

Linguaggio Assembly per PC IBM

SECONDA EDIZIONE

PETER
NORTON

Peter Norton
John Socha



CONTIENE DISCHI

5 1/4" E 3 1/2"

GRUPPO EDITORIALE
JACKSON

Linguaggio Assembly per PC IBM

**Peter Norton
John Socha**



**GRUPPO
EDITORIALE
JACKSON**
Via Rosellini, 12
20124 Milano

Titolo originale

PETER NORTON'S ASSEMBLY LANGUAGE BOOK FOR PC AND XT, Second Edition

© Copyright per l'edizione originale in lingua inglese

Brady Book, una divisione di Simon & Schuster Inc. – 1989

© Copyright per la traduzione in lingua italiana

Gruppo Editoriale Jackson S.p.A. – 1990

Questa edizione è stata pubblicata in accordo con l'editore originale, Brady Books una divisione di Simon & Schuster Inc.

REDATTORE DI COLLANA

Giovanni Perotti

COPERTINA

Emiliano Bernasconi

IMPAGINAZIONE CON TECNICHE DI DESKTOP PUBLISHING

Link S.r.l.

Tutti i diritti sono riservati. Nessuna parte di questo libro può essere riprodotta, memorizzata in sistemi d'archivio, o trasmessa in qualsiasi forma o mezzo, elettronico, meccanico, fotocopia, registrazione o altri, senza la preventiva autorizzazione scritta dell'editore.

Gli autori e l'editore di questo volume si sono fatti carico della preparazione del libro e dei programmi in esso contenuti. Questa attività ha compreso la ricerca, lo sviluppo e il test di teorie e di programmi per determinare le loro funzionalità.

Gli autori e l'editore non si assumono alcuna responsabilità, esplicita o implicita, riguardante questi programmi o il contenuto del testo.

Gli autori e l'editore non potranno in alcun caso essere ritenuti responsabili per incidenti o conseguenti danni che derivino o siano causati dall'uso dei programmi o dal loro funzionamento.

INDICE

Introduzione	XI
Parte I Linguaggio Macchina	1
Capitolo 1 DEBUG E ARITMETICA	3
Numerazione di Base	3
Numeri Esadecimali	4
Debug	4
Aritmetica Esadecimale	5
Conversione da Esadecimale a Decimale	7
Numeri Esadecimali a 5 Cifre	10
Conversione da Decimale a Esadecimale	11
Numeri Negativi	12
Bit, Byte, Parole, e Notazione Binaria	14
Complemento a Due - Uno Strano Tipo di Numero Negativo	16
Sommario	18
Capitolo 2 ARITMETICA DELL'8088	19
Registri come Variabili	19
La Memoria e l'8088	20
Addizione con l'8088	23
Sottrazione con l'8088	25
Numeri Negativi nell'8088	26
I Byte nell'8088	26
Moltiplicazione e Divisione con l'8088	28
Sommario	30
Capitolo 3 VISUALIZZAZIONE DEI CARATTERI	31
INT - La Potenza dell'Interrupt	31
Un'Uscita Elegante - INT 20h	33
Un Programma di Due Righe - Unire le Parti	34
Inserire i Programmi	35

Spostare i Dati nei Registri	36
Scrivere una Stringa di Caratteri	38
Sommario	40
Capitolo 4 VISUALIZZARE DEI NUMERI BINARI	41
Rotazioni e Flag di Riporto	41
Sommare con il Flag di Riporto	43
Cicli	44
Scrivere un Numero Binario	46
Il Comando Proceed	47
Sommario	47
Capitolo 5 VISUALIZZARE I NUMERI IN ESADECIMALE	49
Bit di Confronto e Bit di Stato	49
Visualizzare una Singola Cifra Esadecimale	52
Un'Altra Istruzione di Rotazione	54
AND Logico	56
Riunire le Parti	57
Sommario	58
Capitolo 6 LETTURA DEI CARATTERI	59
Leggere un Carattere	59
Leggere un Numero Composto da Una Cifra Esadecimale	60
Leggere un Numero Composto da Due Cifre Esadecimali	60
Sommario	62
Capitolo 7 LE PROCEDURE - PARENTI DELLE SUBROUTINE	63
Le Procedure	63
Lo Stack e gli Indirizzi di Ritorno	65
Le Istruzioni PUSH e POP	67
Leggere dei Numeri Esadecimali in Modo Elegante	68
Sommario	70
Parte II Linguaggio Assembly	71
Capitolo 8 L'ASSEMBLATORE	73
Un programma senza debug	73
Creare i file sorgente	76
Linking	76
Ancora il debug	78

Commenti	78
Etichette	79
Sommario	81
Capitolo 9 LE PROCEDURE E L'ASSEMBLATORE	83
Le Procedure dell'Assemblatore	83
Le Procedure di Output Esadecimale	85
Introduzione alla Progettazione Modulare	88
Lo Scheletro di un Programma	89
Sommario	90
Capitolo 10 VISUALIZZAZIONE IN DECIMALE	91
Ritorno alla Conversione	91
Alcuni Accorgimenti	93
Il Funzionamento Interno	95
Sommario	96
Capitolo 11 I SEGMENTI	97
Dividere la Memoria dell'8088	97
Lo Stack	101
Il Program Segment Prefix (PSP)	102
La Direttiva DOSSEG	104
Chiamate Near e Far	104
Altre Informazioni sull'Istruzione INT	106
I Vettori di Interrupt	108
Sommario	108
Capitolo 12 IMPOSTAZIONE DEL LAVORO	111
Dischetti, Settori e Argomenti Simili	111
Il Punto della Situazione	114
Sommario	114
Capitolo 13 LA PROGETTAZIONE MODULARE	117
Assemblaggio Separato	117
Le Tre Leggi della Progettazione Modulare	120
Sommario	123
Capitolo 14 VISUALIZZAZIONE DELLA MEMORIA	125
Modi di Indirizzamento	125
Il Segmento Dati	128

Indirizzamento Base-Relativo	130
Impostazione di DS	131
Aggiungere dei Caratteri alla Stampa	132
Visualizzare 256 Byte di Memoria	134
Sommario	138
Capitolo 15 VISUALIZZARE UN SETTORE DEL DISCO	141
Semplificare il Lavoro	141
Formato del File Make	142
Il Make di OPTASM	143
Modificare DISP_SEC	144
Leggere un Settore	145
La Direttiva .DATA?	149
Sommario	150
Capitolo 16 MIGLIORARE LA VISUALIZZAZIONE DEI SETTORI	153
Aggiungere dei Caratteri Grafici	153
Aggiungere gli Indirizzi alla Visualizzazione	155
Aggiungere delle Linee Orizzontali	159
Aggiungere dei Numeri alla Visualizzazione	163
Sommario	166
Parte III Rom Bios del Pc IBM	167
Capitolo 17 LE ROUTINE DELLA ROM BIOS	169
VIDEO_IO, le Routine della ROM BIOS	169
Cancellare lo Schermo	172
Spostare il cursore	174
Modifica dell'Uso delle Variabili	176
Scrivere l'Intestazione	179
Sommario	183
Capitolo 18 WRITE_CHAR	185
La Nuova WRITE_CHAR	186
Cancellare Fino alla Fine della Riga	189
Sommario	191
Capitolo 19 LE ROUTINE DI SMISTAMENTO	193
Le Routine di Smistamento	193
Leggere Altri Settori	199
Filosofia dei Capitoli Successivi	201

Capitolo 20	UNA SFIDA ALLA PROGRAMMAZIONE	203
Il Corsore Fantasma	203
Un Semplice Editing	204
Aggiunte e Cambiamenti a Dskpatch	205
Capitolo 21	I CURSORI FANTASMA	207
I Cursori Fantasma	207
Cambiare gli Attributi del Carattere	213
Sommario	214
Capitolo 22	UN SEMPLICE EDITING	215
Spostare i Cursori Fantasma	215
Un Semplice Editing	218
Sommario	221
Capitolo 23	INPUT ESADECIMALE E DECIMALE	223
Input Esadecimale	223
Input Decimale	230
Sommario	233
Capitolo 24	MIGLIORAMENTO DELL'INPUT DI TASTIERA	235
Una Nuova READ_STRING	235
Semplice per l'Utente o Semplice per il Programmatore	241
Sommario	242
Capitolo 25	ALLA RICERCA DEGLI ERRORI	243
Risolvere i Problemi di DISPATCHER	243
Sommario	245
Capitolo 26	SCRIVERE I SETTORI MODIFICATI	247
Scrivere sul Disco	247
Altre Tecniche di Collaudo	249
Costruire una Mappa	250
Tracciare gli Errori	252
Collaudo a Livello Sorgente	254
Microsoft CodeView	254
Borland Turbo Debugger	256
Sommario	260

Capitolo 27	L'ALTRO MEZZO SETTORE	261
	Scorrere di Mezzo Settore	261
	Sommario	264
Parte IV	Caratteristiche Avanzate	265
Capitolo 28	RILOCAZIONE	267
	Programmi .COM	267
	Rilocazione	268
	Programmi .COM e Programmi .EXE	271
Capitolo 29	DETTAGLI SUI SEGMENTI E SU ASSUME	275
	Sovrapposizione del Segmento	275
	Un Altro Sguardo ad ASSUME	277
	Sommario	278
Capitolo 30	UNA WRITE_CHAR MOLTO VELOCE	279
	Il Segmento di Schermo	279
	Organizzazione della Memoria Video	281
	Alta Velocità	283
	Sommario	290
Capitolo 31	PROCEDURE C IN ASSEMBLY	291
	Una Procedura per Cancellare lo Schermo per il C	291
	Passare i Parametri	295
	Un Esempio a Due Parametri	300
	Fornire i Valori delle Funzioni	301
	Sommario	302
Capitolo 32	DISKLITE, UN PROGRAMMA RESIDENTE IN RAM	303
	I Programmi Residenti in RAM	303
	Intercettare gli Interrupt	303
	Disklite	305
APPENDICE A GUIDA AL DISCO	311	
	Gli Esempi dei Capitoli	311
	Una versione Avanzata di Dskpatch	312

APPENDICE B LISTATI DI DSKPATCH	317
Descrizione delle Procedure	317
Listati del Programma per le Procedure di Dskpatch	322
DSKPATCH Make File	322
DSKPATCH Linkinfo File	322
CURSOR.ASM	323
DISK_IO.ASM	327
DISPATCH.ASM	330
DISP_SEC.ASM	332
DSKPATCH.ASM	338
EDITOR.ASM	340
KBD_IO.ASM	342
PHANTOM.ASM	350
VIDEO_IO.ASM	356
APPENDICE C MESSAGGI DI ERRORE COMUNI	363
MASM	363
LINK	364
EXE2BIN	365
APPENDICE D TABELLE VARIE	367
Codici ASCII	367
Codici dei Colori	369
Codici Estesi di Tastiera	370
Tabella delle Modalità di Indirizzamento	371
Le Funzioni di INT 10h	372
Le Funzioni di INT 16h	375
Le Funzioni di INT 21h	376
Le Funzioni per Leggere/Scrivere i Settori	378
Indice Analitico	379

INTRODUZIONE

MARCHI REGISTRATI

IBM, IBM PC, XT, e AT sono marchi registrati della International Business Machines Corporation.

COMPAQ è un marchio registrato della Compaq Computer Corporation.

MS-DOS e Microsoft sono marchi registrati della Microsoft Corporation.

SideKick e SuperKey sono marchi registrati della Borland International.

ProKey è un marchio registrato della Rosesoft.

Lotus e 1-2-3 sono marchi registrati della Lotus Development Corporation.

Intel è un marchio registrato della Intel Corporation.

LIMITI DI RESPONSABILITÀ E TERMINI DI GARANZIA

Gli autori e l'editore di questo libro hanno prodotto il massimo sforzo nella preparazione di questo libro e dei programmi in esso contenuti. Questo sforzo include lo sviluppo, la ricerca, e il controllo dei concetti e dei programmi esposti nel libro. Gli autori e l'editore non forniscono garanzia di alcun tipo, esplicita o implicita, riguardo alla documentazione e ai programmi contenuti in questo libro. Gli autori e l'editore non possono essere considerati responsabili in nessun caso di danni conseguenti o derivanti dalla fornitura, dalla prestazione o dall'utilizzo di questi programmi.

INTRODUZIONE

Una volta finito di leggere questo libro, sarete in grado di scrivere dei veri e propri programmi in assembly: editor di testo, utility, e così via. Durante la lettura, imparerete molte delle tecniche utilizzate dai programmatori professionisti che risulteranno molto utili per semplificare il lavoro. Queste tecniche, che includono la progettazione modulare e il perfezionamento a fasi, raddoppieranno o triplicheranno la vostra velocità di programmazione, permettendovi di scrivere programmi più leggibili e affidabili.

La tecnica del perfezionamento a fasi, in particolare, semplifica notevolmente la stesura di programmi complessi. Questa tecnica è un modo semplice e naturale (e anche divertente!) per scrivere dei programmi. La troverete utile soprattutto nei momenti in cui vi chiederete: "Da dove comincio?".

Questo libro, non è solo teoria. Per spiegare e illustrare meglio determinati concetti, vi faremo creare un programma, chiamato Dskpatch (per Disk Patch, cioè correzione di disco), che troverete utile per diverse ragioni. Innanzitutto, durante la creazione di questo programma avrete l'opportunità di vedere praticamente come vengono utilizzate le tecniche della progettazione modulare e del perfezionamento a fasi; in secondo luogo, il programma che creerete non sarà fine a se stesso ma, come editor di settori di disco, risulterà utile in svariate circostanze.

PERCHÉ LINGUAGGIO ASSEMBLY?

Presumiamo che abbiate acquistato questo libro per apprendere il linguaggio assembly; ma forse non sapete esattamente perché volete imparare questo linguaggio. Una ragione, forse la meno ovvia, è che i programmi in linguaggio assembly sono alla base di qualsiasi computer PC IBM (con tale dicitura, in questo libro, ci riferiremo sempre a qualsiasi computer PC, AT, PS/2 o compatibile). Il linguaggio assembly è quello che si differenzia maggiormente tra tutti i linguaggi disponibili e, rispetto ai linguaggi di alto livello, è quello che si avvicina di più alla macchina. Per questo motivo, imparare il linguaggio assembly significa anche imparare a conoscere il microprocessore del computer che può essere un 8088, 80286 o 80386. (8088, in questo libro, si riferirà sempre alla famiglia dei microprocessori 8088, 80286 e 80386.) Imparerete le istruzioni del microprocessore 8088 e verrete a conoscenza di argomenti avanzati che saranno di incalcolabile valore durante la stesura dei vostri programmi in assembly.

Una volta conosciuto il microprocessore del vostro computer, molti elementi visti in altri programmi e in linguaggi di alto livello acquisteranno un nuovo significato. Per esempio, potreste aver notato che il numero intero più grosso gestibile in BASIC è 32767. Da dove spunta questo numero? E' un numero strano per essere un limite massimo ma, come vedrete in seguito, il numero 32767 è direttamente collegato al modo in cui il PC IBM memorizza i numeri.

Inoltre, potreste essere interessati alla velocità e alla dimensione dei programmi. Un

programma in assembly, infatti, risulta molto più veloce di un programma scritto in qualsiasi altro linguaggio. Generalmente, i programmi in assembly risultano due o tre volte più veloci dei programmi equivalenti scritti in C o Pascal, e addirittura 15 volte più veloci di quelli in BASIC interpretato. Anche le dimensioni sono decisamente più contenute. Il programma che creerete in questo libro, DSKPATCH, sarà di circa un kilobyte (veramente poco rispetto ad altri programmi). Un programma simile scritto in C o in Pascal occuperebbe circa dieci volte tanto. Queste sono alcune delle ragioni per cui la Lotus Development Corporation ha scritto 1-2-3 interamente in linguaggio assembly.

I programmi in assembly offrono inoltre un accesso totale a tutte le capacità del computer. Alcuni programmi, tra cui SideKick, ProKey e SuperKey, una volta caricati restano residenti in memoria. Questi programmi cambiano il modo in cui lavora il computer e sfruttano delle caratteristiche disponibili solamente tramite l'assembly. Mostriamo come scrivere questi programmi alla fine del libro.

DSKPATCH

Con questo programma sarete in grado di leggere direttamente i settori di un dischetto, visualizzando i numeri e i caratteri memorizzati dal DOS in notazione esadecimale. Dskpatch è un editor a pieno schermo per i dischi, e permette di modificare le informazioni contenute nei settori del disco stesso. Con Dskpatch, per esempio, potreste leggere il settore in cui il DOS memorizza la directory del disco, e cambiare i nomi dei file o altre informazioni. In questo modo imparerete anche a conoscere il modo in cui il DOS memorizza i dati sul disco.

Con Dskpatch si avrà più di un programma, dato che sono incluse circa 50 subroutine, molte delle quali sono multiuso e possono essere utilizzate in altri programmi. Questo libro, quindi, non è solo un'introduzione all'8088 e al linguaggio assembly, ma è anche una fonte di utili subroutine.

Inoltre, qualsiasi editor a pieno schermo deve utilizzare delle caratteristiche specifiche della famiglia dei computer PC IBM. Attraverso gli esempi di questo libro, imparerete a scrivere programmi di utilità per i computer IBM PC, AT o compatibili (come, ad esempio, Olivetti e Compaq).

REQUISITI

Di che cosa avete bisogno per eseguire gli esempi contenuti in questo libro? E' necessario un IBM PC o compatibile con almeno 256K di memoria, un drive e la versione 2.0 o superiore del DOS (MS-DOS o PC-DOS). Per eseguire gli esercizi della seconda parte di questo libro, avete bisogno di un assembler che può essere l'assembler IBM, il Microsoft Macro Assembler versione 5.0 o superiore, il Turbo Assembler della Borland International, o OPTASM della SLR Systems.

ORGANIZZAZIONE DEL LIBRO

Questo libro è diviso in quattro parti, ciascuna delle quali tratta un argomento specifico. Indipendentemente dal fatto che conosciate o meno i microprocessori o il linguaggio assembly, troverete in ogni caso dei paragrafi interessanti.

La prima parte è incentrata sul microprocessore 8088. In questa sezione, svelerete il mistero dei bit, dei byte e del linguaggio macchina. Ciascuno dei sette capitoli contiene una serie di esempi che utilizzano un programma chiamato Debug che viene fornito con il disco del DOS e permette di guardare *all'interno* del microprocessore 8088. Per quanto riguarda la prima sezione, è richiesta solamente una rudimentale conoscenza del BASIC e, ovviamente, del vostro computer.

La seconda parte, che comprende i capitoli da 8 a 16, si sposta sul linguaggio assembly e sulla scrittura di programmi utilizzando un assembler. L'approccio è graduale e, invece di trattare tutte le funzionalità disponibili in un assembler, viene privilegiata una serie di comandi necessaria per scrivere programmi di utilità.

Utilizzerete l'assembler per riscrivere alcuni dei programmi visti nella prima parte del libro e inizierete a creare il programma Dskpatch. Costruirete questo programma lentamente, in modo da apprendere pienamente la tecnica del perfezionamento a fasi. Sarà inoltre analizzata la tecnica della progettazione modulare, molto utile per scrivere programmi comprensibili. Come detto precedentemente, queste tecniche semplificano il lavoro, eliminando le tipiche difficoltà che si incontrano durante la scrittura di programmi in assembly.

Nella terza parte, che comprende i capitoli da 17 a 28, saranno analizzate delle caratteristiche più avanzate come, per esempio, lo spostamento del cursore e la cancellazione dello schermo.

In questa sezione saranno trattate anche le tecniche usate per collaudare e correggere un programma. I programmi in assembly crescono rapidamente e possono raggiungere una lunghezza di due o tre pagine senza produrre ancora i risultati sperati (Dskpatch sarà più lungo). Anche se queste tecniche di correzione saranno utilizzate su programmi di lunghezza superiore a due o tre pagine, potrete trovarle utili anche per revisionare e correggere piccoli programmi.

La quarta parte, infine, analizzerà alcuni argomenti avanzati necessari per scrivere dei veri programmi. I primi due capitoli si soffermeranno sui programmi .COM e sui segmenti, mentre nel terzo capitolo vedrete come scrivere direttamente nella memoria di schermo per ottenere una rapidissima visualizzazione. Imparerete quindi a scrivere delle procedure in assembler da utilizzare nei programmi scritti in linguaggio C, e infine saranno trattati i programmi residenti in memoria tramite un esempio.

Ora, senza ulteriore indugio, addentriamoci nel mondo dell'8088 e vediamo il modo in cui vengono memorizzati i numeri.

PARTE I

LINGUAGGIO MACCHINA

DEBUG E ARITMETICA

Prima di iniziare a parlare del linguaggio Assembly, spendiamo qualche parola sui microprocessori. Al giorno d'oggi (1990), ci sono tre tipi principali di microprocessori usati nei computer IBM PC, AT, PS/2 e compatibili: 8088, 80286 e 80386. Il microprocessore 8088 è stato il primo ad essere usato sull'IBM PC ed è, fra quelli citati, il microprocessore più lento e meno potente. Successivamente è stato presentato l'IBM AT, dotato di microprocessore 80286 (circa quattro volte più veloce dell'8088) e che è stato il primo computer in grado di utilizzare il nuovo sistema operativo OS/2. Infine sono usciti i computer con microprocessore 80386, che sono, naturalmente, molto più veloci e potenti dei predecessori.

Sia l'80286 che l'80386 sono compatibili con il microprocessore 8088; ciò significa che i programmi scritti utilizzando le istruzioni dell'8088 possono essere tranquillamente utilizzati con questi microprocessori. Generalmente, nei programmi scritti per i computer MS-DOS, sono state utilizzate solo le istruzioni dell'8088; questo garantisce una piena compatibilità su tutti i computer. Dato che tutti i programmi presentati in questo libro si riferiscono esclusivamente al microprocessore 8088, potrete utilizzarli su qualsiasi computer MS-DOS. Quindi, quando parliamo di 8088, ci riferiamo anche ai microprocessori 80286 e 80386 (e, tra non molto, anche 80486).

NUMERAZIONE DI BASE

Iniziamo l'avventura nel linguaggio assembly imparando come contano i computer. La questione potrebbe sembrare già risolta; in fondo noi contiamo fino a 11 partendo da uno e proseguendo nel modo seguente: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11.

Ma un computer non conta in questo modo; per contare fino a cinque procede nel modo seguente: 1, 10, 11, 100, 101. I numeri 10, 11, 100 ecc. sono numeri binari, basati su un sistema numerico con sole due cifre (0 e 1). In pratica si usa un sistema in base due invece del consueto sistema numerico decimale. Quindi, il numero binario 10 è l'equivalente del numero 2 nel sistema decimale.

Il microprocessore 8088 utilizza il sistema binario per manipolare i numeri. Tuttavia, è impensabile per un uomo l'utilizzo di questo sistema; infatti, nel momento in cui si devono utilizzare dei grossi numeri, è necessario scrivere una lunghissima serie di zero e uno. Per questo motivo è stato introdotto il sistema esadecimale, un metodo molto più compatto per scrivere i numeri binari. In questo capitolo, imparerete a utilizzare i numeri esadecimali e quelli binari e il metodo usato dal computer per

memorizzarli (in bit, byte e parole).

Se conoscete già il sistema binario e il sistema esadecimale, i bit, i byte e le parole, potete passare al capitolo successivo.

NUMERI ESADECIMALI

Dato che i numeri esadecimale possono essere gestiti più facilmente dei numeri binari (almeno in termini di lunghezza), inizierete usando gli esadecimale e il programma DEBUG.COM, che potrete trovare sul dischetto del DOS. Il programma Debug sarà usato in questo capitolo e nei successivi per inserire ed eseguire programmi in linguaggio macchina, una istruzione alla volta. Come il BASIC, Debug fornisce un ambiente interattivo ma, a differenza del BASIC, non riconosce i numeri decimali. Nell'ambiente del Debug, 10 corrisponde a un numero esadecimale e dato che Debug accetta solamente numeri esadecimale, dovrete imparare questo nuovo sistema. Ma prima di questo, vediamo brevemente come funziona il programma Debug.

DEBUG

Da dove deriva questo nome? Bug (letteralmente insetto), nel gergo informatico significa errore in un programma. Un programma che funziona non ha bug, mentre un programma che non funziona o funziona parzialmente contiene almeno un errore (bug). Con il programma Debug, è possibile eseguire un programma una istruzione alla volta controllando quindi il funzionamento del programma in modo da correggerne gli errori. Questa tecnica è conosciuta come debugging (collaudo) e da qui deriva il nome.

Si dice che il termine debugging sia nato agli albori dell'informatica, quando un giorno il computer Mark I di Harvard si bloccò. Dopo una lunga ricerca, i tecnici trovarono l'origine del guasto: una piccola tarma era rimasta intrappolata tra i contatti di un relè. I tecnici rimossero la tarma e scrissero un appunto sull'eliminazione degli insetti (debugging) dal Mark I.

Potrete trovare il programma Debug sul dischetto supplementare del DOS. Se non si dispone di un disco fisso, converrà copiare DEBUG.COM su un dischetto nuovo formattato dato che in questa sezione si farà un uso intensivo di questo programma.

Nota: Da questo momento in poi, nelle sessioni interattive come questa, il testo che dovete digitare sarà stampato in grassetto, in modo da distinguerlo dalle risposte del computer:

```
A>DEBUG
```

In questo caso, dovrete digitare il testo in grassetto e premere Invio; dovrete quindi ottenere una risposta simile a quelle riportate in queste

sessioni. Potreste non vedere sempre le stesse risposte, dato che il computer potrebbe avere una situazione di memoria differente da quella mostrata negli esempi. (Inizierete a trovare le prime differenze nel prossimo capitolo). Inoltre, notate che vengono utilizzate le lettere maiuscole in tutti gli esempi. Questo viene fatto solamente per evitare confusione tra alcune lettere e numeri (per esempio, l [elle] e 1 [uno]). Comunque, se preferite, potete usare indifferentemente le lettere maiuscole o minuscole.

Ora, viste queste poche convenzioni, eseguite Debug digitandone il nome dopo il prompt del DOS (nell'esempio A>):

```
A>DEBUG
-
```

Il trattino che appare in risposta al comando è il simbolo di prompt del Debug (proprio come A> che è il prompt del DOS). Questo trattino significa che Debug sta aspettando un comando.

Per uscire da Debug e ritornare al DOS, digitate Q (per Quit, Uscita) e premete Invio. Provate a uscire e, successivamente, richiamate Debug:

```
-Q
A>DEBUG
-
```

Vediamo ora di imparare l'aritmetica esadecimale.

ARITMETICA ESADECIMALE

Userete ora un comando di Debug chiamato H. H è la forma abbreviata di Hexarithmic (aritmetica esadecimale); questo comando viene utilizzato per sommare e sottrarre due numeri esadecimali. Per vedere come funziona questo comando, provate a fare la somma 2+3. Nel sistema decimale il risultato sarebbe 5. E in esadecimale? Assicuratevi di essere ancora nel programma Debug e digitate al prompt (il trattino) il testo seguente:

```
-H 3 2
0005    0001
```

Debug risponde visualizzando la somma (0005) e la differenza (0001) di 3 e 2. Il comando H esegue sia la somma che la sottrazione di due numeri e i risultati sono gli stessi che si sarebbero ottenuti con il sistema decimale: 5 (la somma di 3 e 2) e 1 (la sottrazione di 3 e 2). A volte, però, si possono incontrare delle sorprese. Per esempio, provate a digitare H 2 3; in questo caso Debug calcolerà 2+3 e 2-3 (invece di 3-2):

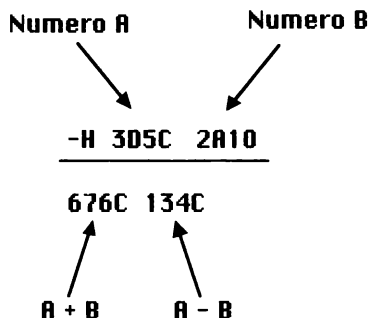


Figura 1-1 . Il Comando H

```
-H 2 3
0005    FFFF
```

Debug ha risposto con FFFF invece di -1 (2-3). Per quanto possa sembrare strano, FFFF è un numero ed è l'equivalente esadecimale di -1.

Torneremo tra poco su questa forma inusuale di -1. Vediamo prima altri esempi utilizzando numeri più grandi, per capire quando può apparire questa F.

Provate a utilizzare il comando H utilizzando i numeri 9 e 1; la somma, in notazione decimale, produrrebbe 10.:

```
-H 9 1
000A    0008
```

Perché 9+1=A? Perché A è l'equivalente esadecimale di 10. Provate ora a generare il numero 15:

```
-H 9 6
000F    0003
```

Se provate altri numeri, troverete in tutto 16 cifre, da 0 a 9 e da A a F. Il nome esadecimale deriva da esa (6) più deca (10) che, combinati, rappresentano 16. I numeri da 0 a 9 sono uguali in entrambi i sistemi (decimale e esadecimale), mentre le cifre da A a F esadecimali corrispondono ai decimali da 10 a 15.

Perché Debug utilizza il sistema esadecimale? Vedrete presto che è possibile scrivere 256 numeri differenti con due sole cifre. Come forse sospettate, 256 ha anche una relazione con l'unità chiamata byte, e il byte ha un ruolo importante nei computer e in questo libro. Troverete delle informazioni sui byte verso la fine di questo capitolo; per ora concentratevi sui numeri esadecimali, l'unico sistema numerico che Debug è in grado di utilizzare.

<u>Decimale</u>	<u>Numeri esadecimali</u>
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F

Figura 1-2. Numeri esadecimali

CONVERSIONE DA ESADECIMALE A DECIMALE

Poco fa abbiamo analizzato dei numeri esadecimali a una cifra; vediamo ora come gestire numeri esadecimali più grandi e come convertirli in numeri decimali. Come nel sistema decimale, per costruire un numero esadecimale a più cifre è necessario aggiungere le cifre a sinistra. Supponete, per esempio, di voler sommare al numero 1 il più grosso numero decimale a una cifra, 9. Il risultato sarà un numero a due cifre, 10. Cosa succede se sommiamo a 1 il numero più grosso esadecimale a una cifra (cioè F)? Il risultato sarà ancora 10.

Ma attenzione perché il numero 10 in esadecimale equivale a 16 nel sistema decimale. Questo potrebbe creare un po' di confusione; per questo motivo, d'ora in avanti, aggiungeremo una h di fianco a un numero esadecimale. Saprete quindi che 10h corrisponde a 16 decimale, mentre 10 è dieci nel sistema decimale.

$$7 \rightarrow 7 * 16 = 112$$

$$C \rightarrow 12 * 1 = 12$$

$$7Ch = 124$$

$$3 \rightarrow 3 * 256 = 768$$

$$F \rightarrow 15 * 16 = 240$$

$$9 \rightarrow 9 * 1 = 9$$

$$3F9h = 1,017$$

$$A \rightarrow 10 * 4,096 = 40,960$$

$$F \rightarrow 15 * 256 = 3,840$$

$$1 \rightarrow 1 * 16 = 16$$

$$C \rightarrow 12 * 1 = 12$$

$$AF1Ch = 44,828$$

$$3 \rightarrow 3 * 65,536 = 196,608$$

$$B \rightarrow 11 * 4,096 = 45,056$$

$$8 \rightarrow 8 * 256 = 2,048$$

$$D \rightarrow 13 * 16 = 208$$

$$2 \rightarrow 2 * 1 = 2$$

$$3B8D2h = 243,922$$

Figura 1-3. Conversioni esadecimale-decimale

Vediamo ora di spiegare come convertire i numeri da esadecimale a decimale e viceversa. Sapete che 10h corrisponde a 16, ma si possono convertire numeri esadecimale più grossi come, per esempio, D3h senza contare da 10h fino a D3h? O come si fa a convertire il numero decimale 173 nel corrispondente esadecimale?

Non possiamo fare affidamento su Debug, dato che questo accetta solo notazione esadecimale. Nel capitolo 10, scriverete un programma per convertire un numero esadecimale in decimale in modo che i programmi possano "parlarci" in decimale. Ma ora dovete fare questa conversione a mano; iniziate ritornando al mondo decimale, che ci è senz'altro più familiare.

Che cosa significa il numero 276? A scuola avete imparato che 276 significa due centinaia, sette decine e sei unità o, graficamente:

$$\begin{array}{r}
 2 \quad * \quad 100 = \quad 200 \\
 7 \quad * \quad 10 = \quad 70 \\
 \hline
 6 \quad * \quad 1 = \quad 6 \\
 276 \quad \quad \quad = \quad 276
 \end{array}$$

Questa rappresentazione aiuta a capire il significato delle cifre. E' possibile usare lo stesso metodo grafico con un numero esadecimale? Certamente.

Considerate il numero D3h menzionato precedentemente. D è l'equivalente di 13 e, dato che ci sono 16 cifre esadecimali contro le 10 decimali, D3h significa tredici volte sedici e tre unità o, rappresentato graficamente:

$$\begin{array}{r}
 D \rightarrow 13 \quad * \quad 16 = \quad 208 \\
 3 \rightarrow 3 \quad * \quad 1 = \quad 6 \\
 \hline
 D3h \quad \quad \quad = \quad 211
 \end{array}$$

Per quanto riguarda il numero decimale 276, abbiamo moltiplicato le cifre per 100, 10, e 1; per il numero esadecimale D3, abbiamo moltiplicato le cifre per 16 e 1. Se avessimo avuto quattro cifre decimali le avremmo moltiplicate per 1000, 100, 10, e 1. Che numeri avremmo dovuto usare per un numero esadecimale a quattro cifre?

Nel caso dei decimali, 100, 100, 10, e 1 sono tutte potenze di 10:

$$\begin{array}{l}
 10^3 = 1000 \\
 10^2 = 100 \\
 10^1 = 10 \\
 10^0 = 1
 \end{array}$$

Possiamo usare lo stesso metodo per le cifre esadecimali servendoci però delle potenze di 16 invece che di 10. I numeri saranno quindi:

$$\begin{array}{l}
 16^3 = 4096 \\
 16^2 = 256 \\
 16^1 = 16 \\
 16^0 = 1
 \end{array}$$

Provate a convertire 3AC8h in decimale usando i quattro numeri appena calcolati:

$$\begin{array}{r}
 3 \quad \rightarrow \quad 3 \quad * \quad 4096 = \quad 12288 \\
 A \quad \rightarrow \quad 10 \quad * \quad 256 = \quad 2560 \\
 C \quad \rightarrow \quad 12 \quad * \quad 16 = \quad 192 \\
 8 \quad \rightarrow \quad 8 \quad * \quad 1 = \quad 8 \\
 \hline
 3AC8h \quad \quad \quad = \quad 15048
 \end{array}$$

Guardate ora cosa succede quando sommate due numeri esadecimali che hanno più di una cifra. Per questo, usate Debug e i numeri 3A7h e 1EDh:

```
-H 3A7 1Ed
0594 01BA
```

$\begin{array}{r} 1 \\ 3A7 \\ + 92A \\ \hline CD1 \end{array}$	$\begin{array}{r} 1 \\ F451 \\ + CB03 \\ \hline 1BF54 \end{array}$	$\begin{array}{r} 1 \\ C \\ + D \\ \hline 19 \end{array}$
$\begin{array}{r} 1111 \\ BCD8 \\ + FAE9 \\ \hline 1B7C1 \end{array}$	$\begin{array}{r} 11 \\ BCD8 \\ + 0509 \\ \hline C1E1 \end{array}$	

Figura 1-4. Esempi di addizioni esadecimali

Avete visto che $3A7h + 1EDh = 594h$. Potete controllare il risultato convertendo questi numeri in decimali e facendo l'addizione (e la sottrazione) in forma decimale; se ve la sentite, fate il calcolo direttamente in esadecimale.

NUMERI ESADECIMALI A 5 CIFRE

Al momento l'aritmetica esadecimale è abbastanza comprensibile. Ma cosa succede quando si vogliono gestire numeri ancora più grossi? Provate con un numero esadecimale a cinque cifre:

```
-H 5C3F0 4BC6
      ^ Error
-
```

Una risposta inaspettata. Perché Debug visualizza un errore? La ragione va ricercata nell'unità di memorizzazione chiamata *parola*. Il comando di aritmetica esadecimale di Debug funziona solo con parole e queste parole possono contenere al massimo quattro cifre.

Torneremo su questo argomento più avanti; al momento ricordatevi che potete lavorare solo con numeri esadecimali a quattro cifre. Quindi, se cercate di sommare due numeri a quattro cifre, per esempio C000h e D000h, otterrete 9000h invece di 19000h:

```
-H C000 D000
9000 F000
```

Debug conserva solo le ultime quattro cifre del numero.

CONVERSIONE DA DECIMALE A ESADECIMALE

Poco fa avete imparato a convertire un numero da esadecimale a decimale. Vediamo ora come effettuare l'operazione inversa: convertire un decimale in esadecimale. Nel capitolo 23 scriverete un programma che effettuerà automaticamente questa conversione ma, come già fatto in precedenza, iniziate imparando a fare questa conversione a mano. Richiamiamo ancora una volta l'aritmetica della scuola elementare. A scuola avete imparato che dividendo 9 per 2 si ottiene 4 con il resto di 1. Sarà proprio il resto che dovrete utilizzare per convertire un numero da decimale a esadecimale. Guardate cosa succede quando un numero (in questo caso 493) viene diviso ripetutamente per 10:

$$493 / 10 = 49 \quad \text{resto } 3$$

$$49 / 10 = 4 \quad \text{resto } 9$$

$$4 / 10 = 0 \quad \text{resto } 4$$

4 9 3

Le cifre di 493 sono le stesse del resto in ordine inverso (394). Avete visto nell'ultima sezione che per convertire un numero da esadecimale a decimale è sufficiente cambiare la potenza 10 con la potenza 16. Per la conversione da decimale a esadecimale è possibile dividere per 16 invece che per 10? Certamente! Questo è proprio il metodo di conversione.

Per esempio, trovate l'equivalente esadecimale di 493 dividendo per 16, come mostrato di seguito:

$$493 / 16 = 30 \text{ resto } 13 \quad (\text{Dh})$$

$$30 / 16 = 1 \text{ resto } 14 \quad (\text{Eh})$$

$$1 / 16 = 0 \text{ resto } 1 \quad (\text{1h})$$

$$493 = 1 \text{ E D h}$$

1EDh è l'equivalente esadecimale di 493. In altre parole, continuate a dividere per 16 e formate il numero esadecimale finale dal resto. Questo è tutto ciò che bisogna fare.

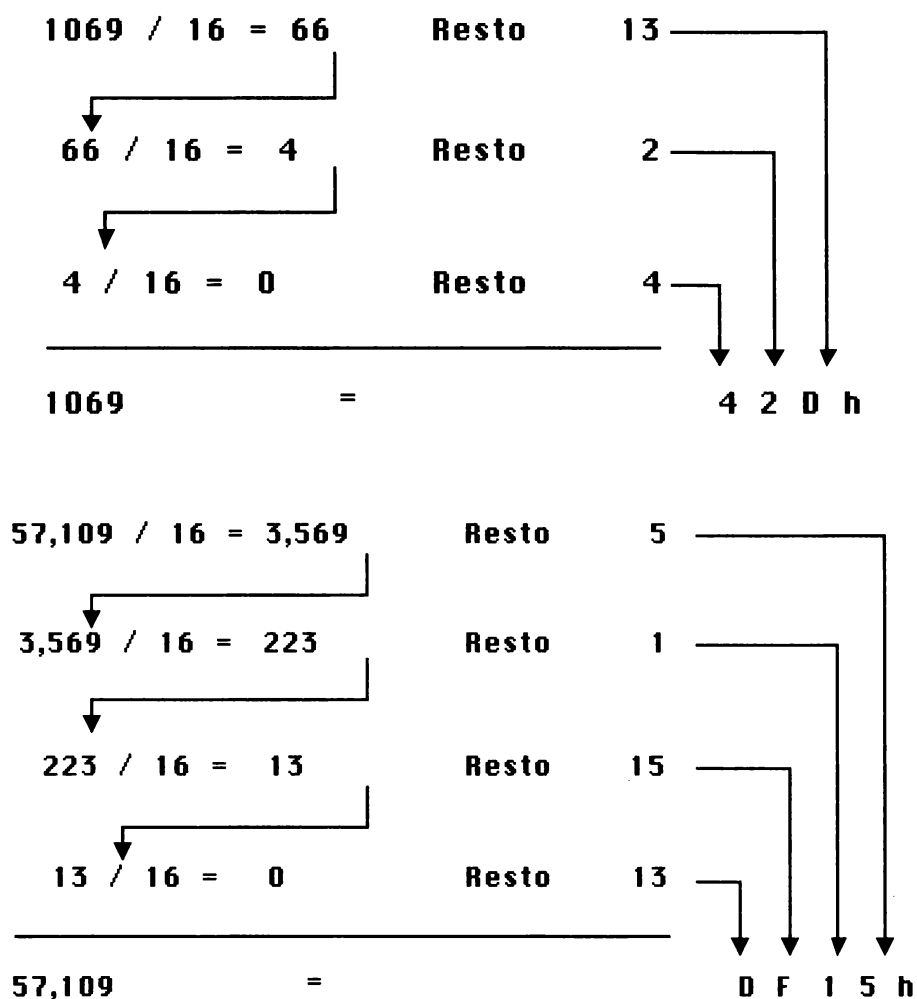


Figura 1-5. Esempi di conversioni esadecimale

NUMERI NEGATIVI

Se vi ricordate, c'è ancora un quesito irrisolto con il numero FFFFh. Abbiamo detto che FFFFh corrisponde a -1 ma, se convertite questo numero, ottenete 65535. Come può essere? Non doveva essere un numero negativo?

Provate a usare il comando H di Debug per sommare 5 e FFFFh; dato che FFFFh corrisponde a -1, il risultato dovrebbe essere 4 (5-1=4):

```
-H 5 FFFF
0004    0006
-
```

Debug *sembra* trattare FFFFh come -1 ma, come vedrete in seguito, FFFFh non sarà sempre trattato come -1 nei vostri programmi. Provate a fare manualmente questa addizione per capire perché Debug agisce in questo modo.

Quando si sommano due numeri decimali capita spesso di dover *riportare* un 1 sulla colonna a sinistra, come nell'esempio seguente:

```
  1 1
   9 5
+  5 8
-----
  1 5 3
```

La somma di due numeri esadecimali non è molto differente. Se sommate 3 a F ottenete 2, con un riporto da aggiungere alla colonna successiva:

```
  1
   F
+  3
-----
  12
```

Guardate ora cosa succede a sommare 5 a FFFFh:

```
  1 1 1 1
   0 0 0 5 h
+  F F F F h
-----
  1 0 0 0 4 h
```

Dato che $Fh + 1h = 10h$, i riporti successivi portano un 1 nella colonna più a sinistra e, se questo 1 viene ignorato, si ottiene il risultato corretto di 5 - 1, cioè 4. Anche se può sembrare strano, FFFFh si comporta come -1 se viene ignorato il riporto. Debug agisce in questo modo perché può gestire solamente numeri esadecimali di quattro cifre; per questo motivo vengono considerate solamente le ultime quattro cifre a destra e viene tralasciato il riporto.

Il risultato ottenuto deve quindi essere considerato esatto o sbagliato? Anche se sembra una contraddizione, bisogna considerare valide entrambe le alternative;

infatti, questo numero, può essere considerato in due modi.

Supponete di considerare FFFFh come 65535. Questo è un numero positivo ed è il numero più grosso rappresentabile con quattro cifre esadecimale. Considerate FFFFh come un numero *senza segno* poiché tutti i numeri di quattro cifre sono stati definiti come numeri positivi. Se aggiungete 5 a FFFFh ottenete 100004h; nessun altro risultato è possibile. Nel caso dei numeri senza segno, un riporto mancato diventa un errore.

D'altro canto, il numero FFFFh può essere trattato come numero negativo, cosa fatta da Debug quando viene utilizzato il comando H. FFFFh diventa -1 ogniqualvolta viene ignorato il riporto. Quindi, tutti i numeri compresi tra 8000h e FFFFh diventano numeri negativi se non viene considerato il riporto. Per i numeri *con segno*, quindi, un riporto mancato non è un errore.

Il microprocessore 8088 può trattare i numeri sia come numeri con segno che come numeri senza segno; la scelta è solo vostra. Ci sono istruzioni differenti per ciascuna forma; imparerete queste differenze più avanti in questo libro, quando inizierete a usare i numeri nella programmazione. Per ora, prima di imparare a scrivere i numeri negativi, (per esempio, 3C8h), dovete conoscere il significato di bit e il suo ruolo all'interno dei byte, delle parole e dei numeri esadecimale.

BIT, BYTE, PAROLE E NOTAZIONE BINARIA

E' venuto il momento di approfondire la conoscenza dell'IBM PC, imparando l'aritmetica dell'8088: i numeri binari. Il microprocessore 8088, nonostante la sua potenza, non è molto intelligente; conosce infatti solamente le cifre 0 e 1. Qualsiasi numero deve essere formato da una lunga stringa di zero e uno. Questo è il sistema numerico *binario* (in base 2).

Quando Debug visualizza un numero esadecimale, usa un piccolo programma incorporato per convertire i numeri dalla forma binaria a quella esadecimale. Nel capitolo 5, costruirete un programma per convertire i numeri da binario a esadecimale; per ora, vediamo di approfondire un po' la conoscenza sul sistema binario.

Prendete il numero binario 1011b (b sta per binario). Questo numero corrisponde al numero decimale 11, o all'esadecimale Bh. Per capire il funzionamento del sistema binario, moltiplicate le cifre di 1011b per la base del numero, 2:

Potenza di 2:

$$\begin{aligned}2^3 &= 8 \\2^2 &= 4 \\2^1 &= 2 \\2^0 &= 1\end{aligned}$$

Quindi:

1	*	8	=	8
0	*	4	=	0
1	*	2	=	2
1	*	1	=	1
<hr/>				
1011b	=	11	o	Bh

<u>Binario</u>	<u>Decimale</u>	<u>Numeri esadecimali</u>
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1111	14	E
1111	15	F

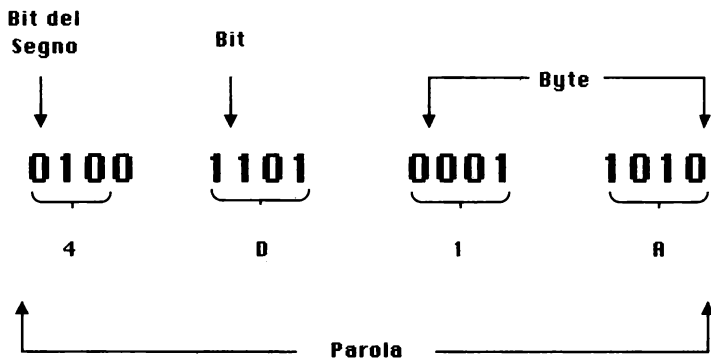


Figura 1-7. Una parola è composta da bit e byte

Similmente, 1111b corrisponde a Fh o 15. Inoltre, 1111b è il numero binario senza segno di quattro cifre più grosso che si possa scrivere, mentre 0000b è il più piccolo. Quindi, con quattro cifre binarie, si possono scrivere 16 numeri differenti. Dato che ci sono esattamente 16 cifre esadecimali, si può scrivere una cifra esadecimale per ogni quattro cifre binarie.

Un numero esadecimale a due cifre, per esempio 4Ch, può essere scritto come 0100 1100b. Questo numero è composto da otto cifre che sono state separate in due gruppi da quattro cifre solo per renderlo più leggibile. Ciascuna di queste cifre binarie è conosciuta come un bit; quindi, un numero come 0100 1100b (o 4Ch), è lungo otto bit.

Molto spesso capiterà di numerare ciascun bit in una stringa lunga, assegnando il numero 0 al bit più a destra. Il numero 1 in 10b sarà il bit numero 1 e il bit più a sinistra in 1011b sarà il bit numero 3. Numerare i bit in questo modo permetterà di parlare di un bit specifico, senza creare confusione.

Un gruppo di otto cifre binarie viene chiamato *byte*, mentre un gruppo di 16 cifre binarie, o due byte, è una *parola*. Questi termini saranno usati molto spesso in questo libro, dato che i bit, i byte e le parole sono la base del microprocessore 8088.

E' ora possibile capire perché è conveniente la notazione esadecimale; due cifre esadecimali formano esattamente un byte (quattro bit per cifra esadecimale), mentre quattro cifre formano una parola. Non si può dire lo stesso per i numeri decimali. Se cercate di usare due cifre decimali per un byte, non potrete scrivere numeri superiori a 99 perdendo quindi i valori compresi tra 100 e 255. Inoltre, usando i numeri decimali a tre cifre, non si potrebbero usare più della metà dei numeri decimali a tre cifre, dato che i numeri compresi tra 256 e 999 non possono essere contenuti in un byte.

COMPLEMENTO A DUE: UNO STRANO TIPO DI NUMERO NEGATIVO

Siete ora pronti per affrontare i numeri negativi. Abbiamo detto precedentemente che i numeri compresi tra 8000h e FFFFh vengono considerati negativi se viene ignorato il riporto. C'è un modo molto semplice per localizzare i numeri negativi quando questi sono scritti in forma binaria:

Numeri Positivi:

0000h	0000 0000 0000 0000b
.	.
.	.
.	.
7FFFh	0111 1111 1111 1111b

Numeri Negativi:	
8000h	1000 0000 0000 0000b
.	.
.	.
.	.
FFFFh	1111 1111 1111 1111b

Nella forma binaria, il bit più a sinistra (il bit 15) per tutti i numeri positivi è sempre 0. Per tutti i numeri negativi, invece, il bit più a sinistra è sempre 1. Questo è il modo utilizzato dal microprocessore 8088 per distinguere i numeri negativi da quelli positivi: il bit 15 è il *bit del segno*. Se in un programma vengono utilizzate delle istruzioni per i numeri senza segno, questo bit viene ignorato. E' quindi compito vostro scegliere se usare numeri con segno o senza segno.

I numeri negativi sono conosciuti come il *complemento a due* dei numeri positivi. Perché complemento? Perché la conversione da un numero positivo (per esempio 3C8h) nella forma complemento a due è un processo a due fasi, la prima delle quali è la conversione del numero nel suo *complemento*.

Non capiterà spesso di trasformare un numero in negativo; imparerete comunque a effettuare questa conversione per vedere come il microprocessore 8088 rende un numero da positivo a negativo. La conversione potrebbe sembrare un po' strana; non vedrete perché funziona, ma vedrete che funziona.

Per trovare il complemento a due (il negativo di) di qualsiasi numero, dovete prima scriverlo in forma binaria, ignorando il segno. Per esempio, 4Ch corrisponde a 0000 0000 0100 1100b.

Per rendere questo numero negativo, dovete innanzitutto cambiare tutti gli zero in uno e viceversa. Questo processo di inversione porta al complemento del numero 4Ch, nel modo seguente:

	0000 0000 0100 1100
diventa:	1111 1111 1011 0011

secondo passo della conversione, consiste nell'aggiungere un 1:

	1 1	
	1 1 1 1 1 1 1 1 1 1 0 0 1 1	
+	1	
	1 1 1 1 1 1 1 1 1 1 0 1 0 0	
	-4Ch = FFB4h	

La risposta, FFB4h, è il risultato che si otterrebbe usando il comando H di Debug per sottrarre 4Ch da 0.

Se desiderate, potete sommare manualmente FFB4h a 4Ch per verificare che la risposta sia 10000h. Da quanto trattato precedentemente sapete che, quando sommate un complemento a due, dovete ignorare l'uno più a sinistra. Quindi, ignorando il riporto, ottenete 0; infatti $(4C + (-4C) = 0)$.

SOMMARIO

In questo capitolo siete stati introdotti nel mondo dei numeri esadecimali e dei numeri binari; se questa è stata la prima volta in cui siete venuti a contatto con questo tipo di numeri, potreste aver incontrato qualche difficoltà. Tuttavia, quando avrete appreso abbastanza per poter conversare in esadecimale con il programma Debug, le difficoltà diminuiranno notevolmente. Ora, rilassatevi un momento e assicuratevi di aver appreso tutti i concetti esposti in questo capitolo prima di continuare.

Avete incontrato il programma Debug e, dato che Debug accetta solo numeri in notazione esadecimale, avete dovuto imparare un nuovo sistema numerico.

Studiando questo sistema, avete imparato a convertire i numeri da esadecimale a decimale e viceversa. Avete quindi appreso il significato dei termini bit, byte, parola e numeri binari (concetti che risulteranno fondamentali per il prosieguo di questo libro).

Avete imparato infine alcune caratteristiche dei numeri negativi, una delle quali è il complemento a due. Siete venuti a contatto con due tipi di numeri, quelli con segno e quelli senza segno; per i primi bisogna sempre considerare il riporto, per gli altri, invece, bisogna ignorarlo.

Tutte queste nozioni risulteranno indispensabili nei capitoli successivi per poter conversare con Debug; Debug farà da interprete tra voi e il microprocessore 8088. Nel prossimo capitolo, vi servirete dei concetti appena appresi per utilizzare il microprocessore 8088. Utilizzerete nuovamente Debug e userete i numeri esadecimali, invece dei numeri binari, per conversare con il computer. Saranno quindi analizzati i registri del microprocessore (dove vengono memorizzati i numeri) e, nel capitolo 3, sarete pronti per scrivere un programma che visualizzerà un carattere sullo schermo. Vedrete anche come effettuare dei calcoli con l'8088; una volta raggiunto il capitolo 10, sarete in grado di scrivere un programma per convertire i numeri binari in numeri decimali.

ARITMETICA DELL'8088

Ora che conoscete un po' di aritmetica esadecimale e binaria, potete iniziare a capire come il microprocessore 8088 effettua i calcoli. L'8088 usa dei comandi interni chiamati *istruzioni*.

REGISTRI COME VARIABILI

Debug, nostro interprete e guida, conosce bene il microprocessore 8088. Userete Debug per capire il funzionamento dell'8088, e inizierete chiedendo a Debug di visualizzare delle piccole aree di memoria, chiamate *registri*, in cui vengono memorizzati i numeri. I registri sono simili alle variabili del BASIC, anche se non sono esattamente la stessa cosa. A differenza del linguaggio BASIC, il microprocessore 8088 contiene un numero fisso di registri, e questi registri non fanno parte delle memoria del computer.

Per visualizzare i registri dell'8088, dovete usare il comando R (per *Registro*) di Debug:

```
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3756 ES=3756 SS=3756 CS=3756 IP=0100 NV UP DI PL NZ NA PO NC
3756:0100 E485          IN      AL,85
-
```

(Potreste vedere dei numeri differenti nella seconda e nella terza riga; questi numeri riflettono la quantità di memoria disponibile nel computer. Continuerete a vedere queste differenze, e successivamente imparerete a capirle).

Per ora Debug ha fornito parecchie informazioni. Concentratevi sui primi quattro registri (AX, BX, CX e DX) che hanno tutti un valore uguale a 0000. Questi registri sono registri per *uso generale*. Gli altri registri, SP, BP, SI, DI, DS, ES, SS, CS e IP sono registri per uso speciale e saranno analizzati in seguito.

Il numero a quattro cifre che segue il nome di ciascun registro è in notazione esadecimale. Nel capitolo 1, avete imparato che una parola è composta da quattro cifre esadecimali. Potete ora vedere che tutti e 13 i registri dell'8088 hanno una lunghezza pari a una parola (o a 16 bit). Questo è il motivo per cui i computer basati sul microprocessore 8088 sono conosciuti come macchine a 16-bit.

Abbiamo detto che i registri sono simili alle variabili del BASIC; questo significa che è possibile modificarli. Il comando R di Debug non si limita a visualizzare i registri ma, se viene seguito dal nome di un registro, oltre a visualizzarlo permette di modificarlo. Per esempio, provate a cambiare il contenuto del registro AX in questo modo:

```
-R AX
AX 0000
:3A7
-
```

Visualizzate nuovamente il registro per vedere se AX contiene ora il valore 3A7h:

```
-R
AX=03A7 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3756 ES=3756 SS=3756 CS=3756 IP=0100 NV UP DI PL NZ NA PO NC
3756:0100 E485          IN      AL, 85
-
```

Lo contiene. E' quindi possibile inserire un qualsiasi numero in un registro utilizzando il comando R, specificando il nome del registro e inserendo il nuovo valore dopo i due punti (:). Da questo momento in avanti, userete sempre il comando R quando sarà necessario inserire dei numeri nei registri dell'8088.

Se vi ricordate, nel capitolo 1 avete usato il comando H per sommare il numero 3A7h a 1EDh; in quel caso, Debug aveva fatto il lavoro da solo. Questa volta, invece, userete Debug solamente come interprete in modo da poter lavorare direttamente con il microprocessore 8088. Dopo aver inserito un valore nel registro BX, imparerete le istruzioni necessarie per sommare il contenuto dei due registri (AX e BX) e per memorizzare il risultato nuovamente nel registro AX. Inserite innanzitutto un numero nel registro BX. Dato che vogliamo sommare 3A7h e 92Ah, usate il comando R per inserire nel registro BX il numero 92Ah.

LA MEMORIA E L'8088

Ora i registri AX e BX dovrebbero contenere rispettivamente 37Ah e 92Ah; verificatelo usando il comando R:

```
-R
AX=03A7 BX=092A CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3756 ES=3756 SS=3756 CS=3756 IP=0100 NV UP DI PL NZ NA PO NC
3756:0100 E485          IN      AL, 85
-
```

Ora che avete inserito due numeri nei registri AX e BX, come potete indicare all'8088 di sommarne il contenuto? Dovete inserire alcuni numeri nella memoria del computer. Il vostro computer avrà almeno 128K di memoria, molta di più di quanta necessaria in questo caso. Inserirete ora due byte di *codice macchina* in un angolo di questa vasta area di memoria. In questo caso, il codice macchina sarà composto da due numeri binari che indicheranno all'8088 di sommare i registri AX e BX. *Eseguirete* quindi questa istruzione per vedere che cosa succede.

In quale parte della memoria bisogna inserire l'istruzione a due byte e come si può indicare al microprocessore il punto in cui trovarla? L'8088 divide la memoria in blocchi da 64K, chiamati *segmenti*. Nella maggior parte dei casi, accederete alla memoria in uno di questi segmenti senza sapere dove inizia effettivamente il segmento. Questo può essere fatto grazie al modo in cui l'8088 etichetta la memoria. Tutti i byte in memoria sono etichettati con dei numeri progressivi che partono da 0h. Ma vi ricordate del limite delle quattro cifre esadecimali? Questo significa che il numero più alto che l'8088 può gestire è l'equivalente del numero decimale 65535; quindi il numero massimo di etichette utilizzabili è 64K (cioè, FFFFh). Nonostante questo, sappiamo che l'8088 può gestire più di 64K di memoria; come può essere? Questo è possibile grazie a un piccolo trucco: l'8088 utilizza due numeri, uno per ciascun segmento di 64K, e uno per ciascun byte, o *scarto* (offset), all'interno del segmento. Ciascun segmento inizia con un multiplo di 16 quindi, sovrapponendo segmenti e scarti, l'8088 può effettivamente etichettare più di 64K di memoria. In questo modo l'8088 può gestire fino a un milione di byte di memoria.

Tutti gli indirizzi (etichette) che utilizzerete, corrispondono alla distanza dall'inizio di un segmento. Scriverete gli indirizzi come un numero di segmento seguito dalla distanza all'interno del segmento. Per esempio, 3756:0100 significa una distanza di 0100h all'interno del segmento 3756h.

Più avanti, nel capitolo 11, analizzeremo più dettagliatamente i segmenti. Per ora affidate a Debug la gestione dei segmenti, in modo da poter lavorare in un segmento senza dovervi preoccupare del suo numero. Inoltre, al momento, i riferimenti agli indirizzi saranno fatti solo in relazione alla loro distanza (scarto) dall'inizio del segmento. Ciascuno di questi indirizzi si riferisce a un byte contenuto in un segmento e, dato che gli indirizzi sono sequenziali, 101h corrisponderà al byte numero 100h nella memoria.

L'istruzione a due byte necessaria per sommare il contenuto dei registri AX e BX assomiglierà alla seguente: ADD AX,BX. Inserite questa istruzione nelle locazioni 100h e 101h, lasciando a Debug la scelta del segmento. Facendo riferimento all'istruzione ADD, diremo che questa si trova nella locazione 100h perché questa è la locazione in cui è contenuto il primo byte dell'istruzione. Il comando di Debug per esaminare e modificare una locazione di memoria è E (per Enter, inserimento). Usate questo comando per inserire i due byte dell'istruzione ADD nel modo seguente:

-E 100

3756:0100 **E4.01**

-E 101

3756:0101 **85.D8**

I numeri 01h e D8h sono l'equivalente in linguaggio macchina dell'istruzione ADD nella locazione di memoria 3756:0100 e 3756:0101. Il numero di segmento potrebbe essere differente, ma questa differenza non influirà sul programma. Similmente, Debug potrebbe visualizzare un diverso numero a due cifre per ciascun comando E impartito. Questi numeri (E4h e 85H nell'esempio) sono i numeri contenuti in quel momento in memoria negli indirizzi 100h e 101h del segmento scelto da Debug; in pratica sono i valori lasciati dal programma che era in memoria prima di eseguire Debug (se avete appena acceso il computer, questi numeri dovrebbero essere 00).

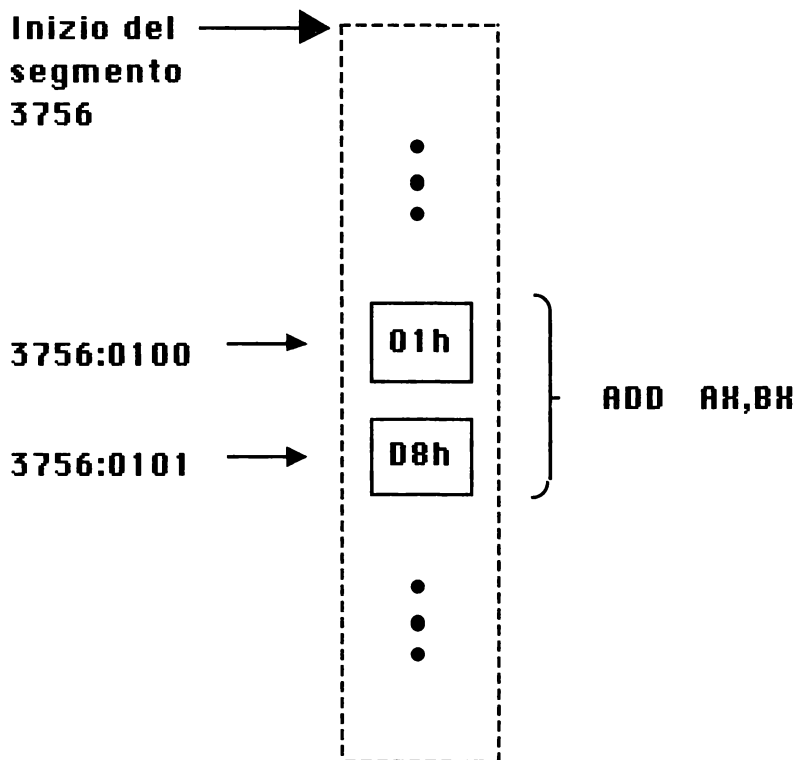


Figure 2-1. L'istruzione inizia 100 byte dall'inizio del segmento

ADDIZIONE CON L'8088

Ora i registri dovrebbero apparire in questo modo:

```
-R
AX=03A7  BX=092A  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3756  ES=3756  SS=3756  CS=3756  IP=0100  NV UP DI PL NZ NA PO NC
3756:0100 01D8          ADD      AX,BX
-
```

L'istruzione ADD è stata posizionata correttamente in memoria. Potete verificarlo guardando la terza riga visualizzata. I primi due numeri, 3756:0100, forniscono l'indirizzo (100h) del primo byte e dell'istruzione ADD. Di fianco a questi numeri potete vedere i due byte che rappresentano l'istruzione ADD:01D8. Il byte uguale a 01h si trova nella locazione 100h, mentre D8h si trova in 101h. Infine, dato che avete inserito l'istruzione in *linguaggio macchina* (numeri che non hanno un significato per voi, ma che vengono interpretati dall'8088 come un'istruzione di somma), il messaggio ADD AX,BX conferma che l'istruzione inserita è corretta.

Anche se avete inserito l'istruzione ADD in memoria, non siete ancora pronti per eseguirla. Dovete infatti dire al microprocessore dove trovare l'istruzione.

L'8088 trova i segmenti e gli indirizzi in due registri speciali, CS e IP, che potete vedere nella lista precedentemente visualizzata. Il numero di segmento viene memorizzato nel registro CS (*Code Segment*, Segmento Codice), che analizzeremo brevemente. Se controllate il valore di questo registro, potete vedere che Debug ha già impostato questo registro (CS=3756 nell'esempio). L'intero indirizzo dell'istruzione, tuttavia, è 3756:0100.

La seconda parte di questo indirizzo (la distanza dall'inizio del segmento), è memorizzata nel registro IP (*Instruction Pointer*, Puntatore di Istruzione). L'8088 usa lo scarto contenuto nel registro IP per trovare il primo byte dell'istruzione. Potete quindi dire al microprocessore dove trovare il primo byte dell'istruzione impostando correttamente questo registro (in questo caso, IP=0100).

Ma, come potete vedere, il registro IP è già impostato correttamente. Debug, infatti, inizialmente imposta sempre questo registro sul valore 0100h. Sapendo questo, vi abbiamo detto di scegliere 100h come primo byte dell'istruzione, in modo da non dover impostare questo registro.

Ora, dato che l'istruzione e i registri sono impostati correttamente, potete eseguire il comando. Userete il comando T (*Trace*, Traccia) di Debug, che esegue un'istruzione alla volta e visualizza quindi i registri. Dopo ciascun passo, il registro IP punterà l'istruzione successiva; in questo caso, punterà 102h. Dato che non avete inserito alcuna istruzione in questa locazione, nell'ultima riga della visualizzazione dei registri vedrete un'istruzione lasciata da qualche programma precedente.

Utilizzate il comando T di Debug per eseguire il comando un'istruzione alla volta:

```

-T
AX=0CD1 BX=092A CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3756 ES=3756 SS=3756 CS=3756 IP=0102 NV UP DI PL NZ AC PE NC
3756:0102 AC          LODSB
-

```

Il registro AX contiene ora CD1h, che è la somma di 3A7h e 92Ah, mentre il registro IP punta l'indirizzo 102h; per questo motivo, nella terza riga vedete l'istruzione contenuta nella locazione 102h invece di in 100h.

Abbiamo detto precedentemente che il puntatore di istruzione, insieme al registro CS, punta sempre l'istruzione successiva da eseguire. Se digitaste nuovamente T, verrebbe eseguita l'istruzione successiva; non fatelo ora, dato che questa operazione potrebbe bloccare il computer.

AX: 0CD1 BX: 092A

 **ADD AX,BX**
LODSB

Figura 2-2. Prima di aver eseguito l'istruzione ADD

AX: 0CD1 BX: 092A

 **ADD AX,BX**
LODSB

Figura 2-3. Dopo l'esecuzione di ADD

Cosa succederebbe invece se eseguite nuovamente l'istruzione ADD, sommando 92Ah a CD1h e memorizzando il nuovo risultato in AX? Per fare questo, dovete dire all'8088 dove trovare l'istruzione successiva, che deve essere ancora l'istruzione ADD nella locazione 0100h. E' sufficiente impostare il registro IP su 0100h? Provate. Usate il comando R per impostare IP su 100 e visualizzare i registri:

```
AX=0CD1  BX=092A  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3756  ES=3756  SS=3756  CS=3756  IP=0100  NV UP DI PL NZ AC PE NC
3756:0100          ADD      AX,BX
```

Sembra tutto a posto. Provate a eseguire nuovamente il comando T e guardate se il registro AX contiene 15FBh. Funziona.

Nota: Dovreste sempre controllare il registro IP e l'istruzione mostrata nell'ultima riga visualizzata dopo il comando R, prima di eseguire il comando T. In questo modo, sarete sicuri che l'8088 esegua l'istruzione desiderata.

Ora, impostate nuovamente il registro IP su 100h e assicuratevi che i registri AX e BX abbiano i valori seguenti: AX=15FB, BX=092A.

SOTTRAZIONE CON L'8088

Imparerete ora a inserire un'istruzione per sottrarre BX da AX in modo che, dopo due sottrazioni, nel registro AX ritorni il numero 3A7h (il valore da cui siete partiti prima delle due addizioni). Imparerete anche un modo per inserire due byte in memoria più velocemente.

Quando avete inserito i due byte per l'istruzione ADD, avete digitato il comando E due volte: una volta con 0100h come primo indirizzo, e un'altra con 0101h come secondo indirizzo. La procedura ha funzionato, ma è possibile inserire il secondo byte senza impartire nuovamente il comando E; per far questo, è sufficiente separare con uno spazio i due byte. Una volta finito di inserire i due byte, premete il tasto Invio per uscire dalla modalità di inserimento. Provate questo metodo per inserire l'istruzione di sottrazione:

```
-E 100
3756:0100      01.29 D8.D8
```

Se ora visualizzate i registri (ricordatevi di impostare il registro IP su 100h) viene mostrata l'istruzione SUB AX,BX, che sottrae il valore del registro BX da quello del registro AX lasciando il risultato in AX. L'ordine di AX e BX può sembrare invertito, ma l'istruzione è analoga a quella del BASIC (AX=AX-BX) ad eccezione del fatto che

l'8088, a differenza del BASIC, pone sempre il risultato nella prima variabile (registro). Eseguite questa istruzione con il comando T. AX dovrebbe contenere CD1. Cambiate nuovamente il registro IP in modo che punti ancora questa istruzione, ed eseguitemela nuovamente (ricordatevi di controllare prima l'istruzione che appare nell'ultima riga dopo aver impartito il comando R). AX dovrebbe ora essere 03A7.

NUMERI NEGATIVI NELL'8088

Nell'ultimo capitolo avete visto come l'8088 si serve del complemento a due per i numeri negativi. Utilizzate ora l'istruzione SUB per calcolare direttamente i numeri negativi. Provate a vedere se l'8088 considera FFFFh come -1. Se così fosse, sottraendo uno da zero, dovrete ottenere come risultato nel registro AX FFFFh (-1). Impostate AX su zero, BX su uno, IP su 100h ed eseguite quindi il comando T. Succede proprio quello che ci si aspettava: AX=FFFFh.

Provate ora a fare qualche altra sottrazione per farvi un'idea di come funziona l'aritmetica del complemento a due. Provate, per esempio, a sottrarre un due.

I BYTE NELL'8088

Fino a questo momento, tutte le operazioni matematiche sono state fatte utilizzando le parole, cioè quattro cifre esadecimali. Ma il microprocessore 8088 è in grado di effettuare dei calcoli usando i byte.

Dato che una parola è composta da due byte, ciascun registro di uso generale può essere diviso in due byte, conosciuti come *byte alto* (le prime due cifre esadecimali), e *byte basso* (le seconde due cifre esadecimali). Ciascuno di questi registri può essere definito con una lettera (da A a D), seguita da X (che indica una parola), da H (che

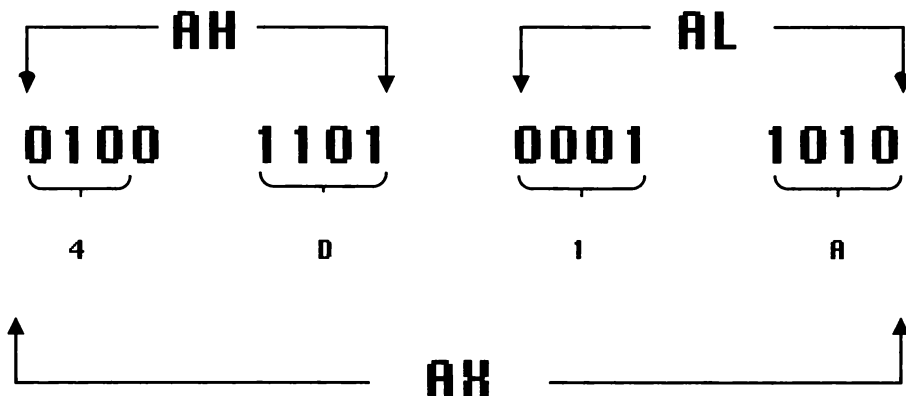


Figura 2-4. Un registro (AX) può essere diviso in due registri di un Byte (AH e AL).

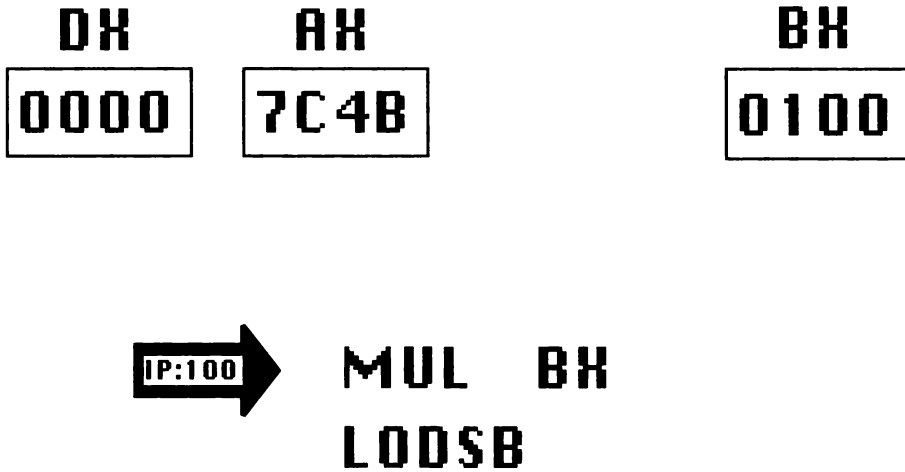


Figura 2-5. Prima dell'esecuzione dell'istruzione MUL

indica un byte alto) o da L (che indica un byte basso). Per esempio, DL e DH sono due registri a un byte (i due byte di un registro), mentre DX è una parola (un registro a due byte). (Questa terminologia potrebbe creare un po' di confusione dato che le parole vengono caricate in memoria ponendo prima il byte basso e dopo il byte alto).

Provate a utilizzare l'istruzione ADD con i byte. Inserite i due byte 00h e C4h partendo dalla locazione 100h. Dopo aver visualizzato i registri, vedrete nell'ultima riga l'istruzione *ADD AH,AL*; questo comando sommerà i due byte del registro AX e porrà il risultato nel byte alto AH.

Ora inserite nel registro AX il valore 102h. Questa operazione pone 01h nel registro AH e 02h nel registro AL. Impostate il registro IP su 100h ed eseguite il comando T.

Vedrete che il registro AX contiene ora 0302h. Il risultato di 01h+02h è 03h e questo valore si trova nel registro AH.

Ma supponete di voler sommare 01h e 03h invece di 01h e 02h. Se il registro AX contiene già 0102h, potete usare Debug per cambiare il registro AL in 03h? No. Dovete cambiare l'intero registro AX in 103h. Perché? Perché Debug permette di modificare solo i registri a due byte (cioè le parole). Non esiste un modo con Debug per cambiare solamente il byte alto o basso di un registro. Ma, come visto nell'ultimo capitolo, questo non è un problema. Con i numeri esadecimali potete dividere una parola in due byte ottenendo due numeri esadecimali a due cifre. Quindi, la parola 103h viene divisa nei due byte 01h e 03h.

Per provare questa istruzione ADD, inserite nel registro AX il valore 103h. L'istruzione *ADD AH,AL* è ancora in memoria nella locazione 100h; impostate quindi il registro IP su 100h e, con i numeri 01h e 03h rispettivamente nei registri AH e AL, eseguite il comando T. Questa volta AX conterrà 0403h e la somma di 01h+03h sarà ora nel registro AH.

MOLTIPLICAZIONE E DIVISIONE CON L'8088

Avete visto come l'8088 somma e sottrarre due numeri. Vediamo ora le operazioni di moltiplicazione e divisione. L'istruzione di moltiplicazione è chiamata MUL e il codice macchina necessario per moltiplicare AX e BX è F7h E3h. Inserirete questa istruzione in memoria, ma spendiamo prima due parole sull'istruzione MUL.

L'istruzione MUL, a differenza di ADD e SUB, non memorizza il risultato nel registro AX; questo è dovuto al fatto che moltiplicando due numeri a 16 bit, viene generato un risultato a 32 bit. Per questo motivo, l'istruzione MUL memorizza il risultato in due registri, DX e AX: i 16 bit alti vengono posti in DX, quelli bassi in AX. D'ora in avanti questa combinazione di registri sarà scritta come DX:AX.

Tornate ora al Debug e inserite l'istruzione di moltiplicazione, F7h E3h, nella locazione 100h, nello stesso modo utilizzato per le istruzioni di addizione e sottrazione. Impostate quindi AX=7C4Bh e BX=100b. L'istruzione sarà *MUL BX* e non verrà fatto alcun riferimento al registro AX. Per moltiplicare delle parole, come in questo caso, l'8088 moltiplica *sempre* il registro specificato nell'istruzione per il registro AX, e memorizza il risultato nella coppia di registri DX:AX.

Prima di eseguire l'istruzione MUL, fate la moltiplicazione manualmente. Come si può calcolare 100h*7C4Bh? Le tre cifre che compongono il numero 100 hanno lo stesso significato sia in decimale che in esadecimale. Quindi, per effettuare questa moltiplicazione, è sufficiente aggiungere due zeri a destra del numero. Si avrà pertanto: 100h*7C4Bh=7C4B00h. Il risultato è troppo grosso per essere contenuto in una parola; dividetelo quindi in due parole, 007Ch e 4B00h.



Figura 2-6. Dopo l'esecuzione di MUL. Il risultato è nella coppia di registri DX:AX

Usate Debug per eseguire l'istruzione passo a passo. Potete vedere che DX contiene la parola 007Ch, mentre AX contiene la parola 4B00h. In altre parole, l'8088 ha inserito il risultato della moltiplicazione nella coppia di registri DX:AX. Dato che la moltiplicazione di due parole non produrrà mai un risultato più lungo di due parole, ma molto spesso (come in questo caso) più lungo di una parola, l'istruzione MUL pone sempre il risultato nella coppia di registri DX:AX.

Parliamo ora della divisione. Quando dividete un numero, l'8088 conserva sia il risultato che il resto della divisione. Provate a vedere come funziona la divisione. Innanzitutto, ponete l'istruzione F7h F3h nelle locazioni 100h e 101h. Come l'istruzione MUL, anche l'istruzione DIV utilizza i registri DX:AX per gestire il risultato. L'istruzione che appare *ora* dovrebbe essere DIV BX. Impostate ora i registri nel modo seguente: DX=007Ch e AX=4B12h; BX dovrebbe contenere ancora 100h.

Calcolate prima il risultato manualmente: $7C4B12h/100h=7C4Bh$ con il resto di 12. Se ora eseguite l'istruzione di divisione, vedrete che AX contiene 7C4Bh (il risultato della divisione), mentre DX contiene 0012h (il resto della divisione). (Utilizzerete il resto della divisione nel capitolo 10, quando scriverete un programma per convertire i numeri da decimale a esadecimale servendovi dei resti, come avete fatto nel capitolo 1).



Figura 2-7. Prima dell'esecuzione dell'istruzione DIV. DIV BX calcola DX:AX/BX

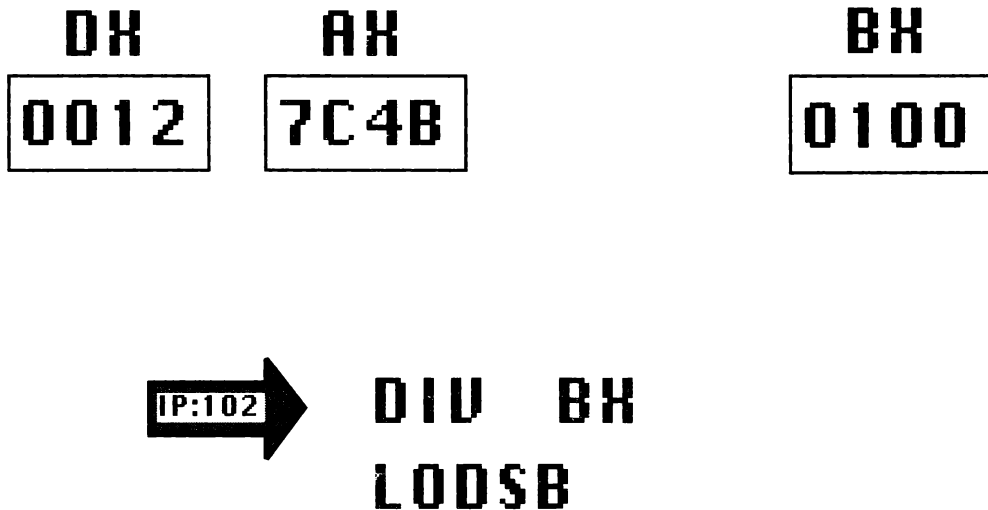


Figura 2-8. Dopo l'esecuzione di *DIV*, il risultato è in *AX*, e il resto è nel registro *DX*

SOMMARIO

E' arrivato il momento di scrivere un vero programma: un programma che visualizzi un carattere sullo schermo. Avete per ora imparato i fondamentali: assicuratevi di avere capito bene i concetti esposti finora prima di proseguire.

In questo capitolo, siete venuti a conoscenza dei registri notando la similitudine con le variabili del BASIC. A differenza del BASIC, tuttavia, avete visto che l'8088 ha un numero fisso di registri. Vi siete soffermati sui registri di uso generale (AX, BX, CX e DX) osservando rapidamente i registri IP e CS, usati per localizzare i segmenti e gli indirizzi.

Dopo aver imparato a cambiare e leggere i registri, avete costruito dei programmi composti da una singola istruzione, inserendo il codice macchina necessario per sommare, sottrarre, moltiplicare e dividere due numeri. Nei capitoli successivi userete molto di ciò che avete appreso qui, ma non dovrete ricordarvi il codice macchina di ciascuna istruzione.

Avete inoltre imparato il comando T di Debug, utile per eseguire un'istruzione passo a passo. Questo comando sarà molto utile anche in futuro. Ovviamente, nel momento in cui un programma comincia ad acquistare una certa dimensione, questo comando diventerà molto più utile ma anche molto più noioso. Imparerete quindi un comando di Debug utile per eseguire più di un'istruzione alla volta.

Torniamo ora ai programmi veri, e vediamo come costruire un programma che visualizzi dei caratteri sullo schermo.

VISUALIZZAZIONE DEI CARATTERI

Avete imparato abbastanza per poter fare qualcosa di concreto; quindi, rimboccatevi le maniche e iniziate a lavorare. Inizierete imparando a visualizzare un carattere sullo schermo, e vi addenterete quindi in operazioni più interessanti. Costruirete un piccolo programma composto da più di un'istruzione e apprenderete un altro metodo per inserire dei dati nei registri (questa volta, direttamente da programma). Vediamo ora come visualizzare un carattere sullo schermo.

INT - LA POTENZA DELL'INTERRUPT

Alle quattro istruzioni che conoscete già (ADD, SUB, MUL e DIV) aggiungiamo l'istruzione INT (per *Interrupt*). INT è simile al comando GOSUB del BASIC. Userete l'istruzione INT per richiedere al DOS di visualizzare sullo schermo il carattere A.

Prima di vedere come funziona INT, facciamo un esempio. Richiamate Debug e inserite 200h nel registro AX e 41h in DX. L'istruzione INT per le funzioni del DOS è INT 21h (in codice macchina, CDh 21h). Questa è un'istruzione a due byte come l'istruzione DIV vista nel capitolo precedente. Inserite INT 21h in memoria partendo dalla locazione 100h, e usate il comando R per assicurarvi che venga visualizzato INT 21h (ricordatevi anche di impostare IP su 100h).

Siete ora pronti per eseguire questa istruzione, ma in questo caso non si può utilizzare il comando T di Debug. Il comando T, infatti, esegue un'istruzione alla volta, ma l'istruzione INT richiama un programma del DOS che effettua l'operazione desiderata (proprio come un'istruzione GOSUB richiama una subroutine).

Dato che non è il caso di seguire passo a passo tutte le istruzioni contenute nella "subroutine" DOS, vediamo come *eseguire* un programma di una riga fermando l'esecuzione prima dell'istruzione contenuta in 102h. Potete usare il comando di Debug G (*GO till*, vai fino) seguito dall'indirizzo in cui si vuole fermare l'esecuzione del programma:

- G 102

A

```
AX=0241 BX=0000 CX=0000 DX=0041 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3970 ES=3970 SS=3970 CS=3970 IP=0102 NV UP DI PL NZ NA PO NC
3970:0102 8BE5          MOV     SP,BP
```

Il DOS ha visualizzato il carattere A e ha ripassato il controllo al vostro programma. (Ricordatevi che l'istruzione in 102h è stata lasciata dal programma precedentemente in memoria; per questo motivo, potreste vedere un'istruzione differente).

Il programma seguente è un programma composto da due istruzioni (la seconda istruzione inizia alla locazione 102h):

```
INT      21
MOV      SP,BP          (o quello che appare sul vostro computer)
```

Sostituirete presto questa seconda istruzione; al momento, dato che non c'era nient'altro da eseguire, avete detto a Debug di eseguire il programma, di fermarsi alla seconda istruzione e di visualizzare i registri.

Ma come faceva il DOS a sapere che volevate visualizzare il carattere A? Il numero 02h nel registro AH è la funzione necessaria per far stampare al DOS un carattere. Un altro numero nel registro AH avrebbe eseguito una funzione differente. (Vedrete altre opzioni successivamente ma, se siete curiosi, potete trovare una lista di funzioni nel manuale tecnico del DOS o nell'appendice E di questo libro).

Il DOS, inoltre, usa il numero contenuto nel registro DL come codice ASCII per il carattere da visualizzare. Il numero inserito in questo registro, 41h, è il codice ASCII della lettera A maiuscola.

Nell'appendice E troverete una tabella contenente i codici ASCII di tutti i caratteri visualizzabili su un IBM PC. Per comodità i codici sono stati riportati sia in notazione decimale che esadecimale. Tuttavia, dato che Debug accetta solo numeri esadecimali, questa potrebbe essere una buona occasione per fare un po' di pratica. Scegliete un carattere dalla tabella e convertite manualmente il codice da decimale a esadecimale. Verificate a questo punto la conversione, inserendo il valore ottenuto nel registro DL ed eseguendo l'istruzione INT (ricordatevi di riportare IP su 100h).

Vi sarete chiesti che cosa sarebbe successo se aveste eseguito il comando T invece del comando G. Provate a fare qualche passo, ma non andate troppo avanti; potreste vedere qualcosa di strano. Comunque, dopo aver eseguito qualche istruzione, uscite da Debug con il comando Q, in modo da eliminare qualsiasi modifica accidentale.

-R

```
AX=0200 BX=0000 CX=0000 DX=0041 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3970 ES=3970 SS=3970 CS=3970 IP=0100 NV UP DI PL NZ NA PO NC
3970:0100 CD21          INT      21
```

-T

```
AX=0200 BX=0000 CX=0000 DX=0041 SP=FFE8 BP=0000 SI=0000 DI=0000
DS=3970 ES=3970 SS=3970 CS=3372 IP=0180 NV UP DI PL NZ NA PO NC
3372:0180 80FC4B      CMP      AH,4B
```

-T

```
AX=0200 BX=0000 CX=0000 DX=0041 SP=FFE8 BP=0000 SI=0000 DI=0000
DS=3970 ES=3970 SS=3970 CS=3372 IP=0183 NV UP DI NG NZ AC PE CY
3372:0183 7405          JZ      018A
```

-T

```

AX=0200 BX=0000 CX=0000 DX=0041 SP=FFE8 BP=0000 SI=0000 DI=0000
DS=3970 ES=3970 SS=3970 CS=3372 IP=0185 NV UP DI NG NZ AC PE CY
3372:0185 2E          CS:
3372:0186 FF2EAB0B    JMP     FAR [0BAB]          CS:0BAB=0BFF

```

-Q

Notate che il primo numero dell'indirizzo è cambiato, da 3970 a 3372. Le ultime tre istruzioni fanno parte del DOS e il programma per il DOS si trova in un altro segmento. Ci sono moltissime altre istruzioni che il DOS esegue prima di visualizzare un carattere; anche se sembra un'operazione semplice, non è così facile come appare. Potete ora capire perché è stato usato il comando G fermando l'esecuzione all'istruzione 102h. In caso contrario, avreste dovuto eseguire numerose istruzioni DOS. (Se state utilizzando una versione DOS differente da quella usata per questo esercizio, potreste vedere qualche differenza).

UN'USCITA ELEGANTE - INT 20H

Ricordate l'istruzione INT 21h? Provate a cambiare 21h in 20h. INT 20h è un'altra istruzione di interrupt e serve per comunicare al DOS che si vuole uscire da un programma, in modo che il controllo di tutte le operazioni torni nuovamente al DOS. Nel nostro caso, l'istruzione INT 20h riporterà il controllo a Debug, dato che eseguite i programmi da Debug invece che dal DOS.

Inserite l'istruzione CDh 20h partendo dalla locazione 100h, e provate quindi l'istruzione seguente (ricordatevi di verificare l'istruzione INT 20h con il comando R):

-G 102

Program terminated normally

-R

```

AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3970 ES=3970 SS=3970 CS=3970 IP=0100 NV UP DI PL NZ NA PO NC
3970:0100 CD20          INT     20

```

-G

Program terminated normally

-R

```

AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3970 ES=3970 SS=3970 CS=3970 IP=0100 NV UP DI PL NZ NA PO NC
3970:0100 CD20          INT     20

```

Con il comando G (non seguito da un numero) viene eseguito tutto il programma (che è composto da una sola istruzione, dato che INT 20h è un'istruzione di *uscita*). IP viene automaticamente riportato su 100h (che è il punto da cui siete partiti). I registri in questo esempio contengono 0 solo perché Debug è stato appena avviato. Potete usare l'istruzione INT 20h alla fine di un programma per riportare il controllo al DOS (o a Debug); provate quindi a combinare le due istruzioni INT appena apprese, in modo da creare un programma di due righe.

UN PROGRAMMA DI DUE RIGHE: UNIRE LE PARTI

Partendo dalla locazione 100h, inserite le due istruzioni INT 21h e INT 20h (CDh 21h CDh 20h) una dopo l'altra. (D'ora in avanti, inizierete tutti i programmi partendo da 100h).

Quando si lavora con una sola istruzione è possibile utilizzare il comando R per visualizzarla, ma ora le istruzioni sono due. In questo caso utilizzate il comando U (*Unassemble*, Disassembla), che è simile al comando List del BASIC:

```

-U 100
3970:0100    CD21          INT  21
3970:0102    CD20          INT  20
3970:0104    4F            DEC  DI
3970:0105    53            PUSH BX
3970:0106    207665        AND  [BP+65],DH
3970:0109    7273          JB   017E
3970:010B    69            DB   69
3970:010C    6F            DB   6F
3970:010D    6E            DB   6E
3970:010E    2032          AND  [BP+SI],DH
3970:0110    2E            CS:
3970:0111    3020          XOR  [BX+SI],AH
3970:0113    6F            DB   6F
3970:0114    7220          JB   0136

```

Le prime due istruzioni sono quelle che avete inserito; le altre sono di qualche programma caricato precedentemente in memoria.

Inserite ora nel registro AH 02h e nel registro DL il codice ASCII (in esadecimale) di un carattere qualsiasi (nello stesso modo in cui avete cambiato, precedentemente, i registri AX e DX), ed eseguite quindi il comando G per visualizzare il carattere. Per esempio, se avete inserito in DL 41h, vedrete:

```
-G
A
Program terminated normally
-
```

Provate a inserire altri caratteri nel registro DL prima di continuare nella lettura.

INSERIRE I PROGRAMMI

Da questo momento in avanti, la maggior parte dei programmi che scriverete saranno composti da più di un'istruzione e, per presentare questi programmi, sarà utilizzata una visualizzazione disassemblata (con il comando U). L'ultimo programma, quindi, apparirà nel modo seguente:

```
3970:0100    CD21          INT  21
3970:0102    CD20          INT  20
```

Fino a questo momento, avete inserito delle istruzioni digitando direttamente dei numeri (il codice macchina) come, per esempio, CDh 21h. Esiste però un metodo molto più semplice per inserire delle istruzioni.

Accanto al comando U, Debug offre il comando A (*Assemble*, *Assembla*) che permette di inserire delle istruzioni in un formato più facile da ricordare (l'Assembler). Quindi, invece di inserire i numeri del codice macchina, potete usare il comando A e procedere nel modo seguente:

```
-A 100
3970:0100  INT 21
3970:0102  INT 20
3970:0104
-
```

Una volta finito l'inserimento, è sufficiente premere il tasto Invio per ritornare al prompt di Debug.

In questo caso, il comando A indica a Debug di accettare le istruzioni in forma mnemonica (codice Assembly) mentre il numero 100, che segue l'istruzione A, indica a Debug di partire dalla locazione 100h. Dato che l'inserimento dei programmi in codice Assembly risulta molto più semplice, d'ora in avanti inserirete le istruzioni in questo modo.

SPOSTARE I DATI NEI REGISTRI

Fino a questo momento vi siete basati esclusivamente su Debug per eseguire dei programmi; tuttavia, non utilizzerete sempre Debug. Normalmente un programma imposta da solo i registri AH e DL prima di un'istruzione INT 21h. Per far questo, dovete imparare un'altra istruzione: MOV. Una volta imparata a fondo questa istruzione, sarete in grado di creare un vero programma (eseguibile dal DOS) che visualizzi un carattere sullo schermo.

Userete l'istruzione MOV per caricare dei numeri nei registri AH e DL. Vediamo ora come funziona l'istruzione MOV. Inserite 1234h nel registro AX (12h in AH e 34h in AL) e ABCDh in DX (ABh in DH e CDh in DL). Inserite ora, con il comando A, l'istruzione seguente:

```
396F:0100 88D4                MOV  AH,DL
```

Questa istruzione *muove* il numero contenuto in DL in AH facendone una copia in DL. DL non viene quindi modificato. Potete ora vedere che AX=CD34h e DX=ABCDh. Solo AH è cambiato e contiene ora una copia del numero memorizzato in DL.

Come l'istruzione BASIC LET AH=DL, MOV copia un numero dal secondo registro nel primo ed è questa la ragione per cui avete scritto AH prima di DL. Benché ci siano alcune restrizioni (che vedrete in seguito), potete usare altre forme dell'istruzione MOV per copiare dei numeri tra altre coppie di registri. Per esempio, reimpostate il registro IP e provate questo:

```
396F:0100 89C3                MOV  BX,AX
```

Avete appena spostato delle parole (invece che dei byte) tra due registri. L'istruzione MOV opera sempre tra parole e parole, o tra byte e byte (dato che non avrebbe senso spostare una parola in un byte).

Avevate precedentemente inserito un'istruzione MOV per copiare il contenuto del registro DL in AH. Provate ora un'altra forma di quell'istruzione:

```
396F:0100 B402                MOV  AH,02
```

Questa istruzione sposta 02h nel registro AH lasciando inalterato AL. Il secondo byte dell'istruzione, 02h, è il numero che si vuole inserire (muovere) nel registro. Provate a inserire un numero differente in AH cambiando il secondo byte dell'istruzione con il comando E 101.

Riunite ora tutto quello che avete imparato in questo capitolo e costruite un programma più lungo. Questo programma dovrà visualizzare un asterisco (*), dopo aver impostato i registri AH e DL. Dovrete quindi usare un'istruzione MOV per impostare correttamente i registri AH e DL prima di eseguire l'istruzione INT 21h:

```
396F:0100 B402      MOV  AH,02
396F:0102 B22A      MOV  DL,2A
396F:0104 CD21      INT  21
396F:0106 CD20      INT  20
```

Inserite il programma e verificatelo con il comando U (U 100). Assicuratevi che il registro IP sia impostato su 100 e impartite quindi il comando G per eseguire il programma. Dovreste vedere un asterisco sullo schermo:

```
-G
*
Program terminated normally
-
```

Ora che avete costruito un programma completo, scrivetelo su disco in un file .COM in modo da poterlo eseguire direttamente dal DOS. Potete eseguire un file .COM dal DOS digitandone semplicemente il nome. Dato che questo programma non ha ancora un nome, dovete assegnarne uno.

Il comando di Debug N (*Name*, Nome) permette di assegnare un nome a un file prima di scriverlo su disco. Digitate:

```
-N SCRIVE.COM
```

per assegnare il nome SCRIVE.COM al programma. Il comando N non scrive ancora il file su disco; assegna semplicemente un nome.

Dovete ora sapere da quanti byte è composto il programma, per poter dire a Debug quanti dati (di quelli contenuti in memoria) volete salvare in un file su disco. Se utilizzate il comando U, potete vedere che ciascuna istruzione è composta da due byte (tuttavia, non è sempre così); quindi, dato che il programma è composto da quattro istruzioni, la sua lunghezza sarà di otto byte (4*2). (Potreste anche usare il comando H per calcolare la lunghezza di un programma; in questo caso avreste dovuto digitare H 108 100 per sottrarre l'indirizzo iniziale del programma dall'indirizzo finale).

Una volta calcolata la lunghezza del programma, è necessario inserirla in qualche locazione. Debug cerca la lunghezza di un programma nella coppia di registri BX: CX; Inserite quindi 8h in CX e 0 in BX. In questo modo, Debug saprà che il programma deve essere lungo 8 byte.

Una volta impostato il nome e la lunghezza del programma, è possibile scriverlo su disco con il comando W (*Write*, Scrive):

```
-W
Writing 0008 bytes
-
```

Sul disco sarà ora presente un programma chiamato SCRIVE.COM. Uscite da Debug con il comando Q e usate il comando DIR del DOS per controllare:

```
A>DIR SCRIVE.COM
```

```
Volume in drive A has no label
Directory of A:\

SCRIVE.COM      8      30-01-90   12:02p
 1 file(s)  346432 bytes free
```

```
A>
```

Potete vedere che esiste un file chiamato SCRIVE.COM lungo 8 byte. Per eseguire il programma, digitate semplicemente *Scrive* al prompt del DOS. Vedrete un asterisco (*) apparire sullo schermo.

SCRIVERE UNA STRINGA DI CARATTERI

Come esempio finale per questo capitolo, userete l'istruzione INT 21h con un numero di funzione differente nel registro AH, in modo da scrivere un'intera stringa di caratteri. Dovrete scrivere la stringa di caratteri in memoria e indicare al DOS dove trovarla. Con questo esempio conoscerete più a fondo gli indirizzi e la memoria. Avete già visto che il numero di funzione 02h per l'istruzione INT 21h viene utilizzata per visualizzare un carattere sullo schermo. Un'altra funzione, 09h, stampa sullo schermo tutti i caratteri trovati nella locazione indicata fino a quando non viene incontrato il simbolo \$. Inserite ora la stringa in memoria. Partite dalla locazione 200h, in modo che la stringa non venga sovrascritta dalle istruzioni che inserirete. Inserite i numeri seguenti usando l'istruzione E 200:

```
43    69    61    6F
2C    20    63    6F
6D    65    20    73
74    61    69    3F
24
```

L'ultimo numero, 24, è il codice ASCII del segno \$ e serve per indicare la fine della stringa che si vuole visualizzare. Vedrete che cosa significa questa stringa tra poco, dopo aver inserito il programma seguente:


```
396F:0100 B409          MOV  AH,09
396F:0102 BA0002       MOV  DX,0200
396F:0104 CD21        INT  21
396F:0106 CD20        INT  20
```

200h è l'indirizzo della stringa inserita; quindi, assegnando 200h al registro DX, si indica al DOS dove trovare la stringa desiderata. Controllate il programma con il comando U ed eseguitelo quindi con il comando G:

```
-G
Ciao, come stai?
Program terminated normally
```

Ora che avete memorizzato alcuni caratteri in memoria, è giunto il momento di vedere un altro comando di Debug, D (per *Dump*, Stampa). Il comando Dump visualizza il contenuto della memoria sullo schermo nello stesso modo in cui U elenca le istruzioni. Similmente al comando U, fate seguire al comando D un indirizzo per indicare a Debug il punto di partenza. Per esempio, digitate il comando D 200 per visualizzare la stringa appena inserita:

```
-D 200
396F:0200  43 69 61 6F 2C 20 63 6F-6D 65 20 73 74 61 69 3F  Ciao, come stai?
369F:0210  24 5D C3 55 83 EC 30 8B-EC C7 06 10 00 00 00 E8  $].U..0.....
.
.
.
```

Dopo ciascun indirizzo (come 396F:0200 nell'esempio), ci sono 16 byte (in esadecimale) seguiti dai 16 caratteri ASCII corrispondenti. Quindi, nella prima riga vedete tutti i codici ASCII e i caratteri digitati precedentemente ad eccezione del segno \$ che è il primo carattere della seconda riga; il resto della riga comprende vari caratteri.

Un punto (.) nella finestra ASCII, rappresenta un punto o un carattere speciale come, per esempio, la lettera greca pi. Con il comando D vengono visualizzati solamente 96 dei 256 caratteri disponibili sull'IBM; quindi il punto viene usato per rappresentare i rimanenti 160.

Userete successivamente il comando D di Debug per controllare i numeri inseriti come dati; per ulteriori informazioni, fate riferimento alla sezione Debug del manuale del DOS.

Il programma per scrivere una stringa di caratteri è ora completo; potete quindi scriverlo su disco. La procedura da seguire è la stessa usata per il programma SCRIVE.COM; questa volta, però, dovete calcolare la lunghezza del programma in modo da includere la stringa contenuta a partire dalla locazione 200h. Il programma inizia alla locazione 100h e, come potete vedere dal comando D, termina alla

locazione 211h (con il primo carattere che segue il segno \$). Usate il comando H per trovare la differenza tra questi due indirizzi; calcolate 211h-100h e inserite il risultato nel registro CX, impostando nuovamente BX a zero. Usate il comando N per assegnare un nome al programma (aggiungete l'estensione .COM per poterlo eseguire dal DOS) e usate infine il comando W per scriverlo su disco.

A questo punto avete appreso tutti i comandi necessari per visualizzare una stringa di caratteri sullo schermo; vogliamo però aggiungere una nota finale. Avrete notato che il DOS non visualizza mai il segno \$ perché questo viene utilizzato per segnare la fine di una stringa. Questo significa che non è possibile utilizzare una stringa che contenga il segno del dollaro; tuttavia imparerete nei capitoli successivi un metodo per visualizzare il segno \$ e qualsiasi carattere speciale.

SOMMARIO

I primi due capitoli hanno fornito le basi per poter lavorare con un programma vero. In questo capitolo avete utilizzato i numeri esadecimali, il Debug, le istruzioni dell'8088 e la memoria per costruire dei piccoli programmi indipendenti e siete venuti a contatto con nuovi comandi.

Avete innanzitutto imparato le istruzioni INT (non dettagliatamente, ma abbastanza per scrivere due programmi). Nei capitoli successivi conoscerete più a fondo le istruzioni di interrupt aumentando, nello stesso tempo, la conoscenza del microprocessore 8088.

Debug è stato ancora una volta una guida utile e fedele. Avete usato moltissimo Debug per visualizzare il contenuto dei registri e della memoria, e avete imparato il comando G per eseguire i programmi costruiti.

Avete utilizzato l'istruzione di uscita INT 20 e l'istruzione MOV (per spostare dei numeri tra i registri). L'istruzione di uscita (INT 20) ha permesso di costruire un programma completamente indipendente eseguibile direttamente dal DOS, mentre MOV ha permesso di impostare dei registri prima di eseguire l'istruzione INT 21 (che permette di visualizzare un carattere).

Infine avete imparato a visualizzare una stringa di caratteri sullo schermo. Userete moltissimo tutte queste istruzioni durante la lettura del libro e, grazie ai comandi U e A non sarete costretti a ricordarvi il codice macchina di questi comandi.

Nel prossimo capitolo costruirete un piccolo programma che, dopo aver prelevato un byte, lo mostrerà sullo schermo come una stringa composta da cifre binarie (zero e uno).

VISUALIZZAZIONE DEI NUMERI BINARI

In questo capitolo scriverete un programma per visualizzare dei numeri binari sullo schermo, come stringhe di zeri e uno. Conoscete già la maggior parte delle istruzioni necessarie, e il lavoro che svolgerete vi aiuterà a consolidare le nozioni apprese. Imparerete qualche nuovo comando come, per esempio, un'altra forma di ADD o istruzioni che vi consentiranno di ripetere certe operazioni. Iniziate ora con qualcosa di completamente nuovo.

ROTAZIONI E FLAG DI RIPORTO

Nel capitolo 2, quando avete incontrato per la prima volta l'aritmetica esadecimale, avete scoperto che, sommando 1 a FFFFh, potreste ottenere 10000h. Ma questo non succede; infatti solo le ultime quattro cifre esadecimali possono essere contenute in una parola (l'uno più a sinistra non ci sta). Avete anche visto che questo 1 è il riporto e che non viene perso. Il riporto viene memorizzato in una locazione speciale, chiamata *flag* (in questo caso *Flag di Riporto*). I flag contengono numeri a un bit e possono quindi contenere zero o uno. Pertanto, quando è necessario riportare un uno, questo viene inserito nel flag di riporto.

Tornando all'istruzione ADD del capitolo 2 (ADD AX,BX), inserite FFFFh in AX e 1 in BX; eseguite quindi, con il comando T, l'istruzione ADD. Utilizzando il comando R, potete vedere nella seconda riga otto coppie di lettere, l'ultima delle quali (che può essere NC o CY) indica lo stato del flag di riporto. In questo caso, dato che si otterrà un riporto di 1, verrà visualizzato CY (*Carry*, Riporto) che indica che il valore del flag di riporto è 1.

Utilizzate ora l'istruzione ADD per sommare 1 allo zero contenuto nel registro AX; impostate nuovamente IP su 100h e utilizzate il comando T. Il flag di riporto viene influenzato da ciascuna istruzione ADD e in questo caso, dato che non c'è alcun riporto, viene impostato a zero. Se utilizzate il comando R, vedrete che ora appare NC (*No Carry*, Nessun Riporto).

(Imparerete il significato degli altri flag successivamente; tuttavia, se siete curiosi, potete trovare delle informazioni nel manuale del DOS nella sezione che tratta il comando R di Debug).

Vediamo come possono essere utili le informazioni sul riporto, nel caso di una visualizzazione di numeri binari. Supponete di voler visualizzare solo un carattere alla volta, prendendo i bit, uno per uno, da sinistra a destra.

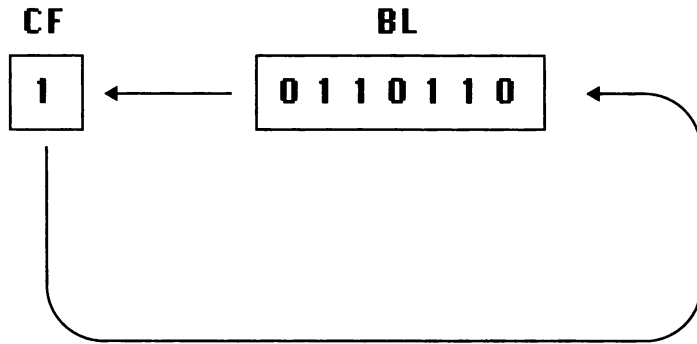


Figura 4-1. L'istruzione *RCL BL,1* sposta i bit a sinistra di una posizione utilizzando il flag di riporto

Considerando per esempio il numero 1000 0000b, il primo numero da mostrare sarebbe 1. Se fosse possibile spostare l'intero byte a sinistra di una posizione, inserendo il bit più a sinistra (in questo caso 1) nel flag di riporto e aggiungendo uno zero a destra, e ripetendo quindi il processo per tutte le cifre del numero, il flag di riporto conterrebbe, di volta in volta, tutte le cifre desiderate. Questa operazione è possibile con una nuova istruzione chiamata *RCL (Rotate Carry Left, Rotazione del Flag a sinistra)*.

Per vedere come funziona, inserite l'istruzione seguente:

```
3985:0100 D0D3          RCL  BL,1
```

Questa istruzione fa *ruotare* il byte in BL di una posizione a sinistra utilizzando il flag di riporto. In pratica l'istruzione RCL sposta il bit più a sinistra nel flag di riporto mentre il bit contenuto in quel momento nel flag stesso viene inserito all'estrema destra del byte. Durante questo processo, tutti gli altri bit vengono spostati (ruotati) di una posizione a sinistra. Dopo un certo numero di rotazioni (17 per una parola, 9 per un byte) i bit ritornano nella posizione originale.

Inserite B7h nel registro BX ed eseguite l'istruzione T alcune volte. Convertendo i risultati in binario, vedrete quanto segue:

<u>Riporto</u>	<u>Registro BL</u>	
0	1011 0111	B7h Inizio
1	0110 1110	6Eh
0	1101 1101	DDh
1	1011 1010	BAh
	.	
	.	
	.	
0	1011 0111	B7h Dopo 9 rotazioni

Dopo la prima rotazione, il bit 7 di BL viene spostato nel flag di riporto, il bit nel flag di riporto viene spostato nel bit 0 di BL, e tutti gli altri bit vengono spostati a sinistra di una posizione. Le rotazioni successive continuano a spostare i bit di una posizione a sinistra e quindi, dopo nove rotazione, viene ripristinato il numero originale in BL. Siete quasi pronti per costruire il programma per scrivere dei numeri binari sullo schermo, ma vediamo prima come convertire il bit del flag di riporto nel carattere 0 o 1.

SOMMARE CON IL FLAG DI RIPORTO

Una normale istruzione ADD, per esempio ADD AX,BX, somma semplicemente due numeri. L'istruzione ADC (*Add with Carry*, Somma con il riporto), invece, somma tre numeri: i due numeri specificati più il bit del flag di riporto. Se guardate la tabella ASCII, scoprirete che 30h è il codice corrispondente allo 0 mentre 31h è il corrispondente di 1. Quindi, sommando il flag di riporto a 30h, si ottiene 0 quando il riporto è nullo, 1 quando è presente un riporto. Pertanto, se DL=0 e il flag di riporto è 1, eseguendo:

```
ADC    DL, 30
```

si ottiene 31h (1); infatti DL(0) più 30h(0) più 1h (il riporto) è uguale a 31h. Con questa istruzione potete convertire il bit del riporto in un carattere stampabile.

A questo punto, invece di fare un esempio con ADC, vediamo di completare il programma. Una volta terminato il programma, eseguirete le istruzioni una alla volta (procedura chiamata *passo a passo*) vedendo così come funziona l'istruzione ADC. Ma prima di poter completare il programma, avete bisogno di un'istruzione per ripetere otto volte (una per ciascun bit) i comandi RCL, ADC e INT 21h.

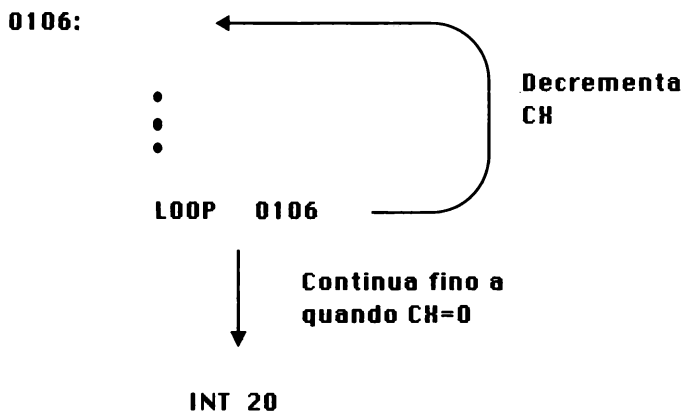


Figura 4-2. L'istruzione LOOP

CICLI

Come già detto, con l'istruzione RCL è possibile non solo ruotare dei byte, ma anche delle parole. Utilizzerete questa caratteristica per vedere come funziona l'istruzione *LOOP*. *LOOP* è simile al comando FOR-NEXT del BASIC; quindi, come per questo comando, anche con *LOOP* bisogna specificare quante volte deve essere eseguita una serie di istruzioni. Per far questo, dovete inserire un contatore nel registro CX. Ogni volta che viene terminato un ciclo, l'8088 sottrae uno da CX e, quando CX diventa zero, *LOOP* esce dal ciclo.

Perché il registro CX? La C di CX significa *Count* (Conta). Questo registro può essere usato come registro di uso generale ma, come vedrete in seguito, CX viene usato con altre istruzioni che effettuano operazioni ripetitive.

Di seguito un semplice programma che ruota a sinistra il registro BX per otto volte, spostando BL in BH (ma non al contrario, dato che la rotazione viene fatta tramite il flag di riporto):

```

396F:0100    BBC5A3          MOV    BX, A3C5
396F:0103    B90800          MOV    CX, 0008
396F:0106    D1D3           RCL    BX, 1
396F:0108    E2FC           LOOP   0106
396F:010A    CD20           INT    20

```

Il ciclo inizia in 106h (RCL BX,1) e finisce con l'istruzione *LOOP*. Il numero che segue *LOOP* (106h) è l'indirizzo dell'istruzione RCL. Quando eseguite il programma, *LOOP* sottrae uno da CX e, se CX non è uguale a zero, salta a 106h. L'istruzione RCL BX,1 viene eseguita otto volte, dato che CX è stato impostato a 8 prima del ciclo.

Avrete notato che, a differenza del ciclo FOR-NEXT del BASIC, il comando *LOOP* si trova alla fine del ciclo (dove in genere viene posizionato NEXT nel BASIC), mentre l'inizio del ciclo (RCL) non ha istruzioni speciali (come FOR nel BASIC). Se conoscete il linguaggio PASCAL, potete vedere che l'istruzione *LOOP* è in qualche modo simile alla coppia di comandi REPEAT-UNTIL, dove REPEAT indica l'inizio del blocco di istruzioni da eseguire.

Ci sono diversi metodi per eseguire un piccolo programma. Se digitate semplicemente G non vedrete alcun cambiamento nei registri, in quanto Debug salva tutti i registri prima di eseguire il comando G. Se poi viene incontrata un'istruzione INT 20 (come in questo caso), tutti i registri vengono ripristinati. Provate il comando G. Vedrete che IP è stato riportato su 100h (come prima dell'esecuzione) e che tutti gli altri registri non hanno subito modifiche.

Se avete pazienza, potete usare il comando T. Eseguendo il programma passo a passo potete vedere come i registri cambiano:

-R

```
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0100 NV UP DI PL NZ NA PO NC
0CDE:0100 BBC5A3          MOV      BX,A3C5
```

-T

```
AX=0000 BX=A3C5 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0103 NV UP DI PL NZ NA PO NC
0CDE:0103 B90800          MOV      CX,0008
```

-T

```
AX=0000 BX=A3C5 CX=0008 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0106 NV UP DI PL NZ NA PO NC
0CDE:0106 D1D3           RCL      BX,1
```

-T

```
AX=0000 BX=478A CX=0008 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0108 OV UP DI PL NZ NA PO CY
0CDE:0108 E2FC           LOOP     0106
```

-T

```
AX=0000 BX=478A CX=0007 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0106 OV UP DI PL NZ NA PO CY
0CDE:0106 D1D3           RCL      BX,1
```

-T

-T

```
AX=0000 BX=C551 CX=0001 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0108 NV UP DI PL NZ NA PO CY
0CDE:0108 E2FC           LOOP     0106
```

-T

```
AX=0000 BX=C551 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=010A NV UP DI PL NZ NA PO CY
0CDE:010A CD20           INT      20
```

Un'altra possibilità è quella di impartire il comando G 10A per eseguire il programma escludendo l'istruzione INT 20 in 10Ah. In questo caso, i registri mostreranno il risultato del programma. Quindi, CX sarà uguale a zero e BX uguale a C551 o C5D1 a seconda dello stato del flag di riporto prima dell'esecuzione del programma. Il valore C5 che il programma (con l'istruzione MOV) ha portato in BL, si trova ora in BH; per contro, BL non contiene A3 poiché la rotazione di BX è stata fatta *attraverso* il riporto. Imparerete successivamente a effettuare delle rotazioni senza utilizzare il riporto. Vediamo ora come visualizzare un numero in notazione binaria.

SCRIVERE UN NUMERO BINARIO

Avete visto come isolare le cifre binarie (una alla volta) e come convertirle in caratteri ASCII. E' ora sufficiente aggiungere un'istruzione INT 21h per completare il programma. Ecco il programma; la prima istruzione inserisce 02 in AH per la funzione INT 21h (se vi ricordate, 02 indica al DOS di visualizzare il carattere contenuto nel registro DL):

```

3985:0100      B402          MOV  AH,02
3985:0102      B90800       MOV  CX,0008
3985:0105      B200         MOV  DL,00
3985:0107      D0D3        RCL  BL,1
3985:0109      80D230      ADC  DL,30
3985:010C      CD21        INT  21
3985:010E      E2F5        LOOP 0105
3985:0110      CD20        INT  20

```

Avete visto come funzionano le parti del programma; ora riunitele. Ruotate BL (con l'istruzione RCL BL,1) per rilevare i bit di un numero, scegliete un numero da visualizzare in binario, caricatelo nel registro BL ed eseguite il programma con il comando G. Dopo l'istruzione INT 20h, il comando G riporta i registri ai loro valori originali quindi, BL, contiene ancora il numero che vedete visualizzato in binario.

L'istruzione ADC DL,30 converte il flag di riporto nel carattere zero o uno. L'istruzione MOV DL,0 imposta DL a zero, quindi l'istruzione ADC somma 30h a DL e aggiunge infine il riporto. Dato che 30h è il codice ASCII di zero, il risultato di ADC DL,30 è il codice dello zero se il riporto è nullo (NC) o dell'uno se il riporto è 1 (CY).

Se volete vedere cosa succede quando eseguite questo programma, usate il comando T. Ricordatevi, però, di interrompere l'esecuzione passo a passo quando raggiungete l'istruzione INT 21h. Questa istruzione, infatti, richiama un programma all'interno del DOS e sarebbero necessari moltissimi passi prima di ritornare al programma principale. Per tracciare il programma "saltando" l'istruzione INT 21h, potete digitare G 10E ogni volta che la raggiungete. Il comando G, seguito da un indirizzo, indicherà a Debug di continuare l'esecuzione del programma fermandosi nel momento in cui IP contiene l'indirizzo 10E. In questo modo, Debug eseguirà l'istruzione INT 21h (non passo a passo) e si fermerà prima di eseguire l'istruzione LOOP in 10E; potrete quindi riprendere a tracciare il programma con il comando T. (Il numero digitato dopo G viene chiamato *breakpoint* - punto di interruzione - nel manuale del DOS; i breakpoint sono molto utili durante il collaudo dei programmi).

Per terminare il programma, digitate semplicemente G quando raggiungete l'istruzione 20h.

IL COMANDO PROCEED

Indipendentemente dal fatto che abbiate provato o meno il comando T per tracciare un programma, avete visto che un comando come G 10E permette di *aggirare* un'istruzione INT che inizi, per esempio, in 10Ch. Ma questo significa che ogni volta che si vuole aggirare un'istruzione INT, è necessario sapere l'indirizzo dell'istruzione che segue INT.

Per fortuna esiste un comando di Debug che permette di "saltare" le istruzioni INT in un modo molto semplice. Il comando P (*Proceed*, Continua) effettua questa operazione. Per vedere come funziona tracciate il programma, ma quando raggiungete l'istruzione INT 21h, digitate P invece di G 10E.

Farete un uso intensivo del comando P nel resto di questo libro, dato che è un ottimo metodo per aggirare istruzioni tipo INT che richiamano grossi programmi (come routine del DOS). Prima di proseguire, però, dobbiamo dire che il comando P non è documentato nei manuali del DOS prima della versione 3.00. Questa mancanza di documentazione può essere stata una dimenticanza o, più probabilmente, è stata dovuta al fatto che la Microsoft non ha avuto abbastanza tempo per testare completamente il comando P prima di rilasciare la versione 2.00 del DOS. Comunque, qualunque sia la ragione, se avete una versione DOS precedente alla 3.00, dovete tener presente che il comando P *potrebbe non* funzionare sempre (anche se noi non abbiamo mai riscontrato alcun problema).

Quanto esposto finora è tutto il necessario per visualizzare dei numeri binari come stringhe di zeri e uno. Per concludere vi proponiamo un piccolo esercizio: provate a modificare il programma appena costruito in modo che venga visualizzata la lettera *b* alla fine del numero binario (**Suggerimento**: il codice ASCII di *b* è 62h).

SOMMARIO

In questo capitolo avete avuto la possibilità di consolidare i concetti appresi nei primi tre capitoli, imparando qualche nuova istruzione.

Avete incontrato il flag di riporto, che è risultato molto importante nella costruzione del programma per visualizzare dei numeri binari. Avete imparato l'istruzione di rotazione RCL, utile per ruotare verso sinistra, un bit alla volta, un byte o una parola. Dopo queste nozioni avete conosciuto una nuova forma dell'istruzione di addizione, ADC, e avete iniziato la costruzione di un programma.

A questo punto è entrata in scena l'istruzione LOOP. Caricando nel registro CX un contatore (otto), avete fatto eseguire all'8088 una serie di istruzioni otto volte.

Userete tutte queste istruzioni ancora nei capitoli successivi. Nel prossimo capitolo imparerete a visualizzare un numero binario in notazione esadecimale (proprio come fa Debug); avrete quindi un'idea più chiara di come Debug converte i numeri. Effettuerete infine l'operazione inversa: convertire i numeri esadecimale in notazione binaria.

VISUALIZZAZIONE DEI NUMERI ESADECIMALI

Il programma sviluppato nel capitolo 4 era abbastanza facile. Siete stati fortunati dal momento che il flag di riporto permette di visualizzare facilmente un numero binario come stringa di zeri e uno. Vediamo ora come possono essere visualizzati dei numeri in notazione esadecimale. In questo caso, il programma sarà un pochino più complicato e dovrete scrivere la stessa serie di istruzioni più di una volta. Nel Capitolo 7, comunque, imparerete a utilizzare le procedure (subroutine) in modo da eliminare la scrittura ripetuta di una serie di comandi. Vediamo ora una serie di istruzioni nuove e come visualizzare dei numeri in esadecimale.

BIT DI CONFRONTO E BIT DI STATO

Nell'ultimo capitolo avete imparato qualcosa sui flag di stato e avete esaminato il flag di riporto (che viene rappresentato da CY o NC nella visualizzazione ottenibile con il comando R). Gli altri flag, che sono ugualmente utili, tengono traccia dello *stato* dell'ultima operazione aritmetica eseguita. Ci sono otto flag; imparerete a utilizzare tutti questi flag nel momento in cui saranno necessari.

Ricordate che CY significa che il flag di riporto è uguale a 1, mentre NC significa che è uguale a zero. In tutti i flag, 1 significa *vero* e 0 significa *falso*. Per esempio, se un'istruzione SUB dà come risultato zero, il flag conosciuto come Flag Zero sarà impostato a 1 (vero) e verrà visualizzato (tramite il comando R) come ZR (*Zero*). In caso contrario, il flag zero sarà impostato a zero e visualizzato come NZ (*Not Zero, Non Zero*).

Diamo un'occhiata a un esempio che controlla il flag zero. L'istruzione SUB viene utilizzata per sottrarre due numeri. Se questa sottrazione dà come risultato zero, il flag zero apparirà come ZR: Inserite l'istruzione di sottrazione:

```
396F:0100      29D8 SUB  AX,BX
```

Ora tracciate questa istruzione con diversi numeri guardando lo stato del flag zero (ZR o NZ). Se inserite lo stesso numero (F5h nell'esempio seguente) nei registri AX e BX, vedrete che il flag zero viene impostato dopo un'istruzione di sottrazione e azzerato dopo un'altra:

-R

```
AX=00F5 BX=00F5 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0100 NV UP DI PL NZ NA PO NC
0CDE:0100 29D8          SUB      AX,BX
```

-T

```
AX=00F5 BX=00F5 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0102 NV UP DI PL ZR NA PE NC
0CDE:0102 3F          AAS
```

-R IP

```
IP 0102
:100
```

-R

```
AX=0000 BX=00F5 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0100 NV UP DI PL ZR NA PE NC
0CDE:0100 29D8          SUB      AX,BX
```

-T

```
AX=FF0B BX=00F5 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0102 NV UP DI NG NZ AC PO CY
0CDE:0102 3F          AAS
```

Se si sottrae uno da zero il risultato è FFFFh che, come visto nel capitolo 1, è il corrispondente di -1 nella forma complemento a due. E' possibile vedere con il comando R se un numero è positivo o negativo? Sì, grazie a un altro flag, chiamato il flag del segno, che cambia da NG (*Negative*, Negativo) a PL (*Plus*, Positivo) e che viene impostato a 1 quando un numero è negativo.

Un altro flag da considerare è il flag di Overflow che cambia da OV (*Overflow*), quando il flag è 1, a NV (*No Overflow*) quando il flag è zero. Questo flag viene impostato se il bit del segno cambia quando non dovrebbe. Per esempio, se si sommassero due numeri positivi come 7000h e 6000h, si otterrebbe un numero negativo (D000h o -12288). Questo è un errore dato che il risultato non può essere contenuto in una parola. Il risultato dovrebbe essere positivo ma, dato che non lo è, l'8088 imposta a 1 il flag di overflow. (Ricordatevi che se utilizzate dei numeri senza segno, dovete ignorare il flag di overflow).

Provate diversi numeri per vedere come questi flag vengono impostati, in modo da prendere confidenza. Per quanto riguarda il flag di overflow, sottraete un grosso numero negativo da un grosso numero positivo (per esempio, 7000h - 8000h dato che 8000h è il corrispondente di -32768 nella forma complemento a due).

Siete ora pronti per imparare una serie di istruzioni chiamate istruzioni di *salto condizionale*. Queste permettono di controllare i flag di stato in modo più comodo di quanto fatto finora. L'istruzione JZ (*Jump if Zero*, Salta se Zero) salta all'indirizzo specificato se il risultato dell'ultima operazione aritmetica effettuata è stato zero. Quindi, se all'istruzione SUB fate seguire il comando JZ (per esempio, JZ 15A), un risultato pari a zero farà saltare l'esecuzione del programma all'indirizzo 15A invece che all'istruzione successiva.

JZ controlla il flag zero e, se questo è impostato (ZR), effettua un salto all'indirizzo specificato. (Questa istruzione è l'equivalente del comando BASIC: IF A = 0 THEN 100). L'opposto di JZ è JNZ (*Jump if Not Zero*, Salta se non è zero). Guardate un piccolo esempio che utilizza JNZ e sottrae uno da un numero fino a quando il risultato diventa uguale a zero:

```
396F:0100      2C01          SUB  A1,01
396F:0102      75FC          JNZ  0100
396F:0104      CD20          INT  20
```

Inserite il valore 3 in AL in modo che il ciclo venga eseguito tre volte, usate quindi il comando T per eseguire il programma passo a passo e guardate come funziona il salto condizionale. L'istruzione INT 20h posta alla fine del programma evita di superare la fine del programma in caso di un'esecuzione accidentale del comando G.

Avrete notato che l'uso dell'istruzione SUB per confrontare due numeri ha lo spiacevole effetto collaterale di cambiare il primo numero. Fortunatamente esiste un'altra istruzione, CMP (*Compare*, Confronta), che permette di effettuare la sottrazione di due numeri senza modificare il primo numero. Il risultato viene usato solamente per cambiare i flag; in questo modo è possibile usare uno dei molti comandi di salto condizionale dopo un'istruzione di confronto. Per vedere come funziona CMP, inserite nei registri AX e BX lo stesso numero (F5h) e tracciate il programma:

```
-A 100
0CDE:0100  CMP AX, BX
0CDE:0102
-T

AX=00F5 BX=00F5 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0102 NV UP DI PL ZR NA PE NC
0CDE:0102 3F          AAS
```

Il flag zero è ora ZR ma F5h resta in entrambi i registri.

Userete ora CMP per visualizzare una singola cifra esadecimale. Vedrete come utilizzare una serie di istruzioni per cambiare lo stato dei flag in modo da cambiare il flusso del programma, come con il comando BASIC IF-THEN. Questa nuova serie di istruzioni si servirà dei flag per controllare determinate condizioni (come minore di, maggiore di e così via). Non dovrete preoccuparvi di sapere quale flag viene modificato durante le varie operazioni; le istruzioni di salto condizionale controllano automaticamente il flag interessato.

VISUALIZZARE UNA SINGOLA CIFRA ESADECIMALE

Iniziate inserendo un piccolo numero (compreso tra 0 e Fh) nel registro BL. Dato che qualsiasi numero compreso tra 0 e Fh equivale a una singola cifra esadecimale, potete convertire la vostra scelta in un carattere ASCII e quindi visualizzarlo. Vediamo come effettuare questa conversione.

Carattere	Codice ASCII (Esa)
/	2F
0	30
1	31
2	32
3	33
4	34
5	35
6	36
7	37
8	38
9	39
:	3A
;	3B
<	3C
=	3D
>	3E
?	3F
@	40
A	41
B	42
C	43
D	44
E	45
F	46
G	47

Figura 5-1. La Parte del codice ASCII che contiene i caratteri usati per le cifre esadecimali

I caratteri ASCII compresi tra 0 e 9 hanno il codice compreso tra 30h e 39h.; i caratteri da A a F, invece, hanno i codici compresi tra 41h e 46h. Qui sta il problema: questi due gruppi di caratteri ASCII sono separati da sette caratteri. Quindi, la conversione in ASCII sarà differente per i due gruppi (da 0 a 9, e da A a F); per questo motivo dovrete manipolare ciascun gruppo separatamente. Un programma in BASIC effettuerebbe questa operazione nel modo seguente:

```
100 IF BL < &H0A
      THEN BL = BL + &H30
      ELSE BL = BL + &H37
```

In BASIC una conversione del genere sarebbe decisamente semplice. Sfortunatamente, il linguaggio macchina dell'8088 non include un comando ELSE. Vediamo quindi come si dovrebbe strutturare un programma in BASIC se non esistesse l'istruzione ELSE:

```
100 BL = BL + &H30
110 IF BL < &H3A
      THEN BL = BL + &H7
```

Per verificare il funzionamento di questo programma, fate qualche prova usando in special modo i numeri 0, 9, Ah e Fh che sono gli estremi dei due gruppi.

Infatti, 0 e Fh sono rispettivamente il numero più piccolo e quello più grosso composto da una sola cifra, mentre 9 e 0Ah (pur essendo adiacenti) richiedono un differente sistema di conversione. Usando questi numeri è quindi possibile verificare il limite inferiore e quello superiore di tutta la gamma, e assicurarsi di aver stabilito correttamente il punto in cui cambiare il tipo di conversione (con 9 e 0Ah).

(Notate che abbiamo scritto 0Ah in quest'ultimo paragrafo invece di Ah; questo è stato fatto per evitare confusione con il registro AH. D'ora in avanti capiterà spesso di aggiungere uno zero davanti a una cifra esadecimale in modo da evitare possibili malintesi. Ricordate che uno zero davanti a un numero esadecimale non altera il numero stesso).

La versione in linguaggio macchina di questo programma richiede qualche passo ulteriore, ma è essenzialmente uguale alla versione BASIC. Si deve utilizzare un'istruzione CMP e l'istruzione di salto condizionale JL (*Jump if Less Than*, Salta se Minore Di). Ecco come deve essere strutturato il programma per prelevare una singola cifra dal registro BL e visualizzarla in esadecimale:

```
3985:0100      B402          MOV  AH,02
3985:0102      88DA          MOV  DL,BL
3985:0104      80C230       ADD  DL,30
3985:0107      80FA3A       CMP  DL,3A
3985:010A      7C03          JL   010F
3985:010C      80C207       ADD  DL,07
3985:010F      CD21          INT  21
3985:0111      CD20          INT  20
```

L'istruzione `CMP`, come visto prima, sottrae due numeri (`DL-3Ah`) per impostare i flag, ma non cambia il numero in `DL`. Quindi, se `DL` è minore di `3Ah`, l'istruzione `JL` fa saltare l'esecuzione del programma in `10F`, dove viene visualizzata la cifra esadecimale. Inserite una singola cifra esadecimale in `BL` e tracciate il programma, in modo da prendere confidenza con `CMP` e con l'algoritmo necessario per convertire un esadecimale in ASCII. Ricordatevi di usare il comando `G` con un punto di interruzione o il comando `P`, quando raggiungete un'istruzione `INT`.

UN'ALTRA ISTRUZIONE DI ROTAZIONE

Questo programma funziona con qualsiasi numero composto da una cifra; se però volete visualizzare un numero esadecimale a due cifre, dovete aggiungere alcune istruzioni. Innanzitutto dovete isolare ciascuna cifra (quattro bit, spesso chiamato un *nibble*) del numero esadecimale. In questo paragrafo vedrete come è facile isolare i primi quattro bit (i bit alti), mentre nel paragrafo successivo imparerete le *operazioni logiche*, che userete per isolare i quattro bit bassi.

Se vi ricordate, nell'ultimo capitolo avete usato l'istruzione `RCL BL,1` per ruotare il contenuto del registro `BL` di un bit a sinistra (attraverso il flag di riporto). E' possibile ruotare più di un bit alla volta, ma non si può scrivere `RCL BL,2`. (**Nota:** `RCL BL,2` non è ammessa dal'8088, ma può essere tranquillamente usata con i microprocessori 80286 e 80386. Dato che stiamo vedendo le istruzioni dell'8088, vediamo come si possono ruotare più bit su questo microprocessore). Per ruotare più di un bit alla volta dovete inserire un contatore nel registro `CL`.

Il registro `CL` viene usato nello stesso modo in cui viene utilizzato il registro `CX` in un'istruzione `LOOP` (per determinare il numero di ripetizioni del ciclo). L'8088 usa il registro `CL` (invece di `CX`) come contatore, perché non avrebbe senso ruotare per più di 16 volte un byte; quindi, il registro a 8 bit `CL` è abbastanza grosso per contenere il contatore.

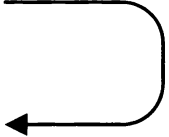
0107	CMP	DL,3A	
010A	JL	010F	
010C	ADD	DL,07	
010F	INT	21	

Figura 5-2. L'istruzione `JL`

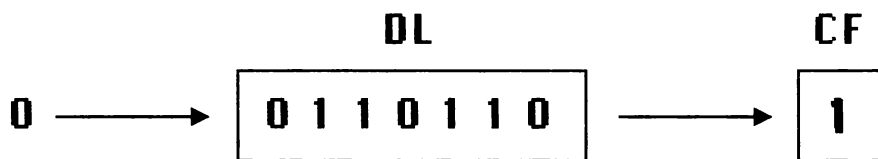


Figura 5-3. L'istruzione SHR DL,1 sposta i bit a destra di una posizione nel flag di riporto

Vediamo ora come ruotare il byte contenuto nel registro DL di 4 bit verso destra. Per far questo, dovete usare un'istruzione di rotazione leggermente differente chiamata SHR (*Shift Right*, Ruota a Destra). Usando SHR potrete spostare i quattro bit alti del numero nel nibble (quattro bit) più a destra.

Impostate inoltre a zero i quattro bit alti di DL, in modo che l'intero registro diventi uguale al nibble che state spostando nel nibble a destra. Se inserite SHR DL,1, l'istruzione ruota il byte contenuto nel registro DL di un bit verso destra e, nello stesso momento, porta il bit 0 nel flag di riporto e uno zero nel bit 7 (il più alto o il più a sinistra). Effettuando tre volte questa operazione, otterrete il risultato desiderato: i quattro bit alti prendono il posto dei quattro bit bassi, e al posto dei quattro bit alti ci sono degli zeri. E' possibile effettuare tutti questi spostamenti con una sola istruzione, usando il registro CL come *contatore*. Impostando CL a quattro prima dell'istruzione SHR DL,CL sarete sicuri che DL diventerà uguale alla cifra esadecimale superiore. Inserite 4 nel registro CL e 5Dh in DL; inserite quindi l'istruzione SHR e tracciatela:

```
3985:0100    D2EA                SHR    DL,CL
```

DL dovrebbe ora contenere 05h, che è appunto la prima cifra del numero 5Dh. E' ora possibile visualizzare questa cifra con un programma simile a quello usato precedentemente. Quindi, unendo le due parti appena sviluppati, potete costruire il seguente programma per prelevare un numero dal registro BL e visualizzare la prima cifra esadecimale:

```
3985:0100    B402                MOV    AH,02
3985:0102    88DA                MOV    DL,BL
3985:0104    B104                MOV    CL,04
3985:0106    D2EA                SHR    DL,CL
3985:0108    80C230              ADD    DL,30
3985:010B    80FA3A              CMP    DL,3A
3985:010E    7C03                JL     0113
3985:0110    80C207              ADD    DL,07
3985:0113    CD21                INT    21
3985:0115    CD20                INT    20
```

AND LOGICO

Ora che siete in grado di visualizzare la prima delle due cifre di un numero esadecimale, vediamo come è possibile isolare e visualizzare la seconda cifra. Imparerete ora ad azzerare i primi quattro bit del numero originale, lasciando quindi DL uguale al valore corrispondente dei quattro bit bassi. Per impostare a zero i quattro bit alti, potete usare l'istruzione AND. AND fa parte delle istruzioni *logiche*. In una formula logica si potrebbe dire: "A è vero se B e C sono veri; ma se B o C sono falsi, anche A deve essere falso". Se in questo enunciato si sostituisce uno con vero e zero con falso, guardando alle varie combinazioni di A, B e C si ottiene la seguente tabella *di verità* (ricavata effettuando un AND logico su due bit):

AND	F	V
F	F	F
V	F	V

=

AND	0	1
0	0	0
1	0	1

In alto e a sinistra ci sono i valori per i due bit. Nella tabella potete vedere i risultati delle varie combinazioni di AND (per esempio, AND tra 0 e 1 dà 0).

L'istruzione AND agisce sui byte e sulle parole facendo un AND tra i bit di ciascun byte o parola che si trovano nella stessa posizione. Per esempio, l'enunciato AND BL,CL effettua un AND tra il bit 0 di BL e il bit 0 di CL, quindi tra il bit 1, il bit 2 e così via, ponendo infine il risultato in BL. Per chiarire questo concetto, ecco un esempio con un numero binario:

	1	0	1	1	0	1	0	1
AND	0	1	1	1	0	1	1	0
	0	0	1	1	0	1	0	0

Inoltre, effettuando un AND logico tra 0Fh e un qualsiasi altro numero, i quattro bit alti vengono sempre impostati a zero:

	0	1	1	1	1	0	1	1
AND	0	0	0	0	1	1	1	1
	0	0	0	0	1	0	1	1

Quindi, se nel programma che preleva un numero dal registro BL effettuate un AND tra 0Fh e i quattro bit alti, potete isolare e visualizzare la cifra esadecimale inferiore. Inserite quindi nel programma precedente questa istruzione AND:

3985:0100	B402	MOV	AH, 02
3985:0102	88DA	MOV	DL, BL
3985:0104	80E20F	AND	DL, 0F
3985:0107	80C230	ADD	DL, 30
3985:010A	80FA3A	CMP	DL, 3A
3985:010D	7C03	JL	0113
3985:010F	80C207	ADD	DL, 07
3985:0112	CD21	INT	21
3985:0114	CD20	INT	20

Provate questa parte di programma inserendo qualche numero esadecimale a due cifre nel registro BL, prima di riunire le varie parti e visualizzare entrambe le cifre. Dovreste vedere la cifra esadecimale più a destra (contenuta in BL) visualizzata sullo schermo.

RIUNIRE LE PARTI

Non ci sono molte modifiche da effettuare per riunire i due programmi appena sviluppati. L'unico cambiamento necessario è l'indirizzo della seconda istruzione JL che viene usata per visualizzare la seconda cifra esadecimale. Ecco il programma completo:

3985:0100	B402	MOV	AH, 02
3985:0102	88DA	MOV	DL, BL
3985:0104	B104	MOV	CL, 04
3985:0106	D2EA	SHR	DL, CL
3985:0108	80C230	ADD	DL, 30
3985:010B	80FA3A	CMP	DL, 3A
3985:010E	7C03	JL	0113
3985:0110	80C207	ADD	DL, 07
3985:0113	CD21	INT	21
3985:0115	88DA	MOV	DL, BL
3985:0117	80E20F	AND	DL, 0F
3985:011A	80C230	ADD	DL, 30
3985:011D	80FA3A	CMP	DL, 3A
3985:0120	7C03	JL	0125
3985:0122	80C207	ADD	DL, 07
3985:0125	CD21	INT	21
3985:0127	CD20	INT	20

Una volta inserito questo programma, dovete digitare *U 100*, seguito da *U*, per vedere il listato disassemblato. Notate che è stata ripetuta una serie di cinque istruzioni: le

istruzioni comprese tra 108h e 113h, e 11Ah e 125h. Nel capitolo 7 vedrete come scrivere questa sequenza una volta sola, usando un'istruzione simile a GOSUB del BASIC.

SOMMARIO

In questo capitolo avete imparato il modo in cui Debug converte i numeri dal formato binario dell'8088 al formato esadecimale, alcuni flag a due lettere (quelli visualizzati con il comando R) e avete capito l'importanza dei bit di stato. Grazie a questi bit, infatti, è possibile ottenere numerose informazioni riguardo alle operazioni aritmetiche. Controllando il bit di stato del flag zero, per esempio, è possibile determinare se l'ultima operazione ha dato come risultato zero. Avete inoltre visto come confrontare due numeri con l'istruzione CMP.

Successivamente avete imparato a visualizzare un numero composto da una singola cifra esadecimale e avete visto come utilizzare l'istruzione SHR per ruotare più byte verso destra. Avete infine appreso l'istruzione AND che vi ha permesso di isolare i bit bassi di un byte. Con tutte queste nozioni, siete stati in grado di costruire un programma per visualizzare un numero composto da due cifre esadecimali.

Avremmo potuto continuare spiegandovi come visualizzare un numero di quattro cifre, ma questo avrebbe causato una ripetizione di istruzioni. Prima di imparare a visualizzare quattro cifre esadecimali, dovrete apprendere l'uso delle procedure (nel capitolo 7). Solo allora sarete in grado di scrivere una procedura che farà tutto il lavoro necessario. Ma ora proseguiamo vedendo ulteriori dettagli sui numeri esadecimali.

LETTURA DEI CARATTERI

Ora che siete in grado di visualizzare un byte in notazione esadecimale, vediamo il processo inverso: come leggere due caratteri dalla tastiera e convertirli in un byte.

LEGGERE UN CARATTERE

L'istruzione INT 21h ha una funzione di input, la numero 1, che permette di leggere un carattere dalla tastiera. Quando avete imparato le funzioni del comando INT, nel capitolo 4, avete visto che il numero di funzione deve essere inserito nel registro AH (prima di eseguire INT). Provate la funzione 1 per l'istruzione INT 21h. Inserite INT 21h nella locazione 100h:

```
396F:0100      CD21                INT     21
```

Inserite quindi 01h in AH e digitate *G 102* oppure *P* per eseguire questa singola istruzione. Vedrete ora un cursore lampeggiante (il DOS, infatti, sospende l'esecuzione fino a quando non viene premuto un tasto). Una volta premuto un tasto, il DOS pone il codice ASCII corrispondente nel registro AL. Userete questa istruzione successivamente, per leggere i caratteri di un numero esadecimale; al momento vediamo cosa succede se viene premuto un tasto particolare come, ad esempio, F1. Provate a premere F1. Il DOS inserisce 0 nel registro AL e, di fianco al prompt di Debug (il trattino) appare un punto e virgola (;).

F1 fa parte di una serie di tasti speciali che hanno un *codice esteso* e che il DOS tratta in maniera diversa dagli altri. (Potete trovare un elenco dei codici estesi nell'appendice D o nel manuale del BASIC). Per ciascuno di questi tasti speciali, il DOS invia *due* caratteri, uno dopo l'altro. Il primo carattere fornito è sempre zero, mentre il secondo è il *codice di scansione* per il tasto speciale.

Per leggere entrambi i caratteri, è necessario eseguire due volte l'istruzione INT 21h. Ma nell'esempio precedente avete letto solamente il primo carattere (zero) lasciando il codice di scansione nel DOS. Quando Debug ha terminato il comando G 102 (o P), ha iniziato a leggere i caratteri e il primo carattere letto è stato il codice di scansione di F1. Questo codice corrisponde a 59 ed è l'equivalente ASCII del punto e virgola. Più avanti, quando svilupperete il programma Dskpatch, userete questi codici estesi per manipolare i tasti cursore e i tasti funzione. Per ora concentratevi solamente sui normali caratteri ASCII.

LEGGERE UN NUMERO COMPOSTO DA UNA CIFRA ESADECIMALE

Invertite ora il processo di conversione visto nel capitolo 5, quando avete trasformato un numero esadecimale a una cifra nel codice ASCII corrispondente, per un carattere compreso tra 0 e 9 o tra A e F. Per convertire un carattere come, per esempio, C o D, da carattere esadecimale a byte, dovete sottrarre 30h (per i caratteri da 0 a 9) o 37h (per i caratteri da A a F). Ecco un semplice programma che legge un singolo carattere ASCII e lo converte in un byte:

```

3985:0100      B401          MOV  AH,01
3985:0102      CD21          INT  21
3985:0104      2C30          SUB  AL,30
3985:0106      3C09          CMP  AL,09
3985:0108      7E02          JLE  010C
3985:010A      2C07          SUB  AL,07
3985:010C      CD20          INT  20

```

Dovreste conoscere tutte le istruzioni di questo programma ad eccezione di JLE (*Jump if Less than or Equal*, Salta se è Minore di o Uguale). In questo programma il salto viene effettuato se AL è minore o uguale a 9.

Per vedere la conversione da carattere esadecimale a ASCII, dovete visualizzare il registro AL prima che venga eseguita l'istruzione INT 20h. Dato che Debug ripristina i registri dopo aver eseguito INT 20h, dovete impostare un punto di interruzione (breakpoint) come avete fatto nel capitolo 4. Digitate quindi *G 10C*; in questo modo potrete vedere il contenuto di AL.

Provate a digitare alcuni caratteri come, per esempio, *k* o la lettera *d* minuscola (che non sono cifre esadecimali) per vedere cosa succede. Noterete che questo programma funziona correttamente solo quando vengono inserite delle cifre comprese tra 0 e 9 o delle lettere maiuscole comprese tra A e F. Correggerete questo errore nel prossimo capitolo, quando verrete a conoscenza delle procedure; al momento ignorate questo errore.

LEGGERE UN NUMERO COMPOSTO DA DUE CIFRE ESADECIMALI

Leggere due cifre esadecimali non è un'operazione molto più complicata; richiede solamente più istruzioni. Iniziate leggendo la prima cifra, ponete quindi il valore esadecimale nel registro DL e moltipicatelolo per 16. Per eseguire questa moltiplicazione, dovete spostare i bit del registro DL di quattro posizioni a sinistra, ponendo quattro zeri al posto dei quattro bit bassi della cifra appena letta. L'istruzione SHL DL,CL con CL impostato a quattro si adatta perfettamente a questa situazione. L'istruzione SHL

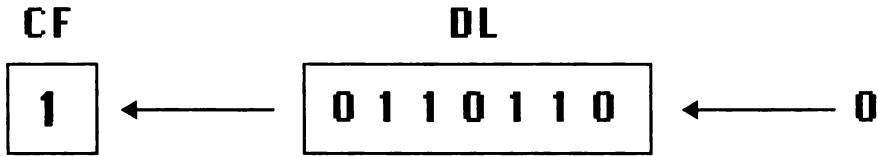


Figura 6-1. L'istruzione SHL DL,1 sposta i bit di una posizione a sinistra nel flag di riporto

(*Shift Left*, Sposta a Sinistra) è conosciuta come *spostamento aritmetico*, dato che ha lo stesso effetto di una moltiplicazione aritmetica per due, quattro, otto e così via, a seconda del numero contenuto in CL.

Infine, dopo aver spostato la prima cifra, sommate la seconda cifra esadecimale al numero in DL (la prima cifra moltiplicata per 16). Ecco come deve essere strutturato il programma:

```

3985:0100    B401            MOV  AH,01
3985:0102    CD21            INT  21
3985:0104    88C2            MOV  DL,AL
3985:0106    80EA30          SUB  DL,30
3985:0109    80FA09          CMP  DL,09
3985:010C    7E03            JLE  0111
3985:010E    80EA07          SUB  DL,07
3985:0111    B104            MOV  CL,04
3985:0113    D2E2            SHL  DL,CL
3985:0115    CD21            INT  21
3985:0117    2C30            SUB  AL,30
3985:0119    3C09            CMP  AL,09
3985:011B    7E02            JLE  011F
3985:011D    2C07            SUB  AL,07
3985:011F    00C2            ADD  DL,AL
3985:0121    CD20            INT  20

```

Ora che avete un programma funzionante, è una buona idea controllare le condizioni di limite per confermare che sia tutto corretto. Per queste condizioni di limite, usate i numeri 00, 09, 0A, 0F, 90, A0, F0, e qualche altro numero come, per esempio, 3C. Usate un punto di interruzione per eseguire il programma fermandovi all'istruzione INT 20h (assicuratevi di usare lettere maiuscole durante l'inserimento).

SOMMARIO

Avete finalmente avuto la possibilità di provare ciò che avete imparato nei capitoli precedenti, senza dover memorizzare una lunga serie di istruzioni nuove. Usando una nuova funzione dell'istruzione INT 21h (la numero 1), avete imparato a leggere dei caratteri dalla tastiera e avete impostato un programma per leggere un numero esadecimale a due cifre.

Ora siete pronti per conoscere l'uso delle procedure nell'8088.

LE PROCEDURE: PARENTI DELLE SUBROUTINE

Nel prossimo capitolo incontrerete MASM, il macro assembler, e inizierete a usare il linguaggio *assembler*. Ma prima di lasciare Debug, vediamo un'ultima serie di esempi, spiegando che cosa sono le procedure e lo stack.

LE PROCEDURE

Una procedura è una lista di istruzioni che può essere eseguita da qualsiasi parte di un programma. In pratica, quando si deve usare la stessa serie di istruzioni in più parti di uno stesso programma, invece di scrivere più volte la stessa sequenza, è possibile creare una procedura che può essere richiamata quando necessario. Una procedura è il corrispondente di una *subroutine* in BASIC; in questo caso, però, vengono chiamate *procedure* per dei motivi che vedrete in seguito.

Per richiamare e uscire da una procedura bisogna agire in modo molto simile al BASIC. Per chiamare una procedura si usa l'istruzione *CALL* (che corrisponde a GOSUB in BASIC), mentre per uscire da una procedura si usa *RET* (che corrisponde a RETURN in BASIC).

Ecco un semplice programma in BASIC che tra poco scriverete in linguaggio macchina. Questo programma chiama una subroutine dieci volte, mostrando ogni volta un carattere (partendo da A fino a J):

```

10  A = &H41                                'ASCII DI A'
20  FOR I = 1 TO 10
30  GOSUB 1000
40  A = A + 1
50  NEXT I
60  END

1000 PRINT CHR$(A) ;
1200 RETURN

```

La subroutine, secondo una pratica comune dei programmi in BASIC, inizia alla riga 1000 in modo da lasciare spazio per l'eventuale aggiunta di istruzioni al programma principale (senza interferire con la subroutine). Seguirete lo stesso metodo con la procedura in linguaggio macchina, facendola iniziare alla locazione 200h (molto

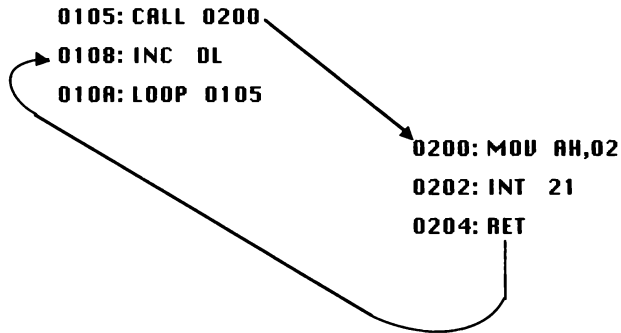


Figura 7-1. Le istruzioni CALL e RET

lontana dall'inizio del programma che è 100h). Sostituirete l'istruzione GOSUB 1000 con CALL 200h, che *chiama* la procedura alla locazione 200h (CALL imposta IP su 200h).

Il ciclo FOR-NEXT può essere scritto con un'istruzione LOOP (come avete visto nel capitolo 4). Le altre parti del programma, ad eccezione dell'istruzione INC, dovrebbero esservi familiari.

3985:0100	B241	MOV DL, 41
3985:0102	B90A00	MOV CX, 000A
3985:0105	E8F800	CALL 0200
3985:0108	FEC2	INC DL
3985:010A	E2F9	LOOP 0105
3985:010C	CD20	INT 20

La prima istruzione inserisce 41h (il codice ASCII di A) nel registro DL, dato che l'istruzione INT 21h visualizza il carattere il cui codice ASCII si trova in DL. INT 21h si trova lontano dal programma, nella procedura che inizia alla locazione 200h. INC DL; l'istruzione nuova, *incrementa* il registro DL. In questo modo, aggiungendo uno a DL, viene impostato il codice ASCII per il carattere successivo. Ecco la procedura da inserire in 200h:

3985:0200	B402	MOV AH, 02
3985:0202	CD21	INT 21
3985:0204	C3	RET

Ricordatevi che 02h in AH indica al DOS di visualizzare il carattere contenuto in DL quando viene eseguita l'istruzione INT 21h. RET è un'istruzione nuova che *ritorna* alla prima istruzione che segue il comando CALL.

Digitate G per vedere l'output del programma; tracciate quindi le istruzioni passo a passo per vedere come funzionano (ricordatevi di usare un punto di interruzione o il comando P per eseguire INT 21h).

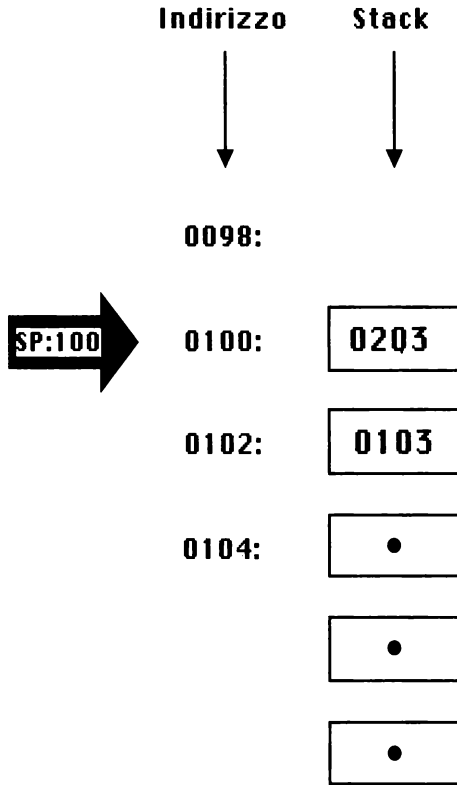


Figura 7-2. Lo stack appena prima di eseguire l'istruzione CALL 400

LO STACK E GLI INDIRIZZI DI RITORNO

L'istruzione CALL in un programma deve salvare *l'indirizzo di ritorno* in qualche locazione di memoria del computer, per poter sapere dove riprendere l'esecuzione dopo aver incontrato l'istruzione RET. La porzione di memoria in cui vengono registrati questi indirizzi è chiamata stack. Per poter gestire i dati nello stack sono previsti due registri, entrambi visualizzati con il comando R: SP (*Stack Pointer*, Puntatore dello Stack) che punta l'inizio dello stack, e SS (*Stack Segment*, Segmento dello Stack) che contiene il numero di segmento.

Possiamo immaginare lo stack come una pila di vassoi in un self service dove, ponendo un vassoio in cima alla pila, si coprono quelli sottostanti. L'ultimo vassoio della pila è il primo ad essere preso; per questo motivo un altro nome per lo stack è LIFO (*Last In, First Out*, ultimo a entrare, primo a uscire).

Questo ordine, LIFO, è quello da considerare per recuperare gli indirizzi di ritorno dopo aver effettuato delle istruzioni CALL *nidificate*, come nell'esempio seguente:

```

396F:0100    E8FD00    CALL 0200
              .
              .
              .
396F:0200    E8FD00    CALL 0300
396F:0203    C3        RET
              .
              .
              .
396F:0300    E8FD00    CALL 0400
396F:0303    C3        RET
              .
              .
              .
396F:0400    C3        RET

```

In questo esempio, l'istruzione in 100h ne chiama una in 200h, che a sua volta ne chiama una in 300h che a sua volta ne chiama un'altra in 400h dove, infine, si trova

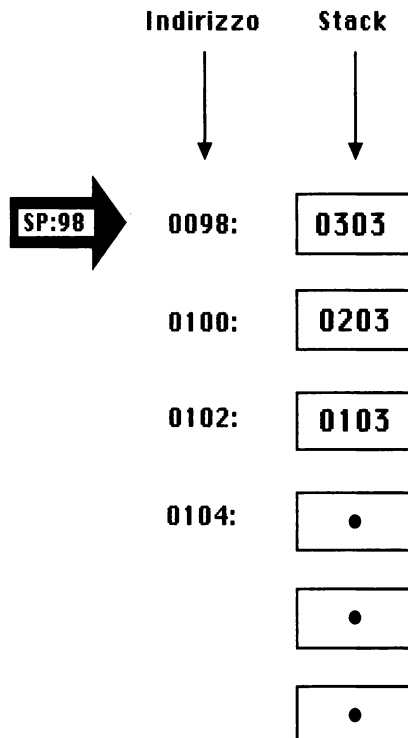


Figura 7-3. Lo stack dopo avere eseguito l'istruzione CALL 400

l'istruzione RET. Questo RET riporta il controllo del programma all'istruzione successiva dell'*ultimo* CALL eseguito (in 300h) e riprende quindi l'esecuzione da 303h. Ma in 303h c'è un altro RET, per cui il programma torna in 203h e, dato che in 203h è presente ancora un RET, il programma riparte da 103h. Ciascuna istruzione RET preleva l'ultimo indirizzo di ritorno presente nello stack riportando il controllo alla precedente istruzione CALL.

Provate a inserire un programma simile al precedente. Usate chiamate multiple e tracciate il programma per vedere come funzionano le istruzioni CALL e RET. Benché il processo possa sembrare poco interessante, una buona conoscenza di come lavora lo stack è necessaria per operazioni che svilupperete più avanti.

LE ISTRUZIONI PUSH E POP

Nello stack è possibile memorizzare temporaneamente dei dati, sempre che ci si ricordi di ripristinare lo stack prima di eseguire un'istruzione RET. Avete visto che CALL *inserisce* l'indirizzo di ritorno all'inizio dello stack, mentre RET *preleva* l'ultimo indirizzo presente nello stack e lo inserisce nel registro IP. E' possibile effettuare queste operazioni con le istruzioni PUSH e POP, che permettono di inserire ed estrarre dei byte dallo stack.

Molto spesso è utile salvare i valori dei registri all'inizio di una procedura e recuperarli alla fine, prima dell'istruzione RET. In questo modo i registri possono essere usati per qualsiasi funzione all'interno della procedura, senza preoccuparsi dei dati precedentemente contenuti.

I programmi sono spesso costruiti con procedure nidificate a più livelli; salvando i registri all'inizio di una procedura e ripristinandoli alla fine, si eliminano interazioni non desiderate tra le procedure nei differenti livelli, rendendo più semplice la programmazione. Ulteriori dettagli sul salvataggio e il ripristino dei registri saranno dati nel capitolo 13, dove verrà discussa la progettazione modulare. Guardate ora un esempio (non inseritelo) su come salvare e ripristinare i registri CX e DX:

396F:0200	51	PUSH CX
396F:0201	52	PUSH DX
396F:0202	B90800	MOV CX, 0008
396F:0205	E8F800	CALL 0300
396F:0208	FEC2	INC DL
396F:020A	E2F9	LOOP 0205
396F:020C	5A	POP DX
396F:020D	59	POP CX
396F:020E	C3	RET

Notate che le istruzioni POP sono in ordine inverso rispetto a PUSH, dato che POP estrae l'ultima parola presente nello stack e il vecchio valore di DX si trova sopra al vecchio valore di CX.

L'aver salvato il contenuto dei registri CX e DX, ha permesso di modificare questi

registri nella procedura che inizia alla locazione 200h, ma senza cambiare i valori usati dalla procedura che l'ha chiamata. Una volta salvato il contenuto di CX e DX, è possibile usare questi registri per memorizzare delle variabili *locali* da utilizzare all'interno della procedura senza modificare i valori usati dal programma chiamante. Userete queste variabili locali per semplificare il lavoro di programmazione. Se vi ricorderete di salvare i valori dei registri, non dovrete temere che le procedure modifichino i registri del programma chiamante. Tutto questo vi sarà più chiaro nel prossimo esempio, in cui userete una procedura per leggere un numero esadecimale. A differenza del programma sviluppato nel capitolo 6, questo programma permetterà di inserire solamente un carattere valido (per esempio, A e non K).

LEGGERE DEI NUMERI ESADECIMALI IN MODO ELEGANTE

Vediamo come creare una procedura che continui a leggere dei caratteri fino a quando non riceve un carattere convertibile in esadecimale (compreso tra 0 e Fh). Dato che sono ammessi solo determinati caratteri, userete una nuova funzione dell'istruzione INT 21h, la numero 8, che legge un carattere ma non lo visualizza sullo schermo. In questo modo potrete visualizzare i caratteri solo se questi sono validi. Inserite 8h nel registro AH e tracciate questa istruzione, digitando una A dopo aver impartito il comando G 102:

```
3985:0100      CD21          INT    21
```

Il codice ASCII della A (41h) si trova ora nel registro AL, ma la A non appare sullo schermo.

Usando questa funzione, il programma può leggere dei caratteri visualizzando solo le cifre esadecimali valide (da 0 a 9 o da A a F). Ecco la procedura per effettuare questa operazione e per convertire il carattere in un numero esadecimale:

```
3985:0200      52              PUSH   DX
3985:0201      B408           MOV    AH, D8
3985:0203      CD21          INT    21
3985:0205      3C30           CMP    AL, 30
3985:0207      72FA           JB     0203
3985:0209      3C46           CMP    AL, 46
3985:020B      77F6           JA     0203
3985:020D      3C39           CMP    AL, 39
3985:020F      770A           JA     021B
3985:0211      B402           MOV    AH, 02
3985:0213      88C2           MOV    DL, AL
3985:0215      CD21          INT    21
3985:0217      2C30           SUB    AL, 30
```

3985:0219	5A	POP	DX
3985:021A	C3	RET	
3985:021B	3C41	CMP	AL, 41
3985:021D	72E4	JB	0203
3985:021F	B402	MOV	AH, 02
3985:0221	88C2	MOV	DL, AL
3985:0223	CD21	INT	21
3985:0225	2C37	SUB	AL, 37
3985:0227	5A	POP	DX
3985:0228	C3	RET	

La procedura legge un carattere in AL (con l'istruzione INT 21h in 203h) e controlla se è valido con CMP e con il salto condizionale. Se il carattere appena letto non è un carattere valido, il salto condizionale riporta l'esecuzione del programma nuovamente in 203h, dove INT 21h legge un altro carattere. JA significa *Jump if Above* (Salta se Sopra) mentre JB significa *Jump if Below* (Salta se Sotto); entrambi trattano i numeri come numeri senza segno a differenza di JL (usata in precedenza) che considera solo numeri con segno.

Quando il programma raggiunge la riga 211h, significa che è stata inserita una cifra compresa tra 0 e 9; in questo caso, viene effettuata la sottrazione per determinare il carattere, il risultato viene messo in AL e l'istruzione POP ripristina il contenuto del registro DX (salvato all'inizio della procedura). Il processo per le cifre esadecimali comprese tra A e F è analogo. Notate che ci sono due istruzioni RET in questa procedura; se ne potrebbero avere di più, o solamente una.

Ecco un semplice programma per verificare la procedura:

3985:0100	E8FD00	CALL	0200
3985:0103	CD20	INT	20

Come avete fatto prima, usate il comando G con un punto di interruzione o il comando P. Dovete eseguire l'istruzione CALL 200h senza eseguire INT 20h in modo da poter vedere il contenuto dei registri prima che questi vengano ripristinati.

Dopo aver eseguito il programma, vedrete il cursore lampeggiare sul lato sinistro del video. Digitate *k* che non è un carattere valido. Non succede niente. Digitate ora un qualsiasi carattere esadecimale (in maiuscolo). Vedrete il valore esadecimale del carattere nel registro AL e il carattere stesso visualizzato sullo schermo. Provate questa procedura con le condizioni di limite: '\ ' (il carattere prima dello zero), 0, 9, ':' (il carattere dopo il nove) e così via.

Ora che avete creato questa procedura, potete scrivere un programma per leggere un numero esadecimale a due cifre nel modo seguente:

3985:0100	E8FD00	CALL	0200
3985:0103	88C2	MOV	DL, AL
3985:0105	B104	MOV	CL, 04
3985:0107	D2E2	SHL	DL, CL
3985:0109	E8F400	CALL	0200

3985:010C	00C2	ADD	DL, AL
3985:010E	B402	MOV	AH, 02
3985:0110	CD21	INT	21
3985:0112	CD20	INT	20

Potete eseguire questo programma dal DOS, dato che legge un numero esadecimale a due cifre e visualizza quindi il carattere ASCII corrispondente al numero digitato. Grazie a questa procedura, questo programma è molto più semplice di quello scritto nel capitolo precedente. Inoltre è stata inserita la gestione degli errori (vengono accettati solo caratteri validi) e non ci sono istruzioni ripetute.

Avete anche visto la ragione per cui è necessario salvare il contenuto del registro DX nello stack. Il programma principale memorizza il numero esadecimale in DL, quindi è necessario che la procedura non modifichi questo registro. D'altro canto, la procedura in 200h deve usare DL per visualizzare i caratteri. Per questo motivo, usando l'istruzione PUSH DX all'inizio della procedura e POP DX alla fine, è stato possibile effettuare entrambe le operazioni senza problemi.

Da questo momento in avanti, per evitare complicazioni, saranno sempre salvati i registri usati dalle procedure.

SOMMARIO

Piano piano la programmazione si sta facendo sempre più sofisticata. Avete imparato le procedure, che permettono di usare la stessa serie di istruzioni senza doverle riscrivere ogni volta. Siete inoltre venuti a conoscenza dello stack e avete visto come l'istruzione CALL memorizza l'indirizzo di ritorno all'inizio dello stack e come RET la preleva per riportare l'esecuzione del programma nel punto corretto.

Avete visto come usare lo stack per altre operazioni; per memorizzare il contenuto dei registri (con PUSH) e per ripristinarli (con POP). Salvando e ripristinando i registri usati nelle procedure, è possibile chiamare delle procedure senza temere di modificare dei registri usati dal programma principale.

Infine, con tutte queste nozioni, avete costruito un programma per leggere dei numeri esadecimali, questa volta con il controllo degli errori. Il programma scritto in questa sede, è simile a un programma che utilizzerete più avanti, quando inizierete a sviluppare Dskpatch.

Ora siete pronti per iniziare la seconda parte del libro, in cui imparerete a usare l'assemblatore. Nel prossimo capitolo vedrete come usare l'assemblatore per convertire un programma in linguaggio macchina. Vedrete inoltre che non è necessario lasciare spazio tra il programma e le procedure (come avete fatto in questo capitolo, inserendo la procedura in 200h).

PARTE II

LINGUAGGIO ASSEMBLY

L'ASSEMBLATORE

Finalmente siete pronti per “incontrare” l'assemblatore, un programma DOS che rende la programmazione molto più semplice. Da questo momento in poi, userete delle istruzioni mnemoniche, molto più vicine al linguaggio dell'uomo e userete l'assemblatore per convertirle in linguaggio macchina.

I primi due capitoli di questa seconda parte saranno necessariamente appesantiti con dettagli sull'assemblatore, ma i risultati che potrete ottenere varranno lo sforzo. Una volta che avrete imparato a usare l'assemblatore, potrete imparare a scrivere dei programmi in linguaggio assembler.

UN PROGRAMMA SENZA DEBUG

Fino a questo momento avete sempre usato DEBUG per scrivere i vostri programmi. Ora state per abbandonare Debug e per scrivere dei programmi senza il suo apporto. Avrete bisogno di un editor di testo o di un programma di word processor per creare i file contenenti le istruzioni in linguaggio assembler.

Inizierete creando un *file sorgente* (il nome per la versione di testo di un programma in assembler). Create ora un file sorgente per il programma che avete costruito nel capitolo 3 e che avete chiamato SCRIVE.COM. Per rinfrescarvi la memoria, ecco come era la versione in Debug:

396F:0100	B402	MOV	AH, 02
396F:0102	B261	MOV	DL, 2A
396F:0104	CD21	INT	21
396F:0106	CD20	INT	20

Usate l'editor per inserire le seguenti righe in un file chiamato SCRIVE.ASM (l'estensione .ASM significa che questo è un file sorgente in linguaggio assembler). Anche in questo caso, come in Debug, si possono usare indifferentemente le lettere maiuscole e minuscole; noi vi consigliamo di usare le maiuscole per evitare di fare confusione, per esempio, tra il numero 1 e la lettera l minuscola (elle):

```
.MODEL SMALL
.CODE
```

```
MOV  AH, 2h
MOV  DL, 2Ah
INT  21h
INT  20h

END
```

Questo è lo stesso programma creato nel capitolo 3, ma contiene alcune modifiche necessarie. Ignorate, al momento, le tre righe nuove contenute nel file sorgente; notate che è stata aggiunta una *b* dopo ciascun numero esadecimale. Questa *b* indica all'assemblatore che si stanno utilizzando numeri in notazione esadecimale. A differenza di Debug, che assume che tutti i numeri siano in esadecimale, l'assemblatore assume che tutti i numeri siano in notazione decimale. Questa *b*, quindi, indica l'utilizzo di numeri esadecimali.

Nota: un avvertimento prima di procedere: l'assemblatore può fare confusione con alcuni numeri come, per esempio, ACh, che assomigliano a un nome o a un'istruzione. Per evitare questo, digitate sempre uno zero prima di un numero che inizia con una lettera. Per esempio, digitate 0ACh e *non* ACh.

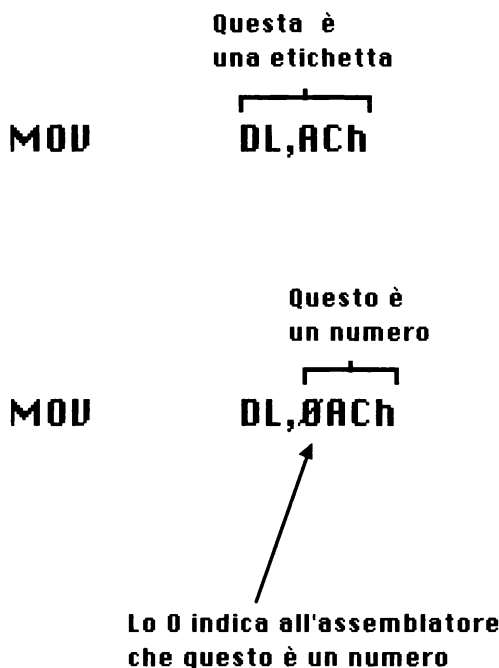


Figura 8-1. Dovete inserire uno 0 prima di un numero esadecimale che inizia con una lettera, altrimenti l'assemblatore tratterà il numero come un nome

Guardate cosa succede quando assemblate un programma con ACh invece di 0ACh. Ecco il programma:

```
.MODEL SMALL
.CODE

        MOV    DL,ACh
        INT    20h

        END
```

Ecco l'output:

```
A> MASM TEST;
Microsoft (R) Macro Assembler Version 5.10
Copyright (C) Microsoft Corp 1981, 1988. All rights reserved.

test.ASM(4) : error A2009: Symbol not defined: ACH

49842 + 224473 Bytes symbol space free

0 Warning Errors
1 Severe Errors
A>
```

Ma se cambiate ACh in 0ACh l'assemblatore funzionerà correttamente. Notate inoltre la spaziatura utilizzata nel programma. Sono stati usati dei tabulatori per allineare le varie istruzioni e rendere il file sorgente più leggibile. Confrontate il programma inserito con questa versione:

```
.MODEL SMALL
.CODE

        MOV    AH,2h
        MOV    DL,2Ah
        INT    21h
        INT    20h
        END
```

Un pasticcio! L'assemblatore non se ne accorge, ma voi sì. Ritornate ora alle tre righe nuove nel file sorgente. Le tre righe nuove sono tutte *direttive* (qualche volta chiamate *pseudo-ops* o pseudo-operazioni). Queste sono chiamate direttive perché, invece di generare istruzioni, forniscono delle informazioni e delle direzioni all'assemblatore. La direttiva END nel file sorgente, indica all'assemblatore che il programma è terminato. Successivamente vedrete che END è utile anche in altri contesti. Al momento, lasciate da parte il discorso sulle direttive e addentratevi nell'assemblatore.

CREARE I FILE SORGENTE

Prima di assemblare il programma appena inserito, vogliamo fare un'ulteriore considerazione: l'assemblatore può processare solamente dei file sorgente che contengono caratteri in ASCII puro. Se usate un word processor, ricordatevi di salvare il file sorgente senza le specifiche di formato. Quindi, prima di assemblare SCRIVE.ASM, assicuratevi che sia in formato ASCII puro.

Dal DOS digitate:

```
A>TYPE SCRIVE.ASM
```

Dovreste vedere lo stesso testo che avete inserito. Se vedete dei caratteri strani nel programma (molti word processor aggiungono delle specifiche di formato che sono considerati errori dall'assemblatore) dovreste usare un editor o un programma di word processor differente. E' anche necessaria una riga vuota dopo il comando END. Ora assemblate il programma SCRIVE.ASM. (Se state usando il Turbo Assembler della Borland, digitate TASM al posto di MASM; se state usando OPTASM della SLR Systems, digitate OPTASM al posto di MASM).. Assicuratevi di digitare il punto e virgola:

```
A> MASM SCRIVE ;
Microsoft (R) Macro Assembler Version 5.10
Copyright (C) Microsoft Corp 1981, 1988. All rights reserved.

49822 + 219323 Bytes symbol space free

0 Warning Errors
0 Severe Errors

A>
```

Non avete ancora finito. A questo punto, l'assemblatore ha prodotto un file chiamato SCRIVE.OBJ, che si trova ora sul disco. Questo è un file di passaggio chiamato *file oggetto*. Questo file contiene il programma in linguaggio macchina e una serie di informazioni che vengono usate da un altro programma DOS chiamato *Linker*.

LINKING

A questo punto, usate il linker per creare un file .EXE dal file oggetto. Copiate LINK.EXE dal dischetto DOS nel disco contenente il file sorgente e l'assemblatore (o sul disco fisso). Digitate quindi:

```
A> LINK SCRIVE ;
Microsoft (R) Overlay Linker Version 3.64
Copyright (C) Microsoft Corp 1981, 1988. All rights reserved.

LINK: warning L4021: no stack segment

A>
```

Anche se il linker comunica che non esiste un segmento di stack, al momento non ne avete bisogno. Capirete successivamente perché, in determinate situazioni, è richiesto un segmento di stack.

Ora avete un file .EXE. ma manca ancora l'ultimo passo. Dovete creare una versione .COM, proprio come quella creata con Debug. Anche in questo caso, vedrete più avanti il perché di questa operazione. Per ora limitatevi a creare una versione .COM di SCRIVE.EXE.

Per questo passo finale, avete bisogno del programma EXE2BIN.EXE che si trova sul dischetto supplementare del DOS. EXE2BIN converte un file .EXE in un file .COM (o binario). C'è una differenza tra i file .EXE e .COM, ma la vedrete nel capitolo 11. Per ora convertire semplicemente questo file:

```
A>EXE2BIN SCRIVE SCRIVE.COM
```

```
A>
```

Non appare alcuna risposta. Per vedere se questo comando ha funzionato, visualizzate tutti i file Scrive che avete creato:

```
A>DIR SCRIVE.*
```

```
Volume in drive A has no label
```

```
Directory of A:\
```

```
SCRIVE.ASM          76    30-01-90   12:02p
SCRIVE.OBJ          105    30-01-90   12:05p
SCRIVE.EXE          520    30-01-90   12:05p
SCRIVE.COM           8    30-01-90   12:06p
      4 file(s)      327432 bytes free
```

```
A>
```

Ci sono quattro file, incluso SCRIVE.COM. Digitate *scrive* per eseguire la versione .COM e verificare che funzioni correttamente (ricordatevi che deve apparire un asterisco sullo schermo). La dimensione dei file riportata dal DOS per i primi tre file potrebbe variare leggermente.

Vi potrà sembrare di essere tornati indietro dato che il risultato è identico a quello ottenuto nel capitolo 3, ma non è così: avete imparato molto. Vi sarà tutto molto più chiaro successivamente. Notate che non avete dovuto preoccuparvi della posizione del programma in memoria e di impostare il registro IP. L'assemblatore ha fatto tutto per voi.

Molto presto inizierete ad apprezzare molto questa caratteristica dell'assemblatore, dato che rende la programmazione molto più semplice. Per esempio, nell'ultimo capitolo avete sprecato spazio ponendo il programma principale in 100h e la procedura in 200h. Vedrete che con l'assemblatore è possibile posizionare la

procedura immediatamente dopo il programma senza alcuno spazio in mezzo. Ma prima tornate un momento al Debug.

ANCORA IL DEBUG

Visualizzate il file .COM appena creato usando Debug e disassemblatelo per vedere come Debug ricostruisce il programma dal codice macchina:

```
A>DEBUG SCRIVE.COM
-U
397F:0100      B402          MOV  AH,02
397F:0102      B261          MOV  DL,2A
397F:0104      CD21          INT  21
397F:0106      CD20          INT  20
```

Esattamente come nel capitolo 3. Questo è tutto ciò che Debug vede nel file SCRIVE.COM. Le tre direttive aggiunte nel file sorgente non appaiono. Dove sono andate?

Queste istruzioni non appaiono nella versione finale in linguaggio macchina proprio perché sono direttive e servono solo per riferimento. L'assemblatore utilizza questi riferimenti per dei servizi interni. Vedrete successivamente (nel capitolo 11) come queste direttive possono semplificare il lavoro.

COMMENTI

Dato che non state lavorando più con Debug, siete liberi di aggiungere dei commenti a un programma, senza "passarli" all'8088. I commenti sono importantissimi per rendere un programma più chiaro e leggibile. Nei programmi in linguaggio assembler, i commenti vanno inseriti facendoli precedere da un punto e virgola (;), che funziona come un apostrofo (') in BASIC. L'assemblatore ignora qualsiasi cosa presente dopo un punto e virgola. Provate ad aggiungere dei commenti al programma:

```
.MODEL  SMALL
.CODE

      MOV  AH,2h   ;Seleziona la funzione DOS 2
      MOV  DL,2Ah ;Carica il codice ASCII da visualizzare (*)
      INT  21h    ;Visualizza il carattere
      INT  20h    ;Esce al DOS

      END
```


In questo modo potete capire il programma senza dover ricordarvi cosa significa ciascuna riga.

ETICHETTE

Per terminare questo capitolo, vediamo un'altra funzione dell'assemblatore per rendere più semplice la programmazione: le etichette.

Fino ad ora, dovevate conoscere l'indirizzo specifico per inserire un'istruzione di salto. Capita però spesso di dover inserire delle istruzioni tra una riga e un'altra e, nel momento in cui si inserisce una riga, bisogna cambiare tutti i riferimenti fatti a indirizzi specifici. Per evitare questo inconveniente, l'assemblatore mette a disposizione dell'utente le *etichette* che permettono di assegnare un nome a un indirizzo o a una locazione di memoria. Un'etichetta prende il posto di un indirizzo. Nel momento in cui il programma viene assemblato, l'assemblatore sostituisce automaticamente le etichette con l'indirizzo corretto.

Le etichette possono essere lunghe al massimo 31 caratteri e possono contenere lettere, numeri e i seguenti simboli: punto di domanda (?), punto (.), chiocciola (@), sottolineato (_), e dollaro (\$).

Non possono iniziare con una cifra (da 0 a 9), e il punto può essere usato solo come primo carattere.

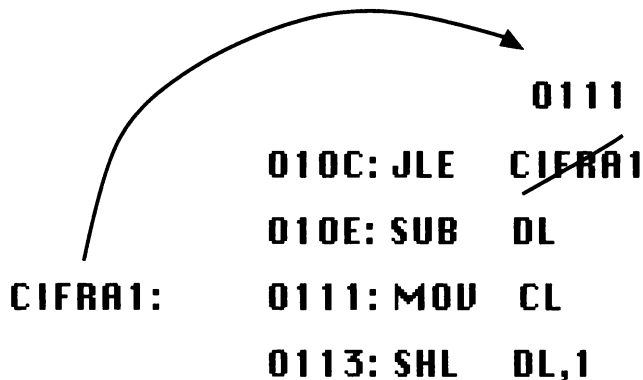


Figure 8-2. L'assemblatore sostituisce le etichette con gli indirizzi corretti.

Come esempio, prendete il programma sviluppato nel capitolo 6 per leggere due cifre esadecimali. Questo contiene due istruzioni di salto, JLE 0111 e JLE 011F. Ecco la versione vecchia:

```

3985:0100      B401          MOV  AH,01
3985:0102      CD21          INT  21
3985:0104      88C2          MOV  DL,AL
3985:0106      80EA30        SUB  DL,30
3985:0109      80FA09        CMP  DL,09
3985:010C      7E03          JLE  0111
3985:010E      80EA07        SUB  DL,07
3985:0111      B104          MOV  CL,04
3985:0113      D2E2          SHL  DL,CL
3985:0115      CD21          INT  21
3985:0117      2C30          SUB  AL,30
3985:0119      3C09          CMP  AL,09
3985:011B      7E02          JLE  011F
3985:011D      2C07          SUB  AL,07
3985:011F      00C2          ADD  DL,AL
3985:0121      CD20          INT  20

```

Non è certamente ovvio il significato di questo programma, e se non l'avete bene in memoria probabilmente farete fatica a capire come funziona esattamente. Aggiungete delle etichette e dei commenti:

```

.MODEL  SMALL
.CODE
MOV  AH,01h  ;Seleziona la funzione 1 del DOS, richiesta carattere
INT  21h     ;Legge un carattere, e inserisce il codice ASCII in AL
MOV  DL,AL   ;Sposta il codice ASCII in DL
SUB  DL,30h  ;Sottrae 30h per convertire le cifre da 0 a 9
CMP  DL,9h   ;E' una cifra compresa tra 0 e 9?
JLE  CIFRA1  ;Sì, abbiamo la prima cifra (4 bit)
SUB  DL,7h   ;No, sottrae 7h per convertire le lettere A-F

CIFRA1:
MOV  CL,4h   ;Prepara la moltiplicazione per 16
SHL  DL,CL   ;Moltiplica per scorrimento
INT  21h     ;Preleva il carattere successivo
SUB  AL,30h  ;Ripete la conversione
CMP  AL,9h   ;E' una cifra compresa tra 0 e 9?
JLE  CIFRA2  ;Sì, abbiamo la seconda cifra
SUB  AL,7h   ;No, sottrae 7h

CIFRA2:
ADD  DL,AL   ;Somma la seconda cifra
INT  20h     ;Esce al DOS

END

```

Le etichette CIFRA1 e CIFRA2 sono del tipo conosciuto come *NEAR* (vicine), poiché il segno dei due punti (:) appare dopo ciascuna etichetta definita. Il termine *NEAR* deriva dai segmenti che vedrete nel capitolo 11, insieme alle direttive `.MODEL` e `.CODE`. In questo caso, se assemblete il programma precedente e lo disassemblate con Debug, vedrete che CIFRA1 è stato sostituito da 0111h e CIFRA2 da 011Fh.

SOMMARIO

Questo capitolo è stato abbastanza impegnativo; è come se foste entrati in un nuovo mondo. L'assemblatore rende molto più semplice il lavoro rispetto a Debug e permette di creare dei grossi programmi senza molta fatica.

In questo capitolo avete imparato a creare dei file sorgente e quindi ad assemblerli e a convertirli (dopo aver usato rispettivamente il programma LINK e il programma EXE2BIN) da file `.OBJ` a `.EXE` e `.COM`. Il programma in linguaggio assembler che avete creato conteneva alcune direttive che non avevate mai incontrato. Queste diventeranno familiari nei prossimi capitoli. Userete le direttive `.MODEL`, `.CODE` e `END` in tutti i programmi che svilupperete da questo momento in avanti, e ne capirete la funzione nel capitolo 11.

Avete inoltre imparato a inserire dei commenti. I commenti sono fondamentali per poter scrivere un programma chiaro e comprensibile. D'ora in avanti, li userete sempre.

Infine avete visto come utilizzare le etichette che a loro volta rendono la programmazione ancora più semplice. Nel prossimo capitolo vedrete come lavorare con l'assemblatore e le procedure.

LE PROCEDURE E L'ASSEMBLATORE

Ora che avete incontrato l'assemblatore, potrete scrivere più facilmente dei programmi in linguaggio assembly. In questo capitolo, vedrete nuovamente le procedure e imparerete a scriverle in un modo molto più semplice, grazie all'aiuto dell'assemblatore. Costruirete quindi qualche procedura utile che potrete utilizzare durante la costruzione del programma Dskpatch.

Inizierete creando due procedure per visualizzare un byte in esadecimale. Durante il lavoro, incontrerete alcune direttive. Ma, come per .MODEL, .CODE e END, non le approfondirete lasciandole in sospeso fino al capitolo 11.

LE PROCEDURE DELL'ASSEMBLATORE

La prima volta che avete usato una procedura, avete lasciato un grosso spazio vuoto tra il programma principale e la procedura stessa, in modo da poter avere la possibilità di effettuare delle modifiche al programma senza correre il rischio di sovrascrivere la procedura. Ma ora che usate l'assemblatore, e dato che questo assegna automaticamente gli indirizzi corretti, non avrete più bisogno di lasciare dello spazio tra le procedure. Con l'assemblatore, ogni volta che vengono effettuate delle modifiche, è sufficiente assemblare nuovamente il programma.

Nel capitolo 7 avete costruito una piccola procedura con un'istruzione CALL. Il programma non faceva altro che visualizzare le lettere da A a J e appariva nel modo seguente:

```

3985:0100      B241                MOV  DL, 41
3985:0102      B90A00             MOV  CX, 000A
3985:0105      E8F800             CALL 0200
3985:0108      FEC2                INC  DL
3985:010A      E2F9                LOOP 0105
3985:010C      CD20                INT  20

3985:0200      B402                MOV  AH, 02
3985:0202      CD21                INT  21
3985:0204      C3                  RET

```

Convertitelo in un programma per l'assemblatore. Dato che è difficile capire un programma senza etichette e commenti, aggiungeteli:

Listato 9-1. Il programma PRINTAJ.ASM

```

.MODEL    SMALL
.CODE

PRINT_A_J PROC
    MOV    DL,'A'           ;Inizia con il carattere A
    MOV    CX,10           ;Visualizza 10 caratteri partendo con A

PRINT_LOOP:
    CALL  WRITE_CHAR      ;Visualizza il carattere
    INC   DL               ;Si sposta sul carattere successivo
    LOOP PRINT_LOOP       ;Continua per 10 caratteri
    INT   20h              ;Ritorna al DOS
PRINT_A_J    ENDP

WRITE_CHAR  PROC
    MOV    AH,2            ;Imposta funzione per visualizzazione
del carattere
    INT   21h              ;Visualizza il carattere in DL
    RET                                ;Ritorna dalla procedura
WRITE_CHAR  ENDP

END    PRINT_A_J

```

In questo programma ci sono due nuove direttive: PROC e ENDP che servono per definire le procedure. Come potete vedere, sia il programma principale che la procedura sono racchiuse dalla coppia di direttive PROC e ENDP.

PROC definisce l'inizio della procedura mentre ENDP definisce la fine. Il nome che si trova davanti alla direttiva, è il nome assegnato alla procedura. Quindi, nella procedura principale (PRINT_A_J) si può sostituire l'istruzione CALL 200 con una più leggibile: CALL WRITE_CHAR. Inserite semplicemente il nome della procedura; l'assemblatore assegnerà automaticamente l'indirizzo corretto.

Dato che ci sono due procedure, bisogna dire all'assemblatore qual è la procedura principale (quella da cui deve partire il programma). La direttiva END serve a questo scopo. Scrivendo END PRINT_A_J avete detto all'assemblatore che questa è la procedura principale. Successivamente, vedrete che la procedura principale può essere inserita in qualsiasi posizione. Per ora, tuttavia, inserite la procedura principale all'inizio del file sorgente.

Siete ora pronti per generare un file .COM. Se non l'avete ancora fatto, inserite il programma in un file PRINTAJ.ASM ed eseguite le stesse operazioni effettuate nell'ultimo capitolo (ricordatevi di sostituire MASM con TASM o OPTASM se usate rispettivamente il Turbo Assembler o OPTASM):

```

MASM PRINTAJ;
LINK PRINTAJ;
EXE2BIN PRINTAJ PRINTAJ.COM

```

Provate ora il programma. (Assicuratevi di aver eseguito EXE2BIN *prima* di lanciare PRINTAJ altrimenti, eseguendo una versione .EXE, il programma si bloccherà nel momento in cui raggiunge l'istruzione INT 20h per ragioni che vedrete nel capitolo 11).

Nota: Se appaiono dei messaggi di errore, controllate di aver digitato il programma correttamente, o consultate l'appendice C che elenca gli errori più comuni.

Se tutto funziona correttamente, usate Debug per disassemblare il programma e per vedere come l'assemblatore ha inserito la procedura. Ricordatevi che per leggere un determinato file dovete far seguire al comando Debug il nome di quel file. In questo caso, digitate *DEBUG PRINTAJ.COM*:

```
-U
3985:0100      B241          MOV    DL, 41
3985:0102      B90A00       MOV    CX, 000A
3985:0105      E80600       CALL  010E
3985:0108      FEC2         INC    DL
3985:010A      E2F9         LOOP  0105
3985:010C      CD20         INT    20
3985:010E      B402         MOV    AH, 02
3985:0110      CD21         INT    21
3985:0112      C3           RET
```

Ora non c'è più dello spazio inutile tra il programma principale e la procedura.

LE PROCEDURE DI OUTPUT ESADECIMALE

Avete visto due metodi per ottenere un output esadecimale: nel capitolo 5, dove avete imparato a visualizzare un numero in esadecimale, e nel capitolo 7, dove avete visto come semplificare il programma usando una procedura. Ora inserirete un'altra procedura per visualizzare un carattere.

Usando una procedura centrale per mostrare un carattere sullo schermo, potete cambiare il modo in cui questa procedura scrive i caratteri, senza influenzare il resto del programma.

Inserite il listato seguente nel file VIDEO_IO.ASM:

Listato 9-2. Il file VIDEO_IO.ASM

```
.MODEL    SMALL
.CODE

TEST_WRITE_HEX  PROC
```

```

MOV    DL,3Fh                ;Verifica con 3Fh
CALL   WRITE_HEX
INT    20h                  ;Ritorna al DOS
TEST_WRITE_HEX ENDP

PUBLIC WRITE_HEX
;-----;
; Questa procedura converte il byte nel registro DL in esadecimale ;
; e scrive le due cifre esadecimali alla posizione corrente del ;
; cursore. ;
; Inserimento: DL      Byte da convertire in esadecimale. ;
; ;
; Usa:      WRITE_HEX_DIGIT ;
;-----;
WRITE_HEX PROC ;Punto di inserimento
PUSH   CX                ;Salva registri usati in questa procedura
PUSH   DX
MOV    DH,DL             ;Copia il byte
MOV    CX,4              ;Preleva il nibble alto in DL
SHR   DL,CL
CALL   WRITE_HEX_DIGIT  ;Visualizza prima cifra esadecimale
MOV    DL,DH             ;Preleva il nibble basso in DL
AND   DL,0Fh            ;Cancella il nibble alto

```

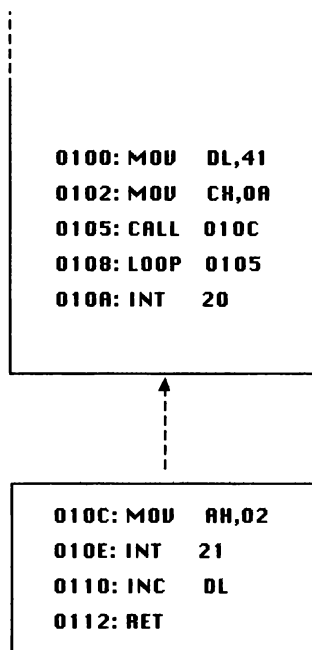


Figura 9-1. Le procedure vengono assemblate senza spazio inutilizzato


```

        CALL  WRITE_HEX_DIGIT      ;Visualizza la seconda cifra esadecimale
        POP   DX
        POP   CX
        RET
WRITE_HEX ENDP

        PUBLIC  WRITE_HEX_DIGIT
;-----;
; Questa procedura converte i 4 bit bassi di DL in una cifra          ;
; esadecimale e la scrive sullo schermo.                             ;
;                                                                       ;
; Inserimento: DL          I 4 bit inferiori contengono numero da    ;
;                                                           visualizzare in esadecimale ;
;                                                                       ;
; Usa:          WRITE_CHAR                                           ;
;-----;
WRITE_HEX_DIGIT PROC
        PUSH  DX                ;Salva i registri utilizzati
        CMP   DL,10             ;Questo nibble è <10?
        JAE  HEX_LETTER        ;No, converte in lettera
        ADD  DL,"0"            ;Sì, converte in una cifra
        JMP  Short WRITE_DIGIT ;Ora scrive questo carattere
HEX_LETTER:
        ADD  DL,"A"-10         ;Converte in lettera esadecimale
WRITE_DIGIT:
        CALL  WRITE_CHAR       ;Visualizza la lettera sullo schermo
        POP   DX               ;Ripristina vecchio valore di AX
        RET
WRITE_HEX_DIGIT ENDP

        PUBLIC  WRITE_CHAR
;-----;
; Questa procedura stampa un carattere sullo schermo usando la      ;
; chiamata di funzione del DOS.                                       ;
;                                                                       ;
; Inserimento:      DL          Byte da visualizzare sullo schermo.  ;
;-----;
WRITE_CHAR PROC
        PUSH  AX
        MOV  AH,2              ;Funzione per visualizzazione carattere
        INT  21h              ;Visualizza il carattere nel registro DL
        POP  AX               ;Ripristina il vecchio valore in AX
        RET                   ;E ritorna
WRITE_CHAR ENDP

        END    TEST_WRITE_HEX

```

Alcuni caratteri vengono trattati in modo speciale. Per esempio, usando la funzione del DOS per visualizzare il carattere con codice 07, verrà emesso un segnale acustico invece del carattere corrispondente (un piccolo diamante). Vedrete una nuova versione di `WRITE_CHAR` (che visualizzerà il diamante) nella parte III di questo libro, dove verrete anche a conoscenza delle ROM BIOS presenti nel PC. Per ora usate la funzione del DOS per visualizzare solo caratteri normali.

La nuova direttiva `PUBLIC` la userete in futuro (nel capitolo 13) quando verrete a contatto con la progettazione modulare. `PUBLIC` indica semplicemente all'assemblatore di generare ulteriori informazioni per il linker. Il linker permette di riunire differenti pezzi di programma, contenuti in file sorgenti separati, in un unico programma. `PUBLIC` indica inoltre all'assemblatore che la procedura dopo la direttiva `PUBLIC` può essere utilizzata da tutte le procedure presenti in altri file.

Al momento, il file `VIDEO_IO` contiene tre procedure che vengono utilizzate per scrivere un byte in esadecimale, e un piccolo programma principale che verifica queste procedure. Aggiungerete nuove istruzioni quando svilupperete il programma `Dskpatch` e, alla fine del libro, in `VIDEO_IO.ASM` saranno inserite molte procedure di uso generale.

La procedura `TEST_WRITE_HEX` serve per verificare la procedura `WRITE_HEX` che utilizza alternativamente `WRITE_HEX_DIGIT` e `WRITE_CHAR`. Quando avrete verificato che tutte queste procedure funzionano correttamente, potrete rimuovere `TEST_WRITE_HEX` dal file `VIDEO_IO.ASM`.

Create una versione `.COM` di `VIDEO_IO` e usate `Debug` per verificare `WRITE_HEX`. Cambiate la locazione `10h` (che contiene `3Fh`) in ciascuna delle condizioni di limite (proprio come avete fatto nel capitolo 5) e usate quindi il comando `G` per eseguire `TEST_WRITE_HEX`.

Userete molti semplici programmi di verifica per controllare le nuove procedure scritte. In questo modo, potrete costruire i programmi pezzo per pezzo, invece di cercare di creare e collaudare un programma tutto in una volta. Questo metodo risulta molto più efficace e più veloce dato che, eventuali errori, vengono confinati in uno spazio ristretto.

INTRODUZIONE ALLA PROGETTAZIONE MODULARE

Notate che davanti a ciascuna procedura nel file `VIDEO_IO` avete inserito un blocco di commenti per descrivere brevemente la funzione della procedura stessa. Molto importante è il fatto che questi commenti indicano quali registri vengono usati e quali altre procedure vengono chiamate. Ecco la prima caratteristica della progettazione modulare: il blocco di commenti permette di utilizzare qualsiasi procedura rilevandone lo scopo dalla descrizione. Non è quindi necessario ristudiare la procedura per capire come funziona. Questo permette anche di riscrivere una procedura senza dover riscrivere qualsiasi altra procedura chiamata dalla stessa.

Avete anche usato le istruzioni `POP` e `PUSH` per salvare e ripristinare i registri usati

all'interno delle procedure. Dovrete effettuare sempre questa operazione, ad eccezione delle procedure usate per la verifica. Anche questo metodo fa parte della progettazione modulare.

Ricordatevi di salvare e ripristinare i registri usati nelle procedure per evitare qualsiasi interazione o conflitto tra le varie parti di un programma. In ciascuna procedura potete usare quanti registri desiderate, *a condizione* di ripristinarli prima dell'istruzione RET. Inoltre, se i registri non vengono salvati e ripristinati, il compito di riscrivere le procedure diventa molto più complesso.

Cercate anche di usare molte procedure piccole invece di una sola molto lunga. Questo rende la programmazione più semplice e il programma più comprensibile. Tuttavia potrà capitare, in casi abbastanza complessi, di dover scrivere una procedura molto lunga; ma questi devono rimanere dei casi isolati.

Questi metodi saranno sviluppati in modo più approfondito nei capitoli successivi. Nel prossimo capitolo, per esempio, aggiungerete un'altra procedura al file VIDEO_IO che servirà per prelevare una parola dal registro DX e visualizzarla come numero decimale sullo schermo.

LO SCHELETRO DI UN PROGRAMMA

Come avete visto in questo capitolo e in quelli precedenti, l'assemblatore impone un certo numero di istruzioni di servizio per ogni programma che viene creato. In altre parole, dovete inserire alcune direttive che indicano all'assemblatore i punti fondamentali. Per riferimento futuro, riportiamo le direttive assolutamente necessarie per ciascun programma:

```
.MODEL    SMALL
.CODE

Procedura  PROC
           .
           .
           .
           INT 20h
Procedura  ENDP

END      Procedura
```

Aggiungerete qualche altra direttiva allo scheletro di questo programma nei capitoli successivi; tuttavia, potete usare queste direttive come punto di partenza per i nuovi programmi che scriverete. O, meglio ancora, potete usare alcune parti di programma e procedure contenute in questo libro, come punto di partenza.

SOMMARIO

State veramente facendo dei progressi. In questo capitolo avete imparato a scrivere delle procedure in linguaggio assembler. Da questo momento in avanti userete sempre le procedure e, usando procedure corte, costruirete dei programmi più gestibili. Avete visto che una procedura inizia con la definizione PROC e finisce con la direttiva ENDP. Avete riscritto PRINT_AJ per verificare la vostra conoscenza delle procedure e avete quindi riscritto il programma per visualizzare un numero esadecimale. Ora che avete familiarizzato con le procedure, non c'è ragione per non dividere un programma in più parti. Avete infatti visto che ci sono molte ragioni a favore dell'utilizzo di piccole procedure.

Alla fine di questo capitolo siete venuti a contatto con la progettazione modulare, una filosofia che permette di risparmiare tempo e fatica. I programmi modulari saranno più facili da scrivere, da leggere e da modificare rispetto ai programmi scritti con la vecchia tecnica che prevedeva procedure molto lunghe.

Siete ora pronti per costruire un'altra procedura utile. Quindi, nel capitolo 11, imparerete più approfonditamente i segmenti. Superato il capitolo 11, inizierete a costruire programmi più estesi utilizzando le tecniche della progettazione modulare.

VISUALIZZAZIONE IN DECIMALE

Scriverete ora una procedura per prelevare una parola e visualizzarla in notazione decimale. In `WRITE_DECIMAL` potete trovare alcuni nuovi accorgimenti che permettono di risparmiare qualche byte e qualche secondo. Forse questi accorgimenti potranno sembrare poco proporzionati allo sforzo richiesto per apprenderli ma, se li memorizzate, potrete usarli per velocizzare un programma e renderlo di dimensioni più contenute. Attraverso questi accorgimenti, imparerete due nuovi tipi di operazioni logiche che si aggiungeranno a `AND` (che avete visto nel capitolo 5). Innanzitutto, ricordiamo il processo necessario per convertire una parola in cifre decimali.

RITORNO ALLA CONVERSIONE

La divisione è la chiave per convertire una parola in cifre decimali. Se vi ricordate, l'istruzione `DIV` calcola sia il risultato che il resto di una divisione. Quindi, calcolando $12345/10$ si ottiene 1234 con resto 5. In questo esempio, il 5 è la cifra più a destra.

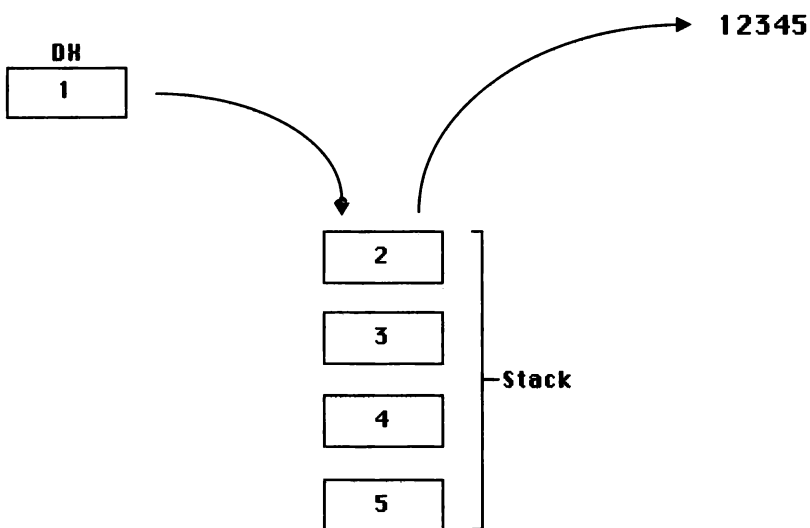


Figura 10-1. L'uso dell'istruzione `PUSH` causa l'inversione del loro ordine

Se dividete ancora per 10, ottenete, come resto, la cifra adiacente a sinistra. Ripetendo la divisione per 10, potete isolare ogni cifra del numero considerandola come resto. Naturalmente le cifre arrivano in ordine inverso, ma con il linguaggio assembly questo non è un problema. Vi ricordate dello stack? Questo è come una pila di vassoi: il primo ad essere preso è l'ultimo inserito sulla pila. Se sostituite le cifre ai vassoi e ponete le cifre una sull'altra (come risultano dal resto), le cifre saranno nell'ordine corretto.

La cifra all'inizio dello stack sarà la prima cifra del numero, e le cifre sottostanti saranno le rimanenti. Quindi, per poter isolare le cifre di un numero e averle nell'ordine corretto, è sufficiente inserire il resto delle varie divisioni nello stack.

Il programma seguente è la procedura completa per visualizzare un numero in notazione decimale. Come già detto, ci sono alcuni accorgimenti nascosti in questa procedura che vi saranno spiegati presto. Per ora provate WRITE_DECIMAL per vedere come funziona.

Inserite WRITE_DECIMAL nel file VIDEO_IO.ASM insieme alle procedure usate per visualizzare un byte in esadecimale. Assicuratevi di inserire WRITE_DECIMAL *dopo* TEST_WRITE_HEX che sostituirete con TEST_WRITE_DECIMAL. Per accorciare il lavoro, WRITE_DECIMAL usa la procedura WRITE_HEX_DIGIT per convertire un nibble (4 bit) in una cifra.

Listato 10-1. Aggiunta al file VIDEO_IO.ASM

```

PUBLIC WRITE_DECIMAL
;-----;
; Questa procedura serve per scrivere un numero a 16 bit senza segno
; in notazione decimale.
; Inserimento:    DX        N : numero senza segno a 16-bit.
;
; Usa:           WRITE_HEX_DIGIT
;-----;
WRITE_DECIMAL PROC NEAR
    PUSH AX                ;Salva i registri utilizzati
    PUSH CX
    PUSH DX
    PUSH SI
    MOV AX,DX
    MOV SI,10              ;Dividerà per 10 usando SI
    XOR CX,CX              ;Conta le cifre inserite nello stack
NON_ZERO:
    XOR DX,DX              ;Imposta la parola superiore di N a 0
    DIV SI                  ;Calcola N/10 e (N mod 10)
    PUSH DX                ;Inserisce una cifra nello stack
    INC CX                  ;Aggiunge un'altra cifra
    OR AX,AX                ;Ancora N = 0?
    JNE NON_ZERO           ;No, continua
WRITE_DIGIT_LOOP:
    POP DX                  ;Preleva le cifre in ordine inverso
    CALL WRITE_HEX_DIGIT
    LOOP WRITE_DIGIT_LOOP

```

```

END_DECIMAL:
        POP     SI
        POP     DX
        POP     CX
        POP     AX
        RET
WRITE_DECIMAL ENDP

```

Notate che è stato incluso un nuovo registro, SI (*Source Index*, Indice Sorgente). Successivamente incontrerete anche il “fratello” di questo registro, DI (*Destination Index*, Indice di Destinazione). Entrambi i registri hanno un uso speciale, ma possono essere usati anche come registri di uso generale. Dato che WRITE_DECIMAL deve utilizzare quattro registri di uso generale, è stato usato SI (anche se si sarebbe potuto usare BX) semplicemente per mostrare che SI (e DI) possono essere usati anche come registri di uso generale.

Prima di provare questa nuova procedura, dovete effettuare due ulteriori modifiche a VIDEO_IO.ASM. Innanzitutto dovete rimuovere la procedura TEST_WRITE_HEX e inserire al suo posto la procedura seguente:

Listato 10-2. Sostituire TEST_WRITE_HEX in VIDEO_IO.ASM con la procedura seguente:

```

TEST_WRITE_DECIMAL      PROC
        MOV     DX,12345
        CALL   WRITE_DECIMAL
        INT    20h                ;Ritorna al DOS
TEST_WRITE_DECIMAL      ENDP

```

Questa procedura verifica WRITE_DECIMAL con il numero 12345 (che l’assemblatore converte nella parola 3039h).

In secondo luogo, dovete cambiare la direttiva END alla fine del file VIDEO_IO.ASM in END TEST_WRITE_DECIMAL, dato che TEST_WRITE_DECIMAL è ora la procedura principale.

Effettuate queste modifiche e provate VIDEO_IO. Convertitelo in un file .COM e guardate come funziona. Se non funzionasse, controllate il file sorgente per eventuali errori (e consultate l’appendice C del libro). Se ve la sentite, provate a trovare l’errore usando Debug. Dopo tutto, Debug serve proprio a questo scopo.

ALCUNI ACCORGIMENTI

Nascosti nella procedura WRITE_DECIMAL ci sono due piccoli “trucchi” presi in prestito dai programmatori delle procedure del BIOS (incontrerete il BIOS nel capitolo 17). Il primo accorgimento è un’istruzione per impostare un registro a zero. Questa non è molto più efficace di MOV AX,0 e forse non vale lo sforzo necessario per impararla, ma è il tipico stratagemma che viene normalmente utilizzato ed è quindi bene conoscerlo. L’istruzione:

```
XOR     AX,AX
```

Imposta il registro AX a zero. Per capire come funziona, dovete imparare un'altra istruzione logica chiamata *OREscusivo* (in inglese Exclusive OR da cui deriva appunto XOR). Un OR esclusivo è simile a OR (che vedrete successivamente), ma il risultato della relazione è differente:

XOR	0	1
0	0	1
1	1	0

Il risultato è vero *solamente* se un bit è vero e non entrambi. Quindi, se si effettua un OR esclusivo tra un numero e se stesso, si ottiene zero:

	1	0	1	1	0	1	0	1
XOR	1	0	1	1	0	1	0	1
	0	0	0	0	0	0	0	0

Questo è il trucco. Non troverete altri usi di XOR in questo libro, ma abbiamo pensato che l'avreste potuto trovare interessante.

Un altro metodo molto utilizzato per impostare un registro a zero, è il seguente:

```
SUB    AX,AX
```

Vediamo ora l'altro accorgimento. Anche questo è abbastanza contorto e utilizza un "cugino" dell'istruzione XOR: la funzione OR.

Per verificare che il registro AX sia zero, potreste usare l'istruzione `CMP AX,0`. Ma vediamo invece questo nuovo metodo che risulta più divertente e un pochino più efficace. Scrivete `OR AX,AX` a fate seguire a questa istruzione il salto condizionale `JNE` (Jump if Not Equal, Salta se Non è Uguale). (Avreste anche potuto usare `JNZ` - Jump if Not Zero, Salta se Non è Zero).

L'istruzione OR, come qualsiasi istruzione matematica, imposta i flag, incluso il flag zero. Come AND, anche OR è un concetto logico ma, in questo caso, si ha un risultato vero se uno o l'altro bit è vero:

OR	0	1
0	0	1
1	1	0

Se prendete un numero ed effettuate un OR su se stesso, ottenete nuovamente lo stesso numero:

	1	0	1	1	0	1	0	1
OR	1	0	1	1	0	1	0	1
	1	0	1	1	0	1	0	1

L'istruzione OR risulta utile anche per impostare un bit in un byte. Per esempio, è possibile impostare a 1 il terzo bit del numero appena usato:

```

      1 0 1 1   0 1 0 1
OR    0 0 0 0   1 0 0 0
-----
      1 0 1 1   1 1 0 1

```

Nel resto del libro troverete altri usi dell'istruzione OR.

IL FUNZIONAMENTO INTERNO

Per vedere come funziona la procedura `WRITE_DECIMAL`, studiate il listato. Vediamo ora di mettere a fuoco alcune particolarità.

Innanzitutto il registro CX viene usato per contare il numero di cifre inserite nello stack (in questo modo è possibile sapere quante se ne devono prelevare). Il registro CX è una scelta particolarmente conveniente dato che è possibile costruire un ciclo con l'istruzione LOOP e usare il registro CX come contatore. Questa scelta rende il ciclo `WRITE_DIGIT_LOOP` quasi superfluo, dato che l'istruzione LOOP usa direttamente il registro CX. Userete molto spesso CX per memorizzare un contatore.

Ora verificate attentamente le condizioni di limite. La condizione 0 non è un problema, come potete facilmente verificare. L'altra condizione di limite è 65535 (FFFFh) che potete verificare con Debug. Caricate `VIDEO_IO.COM` in Debug digitando `DEBUG VIDEO_IO.COM` e modificate il numero 12345 (3039h) alla locazione 101h in 65535 (FFFFh). (`WRITE_DECIMAL` funziona con numeri senza segno. Provate a scrivere una versione che utilizzi i numeri con segno). Potreste aver notato un problema, che però è causato dall'8088 e non dal programma. Debug lavora soprattutto con i byte, mentre voi dovete modificare una parola. Dovete stare attenti dato che l'8088 memorizza i byte in ordine inverso. Vediamo un'istruzione MOV disassemblata:

```
3985:0100      BA3930          MOV     DX,3039
```

Potete vedere dall'istruzione `BA3930` che il byte in 101h è 39h mentre quello in 102h è 30h (BA è l'istruzione MOV). I due byte sono i due byte di 3039h ma sembrano essere in ordine inverso. Anche se potrebbe sembrare strano, questo ordine è logico come vedrete dalla seguente spiegazione.

Una parola è composta da due byte, il byte basso e il byte alto. Il byte basso è il byte meno significativo (39h in 3039h) mentre quello alto è l'altra parte (30h). Ha quindi senso inserire il byte basso nell'indirizzo inferiore della memoria.

(Alcuni microprocessori, come il Motorola 68000 nel Macintosh della Apple, mantengono questi byte nella posizione originale; questo potrebbe creare un po' di confusione se scrivete programmi su diversi tipi di computer).

Provate numeri differenti per la parola che inizia in 101h, e guardate come vengono memorizzati.

MOV DH,3039

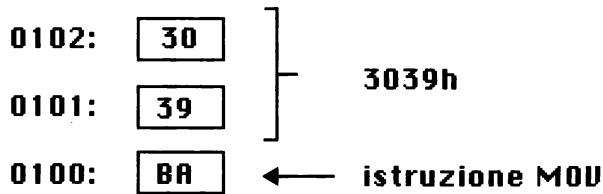


Figura 10-2. L'8088 memorizza i numeri inserendo prima il byte basso

SOMMARIO

Avete imparato alcune nuove istruzioni e qualche piccolo trucco. Avete conosciuto altri due registri, SI e DI, che possono essere usati come registri di uso generale. Questi hanno anche altri usi che vedrete nei capitoli successivi.

Avete imparato le istruzioni logiche XOR e OR che permettono di lavorare con i singoli bit di una parola, e le avete usate nella procedura WRITE_DECIMAL rispettivamente per azzerare e confrontare il registro AX.

Avete imparato infine il metodo utilizzato dall'8088 per memorizzare i numeri e avete creato una nuova procedura, WRITE_DECIMAL, che utilizzerete nei programmi futuri.

A questo punto prendete fiato. Avete davanti alcuni capitoli che trattano argomenti differenti. Il capitolo 11 spiega dettagliatamente i segmenti che sono forse la parte più complicata del microprocessore 8088. Assicuratevi di avere tutto chiaro prima di avventurarvi in questo capitolo che si preannuncia come il più difficile.

In seguito ritornerete allo scopo principale di questo libro, quello di creare il programma Dskpatch. Vi saranno fornite alcune nozioni sui dischi, sulle tracce, sui settori e su argomenti simili.

Inizierete quindi a scrivere una prima, rudimentale versione di Dskpatch. Durante questa costruzione avrete la possibilità di vedere come sviluppare programmi estesi. I programmatori non scrivono un programma intero prima di collaudarlo, ma scrivono delle sezioni del programma collaudandole di volta in volta. Anche voi, in piccolo, avete usato questo metodo scrivendo e controllando le procedure WRITE_HEX e WRITE_DECIMAL. I programmi di verifica diventeranno, da questo momento in avanti, più complessi ma anche più interessanti.

I SEGMENTI

Nei capitoli precedenti avete incontrato alcune direttive che si riferivano ai segmenti. E' arrivato il momento di guardare i segmenti da vicino e di vedere come l'8088 gestisce gli indirizzi fino a un megabyte (1.048.576 byte) di memoria. D'ora in avanti, capirete perché è necessario usare delle direttive nell'assemblatore e nei capitoli successivi inizierete a usare segmenti differenti (fino a questo momento ne avete usato solamente uno).

Iniziamo a vedere come l'8088 costruisce gli indirizzi a 20 bit necessari per gestire un megabyte di memoria.

DIVIDERE LA MEMORIA DELL'8088

I segmenti sono l'unica parte dell'8088 che non abbiamo ancora trattato e sono forse la parte più difficile da capire di questo microprocessore. I segmenti, infatti, vengono chiamati *kludge*, parola che, in gergo informatico, significa soluzione improvvisata di un problema. (Il microprocessore 80386 offre altri modi di indirizzamento che sono molto più semplici e non usa i segmenti. Sfortunatamente non esiste un sistema operativo per l'8088 che utilizzi questi modi di indirizzamento *lineari*. OS/2, che funziona solamente con microprocessori 80286 e 80386 usa un tipo leggermente differente di segmento per poter indirizzare più di un megabyte di memoria).

Il problema, in questo caso, è riuscire a indirizzare più di 64K di memoria (limite imposto da una parola dato che il numero 65535 è appunto il numero più grosso che questa può contenere). I programmatori dell'8088 hanno usato i segmenti e i registri di segmento per risolvere questo problema, rendendo però l'8088 più complicato.

Finora non si era mai posto questo problema. Avete usato il registro IP per memorizzare l'indirizzo dell'istruzione successiva e vi è stato detto che l'indirizzo viene formato anche tramite il registro CS, ma non avete visto come. E' giunto il momento di vederlo. Benché l'indirizzo completo sia formato da due registri, l'8088 non crea un numero composto da due parole per un indirizzo. Unendo i registri CS e IP si otterrebbe un numero a 32 bit (16 bit per numero) e l'8088 sarebbe in grado di indirizzare fino a quattro miliardi di byte (molto di più del milione di byte che può effettivamente indirizzare). Il metodo usato dall'8088 è un po' più complicato. Il registro CS fornisce l'indirizzo di *partenza* del segmento, dove un segmento equivale a 64K di memoria. Ecco come funziona.

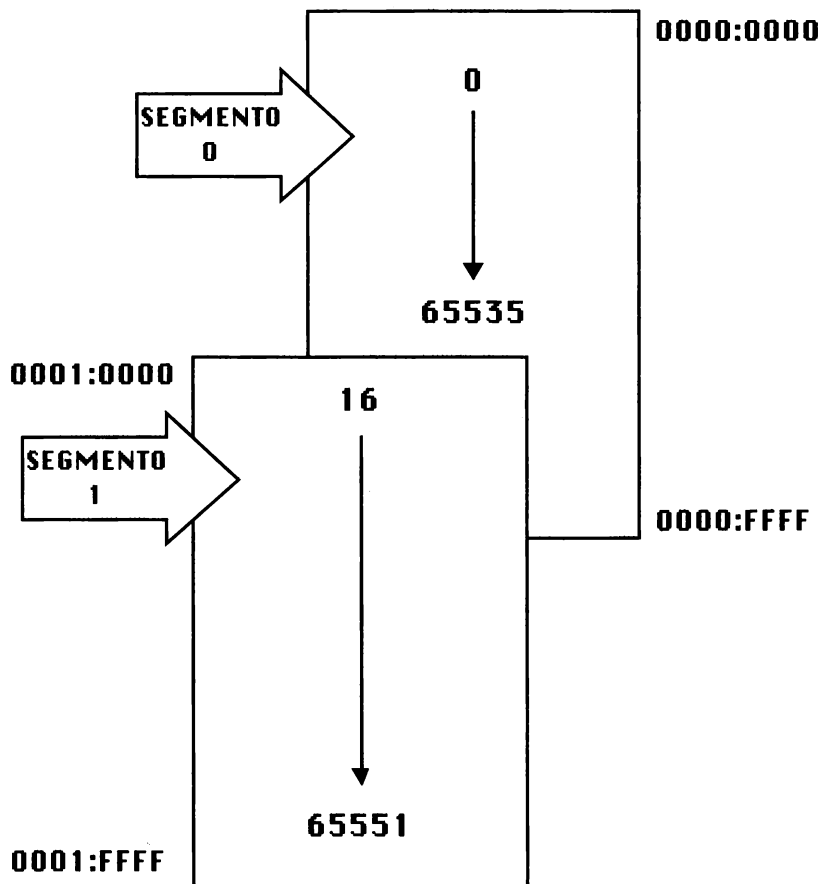


Figura 11-1. I segmenti iniziano ogni 16 byte e sono lunghi 65536 byte

Come potete vedere dalla figura 11-1, l'8088 divide la memoria in molti segmenti sovrapposti, con un nuovo segmento che inizia ogni 16 byte.

Il primo segmento (segmento 0) inizia alla locazione di memoria 0; il secondo segmento (segmento 1) inizia in 10h (16); il terzo inizia in 20h (32) e così via.

L'indirizzo corrente si ottiene moltiplicando CS per 16 e sommando il registro IP ($CS \cdot 16 + IP$). Per esempio, se il registro CS contiene 3FA8 e IP contiene D017, l'indirizzo assoluto sarà:

$$\begin{array}{r}
 CS * 16 : 0011\ 1111\ 1010\ 1000\ 0000 \\
 + IP : \quad\quad\quad 1101\ 0000\ 0001\ 0111 \\
 \hline
 \quad\quad\quad 0100\ 1100\ 1010\ 1001\ 0111
 \end{array}$$

La moltiplicazione per 16 è stata effettuata spostando CS di quattro bit a sinistra e aggiungendo degli zeri a destra.

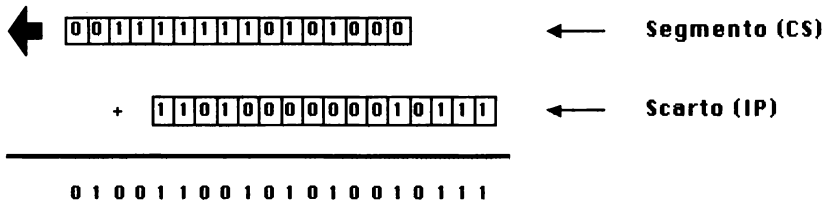


Figura 11-2. L'indirizzo assoluto di CS:IP è $CS * 16 + IP$

Questo vi potrà sembrare un modo strano per indirizzare più di 64K di memoria e, in effetti, lo è (ma funziona). Presto vedrete più dettagliatamente come funziona in realtà.

L'8088 offre quattro registri di segmento: CS (Code Segment, Segmento Codice), DS (Data Segment, Segmento Dati), SS (Stack Segment, Segmento di Stack) e ES (Extra Segment, Segmento Extra). Avete visto che il registro CS contiene il segmento in cui è memorizzata l'istruzione successiva. Analogamente, DS è il registro in cui l'8088 cerca i dati e SS il registro in cui inserisce lo stack.

Prima di proseguire, studiate questo piccolo programma che è abbastanza diverso da quelli visti finora; utilizza infatti due segmenti. Inserite questo programma nel file TEST_SEG.ASM:

Listato 11-1. Il Programma TEST_SEG.ASM

```
DOSSEG
.MODEL    SMALL

.STACK                                ;Assegna 1K di stack

.CODE

TEST_SEGMENT    PROC
    MOV    AH, 4Ch                ;Richiede la funzione per uscire al DOS
    INT    21h                    ;Torna al DOS
TEST_SEGMENT    ENDP

END    TEST_SEGMENT
```

Assemblate questo programma e usate il LINK ma non convertitelo in una versione .COM. Il risultato sarà TEST_SEG.EXE che è leggermente diverso da un file .COM.

Nota: Per ritornare al DOS da un file .EXE avete usato un altro metodo (non il solito INT 20h). INT 20h funziona perfettamente con i file .COM, ma non funziona con i file .EXE a causa della diversa organizzazione dei segmenti (che vedrete in questo capitolo). D'ora in avanti userete INT 21h con funzione 4Ch per uscire dai programmi.

Quando usate Debug con un file .COM, Debug imposta tutti i registri di segmento sullo stesso numero, con l'inizio del programma impostato a una distanza di 100h dall'inizio del segmento. I primi 256 byte (100h) vengono usati per memorizzare alcune informazioni a cui voi non siete interessati, ma a cui daremo ugualmente una breve occhiata.

Provate ora a caricare TEST_SEG.EXE in Debug per vedere cosa succede con i segmenti in un file .EXE:

```
A>DEBUG TEST_SEG.EXE
```

```
-R
```

```
AX=0000 BX=0000 CX=0004 DX=0000 SP=0400 BP=0000 SI=0000 DI=0000  
DS=3985 ES=3985 SS=3996 CS=3995 IP=0000 NV UP DI PL NZ NA PO NC  
3995:0000 B44C          MOV     AH,4C
```

I valori nei registri SS e CS sono diversi da quelli in DS e ES.

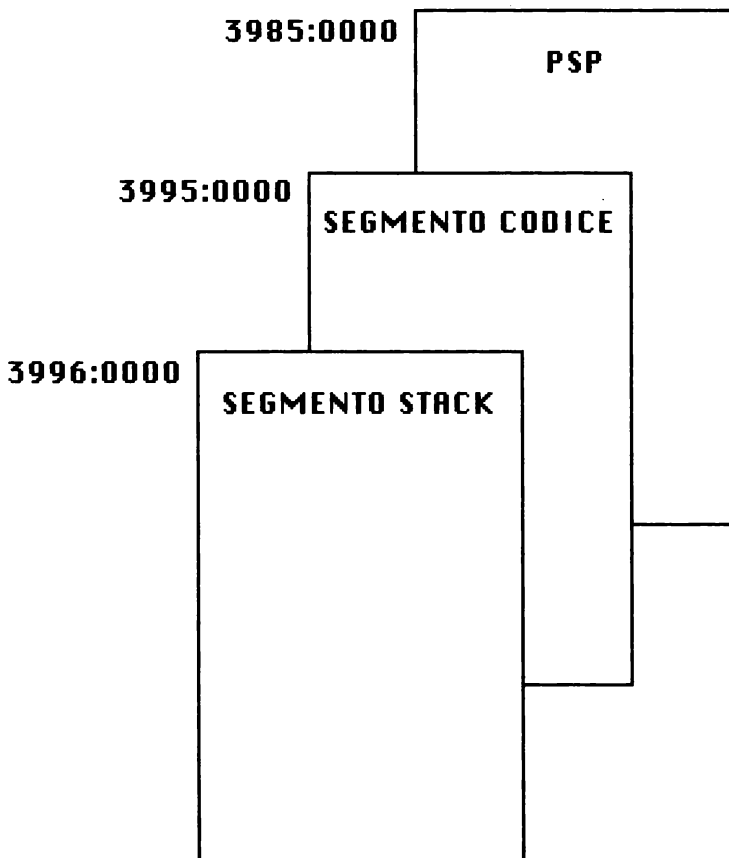


Figura 11-3. Disposizione della memoria con il file TEST_SEG.EXE

LO STACK

In questo programma sono stati definiti due segmenti. Il segmento STACK in cui viene inserito lo stack (da cui .STACK), e il segmento codice (correntemente chiamato _TEXT) in cui sono memorizzate le istruzioni. La direttiva .STACK indica all'assemblatore di creare uno stack di 1024 byte. (E' possibile creare uno stack più grosso o più piccolo inserendo semplicemente un numero dopo .STACK. Per esempio, .STACK 128 crea uno stack di 128 byte).

L'indirizzo della cima dello stack viene fornito dai registri SS:SP, dove SP è il puntatore di stack (come IP e CS) e fornisce la distanza dal segmento di stack corrente.

Dobbiamo dire che "cima dello stack" è un termine improprio dato che lo stack cresce dall'alto verso il basso. Quindi, la *cima* dello stack in realtà si trova in fondo allo stack stesso e i dati inviati vengono inseriti sempre più in basso (in memoria). In questo caso, SP contiene 400h (1024 in decimale) dato che è stata definita un'area di stack di 1024 byte. Poiché non sono stati inseriti dei dati nello stack, la cima dello stack si trova ancora nella parte più alta della memoria: 400h.

Nei programmi .COM sviluppati dei capitoli precedenti, non avete mai definito un segmento di stack. Perché non bisogna definire lo stack con i programmi .COM? E dove si trova lo stack nei programmi .COM? Tutti i programmi .COM che avete creato utilizzavano un solo segmento e tutti i registri di segmento (CS, DS, ES e SS) puntavano a questo segmento. Dato che avevate un solo segmento, non è stato necessario impostare un segmento di stack separato.

Per quando riguarda la seconda domanda, se provate a guardare i registri del programma SCRIVE.COM, vedrete che lo stack si trova alla fine del segmento (SP=FFEE):

```
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3995 ES=3995 SS=3995 CS=3995 IP=0100 NV UP EI PL NZ NA PO NC
3995:0100 B402          MOV     AH,02
- .
```

Il DOS imposta sempre il puntatore di stack alla fine del segmento quando carica un file .COM in memoria. Per questa ragione, non è necessario dichiarare un segmento di stack (.STACK) per i programmi .COM.

Cosa succederebbe se si trogliesse la direttiva .STACK dal file TEST_SEG.ASM?

```
A>DEBUG TEST_SEG.EXE
-R
AX=0000 BX=0000 CX=0004 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=3985 ES=3985 SS=3995 CS=3995 IP=0000 NV UP EI PL NZ NA PO NC
3995:0000 B44C          MOV     AH,4C
-
```

Lo stack si trova ora in 3995:0, che corrisponde all'inizio del programma (CS:0). Questo non va assolutamente bene. Lo stack non deve trovarsi mai vicino al codice

del programma. Inoltre, dato che il puntatore di stack si trova in SS:0, non ha spazio per crescere (dato che lo stack cresce verso il basso). Per questi motivi, *dovete* definire un segmento di stack per i programmi .EXE.

Nota: dovete sempre definire un segmento di stack con la direttiva .STACK nei programmi .EXE.

Tornando all'esempio dei due segmenti, notate che il segmento di stack (SS) è il segmento numero 3996 (potrebbe esse differente dal vostro) mentre il segmento del codice (CS) è il numero 3995 (uno in meno di SS o 16 byte più basso in memoria). Dato che non avete inserito alcun dato nel segmento di stack, se disassemblate partendo da CS:0 verrà visualizzato il programma (MOV AH,4C e INT 21) seguito dal contenuto corrente della memoria:

```
-U CS:0
3995:0000    B44C          MOV    AH, 4C
3995:0002    CD21          INT    21
3995:0004    65           DB     65
3995:0005    2028          AND    [BX+SI], CH
3995:0007    59           POP    CX
3995:0008    2F           DAS
3995:0009    4E           DEC    SI
3995:000A    293F          SUB    [BX], DI
.
.
.
```

IL PROGRAM SEGMENT PREFIX (PSP)

Nella visualizzazione dei registri, notate che ES e DS contengono 3985h, 10 in meno dell'inizio del programma nel segmento 3995h. Moltiplicando per 16 per ottenere il numero di byte, potete vedere che ci sono 100h (256) byte prima dell'inizio del programma. Quest'area è la stessa che viene inserita all'inizio di un file .COM.

Nota: Quest'area viene chiamata PSP (*Program Segment Prefix*, Prefisso del Segmento di Stack) e contiene delle informazioni usate dal DOS. In altre parole, non dovete pensare di poter utilizzare quest'area.

Tra le altre cose, quest'area di 256 byte (PSP) all'inizio dei programmi contiene i caratteri digitati dopo il nome del programma. Per esempio:


```

A>DEBUG TEST_SEG.EXE Ecco alcuni caratteri che appaiono nella memoria
-D DS:80
3985:0080 31 20 45 63 63 6F 20 61-6C 63 75 6E 69 20 63 61 1 Ecco alcuni ca
3985:0090 72 61 74 74 65 72 69 20-63 68 65 20 61 70 70 61 ratteri che appa
3985:00A0 69 6F 6E 6F 20 6E 65 6C-6C 61 20 6D 65 6D 6F 72 iono nella memor
3985:00B0 69 61 0D 65 6C 6C 61 20-6D 65 6D 6F 72 69 61 0D ia.ella memoria.
3985:00C0 02 35 06 35 5C 26 8C 35-96 3A 78 36 14 33 92 35 .5.5\&.5.:x6.3.5

```

Il primo byte indica quanti caratteri sono stati digitati (31h o 49), incluso il primo spazio dopo TEST_SEG.EXE. Non userete queste informazioni in questo libro, ma questo aiuta a far capire perché potrebbe essere necessario un PSP così grosso.

Il PSP contiene anche delle informazioni che il DOS utilizza per uscire da un programma, tramite le istruzioni INT 20h o INT 21h con funzione 4Ch. Ma per ragioni che non sono del tutto chiare, l'istruzione INT 20h si aspetta che il registro CS punti l'inizio del PSP (cosa che viene fatta in un programma .COM, ma *non* in un programma .EXE). Questo è un problema storico. Per questo motivo, infatti, è stata aggiunta la funzione 4Ch (con l'introduzione del DOS versione 2.0) che permette di uscire più facilmente da un programma .EXE tramite l'istruzione INT 21h. Da questo momento in avanti userete sempre INT 21h con funzione 4Ch per uscire da un programma.

Il codice per i file .COM deve sempre iniziare con uno scarto di 100h nel segmento codice in modo da lasciare spazio per l'area di 256 byte (PSP) all'inizio. Questo risulta diverso nei file .EXE in cui il registro IP è impostato su 0000 dato che il segmento codice inizia 100h byte dopo l'inizio dell'area di memoria.

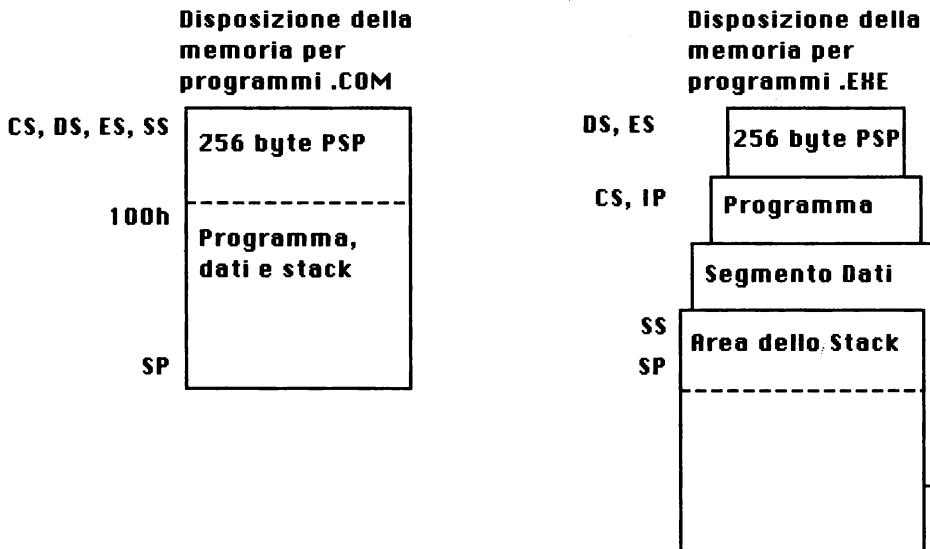


Figura 11-4. Programmi .COM e .EXE

Inizialmente i programmi venivano scritti come .COM perché erano più semplici da realizzare. Al giorno d'oggi, invece, la maggior parte dei programmi viene scritta come .EXE. Nel resto del libro, quindi, lavorerete principalmente con file .EXE.

LA DIRETTIVA DOSSEG

Se guardate nuovamente TESTSEG.EXE, noterete che il segmento di stack è in una parte di memoria più alta rispetto al segmento codice. Nel file sorgente avete definito lo stack (.STACK) *prima* di qualsiasi codice (.CODE). E' questo il motivo per cui lo stack si trova in una zona più alta di memoria?

La direttiva DOSSEG all'inizio del programma indica all'assemblatore che i segmenti del programma devono essere caricati in un ordine specifico, con il codice segmento per primo e il codice di stack per ultimo. Nel capitolo 14 vedrete più dettagliatamente la direttiva DOSSEG e l'ordine dei segmenti (nel momento in cui aggiungerete un altro segmento per memorizzare dei dati).

CHIAMATE NEAR E FAR

Il resto delle informazioni presentate in questo capitolo sono solo per interesse personale, dato che non verranno utilizzate nel libro. Potete saltare le due sezioni successive e leggerle successivamente se ne avrete bisogno.

Osservate più attentamente le istruzioni CALL usate nei capitoli precedenti. In particolare, osservate il breve programma scritto nel capitolo 7, dove avete incontrato per la prima volta l'istruzione CALL.

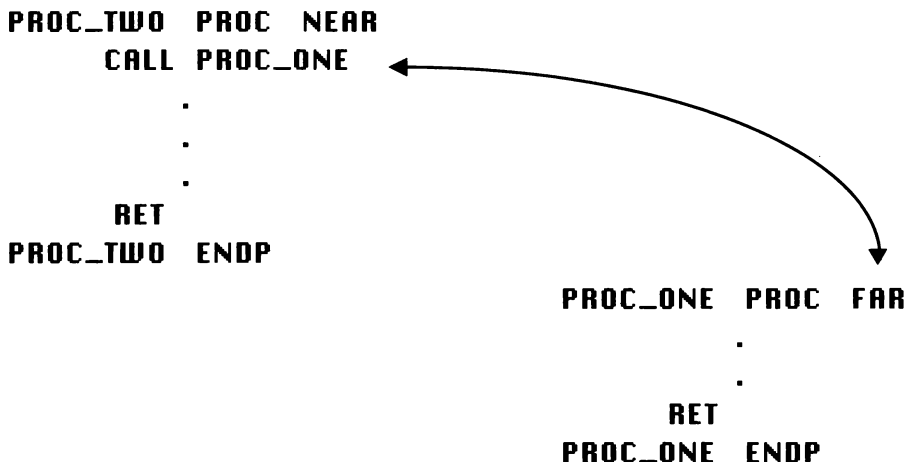


Figura 11-5. L'assemblatore genera una chiamata FAR

Allora avete scritto un programma molto breve che assomigliava al seguente (senza la procedura in 200h):

```

3985:0100    B241          MOV    DL, 41
3985:0102    B90A00       MOV    CX, 000A
3985:0105    E8F800       CALL  0200
3985:0108    E2FB        LOOP  0105
3985:010A    CD20        INT   20

```

Potete vedere osservando il codice macchina sulla sinistra che l'istruzione CALL occupa solo tre byte (E8F800). Il primo byte (E8h) è l'istruzione CALL, mentre gli altri due byte rappresentano una distanza. L'8088 calcola l'indirizzo della chiamata sommando il valore 00F8h (ricordatevi che l'8088 memorizza prima il byte basso e dopo il byte alto) all'indirizzo dell'istruzione successiva (108h nel programma). In questo caso si ottiene F8h+108h=200h. Proprio quello che ci aspettavamo!

Il fatto che questa istruzione utilizzi una singola parola come distanza significa che le "chiamate" sono limitate a un singolo segmento (lungo 64K). Come è possibile, quindi, scrivere un programma più grosso di 64K? Questo può essere fatto con delle chiamate FAR (lontane) invece che con chiamate NEAR (vicine).

Le chiamate NEAR, come avete visto, sono limitate a un singolo segmento. In altre parole, queste modificano il registro IP senza cambiare il registro CS. Per questa ragione, queste chiamate sono anche conosciute (in inglese) come *intra-segment CALLs*.

E' comunque possibile utilizzare delle chiamate FAR per cambiare entrambi i registri: CD e IP. Accanto a queste due versioni dell'istruzione CALL, ci sono anche due versioni dell'istruzione RET.

La chiamata NEAR, vista nel capitolo 7, inserisce una singola parola in cima allo stack (come indirizzo di ritorno), e l'istruzione RET corrispondente preleva questo indirizzo dallo stack e lo pone nel registro IP.

Nel caso di una chiamata FAR, una parola non è sufficiente dato che si sta lavorando su un altro segmento. In altre parole, è necessario salvare un indirizzo di ritorno composto da due parole nello stack: una parola per il puntatore dell'istruzione (IP) e un'altra per il segmento codice (CS). L'istruzione di ritorno FAR, quindi, dovrà prelevare due parole dallo stack inserendole rispettivamente nei registri CS e IP.

Come si può dire all'assemblatore quando usare le chiamate FAR e quando quelle NEAR? La risposta è semplice: è sufficiente inserire una direttiva FAR o NEAR dopo la direttiva PROC.

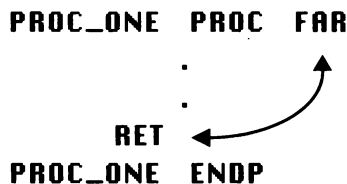


Figura 11-6. L'assemblatore genera un ritorno FAR

Come esempio, osservate il programma seguente:

```

PROC_ONE      PROC      FAR
      .
      .
      .
      RET
PROC_ONE      ENDP

PROC_TWO      PROC      NEAR
      CALL    PROC_ONE
      .
      .
      .
      RET
PROC_TWO      ENDP

```

Quando l'assemblatore trova l'istruzione `CALL PROC_ONE`, cerca la definizione di `PROC_ONE` che, in questo caso, corrisponde a `PROC_ONE PROC FAR`. Questa definizione indica quindi all'assemblatore se si tratta di una procedura `NEAR` o `FAR`. Nel caso di una procedura `NEAR`, l'assemblatore genera una chiamata `NEAR`. Per contro, verrà generata una chiamata `FAR` nel caso sia stata definita una procedura `FAR`. In altre parole, l'assemblatore usa la definizione della procedura che si sta *chiamando* per determinare il tipo di chiamata necessario.

Per quanto riguarda l'istruzione `RET`, l'assemblatore controlla la definizione della procedura che contiene `RET`. Nel programma dell'esempio, l'istruzione `RET` per la procedura `PROC_ONE` sarà di tipo `FAR`, dato che `PROC_ONE` è dichiarata come procedura `FAR`. Analogamente, `RET` in `PROC_TWO` sarà di tipo `NEAR`.

Che cosa succede se non si inserisce una direttiva `NEAR` o `FAR` dopo `PROC`? Se non viene specificata alcuna direttiva per le procedure, l'assemblatore utilizza le informazioni fornite dalla direttiva `.MODEL`. Avete sempre usato la direttiva `.MODEL SMALL` che indica all'assemblatore che viene utilizzato solo un segmento (e quindi tutte le procedure saranno di tipo `NEAR`). Ci sono altri tipi di direttive `.MODEL` (come, per esempio, `MEDIUM`) che indicano all'assemblatore di creare delle procedure `FAR` se non vengono esplicitamente dichiarate come `NEAR`.

ALTRE INFORMAZIONI SULL'ISTRUZIONE INT

L'istruzione `INT` è molto simile all'istruzione `CALL`, ma con una piccola differenza. Il nome *INT* deriva dalla parola *interrupt*. Un interrupt è un segnale esterno che indica all'8088 di eseguire una procedura e di ritornare quindi a ciò che stava facendo prima di ricevere l'interrupt. Un'istruzione `INT` non interrompere l'8088, ma viene trattata come se lo facesse.

Quando l'8088 riceve un interrupt, deve memorizzare più informazioni nello stack del semplice indirizzo di ritorno. Devono essere salvati i valori dei flag di stato (il flag di

riporto, il flag zero e così via). Questi valori vengono memorizzati in una parola conosciuta come Registro Flag, e l'8088 inserisce questo registro nello stack prima dell'indirizzo di ritorno. Questo è il motivo per cui bisogna salvare i flag di stato.

Il computer risponde regolarmente a differenti interrupt. Per esempio, l'8088 riceve un interrupt dalla clock 18,2 volte al secondo. Ognuno di questi interrupt fa in modo che l'8088 interrompa momentaneamente quello che stava facendo e esegua una procedura per contare gli impulsi della clock.

Immaginate ora un interrupt tra queste due istruzioni di programma:

```
CMP    AH, 2
JNE    NOT_2
```

Assumete che AH sia uguale a 2; in questo caso il flag zero sarà impostato dopo l'istruzione CMP (questo significa anche che l'istruzione JNE non salterà a NOT_2). Immaginate ora che la clock interrompa l'8088 tra queste due istruzioni; questo costringerà l'8088 a eseguire la procedura di interrupt senza controllare il flag zero (con l'istruzione JNE). Se l'8088 non salvasse e ripristinasse i flag di stato, l'istruzione JNE userebbe i flag impostati dalla procedura di interrupt, e *non* quelli impostati da CMP. Per evitare un possibile "disastro", l'8088 salva *sempre* (e ripristina) il registro di stato. Un interrupt salva i flag, e un'istruzione IRET (*Interrupt Return*, Ritorno da Interrupt) li ripristina alla fine della procedura.

Queste considerazioni valgono anche per l'istruzione INT. Quindi, dopo aver eseguito l'istruzione:

```
INT    21
```

lo stack assomiglierà al seguente:

```
Cima dello stack→ Vecchio IP (indirizzo di ritorno, prima parte)
                  Vecchio CS (indirizzo di ritorno, seconda parte)
                  Vecchio Registro Flag
```

(Lo stack cresce verso il basso, quindi il Vecchio Registro Flag si trova nella posizione più alta della memoria).

Quando inserite un'istruzione INT in un programma, l'interrupt non è una sorpresa. Perché, dunque, bisogna salvare i flag? Il salvataggio dei flag non è utile solo quando si ha un interrupt esterno che viene inviato in un momento imprevedibile? La risposta è no. Esiste un'ottima ragione per salvare e ripristinare i flag rispettivamente prima e dopo un'istruzione INT. Infatti, senza questa funzione, Debug sarebbe inutilizzabile. Debug usa un flag speciale nel registro flag chiamato Trap Flag (Flag di Interruzione). Questo flag porta l'8088 in una modalità speciale chiamata *passo a passo* che Debug utilizza per tracciare i programmi un'istruzione alla volta. Quando il flag di interruzione (Trap) è impostato, l'8088 invia un INT 1 dopo aver eseguito qualsiasi istruzione.

L'istruzione INT 1 azzerava il flag di interruzione, quindi l'8088 non sarà in modalità passo a passo durante la procedura INT 1 di Debug. Ma dato che INT 1 salva i flag nello stack, l'istruzione IRET per ritornare al programma ripristina il flag di interruzione.

Quindi il processo continua con l'istruzione successiva. Questo è solo un esempio che mostra l'utilità di salvare i flag di stato. Ma, come vedrete successivamente, questa funzione di ripristino dei flag non è sempre appropriata.

Alcune procedure di interrupt aggirano il ripristino dei flag di stato. Per esempio, la procedura INT 21h del DOS, alcune volte modifica i flag cambiando il normale processo di ritorno. Molte delle procedure INT 21h utilizzate per leggere o scrivere su disco, ritornano con il flag di riporto impostato per segnalare un errore come, per esempio, la mancanza di un dischetto nel drive.

I VETTORI DI INTERRUPT

Da dove prendono gli indirizzi per le procedure queste istruzioni di interrupt ?

Ciascuna istruzione di interrupt ha un numero di interrupt come, per esempio, 21h in INT 21h. L'8088 trova gli indirizzi per le procedure di interrupt in una tabella di *vettori di interrupt*, che è localizzata nella parte più bassa della memoria. Per esempio, l'indirizzo composto da due parole per la procedura INT 21h si trova in 0000:0084. Questo indirizzo può essere ottenuto moltiplicando il numero di interrupt per 4 ($4 \times 21h = 84h$).

Questi vettori risultano molto utili per aggiungere delle funzioni al DOS, dato che permettono di intercettare delle chiamate alle procedure di interrupt modificando la tabella dei vettori. Userete proprio questo accorgimento alla fine del libro per simulare sul video l'accensione della luce della unità disco.

Tutti questi metodi diventeranno più chiari dopo aver visto alcuni esempi. Da questo momento in avanti, troverete molti esempi che vi aiuteranno a mettere a fuoco tutti i concetti che avete appreso.

SOMMARIO

Come avevamo detto, questo capitolo ha presentato moltissime informazioni. Non le userete tutte, ma dovrete sapere di più sui segmenti. Il capitolo 13 vi introdurrà alla progettazione modulare e userete alcuni aspetti dei segmenti per facilitare il lavoro. Avete iniziato questo capitolo imparando il modo usato dall'8088 per dividere la memoria. Per capire meglio i segmenti, avete costruito un file .EXE con due segmenti differenti. Avete anche visto la necessità di usare l'istruzione INT 21h con la funzione 4Ch per uscire da un programma .EXE (invece di usare INT 20h). Questo è molto importante dal momento che utilizzerete, d'ora in avanti, solamente file .EXE.

Avete visto che l'area di 100h byte (256), chiamata PSP, che si trova all'inizio dei programmi contiene una copia di quanto è stato digitato sulla riga di comando. Non userete questa caratteristica in questo libro, ma questa vi aiuta a capire perché il DOS imposta un'area di memoria così grossa prima di un programma.

Avete infine imparato il significato delle direttive DOSSEG, .MODEL, .CODE, .STACK, NEAR e FAR. Queste direttive aiutano a lavorare con i segmenti. In questo libro,

sfrutterete raramente il potenziale di queste direttive, dal momento che utilizzerete dei programmi con solo due segmenti. Ma per i programmatori che vorranno scrivere dei programmi estesi (usando la direttiva `.MODEL MEDIUM`), queste direttive saranno di valore incalcolabile. Se siete interessati, potrete trovare ulteriori informazioni nel manuale dell'assemblatore.

Alla fine di questo capitolo vi sono state fornite alcune ulteriori informazioni sull'istruzione `INT`. Ora sapete abbastanza per vedere come scrivere un programma in linguaggio assembler più esteso e più utile.

IMPOSTAZIONE DEL LAVORO

Siete venuti a conoscenza di argomenti nuovi e interessanti e qualche vi sarete probabilmente chiesti se stavate “viaggiando” senza meta. La risposta è ovviamente no. Avete ora abbastanza familiarità con questo nuovo ambiente per potervi avventurare in quello che è lo scopo di questo libro: la creazione del programma Dskpatch. In questo capitolo sarà tracciato un percorso da seguire durante il resto della trattazione e dal prossimo inizierete la costruzione del programma.

La versione finale di Dskpatch non sarà presentata tutta in una volta; scriverete invece delle piccole parti verificandole di volta in volta con dei programmi di verifica appropriati. Per poter fare questo, è necessario sapere dove si vuole arrivare. E' per questo motivo che, prima di iniziare la costruzione del programma, imposteremo il lavoro da eseguire.

Dato che Dskpatch deve gestire le informazioni presenti sui dischi, inizieremo proprio da qui.

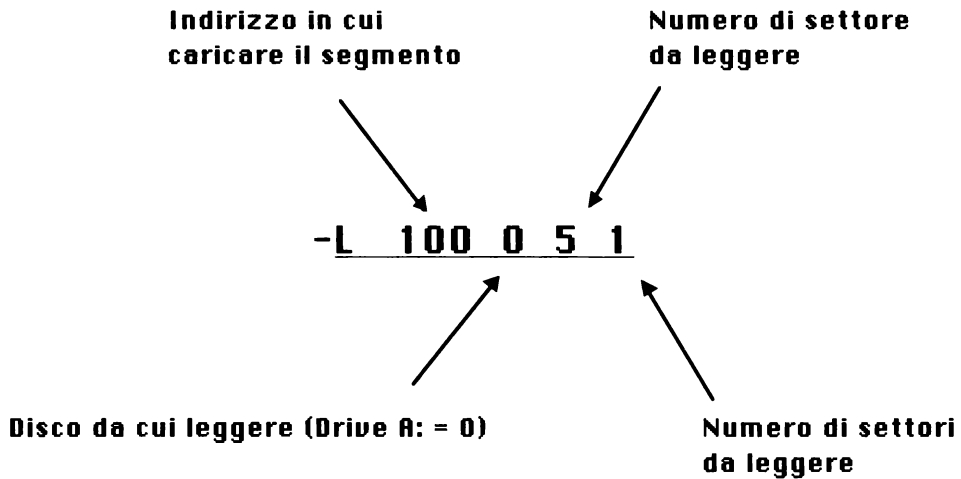
DISCHETTI, SETTORI E ARGOMENTI SIMILI

Le informazioni sui dischetti sono divise in *settori*; ciascun settore contiene 512 byte di dati. Un dischetto da cinque pollici e un quarto a doppia faccia e doppia densità, formattato con il DOS versione 2.0 o superiore, è composto da 720 settori che corrispondono a 360K di informazioni ($720 \times 512 = 368.640$ byte). (Fate riferimento alla tabella 12-1 per gli altri tipi di dischi). Se fosse possibile esaminare direttamente questi settori, potreste leggere la directory o i nomi dei file direttamente sul disco. Voi non potete farlo, ma Dskpatch sì. Usate Debug per approfondire la conoscenza sui settori e per vedere come un settore verrà visualizzato con Dskpatch.

Debug offre un comando, L (*Load*, Carica), che permette di leggere un settore dal disco nella memoria. Come esempio, provate a esaminare la directory che inizia nel settore 5 di un dischetto a doppia faccia (fate riferimento alla tabella 12-1 per determinare quale numero di settore usare per visualizzare la directory, se avete un tipo differente di disco). Caricate il settore 5 dal dischetto nel drive A (che per Debug corrisponde al drive 0) usando il comando L. Assicuratevi di avere un dischetto da 360K (1,2M, 720K o 1,44M) nel drive A e inserite il comando seguente:

Tabella 12-1. Settori di partenza per la root

Tipo di Disco	Settore/disco	Directory
5,25 360K	720	5
5,25 1.2M	2.400	15
3,5 720K	1.440	7
3,5 1.44M	2.880	19



Come potete vedere dalla figura, questo comando carica i settori in memoria, partendo con il settore 5 e continuando con una distanza di 100h dal segmento dati. Per visualizzare il settore 5, potete usare il comando Dump:

```

-D 100
396F:0100 49 42 4D 42 49 4F 20 20-43 4F 4D 27 00 00 00 00 IBMBIO COM'....
396F:0110 00 00 00 00 00 00 00 60-30 0F 02 00 27 5C 00 00 ..... '0...'\..
396F:0120 49 42 4D 44 4F 53 20 20-43 4F 4D 27 00 00 00 00 IBMDOS COM'....
396F:0130 00 00 00 00 00 00 00 60-30 0F 1A 00 A8 77 00 00 ..... '0....w..
396F:0140 43 4F 4D 4D 41 4E 44 20-43 4F 4D 20 00 00 00 00 COMMAND COM ....
396F:0150 00 00 00 00 00 00 00 60-30 0F 38 00 F4 62 00 00 ..... '0.8..b..
396F:0160 41 55 54 4F 45 58 45 43-42 41 54 20 00 00 00 00 AUTOEXECBAT ....
396F:0170 00 00 00 00 00 00 00 A3 99-54 14 51 00 06 00 00 00 ..... T.Q.....
-D
396F:0180 46 57 20 20 20 20 20 20-43 4F 4D 20 00 00 00 00 FW COM ....
396F:0190 00 00 00 00 00 00 00 A8 99-54 14 52 00 80 AF 00 00 ..... T.R.....
396F:01A0 46 57 20 20 20 20 20 20-4F 56 4C 20 00 00 00 00 FW OVL ....
396F:01B0 00 00 00 00 00 00 00 B0 99-54 14 53 00 81 02 00 00 ..... T.S.....
396F:01C0 43 4F 4E 46 49 47 20 20-53 59 53 20 00 00 00 00 CONFIG SYS ....
396F:01D0 00 00 00 00 00 00 00 B6 99-54 14 54 00 0A 00 00 00 ..... T.T.....
396F:01E0 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
396F:01F0 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....

```

Con Dskpatch sar  mostrato un formato simile, ma con qualche miglioramento. Dskpatch sar  l'equivalente di un editor a pieno schermo per i settori di un disco. Sarete in grado di visualizzare i settori sullo schermo, di spostare il cursore tra i dati e di modificare i numeri o i caratteri desiderati. Sarete anche in grado di riscrivere il settore modificato sul disco (da qui deriva il nome Disk patch, correttore di disco che viene ovviamente abbreviato in Dskpatch per il limite imposto dal DOS di otto caratteri).

Disco A Settore 0

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	0123456789ABCDEF
00	EB	34	98	49	42	4D	28	28	33	2E	33	00	02	04	01	00	44E1BM 3.3
10	02	00	02	EF	A9	F8	2B	00	11	00	08	00	11	00	00	00	n-r'+ < . <
20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	12	.3LxLJ !
30	00	00	00	00	01	00	FA	33	C8	8E	D8	BC	00	7C	16	07	x 6+7AV S1+i
40	BB	78	00	36	C5	37	1E	56	16	53	BF	2B	7C	B9	0B	00	z&C= t a� r�-r�
50	FC	AC	26	00	3D	00	74	03	26	8A	05	AA	0A	C4	E2	F1	v�G +iJ= rg�)
60	06	1F	09	47	02	C7	07	2B	7C	FB	CD	13	72	67	A0	10	i�=a ! � i�?
70	7C	98	F7	26	16	7C	03	06	1C	7C	03	06	0E	7C	A3	3F	i�7i? =a<i�!
80	7C	A3	37	7C	B8	28	00	F7	26	11	7C	8B	1E	0B	7C	03	H=� 7i? i?if
90	C3	48	F7	F3	01	06	37	7C	BB	00	05	A1	3F	7C	E8	9F	7 � rvjJ �J
A0	00	B8	01	02	E8	B3	00	72	19	8B	FB	B9	0B	00	BE	D9	}�=u i� J} � �
B0	7D	F3	A6	75	0D	8D	7F	28	BE	E4	7D	B9	0B	00	F3	A6	t^JwEj ZS= ^vA
C0	74	18	BE	77	7D	E8	6A	00	32	E4	CD	16	5E	1F	8F	04	�D =v-}��i- 3v=
D0	8F	44	02	CD	19	BE	C4	7D	EB	EB	A1	1C	05	33	D2	F7	6 !uL�<i?i�=i? �
E0	36	0B	7C	FE	C8	A2	3C	7C	A1	37	7C	A3	3D	7C	BB	00	i?i?i i^! * ;i08
F0	07	A1	37	7C	E8	49	00	A1	18	7C	2A	06	3B	7C	48	38	

Premere un tasto funzione o introdurre carattere o byte esadecimale: _

Figura 12-2. Esempio della visualizzazione di DSKPATCH

Dskpatch   un pretesto per scrivere delle procedure che risulteranno utili anche in altri programmi. Usando Dskpatch come esempio in questo libro, imparerete delle istruzioni e dei concetti che vi serviranno moltissimo durante la creazione di vostri programmi. Svilupperete una serie di procedure di uso generale che potranno facilmente essere adattate in qualsiasi contesto.

Date un'occhiata ad alcuni miglioramenti che farete alla visualizzazione dei settori di Debug. Debug visualizza solamente i caratteri "stampabili" (solo 96 caratteri dei 256 disponibili). Perch  fa questo? Perch  il sistema operativo MS-DOS pu  essere usato su diversi computer, alcuni dei quali possono visualizzare solamente 96 caratteri. Per questo motivo la Microsoft (autrice di Debug) ha deciso di scrivere una versione di Debug che potesse essere compatibile con qualsiasi sistema MS-DOS.

Dskpatch   per il personal computer IBM (o compatibili) e permette di visualizzare tutti i 256 caratteri disponibili. Usando la funzione 2 del DOS per visualizzare un

carattere, potrete gestire quasi tutti i caratteri ad eccezione di qualcuno come, per esempio, il corrispondente del codice ASCII 7 che invece di essere visualizzato causa l'emissione di un segnale acustico. Nella parte 3 di questo libro, vedrete come visualizzare anche questi caratteri speciali.

Farete anche un uso intenso dei tasti funzione (potrete per esempio visualizzare il settore successivo premendo F4), e sarete in grado di cambiare qualsiasi byte spostando il cursore sul byte desiderato e digitando un nuovo valore. Sarà come usare un word processor, in cui è possibile effettuare delle modifiche molto facilmente. Altri dettagli appariranno durante la costruzione di Dskpatch. (La figura precedente mostra quella che sarà la visualizzazione di Dskpatch, una volta ultimato il programma).

IL PUNTO DELLA SITUAZIONE

Nel capitolo 13 imparerete a dividere un programma in molti file sorgente, mentre nel capitolo 14 inizierete a lavorare seriamente su Dskpatch. Alla fine avrete nove file sorgente che dovranno essere collegati insieme. E anche se non userete mai Dskpatch, potrete sempre importare le procedure sviluppate e contenute nei nove file sorgente in altri programmi. In ogni caso, vi farete un'ottima idea di come scrivere dei programmi lunghi durante la lettura di questi ultimi capitoli.

Avete già creato alcune procedure utili come, per esempio, WRITE_HEX (per scrivere un byte come numero esadecimale a due cifre) e WRITE_DECIMAL (per scrivere un numero in decimale). Scriverete ora alcune procedure per visualizzare un blocco di memoria, più o meno nello stesso modo in cui agisce il comando D di Debug. Inizierete visualizzando 16 byte di memoria (una riga della visualizzazione di Debug) e lavorerete quindi su 16 righe di 16 byte (mezzo settore). Un settore intero non sarà contenuto nello schermo a causa del formato di visualizzazione selezionato; per questo motivo Dskpatch sarà equipaggiato con un sistema di scrolling (scorrimento) attraverso il settore, utilizzando gli interrupt del BIOS (e non del DOS). Questo sarà comunque fatto nella parte finale del libro.

Nel momento in cui saranno visualizzati tutti i 256 byte di un settore, costruirete un'altra procedura per leggere un settore dal disco in un'area di memoria. Creerete quindi delle procedure per l'input da tastiera, rendendo possibile l'interazione con l'utente. Migliorerete infine, dal punto di vista estetico, la visualizzazione dei settori. Durante questo processo verrete a conoscenza delle routine del BIOS che permettono di controllare la visualizzazione, lo spostamento del cursore e così via. Sarete quindi pronti per usare altre routine del BIOS per poter visualizzare tutti i 256 caratteri disponibili.

SOMMARIO

In questo capitolo è stata fatta una panoramica generale sul contenuto dei capitoli successivi. Dovreste avere un'idea di quello che vi aspetta; avventuratevi quindi nel

prossimo capitolo in cui apprenderete le basi della progettazione modulare e imparerete a dividere un programma in più file sorgente. Quindi, nel capitolo 14, inizierete a scrivere delle procedure di verifica per visualizzare delle sezioni della memoria.

LA PROGETTAZIONE MODULARE

Senza la progettazione modulare Dskpatch non sarebbe molto divertente da scrivere. La progettazione modulare facilita notevolmente il lavoro nella costruzione di qualsiasi programma (ad eccezione di quelli molto brevi). In questo capitolo apprenderete le nozioni di base della progettazione modulare, e seguirete queste regole durante il resto del libro. Iniziate imparando a dividere un programma abbastanza esteso in più file sorgente.

ASSEMBLAGGIO SEPARATO

Nel capitolo 10 avete aggiunto la procedura WRITE_DECIMAL al file VIDEO_IO.ASM, e avete anche inserito una piccola procedura di verifica chiamata TEST_WRITE_DECIMAL. Estraete questa procedura dal file VIDEO_IO.ASM e inseritela in un altro file chiamato TEST.ASM. Assemblerete quindi questi due file separatamente e li collegherete insieme in un programma. Questo è il file TEST.ASM:

Listato 13-1. Il file TEST.ASM

```
DOSSEG
.MODEL    SMALL

.STACK

.CODE
    EXTRN  WRITE_DECIMAL:PROC

TEST_WRITE_DECIMAL    PROC
    MOV    DX,12345
    CALL  WRITE_DECIMAL
    INT   20h           ;Ritorna al DOS
TEST_WRITE_DECIMAL    ENDP

END    TEST_WRITE_DECIMAL
```

Conoscete già la maggior parte di questo file, ma la direttiva EXTRN è nuova. L'enunciato EXTRN WRITE_DECIMAL:PROC indica all'assemblatore due cose: che WRITE_DECIMAL è un file *esterno* e che è una procedura. Il tipo di procedura (NEAR

o FAR) dipende dalla direttiva `.MODEL`. Dato che avete usato `.MODEL SMALL`, che definisce le procedure come NEAR, `WRITE_DECIMAL` sarà nello stesso segmento. L'assemblatore genera quindi una chiamata NEAR per questa procedura (avrebbe generato una chiamata FAR se aveste inserito FAR dopo `WRITE_DECIMAL`). (E' possibile usare NEAR o FAR al posto PROC nell'enunciato EXTRN se si vuole definire esplicitamente il tipo di una procedura; comunque, è sempre meglio lasciare a `.MODEL` il compito di definire i tipi di procedura).

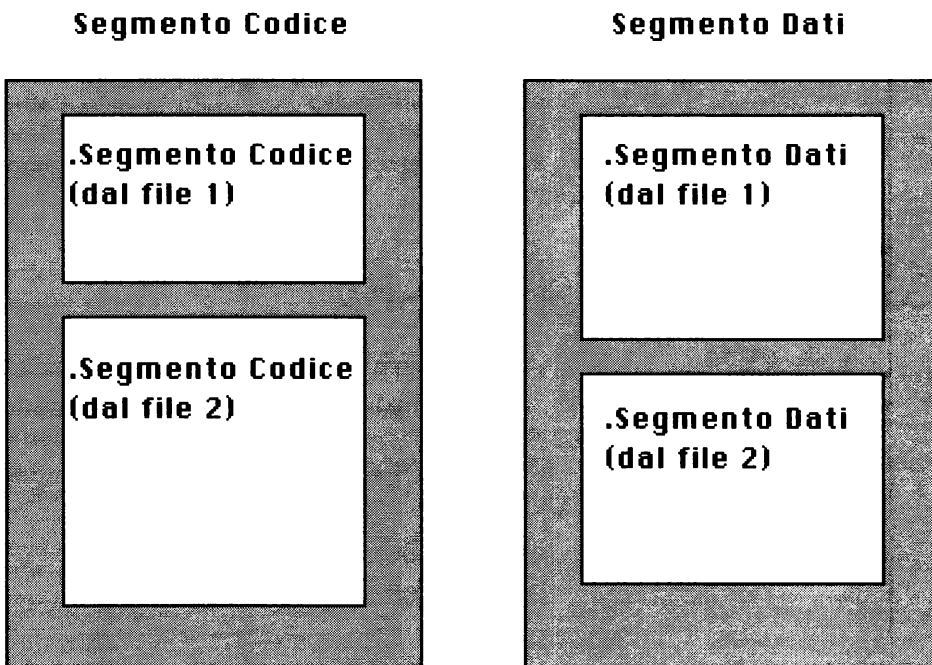


Figura 13-1. LINK collega i segmenti da file diversi

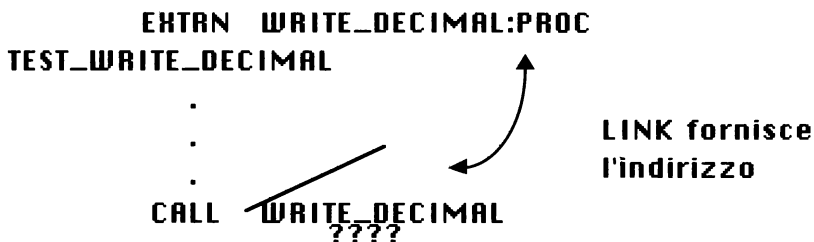


Figura 13-2. LINK assegna gli indirizzi ai nomi esterni

Queste sono le uniche modifiche necessarie per dei file sorgenti separati, fino a quando non si iniziano a inserire dei dati in memoria. A quel punto, dovrete aggiungere un altro segmento per contenere i dati. Modificate ora VIDEO_IO.ASM, quindi assemblate e collegate questi due file.

Rimovete la procedura TEST_WRITE_DECIMAL da VIDEO_IO.ASM (non ne avete più bisogno dato che l'avete inserita in TEST.ASM).

Sostituite l'enunciato END TEST_WRITE_DECIMAL (alla fine del file VIDEO_IO.ASM) con un semplice END. Dato che avete spostato la procedura principale in TEST.ASM, il file VIDEO_IO.ASM contiene ora delle procedure *esterne*. Queste procedure non hanno un significato da sole, ma devono essere chiamate da altri file. Non è necessario un nome dopo la direttiva END nel file VIDEO_IO.ASM, poiché il programma principale si trova ora in TEST.ASM.

Una volta terminate le modifiche, il file sorgente VIDEO_IO.ASM dovrebbe apparire nel modo seguente:

```
.MODEL    SMALL
.CODE

        PUBLIC  WRITE_HEX_DIGIT
        .
        .
        .
WRITE_HEX_DIGIT      ENDP

        PUBLIC  WRITE_HEX
        .
        .
        .
WRITE_HEX            ENDP

        PUBLIC  WRITE_CHAR
        .
        .
        .
WRITE_CHAR           ENDP

        PUBLIC  WRITE_DECIMAL
        .
        .
        .
WRITE_DECIMAL       ENDP

END
```

Assemblate questi due file con la stessa procedura seguita per il file VIDEO_IO. Il file TEST.ASM ha tutti i dati necessari grazie alla direttiva EXTRN. Dovreste ora avere i file TEST.OBJ e VIDEO_IO.OBJ. Usate il comando seguente per collegare questi due file in un programma chiamato TEST.EXE:

```
A>LINK TEST VIDEO_IO;
```

LINK unisce le procedure di questi due file e crea un altro file contenente l'intero programma. Come nome per il file .EXE risultante, viene utilizzato il primo nome di file inserito dopo il comando LINK. In questo caso, verrà creato il file TEST.EXE.

Questo è tutto; avete creato un programma da due file sorgente. Il programma finale (TEST.EXE) è identico, come funzione, alla versione .COM creata nel capitolo 10 dal singolo file VIDEO_IO.ASM (che conteneva la procedura principale TEST_WRITE_DECIMAL).

Da questo momento in avanti userete molto spesso i file sorgente separati, e la loro importanza risulterà chiara durante il lavoro. Nel prossimo capitolo scriverete un programma per visualizzare delle sezioni di memoria in esadecimale. Di norma scriverete una bozza di programma, prima di costruire la versione finale. In questo modo potrete avere un'idea di come scrivere una buona versione finale, risparmiando tempo e fatica.

Ci sono alcuni altri metodi per facilitare il lavoro che possono essere definiti come le *Tre Leggi della Progettazione Modulare*.

LE TRE LEGGI DELLA PROGETTAZIONE MODULARE

Queste leggi sono riassunte nella tabella 13-1. Queste non sono delle vere e proprie *leggi*, ma sono dei suggerimenti (che utilizzerete in questo libro). Definite la vostra legge, se lo desiderate; l'importante è seguire sempre la stessa procedura. Se sarete coerenti, il vostro lavoro sarà più semplice.

Tabella 13-1 *Le Tre Leggi della Progettazione Modulare*

1. Salvate e ripristinate *tutti* i registri, *a meno che* la procedura non fornisca un valore in un registro specifico.
2. Siate coerenti nel passare le informazioni tra i registri. Per esempio:
 - DL,DX - Per inviare i byte e i valori composti da una parola
 - AL,AX - Per riportare i byte e i valori composti da una parola
 - BX:AX - Per riportare i valori composti da due parole
 - DS:DX - Per inviare e riportare gli indirizzi
 - CX - Da usare come contatore
 - CF - E' impostato quando si verifica un errore; un errore può essere fornito in uno dei registri come, per esempio, AL o AX.
3. Definite *tutte* le interazioni esterne nell'intestazione del commento:
 - Le informazioni necessarie in fase di inserimento.
 - Le informazioni fornite (registri modificati).
 - Le procedure chiamate.
 - Le variabili usate (lette, scritte, e così via).

Si può fare un parallelo tra la progettazione modulare nella programmazione e la progettazione modulare nell'ingegneria. Un ingegnere elettrotecnico, per esempio, può costruire un'apparecchiatura molto complicata utilizzando dei componenti che eseguono funzioni differenti, senza sapere come funziona esattamente un determinato componente. Ma se ciascun componente utilizza un voltaggio e un collegamento diverso, questa mancanza di coerenza rende il lavoro dell'ingegnere molto più complesso, dato che diventa necessario fornire diversi voltaggi e creare nuovi collegamenti. Fortunatamente per l'ingegnere esistono degli standard che riducono al minimo il numero dei voltaggi possibili. Quindi, saranno probabilmente necessari solamente quattro voltaggi differenti, invece di uno per ciascun componente.

La progettazione modulare e le interfacce standard sono molto importanti nei programmi in linguaggio assembly, ed è per questo che vengono fornite le tre leggi che seguirete durante questo libro. Vi accorgete alla fine che queste leggi avranno facilitato il vostro lavoro. Vediamo ora più dettagliatamente queste tre leggi.

Salvate e ripristinate tutti i registri, a meno che la procedura non fornisca un valore in un registro specifico. Nell'8088 non ci sono moltissimi registri. Salvando i registri all'inizio di una procedura, è possibile utilizzarli in qualsiasi modo all'interno della procedura stessa. Ma bisogna ricordarsi di ripristinarli alla fine. Effettuerete queste operazioni in tutte le procedure, usando le istruzioni PUSH e POP.

L'unica eccezione è quando le procedure devono fornire alcune informazioni alla procedura chiamante. Per esempio, una procedura che legge un carattere dalla tastiera deve in qualche modo fornire il carattere letto. Non bisogna quindi salvare i registri usati per fornire delle informazioni.

Delle procedure brevi aiutano inoltre a superare il problema derivante dalla esigua quantità di registri a disposizione, e rendono il programma più facile da leggere e da scrivere. Vedrete più approfonditamente questo argomento durante lo sviluppo di Dskpatch.

Siate coerenti nel passare le informazioni tra i registri. Il lavoro diventa semplice se vengono impostati degli standard nello scambio di informazioni tra le procedure. Userete un registro per inviare dei dati e un altro per riceverli. Avrete anche bisogno di passare degli indirizzi e per questo vi servirete della coppia di registri DS:DX, in modo da poter posizionare i dati in qualsiasi area di memoria. Approfondirete questo argomento quando aggiungerete un nuovo segmento per i dati e inizierete a usare il registro DS.

Il registro CX sarà utilizzato solamente come contatore. Costruirete presto una procedura per scrivere un carattere diverse volte, in modo che sia possibile, per esempio, scrivere dieci spazi chiamando una procedura (WRITE_CHAR_N_TIMES) con il registro CX impostato a 10. Userete il registro CX ogni volta in cui sarà necessario un contatore o quando vorrete riportare un conteggio come, per esempio, il numero di caratteri letti dalla tastiera (farete questo quando scriverete una procedura chiamata READ_STRING).

Infine imposterete il flag di riporto (CF) ogni volta che si verifica un errore (e lo azzererete nel caso contrario). Non tutte le procedure usano il flag di riporto. Per esempio, WRITE_CHAR funziona sempre e non c'è quindi bisogno di riportare una condizione di errore. Ma una procedura che scrive su disco può incontrare alcuni errori (manca il disco, il dischetto è protetto in scrittura e così via). In questo caso,

userete un registro per fornire un codice di errore. Non ci sono standard in questo caso, dato che il DOS usa registri differenti per funzioni differenti (è colpa sua, non nostra).

Definite tutte le interazioni esterne nell'intestazione del commento. Non è necessario imparare come funziona una procedura se tutto quello che volete fare è usarla. E' questo il motivo per cui si deve inserire un commento dettagliato prima della procedura. L'intestazione, infatti, contiene *tutte* le informazioni necessarie per poter usare la procedura in un altro programma. Vengono indicati i registri usati e che cosa deve essere inserito in ciascun registro. Alcune procedure usano i registri come variabili locali e altre procedure (che vedrete presto) utilizzano le variabili in memoria. L'intestazione deve fornire tutte le informazioni necessarie sui registri. Infine, ciascuna intestazione deve elencare tutte le procedure chiamate. Ecco un esempio di come un'intestazione fornisce delle informazioni:

```

;-----;
; Questo è un esempio di un'intestazione completa. Questa parte,      ;
; solitamente, contiene una breve descrizione della funzione della   ;
; procedura. Per esempio, questa procedura scriverà il messaggio     ;
; "Settore " sulla prima riga.                                       ;
;                                                                       ;
; Inserimento: DS:DX Indirizzo del messaggio "Settore "             ;
; Ritorno:     AX    Codice di errore (se esistente)                 ;
;                                                                       ;
; Chiamate:    GOTO_XY, WRITE_STRING (procedure chiamate)           ;
; Lettura:     STATUS_LINE_NO (variabili in memoria lette)          ;
; Scrittura:   DUMMY (variabili in memoria modificate)             ;
;-----;

```

Ogni volta che vorrete usare una procedura, sarà sufficiente leggere il commento per imparare a usarla. Non sarà necessario addentrarsi nel funzionamento interno della procedura per capire che cosa fa.

Queste leggi rendono la programmazione in linguaggio assembly più semplice e senz'altro le adatterete anche se non necessariamente dal primo esempio. Vi capiterà infatti di dover scrivere un programma (o una procedura) ma di non sapere esattamente come costruirla. Creerete allora una bozza di programma, senza seguire le leggi. Una volta impostato a grandi linee il lavoro, potrete ritornare indietro e riscrivere ciascuna procedura in conformità delle leggi della progettazione modulare. La programmazione è un processo che prosegue a salti. In questo libro vi mostreremo alcuni degli "scossoni" che ha subito il programma Dskpatch prima di raggiungere la versione definitiva. Purtroppo non c'è spazio a sufficienza per mostrare tutte le versioni intermedie. I primi tentativi, molto spesso, hanno poco a che vedere con quella che sarà la versione finale. Quindi, quando scriverete un programma, non preoccupatevi di dover fare tutto giusto al primo tentativo; preparatevi a riscrivere più volte le vostre procedure.

Nel prossimo capitolo costruirete un piccolo programma per visualizzare un blocco di memoria. Questo non sarà la versione finale; ne creerete delle altre prima di raggiungere quella definitiva e, una volta raggiunta, vedrete che ci saranno altre Avete

modifiche che vi piacerebbe fare. La morale è: Un programma non è mai finito, ma da qualche parte bisogna fermarsi.

SOMMARIO

Questo è un capitolo da ricordare e da utilizzare in futuro. Avete iniziato imparando a dividere un programma in più file sorgente che possono essere assemblati indipendentemente, e quindi riuniti con il programma LINK. Avete usato le direttive PUBLIC e EXTRN per informare il linker che le procedure si trovavano in file separati. PUBLIC indica che gli altri file sorgente possono chiamare le procedure definite dopo la direttiva PUBLIC, mentre EXTRN indica all'assemblatore che le procedure da usare si trovano in un altro file.

VISUALIZZAZIONE DELLA MEMORIA

Da questo momento in avanti, vi concentrerete solamente sulla costruzione del programma Dskpatch. Alcune delle istruzioni che incontrerete potrebbero non esservi familiari e queste verranno spiegate rapidamente. Quindi, per informazioni dettagliate, avete bisogno di un libro che spieghi il significato di tutte le istruzioni disponibili. La maggior parte dei libri di riferimento che trattano i microprocessori 8088, 80286 e 80386 contengono tutte le informazioni necessarie. In questo libro, invece di spiegare dettagliatamente tutte le istruzioni dell'8088, vedremo nuovi concetti come, per esempio, i modi di indirizzamento. Nella terza parte vi allontanerete ancora di più dalle istruzioni e vi saranno fornite informazioni specifiche sul personal computer IBM.

Ora scrivete un piccolo programma per visualizzare 16 byte di memoria in notazione esadecimale, in modo da imparare le *modalità di indirizzamento*. Prima di iniziare, dovete imparare a usare la memoria come variabili.

MODI DI INDIRIZZAMENTO

Avete visto due modi di indirizzamento che sono conosciuti come modi di indirizzamento *immediato* e *di registro*. Il modo di registro utilizza i registri come variabili; per esempio, l'istruzione:

```
MOV    AX, BX
```

usa i due registri AX e BX come variabili.

Avete già visto, invece, il modo immediato, in cui un numero viene direttamente inserito in un registro. Per esempio:

```
MOV    AX, 2
```

Questa operazione sposta il byte o la parola *immediatamente* successiva all'istruzione nel registro. In questo esempio, l'istruzione MOV è composta da tre byte:

```
396F:0100    B80200                MOV    AX, 0002
```

L'istruzione è B8h, mentre i due byte successivi (02h e 00h) rappresentano il dato (ricordatevi che l'8088 memorizza prima il byte basso e poi quello alto).

Ora imparerete a usare la memoria come variabile. Il modo immediato permette di leggere la parte di memoria fissa immediatamente successiva all'istruzione, ma non permette di cambiare la memoria. Per questo motivo, sono necessari altri modi di indirizzamento.

Iniziate con un esempio. Il programma seguente legge 16 byte della memoria, uno alla volta, e visualizza ciascun byte in notazione esadecimale, con un solo spazio tra ciascuno dei 16 numeri. Inserite il programma seguente nel file DISP_SEC.ASM e assemblatelo:

Listato 14-1. Il nuovo file DISP_SEC.ASM

```
DOSSEG
.MODEL    SMALL

.STACK

.DATA

        PUBLIC  SECTOR
SECTOR  DB    10h, 11h, 12h, 13h, 14h, 15h, 16h, 17h
        DB    18h, 19h, 1Ah, 1Bh, 1Ch, 1Dh, 1Eh, 1Fh

.CODE

        EXTRN  WRITE_HEX:PROC
        EXTRN  WRITE_CHAR:PROC

;-----;
; Questo è un semplice programma per visualizzare 16 byte di memoria, ;
; come numeri esadecimali, su una sola riga. ;
;-----;
DISP_LINE  PROC
        MOV    AX,DGROUP      ;Inserisce il segmento dati in AX
        MOV    DS,AX          ;Imposta DS per puntare i dati

        XOR    BX,BX          ;Imposta BX a zero
        MOV    CX,16          ;Visualizza i 16 byte

HEX_LOOP:
        MOV    DL,SECTOR[BX]  ;Preleva 1 byte
        CALL   WRITE_HEX      ;Visualizza il byte in esadecimale
        MOV    DL,' '         ;Scrive uno spazio tra i numeri
        CALL   WRITE_CHAR
        INC    BX
        LOOP   HEX_LOOP
        MOV    AH,4Ch          ;Funzione per tornare al DOS
        INT    21h

DISP_LINE ENDP

        END    DISP_LINE
```

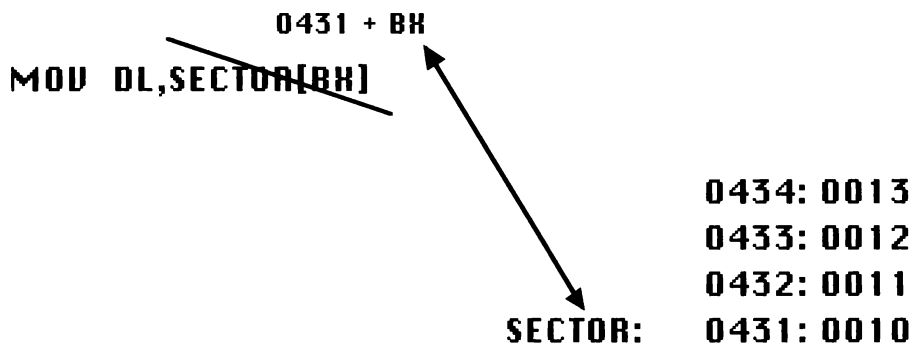



Figura 14-1. Conversione di *SECTOR(BX)*

Provate questo programma per vedere come funziona. Assemblate *DISP_SEC*. Ora collegate *DISP_SEC.OBJ* e *VIDEO_IO.OBJ* e create un file chiamato *DISP_SEC.EXE*. *LINK* creerà il programma inserendo le varie parti nell'ordine in cui i rispettivi nomi compaiono sulla riga di comando. Dato che la procedura principale deve apparire all'inizio del programma, il primo nome deve essere quello del file che contiene la procedura principale (in questo caso *DISP_SEC*). Ricordatevi che un punto e virgola deve terminare l'elenco dei file. Il comando da impartire sarà:

```
A>LINK DISP_SEC VIDEO_IO;
```

E' possibile inserire qualsiasi nome di file sulla riga di comando, ma ricordatevi di inserire sempre per primo il nome del file contenente la procedura principale. In generale, il comando sarà composto nel modo seguente:

```
LINK file1 file2 file3.....;
```

Provate ora a eseguire il file *.EXE*. Se non appare:

```
10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
```

tornate ai file sorgenti e controllate di non aver commesso qualche errore. Vediamo ora come funziona *DISP_SEC*. L'istruzione:

```
MOV DL,SECTOR[BX] ;Preleva 1 byte
```

utilizza un nuovo modo di indirizzamento, conosciuto come *Indirizzamento di Memoria Indiretto*, cioè indirizzamento attraverso il registro *Base* con uno *scarto* (o, più semplicemente, *Base Relativo*). Per vedere che cosa significa precisamente, avete bisogno di qualche altra informazione sui segmenti.

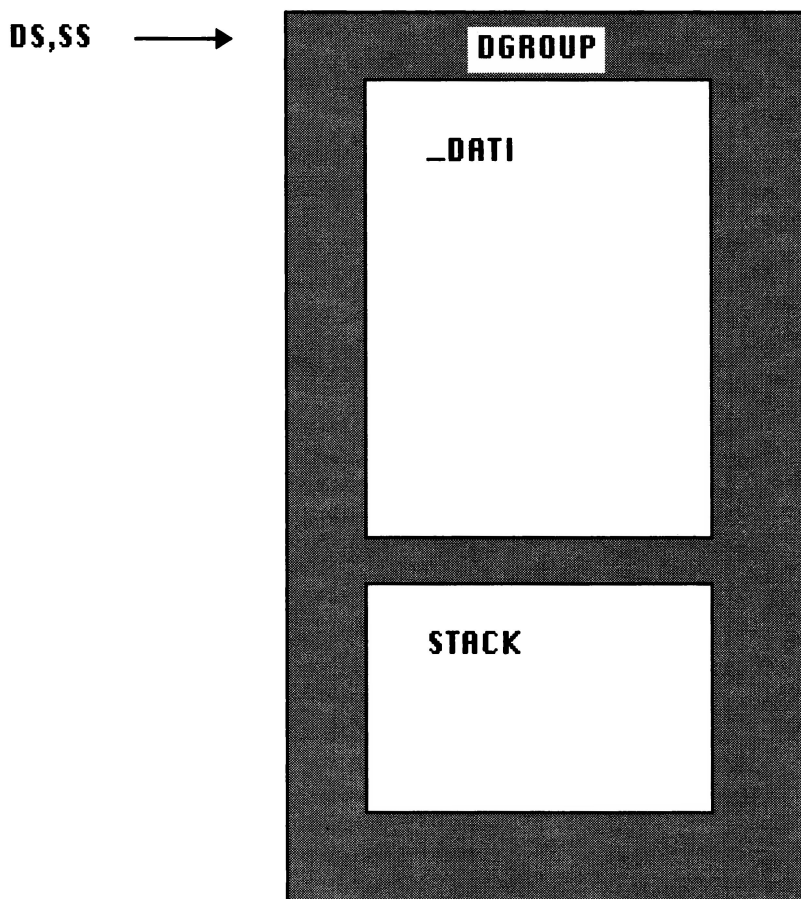


Figura 14-2. Lo stack e i dati si trovano in un gruppo di segmenti (DGROUP).

IL SEGMENTO DATI

Notate che nel file `DISP_SEC`, dopo `.DATA`, appare l'etichetta `SECTOR`. La direttiva `.DATA` definisce un segmento dati da usare per le variabili di memoria. (A proposito, il nome del segmento creato da `.DATA` è `_DATA`). Ogni volta che vorrete memorizzare e leggere dei dati dalla memoria, dovrete utilizzare questo segmento. Torneremo alle variabili di memoria tra qualche istante; vediamo prima qualche ulteriore informazione sui segmenti.

La direttiva `.MODEL SMALL` crea un piccolo programma composto da 64K di codice e 64K di dati (in altre parole, un segmento per il codice e un segmento per i dati). Dato che sia i dati (definiti da `.DATA`) che lo stack (definito da `.STACK`) sono dei dati, questi vengono inseriti in un solo segmento.

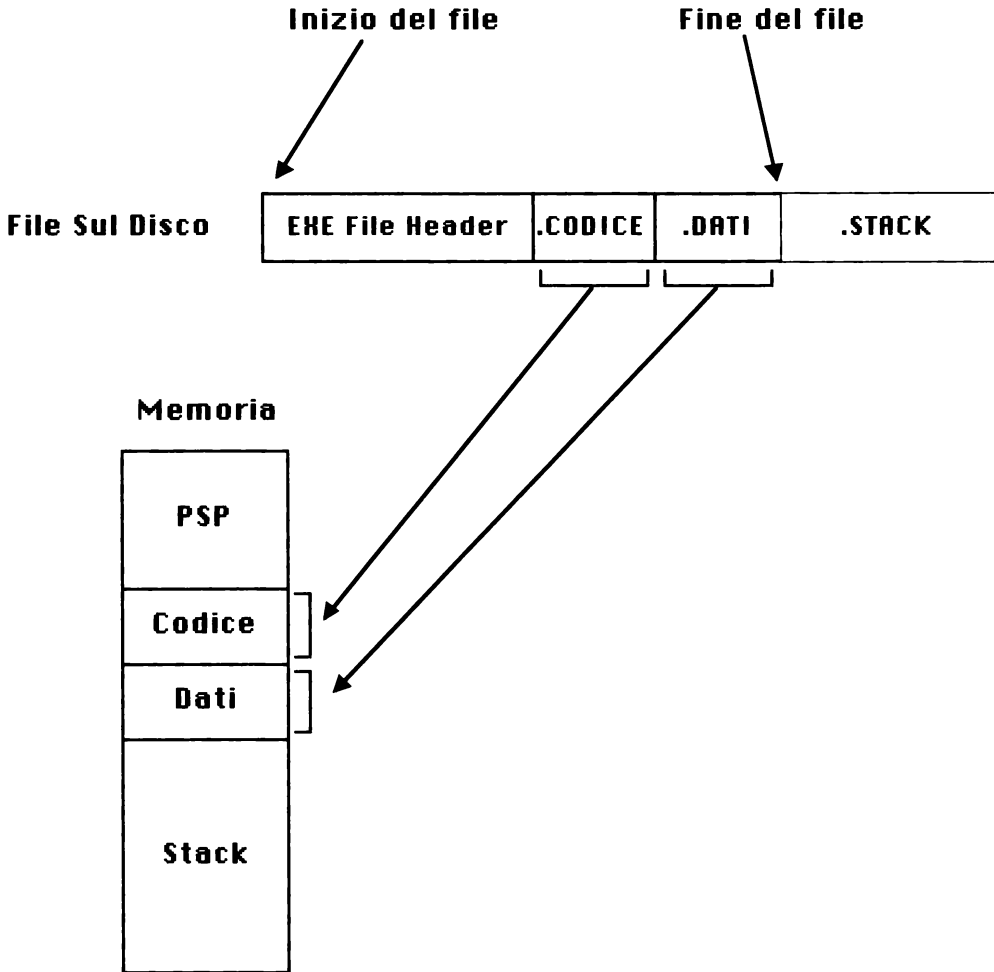


Figura 14-3. Il segmento di stack non usa spazio su disco

Questo raggruppamento dello stack e dei dati in un solo segmento viene gestito da un meccanismo dell'assemblatore chiamato *gruppi*. In particolare, l'assemblatore crea un gruppo chiamato DGROUP (che crea a sua volta un singolo segmento indipendente da tutti i segmenti usati per i dati). Avete visto precedentemente le direttive .STACK e .DATA, ma ci sono altre direttive che creano dei segmenti in questo gruppo. Fortunatamente le direttive .MODEL, .STACK e .DATA gestiscono tutto questo "dietro le quinte". Tuttavia, sapere cosa succede dietro le quinte, risulterà molto utile nel momento in cui dovrete osservare la mappa della memoria per vedere come sono stati inseriti i programmi.

Un'altra cosa che accade automaticamente, come risultato della direttiva DOSSEG, è che il segmento di stack viene caricato in memoria sopra il segmento dati. Esiste una

ragione per questo. Il segmento dati creato nell'esempio, contiene dei dati (10h, 11h, 12h e così via) che devono essere contenuti nel file .EXE in modo da poter essere copiati in memoria quando il programma viene eseguito dal DOS. Lo stack, d'altra parte, ha bisogno di spazio in memoria, ma la memoria dello stack non deve essere inizializzata (bisogna solo impostare SS:SP). Quindi, inserendo il segmento di stack dopo il segmento dati, non bisogna allocare spazio su disco per lo stack (vedi figura 14-3).

INDIRIZZAMENTO BASE-RELATIVO

E' venuto il momento di parlare dell'indirizzamento base-relativo. Le due righe:

```
SECTOR DB 10h, 11h, 12h, 13h, 14h, 15h, 16h 17h
        DB 18h, 19h, 1Ah, 1Bh, 1Ch, 1Dh, 1Eh, 1Fh
```

impostano 16 byte di memoria nel segmento dati, partendo da SECTOR, che l'assemblatore converte nell'indirizzo DB (che sta per *Define Byte*, Definisce Byte). I numeri dopo DB sono valori iniziali. Quindi, nel momento in cui eseguite DISP_SEC, la sezione di memoria che inizia in SECTOR conterrà i valori 10h, 11h, 12h e così via. Se avete scritto:

```
MOV     DL, SECTOR
```

l'istruzione avrebbe spostato il primo byte (10h) nel registro DL. Questo è conosciuto come indirizzamento di memoria *diretto*. Ma non avete scritto questo; avete invece inserito [BX] dopo SECTOR. Questa sembra assomigliare a un indice di un vettore in linguaggio BASIC:

```
K = L(10)
```

che sposta il decimo elemento di L in K.

Infatti, questa istruzione MOV opera nello stesso modo. Il registro BX contiene la *distanza* in memoria da SECTOR. Quindi, se BX è uguale a zero, MOV DL, SECTOR[BX] sposta il primo byte (10h) in DL. Se BX è uguale a 0Ah, questa istruzione MOV sposta l'undicesimo byte (1Ah, ricordatevi che si parte da zero) in DL.

Per contro, l'istruzione MOV DL, SECTOR[BX] sposterebbe la sesta parola in DX, dato che una distanza di 10 byte equivale a 5 parole, e la prima parola è a una distanza di zero. (Per i più interessati: l'ultima istruzione MOV non è corretta, dato che SECTOR è un'etichetta per i byte, mentre DX è un registro per le parole. Avreste dovuto scrivere MOV DX, Word Ptr SECTOR[BX] per indicare all'assemblatore che volete usare SECTOR come etichetta per le parole).

Ci sono molti altri modi di indirizzamento; alcuni li incontrerete successivamente, ma altri mai. Tutti i modi di indirizzamento disponibili, sono riassunti nella tabella seguente.

Tabella 14-1. I modi di indirizzamento

Modo di Indirizzamento	Formato dell'Indirizzo	Registro Segmento Usato
Registro	registro (come AX)	Nessuno
Immediato	dati(come 12345)	Nessuno

Modi di Indirizzamento di Memoria

Registro Indiretto	[BX]	DS
	[BP]	SS
	[DI]	DS
	[SI]	DS
Base Relativo*	etichetta[BX]	DS
	etichetta[BP]	SS
Diretto Indicizzato*	etichetta[DI]	DS
	etichetta[SI]	DS
Base Indicizzato*	etichetta[BX+SI]	DS
	etichetta[BX+DI]	DS
	etichetta[BP+SI]	SS
	etichetta[BP+DI]	SS

Comandi Stringa:
(MOVSW, LODSP, e così via)

Legge da DS:SI
Scriva in ES:DI

*Etichetta[...] può essere sostituita con [disp+...], dove *disp* è uno spostamento. Quindi, si potrebbe scrivere [10+BX] e l'indirizzo sarebbe 10+BX.

IMPOSTAZIONE DI DS

C'è un piccolo dettaglio che abbiamo tralasciato. Nel capitolo 11, avrete notato che i registri DS e ES puntano entrambi al PSP (e non al segmento dati), quando il DOS inizia un programma. Come è possibile impostare DS in modo che punti al segmento dati? Usando le prime due righe di DISP_LINE:

```
MOV     AX, DGROUP      ;Inserisce il segmento dati in AX
MOV     DS, AX          ;Imposta DS per puntare i dati
```

Queste due righe impostano il registro DS in modo che punti al segmento dati. La prima riga sposta l'indirizzo del segmento per il gruppo di dati (chiamato DGROUP)

che contiene .DATA e .STACK nel registro AX. La seconda riga imposta DS in modo da farlo puntare al segmento dati.

Ma c'è ancora un punto oscuro. Se vi ricordate, avevamo detto che il segmento usato nei programmi dipende dalla quantità di memoria già in uso. In altre parole, non è possibile sapere il valore di DGROUP fino a quando il DOS carica il programma in memoria. Come si può quindi sapere quale numero caricare in AX?

Una breve intestazione all'inizio di ciascun file .EXE, contiene una lista di indirizzi che devono essere calcolati. Il DOS usa queste informazioni per calcolare il valore di DGROUP e aggiornare il valore nell'istruzione MOV AX,DGROUP quando viene caricato in memoria il programma DISP_SEC.EXE. Questo processo è conosciuto come rilocazione, e lo vedrete approfonditamente nel capitolo 28.

C'è un ultimo punto da mettere a fuoco prima di proseguire. Notate che il valore del registro DS viene impostato con due istruzioni, invece della singola istruzione:

```
MOVE          DS,DGROUP
```

Perché sono necessarie due istruzioni? Dato che non è possibile spostare un numero direttamente in un registro di segmento nell'8088, è necessario spostare prima il numero di segmento nel registro AX. La richiesta di due istruzioni, invece di una, ha semplificato il progetto del microprocessore 8088 che è risultato più economico ma anche più difficile da programmare.

AGGIUNGERE DEI CARATTERI ALLA STAMPA

Avete quasi finito di scrivere la procedura per creare una visualizzazione simile a quella che si ottiene con il comando Dump di Debug. Poco fa, avete visualizzato dei numeri esadecimali su una riga. Ecco ora la nuova versione di DISP_LINE (da inserire in DISP_SEC.ASM) a cui è stato aggiunto un secondo ciclo:

Listato 14-2. Modifiche a DISP_LINE in DISP_SEC.ASM

```
DISP_LINE      PROC
                MOV     AX,DGROUP      ;Inserisce il segmento dati in AX
                MOV     DS,AX          ;Imposta DS per puntare i dati

                XOR     BX,BX           ;Imposta BX a zero
                MOV     CX,16          ;Visualizza i 16 byte

HEX_LOOP:
                MOV     DL,SECTOR[BX]  ;Preleva 1 byte
                CALL    WRITE_HEX      ;Visualizza il byte in esadecimale
                MOV     DL,' '         ;Scrive uno spazio tra i numeri
                CALL    WRITE_CHAR
                INC     BX
                LOOP    HEX_LOOP
```

```

MOV DL, ' ' ;Aggiunge un altro spazio tra i numeri
CALL WRITE_CHAR
MOV CX,16
XOR BX,BX ;Imposta nuovamente BX a zero
ASCII_LOOP:
MOV DL,SECTOR
CALL WRITE_CHAR
INC BX
LOOP ASCII_LOOP

MOV AH,4Ch ;Funzione per tornare al DOS
INT 21h
DISP_LINE ENDP

```

Assemblete DISP_SEC.ASM, collegatelo a VIDEO_IO e provate il programma. Dovreste ottenere una visualizzazione simile a quella mostrata in figura 14-4. Provate a cambiare i dati in modo da includere 0Dh o 0Ah. Vedrete qualcosa di strano. Questo succede perché 0Ah e 0Dh sono i codici dell'avanzamento riga e del ritorno a capo. Il DOS interpreta questi codici come comandi per spostare il cursore invece di visualizzarli come semplici caratteri. Per poter stampare *tutti* i caratteri disponibili, dovrete modificare la procedura WRITE_CHAR. Effettuerete queste modifiche nella terza parte del libro; per ora modificate leggermente WRITE_CHAR in modo che visualizzi un punto al posto dei caratteri compresi tra 0 e 1Fh:

```

A>disp_sec
10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F  ▶◀!!¶§_↑↑↓→+L+ ▲▼
A>

```

Figura 14-4. L'output di DISP_LINE

```

A>disp_sec
10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F .....
A>

```

Figura 14-5. La versione modificata di DISP_LINE

Sostituite WRITE_CHAR in VIDEO_IO.ASM con questa nuova procedura:

Listato 14-3. Una nuova procedura WRITE_CHAR in VIDEO_IO.ASM

```

PUBLIC WRITE_CHAR
;-----;
; Questa procedura visualizza un carattere sullo schermo usando una ;
; funzione del DOS. WRITE_CHAR sostituisce i caratteri compresi tra 0 ;
; e 1Fh con un punto. ;
; ;
; Inserimento: DL Byte da visualizzare sullo schermo. ;
;-----;
WRITE_CHAR PROC
    PUSH AX
    PUSH DX
    CMP DL,32 ;E' un carattere prima di uno spazio?
    JAE IS_PRINTABLE ;No, quindi visualizzalo
    MOV DL,'.' ;Si, sostituiscilo con un punto
IS_PRINTABLE:
    MOV AH,2 ;Funzione per la visualizzazione del carattere
    INT 21h ;Visualizza il carattere nel registro DL
    POP DX ;Ripristina il vecchio valore in DX e AX
    POP AX
    RET
WRITE_CHAR ENDP

```

Provate questa nuova procedura con DISP_SEC e provate i vari caratteri in modo da controllare tutte le condizioni di limite.

VISUALIZZARE 256 BYTE DI MEMORIA

Ora che avete visualizzato una riga (o 16 byte di memoria), il prossimo passo sarà quello di visualizzare 256 byte. Questi 256 byte corrispondono esattamente a mezzo settore. Lavorerete quindi per costruire un programma che visualizzi mezzo settore di un disco.

Dovete aggiungere due nuove procedure e modificare DISP_LINE. Le nuove procedure sono DISP_HALF_SECTOR, che servirà per visualizzare 256 byte di un settore, e SEND_CRLF che invierà il cursore all'inizio della riga successiva; CRLF sta per *Carriage Return-Line Feed* (Ritorno a Capo e Avanzamento Riga) e sono la coppia di caratteri necessari per spostare il cursore sulla riga successiva.

SEND_CRLF è molto semplice; iniziate quindi con questa. Inserite la procedura seguente in un file chiamato CURSOR.ASM:

Listato 14-4. Il nuovo file CURSOR.ASM

```

CR      EQU    13          ;Ritorno a Capo
LF      EQU    10          ;Avanzamento Riga

.MODEL  SMALL
.CODE

        PUBLIC SEND_CRLF
;-----;
; Questa routine invia allo schermo semplicemente una coppia di ritorno ;
; a capo e avanzamento riga, utilizzando le routine del DOS in modo ;
; da garantire uno scorrimento corretto. ;
;-----;
SEND_CRLF PROC
        PUSH  AX
        PUSH  DX
        MOV   AH,2
        MOV   DL,CR
        INT   21h
        MOV   DL,LF
        INT   21h
        POP   DX
        POP   AX
        RET
SEND_CRLF ENDP

        END

```

Questa procedura invia una coppia di ritorno a capo e avanzamento riga, usando la funzione 2 del DOS per inviare i caratteri. L'enunciato:

```
CR      EQU    13      ;Ritorno a capo
```

utilizza la direttiva EQU per definire il nome CR uguale a 13. Quindi, l'istruzione MOV DL,CR è equivalente a MOV DL,13. Come mostrato nella figura 14-6, l'assemblatore inserisce un 13 ogniqualvolta incontra CR. Analogamente, viene inserito un 10 al posto di LF.

Nota: Da questo momento in avanti, sarà usato un colore per mostrare i cambiamenti fatti in un programma, in modo che non dovrete controllare ciascuna riga per vedere se è stata modificata. Le aggiunte al programma saranno mostrate in grassetto, e il testo da cancellare verrà stampato barrato :

Righe nuove o modificate

~~Testo da cancellare~~

```

CR EQU 13
.
.
.
MOD DI,CR 13

```

Figura 14-6. La direttiva EQU permette di usare i nomi al posto dei numeri

Il file DISP_SEC richiede ancora molto lavoro. Ecco la nuova versione di DISP_SEC.ASM.

Listato 14-5. La nuova versione di DISP_SEC.ASM

```

DOSSEG
.MODEL    SMALL

.STACK

.DATA

SECTOR   DB    10h, 11h, 12h, 13h, 14h, 15h, 16h 17h
          DB    18h, 19h, 1Ah, 1Bh, 1Ch, 1Dh, 1Eh, 1Fh
SECTOR  DB    16 DUP (10h)
          DB    16 DUP (11h)
          DB    16 DUP (12h)
          DB    16 DUP (13h)
          DB    16 DUP (14h)
          DB    16 DUP (15h)
          DB    16 DUP (16h)
          DB    16 DUP (17h)
          DB    16 DUP (18h)
          DB    16 DUP (19h)
          DB    16 DUP (1Ah)
          DB    16 DUP (1Bh)
          DB    16 DUP (1Ch)
          DB    16 DUP (1Dh)
          DB    16 DUP (1Eh)
          DB    16 DUP (1Fh)

.CODE

        PUBLIC DISP_HALF_SECTOR
        EXTRN SEND_CRLF:PROC

;-----;
; Questa procedura visualizza mezzo settore (256 byte) ;
; ; ;
; Usa:          DISP_LINE, SEND_CRLF ;
;-----;

```

```

DISP_HALF_SECTOR PROC
    MOV     AX,DGROUP           ;Inserisce il segmento dati in AX
    MOV     DS,AX              ;Imposta DS per puntare i dati

    XOR     DX,DX              ;Comincia all'inizio di SECTOR
    MOV     CX,16              ;Visualizza 16 righe
HALF_SECTOR:
    CALL    DISP_LINE
    CALL    SEND_CRLF
    ADD     DX,16
    LOOP   HALF_SECTOR

    MOV     AH,4Ch             ;Ritorna al DOS
    INT     21h
DISP_HALF_SECTOR ENDP

        PUBLIC    DISP_LINE
        EXTRN    WRITE_HEX:PROC
        EXTRN    WRITE_CHAR:PROC

;-----;
; Questa procedura visualizza una riga di dati, o 16 byte, prima in ;
; esadecimale e poi in ASCII. ;
; ;
; Inserimento: DS:DX Distanza in SECTOR, in byte. ;
; ;
; Usa:         WRITE_CHAR, WRITE_HEX ;
; Legge:      SECTOR ;
;-----;

DISP_LINE PROC
    MOV     AX,DGROUP           ;Inserisce il segmento dati in AX
    MOV     DS,AX              ;Imposta DS per puntare i dati

    XOR     BX,BX
    PUSH    BX
    PUSH    CX
    PUSH    DX
    MOV     BX,DX              ;La distanza è più utile in BX
    MOV     CX,16              ;Visualizza 16 byte
    PUSH    BX                 ;Salva la distanza per ASCII_LOOP
HEX_LOOP:
    MOV     DL,SECTOR[BX]      ;Preleva 1 byte
    CALL    WRITE_HEX          ;Visualizza il byte in esadecimale
    MOV     DL,' '             ;Scrive uno spazio tra i numeri
    CALL    WRITE_CHAR
    INC     BX
    LOOP   HEX_LOOP

    MOV     DL,' '             ;Aggiunge un altro spazio prima dei caratteri
    CALL    WRITE_CHAR
    MOV     CX,16
    POP     BX                 ;Riporta la distanza in SECTOR
    XOR     BX,BX

```

```

ASCII_LOOP:
    MOV     DL, SECTOR[BX]
    CALL   WRITE_CHAR
    INC    BX
    LOOP   ASCII_LOOP

    POP    DX
    POP    CX
    POP    BX
    RET

    MOV    AH, 4Ch          ;Funzione per tornare al DOS
    INT    21h
DISP_LINE ENDP

    END    DISP_HALF_SECTOR

```

Le modifiche sono abbastanza ovvie. In DISP_LINE avete aggiunto PUSH BX e POP BX agli estremi di HEX_LOOP, dato che è necessario usare nuovamente lo scarto (offset) iniziale in ASCII_LOOP. Avete anche aggiunto delle istruzioni PUSH e POP per salvare e ripristinare tutti i registri usati all'interno di DISP_LINE. Ora, DISP_LINE è quasi terminata; le uniche modifiche da effettuare le farete successivamente e riguardano l'aspetto estetico (aggiungere degli spazi e dei caratteri grafici in modo da avere una visualizzazione migliore).

Quando usate LINK, ricordatevi che ora avete tre file: DISP_SEC, VIDEO_IO e CURSOR. DISP_SEC deve essere il primo file nella lista. Dovreste ottenere una visualizzazione simile a quella mostrata in figura 14-7, nel momento in cui eseguite DISP.SEC.EXE.

Ci sono ancora molti file da costruire, ma ora avventuratevi nel prossimo capitolo in cui scriverete una procedura per leggere un settore direttamente dal disco.

SOMMARIO

Avete imparato i vari modi di indirizzamento disponibili con l'8088, e avete usato l'indirizzamento di memoria indiretto per leggere 16 byte dalla memoria.

Avete usato questo tipo di indirizzamento anche in molti programmi scritti in questo capitolo, iniziando dal programma per visualizzare 16 numeri in esadecimale sullo schermo. Questi 16 numeri sono stati prelevati da un'area di memoria chiamata SECTOR, che successivamente è stata ingrandita per permettere la visualizzazione di 256 byte (mezzo settore).

Alla fine, avete messo a punto il programma DISP_SEC che utilizzerete nei capitoli successivi.

```
A>disp_sec
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 .....
11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 .....
12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 .....
13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 .....
14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 .....
15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 .....
16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 .....
17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 .....
18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 .....
19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 .....
1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A .....
1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B .....
1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C .....
1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D .....
1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E .....
1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F .....
```

A>

Figura 14-7. L'output da disp_sec

VISUALIZZARE UN SETTORE DEL DISCO

Ora che avete un programma che visualizza 256 byte di memoria, potete aggiungere una procedura che legga un settore del disco e lo inserisca in memoria partendo da SECTOR. Quindi, la procedura di visualizzazione appena scritta sarà in grado di mostrare la prima metà di un settore del disco.

SEMPLIFICARE IL LAVORO

Con i tre file sorgente creati nei capitoli precedenti, il processo richiesto per collegarli comincia a diventare un po' più complesso. Nell'ultimo capitolo avrete probabilmente assemblato tutti e tre i file, senza controllare quali erano stati modificati.

Assemblare tutti i file sorgente quando ne è stato modificato solamente uno, causa un'inutile perdita di tempo soprattutto se il programma è molto esteso. Quello che si dovrebbe fare, è assemblare solamente i file che sono stati modificati.

Fortunatamente, tutti gli assembleri trattati in questo libro (MASM, Turbo Assembler e OPTASM) permettono di effettuare questa operazione; la Borland e la Microsoft forniscono un programma chiamato Make, mentre OPTASM ha questa funzione incorporata (che sarà descritta alla fine della sezione successiva). Per poter utilizzare Make, creerete un file (chiamato Makefile) che indicherà a Make come effettuare il lavoro. Dovrete digitare semplicemente:

```
A>MAKE MAKEFILE
```

Nota: Se usate Make della Borland, dovete digitare solo MAKE per assemblare solamente i file che sono stati modificati.

Il file che creerete (MAKEFILE) indicherà a MAKE quali file dipendono da altri file. Ogni volta che sarà cambiato un file, il DOS aggiornerà l'ora della modifica (potete vedere questo con il comando DIR). MAKE controlla semplicemente l'ora abbinata ai file .ASM e .OBJ. Se alla versione .ASM è abbinata un'ora più recente, Make assembla nuovamente quel file.

Questo è tutto quello che si deve fare, ma occorre dare un avvertimento: MAKE funzionerà correttamente solo se imposterete la data e l'ora all'accensione del computer o se il vostro computer ha un orologio incorporato (cosa molto frequente al giorno d'oggi). Senza queste informazioni, MAKE non potrà sapere quali file sono stati modificati.

FORMATO DEL FILE MAKE

Il formato di MAKEFILE, che userete con MAKE, è molto semplice:

Listato 15-1. Il file MAKEFILE

```
disp_sec.obj:    disp_sec.asm
                masm disp_sec;

video_io.obj:   video_io.asm
                masm video_io;

cursor.obj:     cursor.asm
                masm cursor;

disp_sec.exe:   disp_sec.obj video_io.obj cursor.obj
                link disp_sec video_io cursor;
```

Nota: Se usate Make della Borland, le ultime due righe devono trovarsi all'inizio del file invece che alla fine. E' stato inserito un nome di file a sinistra dei due punti (:) e uno o più file a destra. Se uno dei qualsiasi dei file a destra (come DISP_SEC.ASM nella prima riga) è più recente del primo file (DISP_SEC.OBJ), Make eseguirà tutti i comandi rientrati contenuti nelle righe seguenti.

Se il vostro assembler fornisce il programma Make, inserite queste righe nel file Makefile (senza estensione) ed effettuate una piccola modifica a DISP_SEC.ASM. Digitate quindi:

A>**MAKE MAKEFILE**

(digitate solamente MAKE se usate Make della Borland) e vedrete apparire una visualizzazione simile alla seguente:

```
Microsoft (R) Program Maintenance Utility Version 4.06
Copyright (C) Microsoft Corp 1984-1987. All rights reserved.

    masm disp_sec;
Microsoft (R) Macro Assembler Version 5.10
Copyright (C) Microsoft Corp 1981, 1988. All rights reserved.

    49620 + 233303 Bytes symbol space free

    0 Warning Errors
    0 Severe Errors

    link disp_sec video_io cursor;

Microsoft (R) Overlay Linker Version 3.64
Copyright (C) Microsoft Corp 1983-1988. All rights reserved.

A>
```


Make ha effettuato il minimo lavoro necessario per ricostruire il programma. Se avete una versione vecchia del Macro Assembler della Microsoft che non include Make, scoprirete che questo programma vale l'aggiornamento. Riceverete anche un programma chiamato CodeView (che vedremo in seguito) che è un'ottima alternativa a Debug.

IL MAKE DI OPTASM

OPTASM della SLR System ha il programma Make incorporato nell'assemblatore. Ma a differenza del Make della Microsoft, della Borland e dell'IBM, il Make di OPTASM può assemblare solamente i file che sono stati modificati; non può eseguire il linker per costruire un nuovo file .EXE.

Il formato per il Make di OPTASM è leggermente differente dagli altri:

Listato 15-2. Il File Make MAKEFILE di OPTASM

```
disp_sec.obj disp_sec.asm
disp_sec;

video_io.obj video_io.asm
video_io;

cursor.obj cursor.asm
cursor;
```

Il file contiene il nome del file oggetto (come disp_sec.obj) seguito dal nome del file sorgente. Se uno qualsiasi dei file a destra è più recente del file oggetto (per esempio, se avete modificato disp_sec.asm nella prima riga), OPTASM assembla nuovamente il file che appare nella riga successiva. Benché sia differente il formato del file usato, il risultato è lo stesso.

Per assemblare tutti i file modificati, digitate:

```
A>OPTASM @MAKEFILE
```

Questo comando indica a OPTASM di usare le informazioni contenute in MAKEFILE per decidere quali file assemblare.

Dovrete quindi eseguire LINK per creare un nuovo file .EXE:

```
A>LINK DISP_SEC VIDEO_IO CURSOR;
```

Questo è tutto quello che bisogna fare per utilizzare il Make incorporato di OPTASM (troverete altre informazioni nel manuale di OPTASM). Torniamo ora a Dskpatch.

MODIFICARE DISP_SEC

DISP_SEC, come lo avete lasciato, includeva una versione di DISP_HALF_SECTOR che fungeva da procedura di verifica e procedura principale. Cambiate ora DISP_HALF_SECTOR in procedura ordinaria in modo che possa essere chiamata da una nuova procedura che creerete: READ_SECTOR. La procedura di verifica si troverà in DISK_IO.

Modificate innanzitutto DISP_SEC in modo da renderlo un file di procedure (come avevate fatto per VIDEO_IO). Cambiate l'enunciato END DISP_HALF_SECTOR in un semplice END, dato che la procedura principale si troverà ora in DISK_IO. Rimovete quindi le direttive .STACK e DOSSEG all'inizio di DISP_SEC.ASM (dato che le sposterete in un altro file).

Ora, dato che verrà letto un settore in memoria partendo da SECTOR, non è più necessario fornire dei dati. E' quindi possibile sostituire i 16 enunciati DB con una sola riga:

```
SECTOR DB 8192 DUP (0)
```

che riserva 8192 byte per contenere un settore.

Precedentemente, quando avete caricato un settore, avete visto che era lungo 512 byte; perché quindi è stata riservata un'area di memoria così grossa? Perché alcuni dischi fissi (da 300 megabyte per esempio) utilizzano dei settori di dimensione molto più estesa. Anche se non sono molto comuni, per sicurezza è meglio predisporre un'area di memoria adeguata in modo da evitare di leggere un settore che non possa essere contenuto in SECTOR. Nel resto del libro, si assumerà che tutti i settori siano lunghi 512 byte (ad eccezione di SECTOR).

Quello di cui avete bisogno ora, è una nuova versione di DISP_HALF_SECTOR. La versione precedente non ha niente di più che una procedura di verifica usata per controllare DISP_LINE. Nella nuova versione, inserirete un offset (distanza) nel settore in modo da poter visualizzare 256 byte partendo da qualsiasi posizione. Tra le altre cose, questo significa che sarà possibile visualizzare la prima metà, la seconda metà, o la parte centrale di un settore. Anche in questo caso, inserite la distanza in DX. Ecco la nuova versione (e anche finale) di DISP_HALF_SECTOR in DISP_SEC:

Listato 15-3. La Versione Finale di DISP_HALF_SECTOR in DISP_SEC.ASM

```

PUBLIC DISP_HALF_SECTOR
EXTRN SEND_CRLF:PROC
;-----;
; Questa procedura visualizza mezzo settore (256 byte) ;
; ;
; Inserimento: DS:DX Distanza in SECTOR, in byte - deve essere ;
; un multiplo di 16. ;
; ;
; Usa: DISP_LINE, SEND_CRLF ;
;-----;
```

```

DISP_HALF_SECTOR PROC
    MOV     AX,DGROUP           ;Inserisce il segmento dati in AX
    MOV     DS,AX              ;Imposta DS per puntare i dati

    XOR     DX,DX              ;Comincia all'inizio di SECTOR
    PUSH   CX
    PUSH   DX
    MOV     CX,16               ;Visualizza 16 righe
HALF_SECTOR:
    CALL   DISP_LINE
    CALL   SEND_CRLF
    ADD    DX,16
    LOO    HALF_SECTOR
    POP    DX
    POP    CX
    RET

    MOV     AH,4Ch             ;Ritorna al DOS
    INT    21h
DISP_HALF_SECTOR ENDP

```

Vediamo ora la procedura per leggere un settore.

LEGGERE UN SETTORE

Nella prima versione di READ_SECTOR ignorerete deliberatamente il controllo degli errori come, per esempio, l'assenza del dischetto nel drive. Questo non è il modo corretto per procedere, d'altra parte questa non è la versione finale di READ_SECTOR. Non vedrete come gestire gli errori in questo libro, ma potrete trovare delle procedure per la gestione degli errori nella versione di Dskpatch contenuta nel dischetto allegato al libro. Per ora, leggete semplicemente un settore dal disco. Questa è la versione di prova del file DISK_IO.ASM:

Listato 15-4. Il nuovo file DISK_IO.ASM

```

DOSSEG
.MODEL     SMALL

.STACK

.DATA

        EXTRN     DISP_HALF_SECTOR:PROC
;-----;
; Questa procedura legge il primo settore sul disco A e ne visualizza ;
; la prima metà. ;
;-----;

```

```

READ_SECTOR    PROC
                MOV    AX,DGROUP    ;Inserisce il segmento dati in AX
                MOV    DS,AX        ;Imposta DS per puntare ai dati

                MOV    AL,0         ;Drive A (numero 0)
                MOV    CX,1         ;Legge solo 1 settore
                MOV    DX,0         ;Legge il settore numero 0
                LEA    BX,SECTOR    ;Dove memorizzare questo settore
                INT    25h         ;Legge il settore
                POPF           ;Elimina flag inviati allo stack dal DOS
                XOR    DX,DX        ;Imposta a 0 la distanza in SECTOR
                CALL   DISP_HALF_SECTOR ;Visualizza la prima metà

                MOV    AH,4Ch       ;Ritorna al DOS
                INT    21h
READ_SECTOR    ENDP

                END    READ_SECTOR

```

Ci sono tre nuove istruzioni in questa procedura. La prima:

```
LEA    BX,SECTOR
```

sposta l'*indirizzo* di SECTOR (dall'inizio del gruppo dati DGROUP creato da .DATA) nel registro BX; LEA sta per *Load Effective Address* (Carica Indirizzo Effettivo). Dopo questa istruzione LEA, DS:BX conterrà l'indirizzo completo di SECTOR che il DOS userà per la seconda nuova istruzione (INT 25h). (LEA carica la distanza nel registro BX senza impostare DS; è compito vostro assicurarvi che DS punti al segmento dati). SECTOR non si trova nello stesso file sorgente di READ_SECTOR, ma si trova in DISP_SEC.ASM. Come si può quindi informare l'assemblatore di questa posizione? Con una direttiva EXTRN:

```
.DATA

EXTRN SECTOR:BYTE
```

Questa serie di istruzioni indicano all'assemblatore che SECTOR è stato definito nel segmento dati creato da .DATA (che viene a sua volta definito in un altro file sorgente) e che SECTOR è una variabile di byte (invece che di parole). La direttiva EXTRN sarà usata molto spesso nei capitoli successivi; questo è un modo per usare le stesse variabili in più file sorgente.

Bisogna solo stare attenti nel definire le variabili in un solo punto.

Ritornate ora all'istruzione INT 25h. INT 25h è una funzione speciale del DOS utilizzata per leggere i settori dal disco. Quando il DOS riceve una chiamata INT 25h, usa le informazioni contenute nei registri nel modo seguente:

```
AL    Numero del drive (0=A, 1=B, e così via)
CX    Numero dei settori da leggere, uno alla volta
```

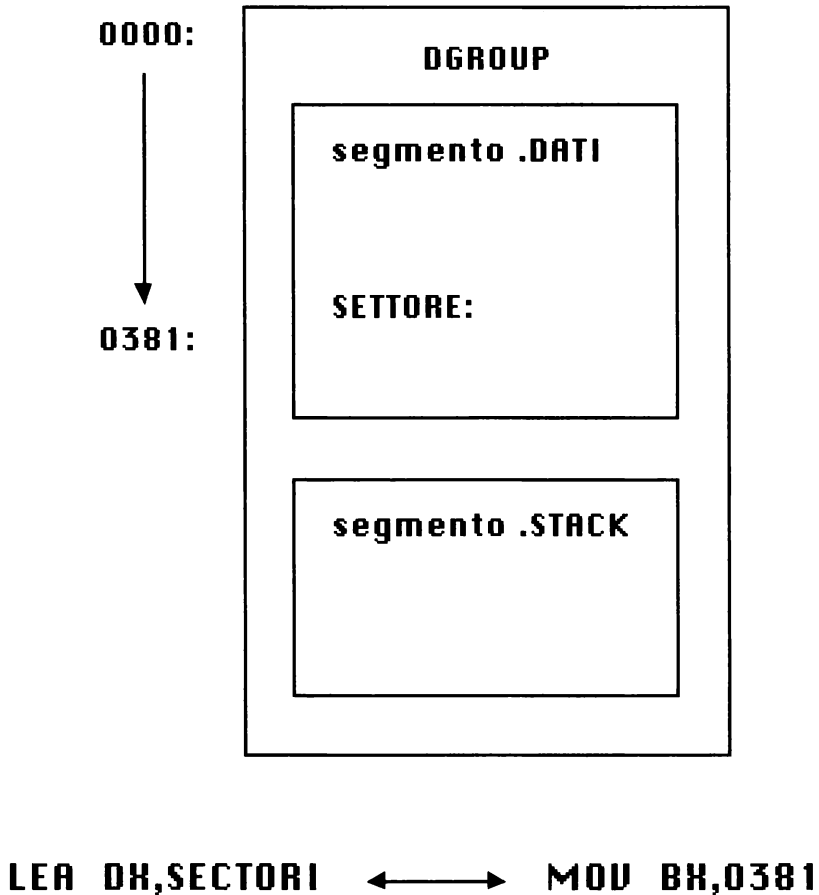


Figura 15-1. LEA carica l'indirizzo effettivo

DX Numero del primo settore da leggere (il primo settore è 0)
 DS:BX Indirizzo di trasferimento: dove devono essere inseriti i settori letti

Il numero nel registro AL determina il drive da cui il DOS deve leggere i settori. Se AL è uguale a 0, il DOS legge dal drive A.

Nota: Alcune versioni recenti del DOS (come il COMPAQ DOS 3.31 e il DOS 4.0) supportano hard disk con partizioni superiori ai 32 megabyte modificando il modo di funzionamento della chiamata INT 25h. Questo non è un problema se si leggono dei settori da un dischetto, ma lo può diventare se utilizzate Dskpatch sul disco fisso.

.DATI

EXTRN SETTORE:BYTE



**Una variabile per i byte.
LINK fornisce l'indirizzo.**

Figura 15-2. La direttiva *EXTRN*

Il DOS può leggere più di un settore con una singola chiamata, e legge il numero di settori fornito da *CX*. In questo caso, *CX* è impostato a 1, per cui il DOS leggerà solamente un settore di 512 byte.

DX è stato impostato a zero in modo che il DOS legga il primo settore del disco. Potete modificare questo numero, se lo desiderate, per leggere un settore differente.

DS:BX è l'indirizzo completo per l'area di memoria in cui si vuole inserire il settore(i) letto. In questo caso, *DS:BX* è stato impostato sull'indirizzo di *SECTOR* in modo che, chiamando *DISP_HALF_SECTOR*, verrà visualizzata la prima metà del primo settore letto dal dischetto nel drive A.

Notate infine l'istruzione *POPF* (subito dopo *INT 25h*). Come già detto, l'8088 ha un registro di stato in cui memorizza i vari flag come, per esempio, il flag zero e il flag di riporto. *POPF* è un'istruzione *POP* speciale che preleva una parola dal registro di stato. Perché è necessaria un'istruzione *POPF*?

L'istruzione *INT 25h* inserisce nello stack prima i registri di stato, e successivamente l'indirizzo di ritorno. Quando il DOS ritorna da questo *INT 25h*, lascia il registro di stato nello stack. Il DOS agisce in questo modo per poter impostare il flag di riporto nel caso si verifichi una condizione di errore. Dato che in questo programma non verrà inserita una gestione degli errori, dovete rimuovere il registro di stato dallo stack (con l'istruzione *POPF*). (Nota: *INT 25h*, insieme a *INT 24h* che *scrive* un settore su disco, sono le uniche routine del DOS che lasciano il registro di stato nello stack).

Potete ora assemblare *DISK_IO.ASM*, e riassemblare *DISP_SEC.ASM*. Quindi, collegate i quattro file *DISK_IO*, *DISP_SEC*, *VIDEO_IO* e *CURSOR* con *DISK_IO* elencato per primo. Oppure, se volete utilizzare *MAKE*, aggiungete queste due righe a *MAKEFILE*:

```
disk_io.obj:   disk_io.asm
              masm disk_io;
```

(per il *Make* di *OPTASM* dovete far rientrare la prima riga e rimuovere gli spazi in testa dalla seconda riga) e cambiate le ultime due righe (le prime due per il *Make* della Borland) in:

```
disk_io.exe:    disk_io.obj disp_sec.obj video_io.obj cursor.obj
               link disk_io disp_sec video_io cursor;
```

Dopo aver creato la versione .EXE di DISK_IO, dovrete ottenere una visualizzazione simile a quella mostrata in figura 15-3 (ricordatevi di inserire un dischetto nel drive A prima di eseguire il programma).

```
A>disk_io
EB 34 90 49 42 4D 20 20 33 2E 33 00 02 04 01 00  64EIBM .3.3....
02 00 02 EF A9 F8 2B 00 11 00 00 00 11 00 00 00  ...n°+.<...<...
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 12  .....
00 00 00 00 01 00 FA 33 C0 8E D0 BC 00 7C 16 07  ....3LÀL.1..
BB 78 00 36 C5 37 1E 56 16 53 BF 2B 7C B9 0B 00  x.6+7AV.S+|f..
FC AC 26 00 3D 00 74 03 26 8A 05 AA 8A C4 E2 F1  %&ç=.t.è.-è-Γ±
06 1F 89 47 02 C7 07 2B 7C FB CD 13 72 67 A0 10  .v8G.+.+j= rgá>
7C 98 F7 26 16 7C 03 06 1C 7C 03 06 0E 7C A3 3F  iy=&.!...!...!ú?
7C A3 37 7C B8 20 00 F7 26 11 7C 0B 1E 0B 7C 03  ú7!q .=&<!YΔ.!.
C3 48 F7 F3 01 06 37 7C BB 00 05 A1 3F 7C E8 9F  |H=<...iq..i?i8f
00 B8 01 02 E8 B3 00 72 19 0B FB B9 0B 00 BE D9  .γ..8|.rvifj..j
7D F3 A6 75 0D 0D 7F 20 BE E4 7D B9 0B 00 F3 A6  }{u.ìδ jΣ}..<u
74 18 BE 77 7D E8 6A 00 32 E4 CD 16 5E 1F 0F 04  t^jw)8j.2Σ= ^vÁ.
8F 44 02 CD 19 BE C4 7D EB EB A1 1C 05 33 D2 F7  ÁD =v^ -)δδíL 3T=
36 0B 7C FE C0 A2 3C 7C A1 37 7C A3 3D 7C BB 00  6.!L6<íí7!ú=|q.
07 A1 37 7C E8 49 00 A1 18 7C 2A 06 3B 7C 40 38  .í7!8I.í^!*.:!08

A>
```

Figura 15-3. Una schermata presa da DISK_IO.COM

LA DIRETTIVA .DATA?

Se vi ricordate, quando avete definito SECTOR in DISP_SEC, avete riservato 8192 byte di zeri. Questo significa che è stato occupato dello spazio su disco con il file DISK_IO.EXE:

```
A>DIR DISK_IO.EXE

Volume in drive A has no label
Directory of A:\

DISK_IO EXE      8922      30-01-90  12:02p
1 file(s) 207432 bytes free

A>
```

Come potete vedere, DISK_IO.EXE è lungo 8922 byte, la maggior parte dei quali sono degli zeri. E' stato occupato moltissimo spazio solamente per degli zeri, spazio che inoltre non viene utilizzato fino a quando non viene letto un settore in memoria. E' quindi necessario utilizzare così tanto spazio su disco per SECTOR? No.

Esiste un'altra direttiva, .DATA?, che permette di definire delle variabili di memoria che occupano spazio in memoria ma non sul disco. E' possibile far questo indicando all'assemblatore di non preoccuparsi di quali valori contiene una variabile di memoria. Cambiate le tre righe in DISP_SEC che definiscono SECTOR nel modo seguente:

```
.DATA?  
  
    SECTOR    DB    8192 DUP (?)
```

Sono stati fatti due cambiamenti. Innanzitutto c'è un punto di domanda (?) dopo la direttiva .DATA che indica all'assemblatore di definire delle variabili che non hanno un valore iniziale e, di conseguenza, di non occupare spazio su disco. In secondo luogo c'è un punto di domanda invece di uno zero come valore per ciascun byte in SECTOR. L'enunciato DUP(?) indica all'assemblatore di non preoccuparsi di quale valore è contenuto in ciascun byte.

Nota: Dovete definire le variabili nella sezione .DATA? con DUP(?). Se definite delle variabili con un valore (come, per esempio, VAR DB 0), o se usate VAR DB ?, l'assemblatore riserverà dello spazio nel file .EXE per *tutte* le variabili in .DATA?. In altre parole, inserite tutte le variabili con valore iniziale in .DATA, e tutte le variabili con DUP(?) in .DATA?.

Dopo aver fatto queste modifiche, ricostruite DISK_IO.EXE. Dovrebbe ora essere lungo 729 byte. La direttiva .DATA? permette di mantenere dei programmi su disco con una dimensione più contenuta.

Modificherete successivamente DISK_IO; per ora avete fatto abbastanza. Nel prossimo capitolo migliorerete la visualizzazione dei settori aggiungendo dei caratteri grafici e qualche informazione.

SOMMARIO

Ora che avete quattro file sorgente differenti, Dskpatch comincia a prendere una forma. In questo capitolo avete visto come funziona il programma Make, che rende più semplice l'assemblaggio dei programmi.

Avete scritto una nuova procedura, READ_SECTOR, in un file diverso da quello che contiene SECTOR; per questo motivo avete usato la direttiva EXTRN in DISK_IO, in modo da indicare all'assemblatore che SECTOR si trova in un altro file.

Avete imparato l'istruzione LEA che vi ha permesso di caricare l'indirizzo di SECTOR nel registro BX.

DISK_IO usa una nuova funzione dell'istruzione INT (la numero 25h) per leggere i

settori dal disco. Avete usato INT 25h per leggere un settore in una variabile di memoria, SECTOR, che avete visualizzato con DISP_HALF_SECTOR. Avete anche imparato l'istruzione POPF che preleva una parola (il registro di stato) dallo stack. Avete usato questa istruzione per rimuovere i flag che il DOS non rimuove dallo stack dopo essere ritornato dalla chiamata INT 25h. Nel prossimo capitolo userete alcuni dei caratteri grafici disponibili sull'IBM per migliorare esteticamente la visualizzazione dei settori.

MIGLIORARE LA VISUALIZZAZIONE DEI SETTORI

Siete arrivati all'ultimo capitolo della seconda parte del libro. Tutto quello che avete imparato finora, può essere utilizzato sui sistemi MS-DOS con microprocessore 8088 (o 8086, 80286 e così via). Nella terza parte inizierete a scrivere delle procedure per lavorare direttamente con le routine del BIOS.

Ma prima di addentrarvi in questi argomenti, aggiungerete alcune procedure al file VIDEO_IO. Modificherete anche DISP_LINE in DISP_SEC. Tutte queste modifiche e aggiunte serviranno per migliorare la visualizzazione dal punto di vista estetico e consisteranno nell'aggiungere dei caratteri grafici e del testo.

AGGIUNGERE DEI CARATTERI GRAFICI

Il Personal Computer IBM offre una serie di caratteri grafici che permettono di tracciare dei riquadri e delle linee nella varie parti dello schermo. In questo paragrafo imparerete a tracciare due riquadri: il primo intorno ai numeri esadecimali visualizzati, e l'altro intorno ai caratteri ASCII. Questa modifica richiede un piccolissimo sforzo. Inserite le seguenti definizioni all'inizio del file DISP_SEC.ASM, tra la direttiva .MODEL e la direttiva .DATA?, lasciando una o due righe vuote sopra e sotto:

Listato 16-1. *Aggiunta al File DISP_SEC.ASM*

```

;-----;
; Caratteri grafici per la cornice del settore. ;
;-----;
VERTICAL_BAR EQU 0BAh
HORIZONTAL_BAR EQU 0CDh
UPPER_LEFT EQU 0C9h
UPPER_RIGHT EQU 0BBh
LOWER_LEFT EQU 0C8h
LOWER_RIGHT EQU 0BCh
TOP_T_BAR EQU 0CBh
BOTTOM_T_BAR EQU 0CAh
TOP_TICK EQU 0D1h
BOTTOM_TICK EQU 0CFh

```

Queste sono le definizioni per i caratteri grafici. Notate che è stato inserito uno zero prima di ciascun numero esadecimale in modo che l'assemblatore possa riconoscerli

come numeri invece che come etichette.

Avreste potuto scrivere semplicemente i numeri invece delle definizioni, ma in questo modo la procedura risulta più chiara e leggibile. Per esempio, confrontate le istruzioni seguenti:

```
MOV DL,VERTICAL_BAR
MOV DL,0BAh
```

La prima istruzione è senz'altro più chiara.

Ecco ora la nuova procedura DISP_LINE, utilizzata per dividere le varie parti della visualizzazione di un settore. In questa procedura viene usato il carattere VERTICAL_BAR il cui codice è 186 (0BAh). Come di consueto, le nuove righe saranno presentate su uno sfondo grigio:

Listato 16-2. Modifiche a DISP_LINE in DISP_SEC.ASM

```
DISP_LINE PROC
    PUSH BX
    PUSH CX
    PUSH DX
    MOV BX,DX ;La distanza è più utile in BX
                ;Scrive separatore
    MOV DL,' '
    CALL WRITE_CHAR
    MOV DL,VERTICAL_BAR ;Traccia il lato sinistro del riquadro
    CALL WRITE_CHAR
    MOV DL,' '
    CALL WRITE_CHAR
                ;Scrive ora 16 byte
    MOV CX,16 ;Visualizza 16 byte
    PUSH BX ;Salva la distanza per ASCII_LOOP
HEX_LOOP:
    MOV DL,SECTOR[BX] ;Preleva 1 byte
    CALL WRITE_HEX ;Visualizza il byte in esadecimale
    MOV DL,' ' ;Scrive uno spazio tra i numeri
    CALL WRITE_CHAR
    INC BX
    LOOP HEX_LOOP
    MOV DL,' ' ;Scrive separatore
    CALL WRITE_CHAR
    MOV DL,' ' ;Aggiunge un altro spazio prima dei caratteri
    CALL WRITE_CHAR
    MOV CX,16
    POP BX ;Riporta la distanza in SECTOR
ASCII_LOOP:
    MOV DL,SECTOR[BX]
    CALL WRITE_CHAR
    INC BX
    LOOP ASCII_LOOP
```

```

MOV     DL, ' '           ;Traccia il lato destro del riquadro
CALL    WRITE_CHAR
MOV     DL, VERTICAL_BAR
CALL    WRITE_CHAR

POP     DX
POP     CX
POP     BX
RET

DISP_LINE     ENDP
    
```

Assemblete questa nuova versione di DISP_SEC e collegate i quattro file (ricordatevi di inserire DISK_IO per primo nell'elenco dopo il comando LINK). Dovreste vedere tre linee doppie che separano le due parti del settore visualizzato (vedi figura 16.1).

AGGIUNGERE GLI INDIRIZZI ALLA VISUALIZZAZIONE

Provate ora qualcosa di più complesso: aggiungete gli indirizzi esadecimali sul lato sinistro dello schermo. Questi numeri devono rappresentare la distanza dall'inizio del settore, in modo che il primo numero sia 00, il successivo 10, quindi 20 e così via.

```

A>disk_io
EB 34 90 49 42 4D 20 20 33 2E 33 00 02 04 01 00  64ÉIBM .3.3....
02 00 02 EF A9 F8 2B 00 11 00 00 00 11 00 00 00  ...n°+.<...<...
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 12  ....
00 00 00 00 01 00 FA 33 C0 0E D0 BC 00 7C 16 07  .....3LÄJ...
BB 78 00 36 C5 37 1E 56 16 53 BF 2B 7C B9 0B 00  x.6+7AU.S+!J..
FC AC 26 00 3D 00 74 03 26 8A 05 AA 0A C4 E2 F1  %&G=.t.èè.-è-f±
06 1F 89 47 02 C7 07 2B 7C FB CD 13 72 67 A0 10  .vèG.+.!J= rgá)
7C 98 F7 26 16 7C 03 06 1C 7C 03 06 0E 7C A3 3F  iÿ=a...l...ú?
7C A3 37 7C B8 20 00 F7 26 11 7C 0B 1E 0B 7C 03  iú7iq .#&<|YA.!.
C3 48 F7 F3 01 06 37 7C BB 00 05 A1 3F 7C E8 9F  |H=#...iÿ...i?iÿf
00 00 01 02 E8 B3 00 72 19 0B FB B9 0B 00 BE D9  .j..è|.rvÿiÿf...j
7D F3 A6 75 0D 0D 7F 20 BE E4 7D B9 0B 00 F3 A6  }<èu.ia jΣ}...èè
74 18 BE 77 7D E8 6A 00 32 E4 CD 16 5E 1F 0F 04  t^jwÿj.JΣ=^vâ.
0F 44 02 CD 19 BE C4 7D EB EB A1 1C 05 33 D2 F7  ÅD =j-}èèiL 3ÿ=
36 0B 7C FE C0 A2 3C 7C A1 37 7C A3 3D 7C BB 00  6.!=Ló<|i7iú=iÿ.
07 A1 37 7C E8 49 00 A1 18 7C 2A 06 3B 7C 40 38  .i7iúI.i^!*.;i00
    
```

A>_

Figura 16-1. Aggiunta di barre verticali

Il processo è molto semplice, dato che avete già scritto la procedura WRITE_HEX per visualizzare dei numeri in esadecimale. C'è tuttavia un problema con i settori lunghi 512 byte: WRITE_HEX visualizza solo dei numeri esadecimali composti da due cifre, mentre è necessario visualizzarne alcuni di tre cifre (quelli superiori a 255).

Ecco la soluzione. Dato che questi numeri sono compresi tra zero e 511 (da 0h a 1FFh), la prima cifra sarà uno spazio se il numero è minore di 100h, o un uno (1) nel caso contrario. Quindi, se il numero da visualizzare è superiore a 255, scriverete semplicemente un 1 seguito dal numero esadecimale che rappresenta il byte basso. Queste sono le modifiche da fare a DISP_LINE, per poter visualizzare un numero composto da tre cifre:

Listato 16-3. Aggiunte a DISP_LINE in DISP_SEC.ASM

```

DISP_LINE PROC
    PUSH    BX
    PUSH    CX
    PUSH    DX
    MOV     BX,DX                ;La distanza è più utile in BX
    MOV     DL,' '              ;Scrive la distanza in esadecimale
                                ;La prima cifra è 1?
    CMP     BX,100h
    JB     WRITE_ONE           ;No, uno spazio è già in DL
    MOV     DL,'1'             ;Sì, allora inserisci 1 in DL per l'output
WRITE_ONE:
    CALL    WRITE_CHAR
    MOV     DL,BL               ;Copia il byte basso in DL per l'output HEX
    CALL    WRITE_HEX
                                ;Scrive separatore
    MOV     DL,' '
    CALL    WRITE_CHAR
    MOV     DL,VERTICAL_BAR     ;Traccia il lato sinistro del riquadro
    .
    .
    .

```

Potete vedere il risultato nella figura 16-2.

Vi state avvicinando alla versione finale. Potete ora notare che ciò che viene visualizzato non è centrato sul video. Dovreste spostare tutto di circa tre spazi verso destra. Fate quest'ultima modifica a DISP_LINE.

Potreste effettuare questa modifica chiamando WRITE_CHAR tre volte, ma non è il modo migliore. Aggiungete invece un'altra procedura, chiamata WRITE_CHAR_N_TIMES a VIDEO_IO. Come potrete capire dal nome, questa procedura scrive un determinato carattere N volte. In questo modo, inserendo il numero N nel registro CX e il codice del carattere in DL, la procedura WRITE_CHAR_N_TIMES scrive N copie del carattere il cui codice si trova in DL. Quindi, grazie a questa procedura, sarete in grado di scrivere tre spazi inserendo 3 in CX e 20h (il codice ASCII dello spazio) in DL.

```

A>disk_io
00 EB 34 90 49 42 4D 20 20 33 2E 33 00 02 04 01 00 | 64ÉIBM .3.3....
10 02 00 02 EF A9 F8 2B 00 11 00 08 00 11 00 00 00 | ...nr°+.<...<...
20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 12 | .....
30 00 00 00 00 01 00 FA 33 C0 8E D0 BC 00 7C 16 07 | .....3LÀL.!.
40 BB 78 00 36 C5 37 1E 56 16 53 BF 2B 7C B9 0B 00 | x.6+7AU.S+!q..
50 FC AC 26 00 3D 00 74 03 26 8A 05 AA 8A C4 E2 F1 | %AQ=.t.&è.è-±
60 06 1F 09 47 02 C7 07 2B 7C FB CD 13 72 67 A0 10 | .vèG.+.+|=rgá>
70 7C 98 F7 26 16 7C 03 06 1C 7C 03 06 0E 7C A3 3F | !ù=à.!.!.!ú?
80 7C A3 37 7C B8 20 00 F7 26 11 7C 0B 1E 0B 7C 03 | !ú7!q .=&<!iA.!.
90 C3 48 F7 F3 01 06 37 7C BB 00 05 A1 3F 7C E8 9F | |H=ç...!q.?!òf
A0 00 B8 01 02 E8 B3 00 72 19 0B FB B9 0B 00 BE D9 | .q.è|.rv!f|.d
B0 7D F3 A6 75 0D 0D 7F 20 BE E4 7D B9 0B 00 F3 A6 | }çªu.ìª dΣ}..çª
C0 74 18 BE 77 7D E8 6A 00 32 E4 CD 16 5E 1F 8F 04 | t^d_w}èj.2Σ= ^vA.
D0 8F 44 02 CD 19 BE C4 7D EB EB A1 1C 05 33 D2 F7 | ÅD =v-}èèíL 3qz
E0 36 0B 7C FE C0 A2 3C 7C A1 37 7C A3 3D 7C BB 00 | 6.!=L6<!í7!ú=iq.
F0 07 A1 37 7C E8 49 00 A1 18 7C 2A 06 3B 7C 40 38 | .!7!èI.í^!*.;!èB

```

A>

Figura 16-2. Aggiunta di numeri

Ecco la procedura da aggiungere a VIDEO_IO.ASM:

Listato 16-4. La procedura da aggiungere a VIDEO_IO.ASM

```

PUBLIC WRITE_CHAR_N_TIMES
;-----;
; Questa procedura scrive più di una copia di un carattere ;
; ;
; Inserimento: DL Codice carattere ;
; CX Numero di copie del carattere ;
; ;
; Usa: WRITE_CHAR ;
;-----;
WRITE_CHAR_N_TIMES PROC
PUSH CX
N_TIMES:
CALL WRITE_CHAR
LOOP N_TIMES
POP CX
RET
WRITE_CHAR_N_TIMES ENDP

```

Potete vedere come è semplice questa procedura, dato che avete già creato WRITE_CHAR. Se vi state chiedendo perché dovete scrivere una procedura per qualcosa di così semplice, la risposta è immediata: il programma Dskpatch risulterà!

molto più chiaro chiamando `WRITE_CHAR_N_TIMES`, invece di scrivere un breve ciclo per effettuare copie multiple di un carattere. Inoltre, userete questa procedura in molte occasioni.

Ecco le modifiche da effettuare a `DISP_LINE` per aggiungere tre spazi a sinistra dello schermo. Fate queste modifiche in `DISP_SEC.ASM`:

```

PUBLIC  DISP_LINE
EXTRN  WRITE_HEX:PROC
EXTRN  WRITE_CHAR:PROC
EXTRN  WRITE_CHAR_N_TIMES:PROC
;-----;
; Questa procedura visualizza una riga di dati, o 16 byte, prima in
; esadecimale e poi in ASCII.
;
; Inserimento: DS:DX  Distanza in SECTOR, in byte.
;
; Usa:          WRITE_CHAR, WRITE_HEX, WRITE_CHAR_N_TIMES
; Legge:       SECTOR
;-----;
DISP_LINE PROC
    PUSH  BX
    PUSH  CX
    PUSH  DX
    MOV   BX,DX                ;La distanza è più utile in BX
    MOV   DL,' '
    MOV   CX,3                ;Scrive 3 spazi prima della linea
    CALL  WRITE_CHAR_N_TIMES
                                ;Scrive la distanza in esadecimale
    CMP   BX,100h             ;La prima cifra è 1?
    JB   WRITE_ONE           ;No, uno spazio è già in DL
    MOV   DL,'1'             ;Sì, allora inserisci 1 in DL per l'output
WRITE_ONE:
    .
    .
    .

```

Avete fatto tre modifiche. Innanzitutto avete aggiunto l'enunciato `EXTRN` per indicare all'assemblatore che `WRITE_CHAR_N_TIMES` si trova in `VIDEO_IO` e non nel file corrente. In secondo luogo, avete modificato il blocco di commento, in modo da includere questa nuova procedura. Infine avete aggiunto due righe per utilizzare `WRITE_CHAR_N_TIMES`, che dovrebbero esservi chiare senza ulteriori spiegazioni. Provate questa nuova versione per vedere come la visualizzazione risulta ora centrata. Nel prossimo paragrafo modificherete ulteriormente questo programma, aggiungendo delle linee nella parte superiore e inferiore del riquadro.

AGGIUNGERE DELLE LINEE ORIZZONTALI

Aggiungere delle linee orizzontali non è così semplice come può sembrare, dato che bisogna prendere in considerazione alcuni casi speciali. Il primo problema è rappresentato dalla fine delle linee che devono formare un angolo con le linee verticali che incontrano, mentre un altro problema può essere individuato nei punti di incontro a forma di T nella parte superiore e nella parte inferiore della linea che divide la finestra esadecimale da quella ASCII.

Potreste scrivere una lunga serie di istruzioni (con `WRITE_CHAR_N_TIMES`) per creare le linee orizzontali, ma non è il caso, dato che esiste un metodo più rapido. Scriverete una nuova procedura, chiamata `WRITE_PATTERN`, che visualizzerà un modello sullo schermo. Sarà quindi necessaria solamente una piccola area di memoria che dovrà contenere una descrizione di ciascun modello. Usando questa nuova procedura, potrete aggiungere facilmente delle piccole tacche per suddividere la finestra esadecimale (come vedrete alla fine di questo capitolo).

`WRITE_PATTERN` contiene due istruzioni completamente nuove, `LODSB` e `CLD`. Imparerete il significato di queste istruzioni dopo aver visto come funziona `WRITE_PATTERN`. Per ora, inserite questa procedura in `VIDEO_IO.ASM`:

Listato 16-6. La nuova procedura `WRITE_PATTERN` in `VIDEO_IO`.

```

PUBLIC WRITE_PATTERN
;-----;
; Questa procedura traccia una linea sullo schermo sulla base ;
; dei dati seguenti ;
; ;
; DB (carattere, numero di copie del carattere), 0 ;
; Dove {x} significa che x può essere ripetuto un qualsiasi numero di volte. ;
; ;
; Inserimento: DS:DX Indirizzo del modello da tracciare ;
; ;
; Usa: WRITE_CHAR_N_TIMES ;
;-----;
WRITE_PATTERN PROC
    PUSH AX
    PUSH CX
    PUSH DX
    PUSH SI
    PUSHF ;Salva il flag di direzione
    CLD ;Imposta il flag direzione per l'incremento
    MOV SI,DX ;Sposta la distanza nel registro SI per LODSB
PATTERN_LOOP:
    LODSB ;Carica il carattere in AL
    OR AL,AL ;I dati sono finiti (0h)?
    JZ END_PATTERN ;Si, ritorna
    MOV DL,AL ;No, imposta la scrittura del carattere N volte
    LODSB ;Carica il contatore in AL
    MOV CL,AL ;E lo invia in CX per WRITE_CHAR_N_TIMES

```

```

XOR    CH,CH                ;Imposta a zero il byte alto di CX
CALL   WRITE_CHAR_N_TIMES
JMP    PATTERN_LOOP
END_PATTERN:
POPF                   ;Ripristina il flag direzione
POP    SI
POP    DX
POP    CX
POP    AX
RET
WRITE_PATTERN    ENDP

```

Prima di vedere come funziona la procedura, dovete imparare a scrivere i dati per il modello. Inserirete i dati per la linea superiore nel file DISP_SEC. A questo scopo, aggiungerete un'altra procedura, chiamata INIT_SEC_DISP, per inizializzare la visualizzazione del settore scrivendo mezzo settore, e modificherete quindi READ_SECTOR in modo che chiami la procedura INIT_SEC_DISP. Innanzitutto inserite i dati seguenti prima della direttiva .DATA?, in cui viene definito SECTOR (in DISP_SEC.ASM):

Listato 16-7. Aggiunte a DISP_SEC.ASM

```

.DATA
TOP_LINE_PATTERN LABEL    BYTE
DB    '\ ',7
DB    UPPER_LEFT,1
DB    HORIZONTAL_BAR,12
DB    TOP_TICK,1
DB    HORIZONTAL_BAR,11
DB    TOP_TICK,1
DB    HORIZONTAL_BAR,11
DB    TOP_TICK,1
DB    HORIZONTAL_BAR,12
DB    TOP_T_BAR,1
DB    HORIZONTAL_BAR,18
DB    UPPER_RIGHT,1
DB    0
BOTTOM_LINE_PATTERN LABEL    BYTE
DB    '\ ',7
DB    LOWER_LEFT,1
DB    HORIZONTAL_BAR,12
DB    BOTTOM_TICK,1
DB    HORIZONTAL_BAR,11
DB    BOTTOM_TICK,1
DB    HORIZONTAL_BAR,11
DB    BOTTOM_TICK,1
DB    HORIZONTAL_BAR,12
DB    BOTTOM_T_BAR,1
DB    HORIZONTAL_BAR,18
DB    LOWER_RIGHT,1
DB    0

```

```
.DATA?
```

```
SECTOR DB 8192 DUP (?)
```

Notate che tutti i dati sono stati inseriti in `.DATA` invece che in `.DATA?`, dato che devono essere impostati i valori per tutte queste variabili.

Ciascun enunciato DB contiene una parte dei dati per una linea. Il primo byte è il carattere da visualizzare; il secondo byte indica a `WRITE_PATTERN` il numero di ripetizioni del carattere. Per esempio, la riga superiore viene iniziata con 7 spazi, seguiti dal carattere grafico che rappresenta un angolo in alto a sinistra, da dodici barre orizzontali e così via. L'ultimo DB contiene uno zero, che indica la fine del modello.

Continuate le modifiche e osservate il risultato prima di analizzare il funzionamento interno di `WRITE_PATTERN`. Ecco una versione di prova di `INIT_SEC_DISP`. Questa procedura scrive la linea superiore, visualizza metà settore e traccia infine la linea inferiore. Inserite questa procedura nel file `DISP_SEC.ASM`, prima di `DISP_HALF_SECTOR`:

Listato 16-8. Aggiungere la procedura in `DISP_SEC.ASM`

```

PUBLIC INIT_SEC_DISP
EXTRN WRITE_PATTERN:PROC, SEND_CRLF:PROC
;-----;
; Questa procedura inizializza la visualizzazione di mezzo settore. ;
; ;
; Usa: WRITE_PATTERN, SEND_CRLF, DISP_HALF_SECTOR ;
; Legge: TOP_LINE_PATTERN, BOTTOM_LINE_PATTERN ;
;-----;
INIT_SEC_DISP PROC
    PUSH DX
    LEA DX, TOP_LINE_PATTERN
    CALL WRITE_PATTERN
    CALL SEND_CRLF
    XOR DX, DX ;Comincia all'inizio del settore
    CALL DISP_HALF_SECTOR
    LEA DX, BOTTOM_LINE_PATTERN
    CALL WRITE_PATTERN
    POP DX
    RET
INIT_SEC_DISP ENDP

```

Avete usato l'istruzione `LEA` per caricare un indirizzo nel registro `DX`; in questo modo `WRITE_PATTERN` sa dove trovare il modello di dati.

Dovete infine effettuare una modifica a `READ_SECTOR` nel file `DISK_IO.ASM`, in modo da chiamare `INIT_SECTOR_DISP` invece di `WRITE_HALF_SECTOR_DISP` (in questo modo sarà tracciato un intero riquadro intorno al mezzo settore visualizzato):

Listato 16-9. Modifiche a READ_SECTOR in DISK_IO.ASM

```

EXTRN  INIT_SEC_DISP:PROC
;-----;
; Questa procedura legge il primo settore sul disco A e ne          ;
; visualizza la prima metà.                                         ;
;-----;
READ_SECTOR  PROC
    MOV     AX,DGROUP          ;Inserisce il segmento dati in AX
    MOV     DS,AX              ;Imposta DS per puntare ai dati

    MOV     AL,0                ;Drive A (numero 0)
    MOV     CX,1                ;Legge solo 1 settore
    MOV     DX,0                ;Legge il settore numero 0
    LEA     BX,SECTOR          ;Dove memorizzare questo settore
    INT     25h                ;Legge il settore
    POPF                                ;Elimina i flag inviati allo stack dal DOS
    XOR     DX,DX              ;Imposta a 0 la distanza in SECTOR
    CALL   INIT_SEC_DISP      ;Visualizza la prima metà

    MOV     AH,4Ch              ;Ritorna al DOS
    INT     21h
READ_SECTOR  ENDP

```

Questo è tutto il necessario per scrivere la linea superiore e inferiore nella visualizzazione di un settore. Assemblate e collegate tutti questi file (ricordatevi di assemblare i tre file modificati) e fate una prova. La figura 16-3 mostra quello che dovrete ottenere.

Vediamo ora come funziona WRITE_PATTERN. Come già detto, vengono utilizzate due nuove istruzioni. LODSB sta per *Load String Byte* (Carica Byte Stringa) ed è una delle istruzioni di tipo stringa (istruzioni speciali per lavorare con le stringhe). Questo non è esattamente quello che viene eseguito qui, ma all'8088 non interessa se si sta utilizzando una stringa di caratteri o di numeri; quindi, questa istruzione, soddisfa pienamente le esigenze richieste.

LODSB carica un singolo byte nel registro AL dalla locazione di memoria specificata da DS:SI, una coppia di registri mai utilizzata prima. (Avete già impostato il registro DS in READ_SECTOR per puntare ai dati). Prima dell'istruzione LODSB, avete impostato il registro SI con l'istruzione MOV SI,DX.

L'istruzione LODSB è in qualche modo simile all'istruzione MOV ma molto più potente. Con un'istruzione LOADSB, l'8088 sposta un byte nel registro AL e quindi incrementa o decrementa il registro SI. Incrementando SI viene puntato il byte successivo in memoria, mentre decrementandolo viene puntato il byte precedente. L'incremento è proprio quello che serve in questo programma, dato che bisogna scorrere attraverso il modello, un byte alla volta, partendo dall'inizio. Questo è quello che fa l'istruzione LODSB, dato che è stata usata un'altra istruzione nuova, CLD (*Clear Direction Flag*, Azzerare il Flag di Direzione), per azzerare il flag di direzione. Se aveste impostato il flag di direzione, l'istruzione LODSB avrebbe decrementato il registro SI.

LODSB è stato utilizzato in diverse parti di Dskpatch, sempre con il flag di direzione azzerato.

Insieme a LODSB e CLD, notate che sono state usate anche le istruzioni PUSHF e POPF che salvano e ripristinano i registri. Questo è stato fatto nel caso si decidesse successivamente di usare il flag di direzione in una procedura che chiama WRITE_PATTERN.

A>disk_io

00	EB 34 90 49 42 4D 20 20 33 2E 33 00 02 04 01 00	δ4ÉIBM .3.3....
10	02 00 02 EF A9 F8 2B 00 11 00 08 00 11 00 00 00	...n-r°+.<...<...
20	00 00 00 00 00 00 00 00 00 00 00 00 00 00 12
30	00 00 00 00 01 00 FA 33 C0 0E D0 BC 00 7C 16 073LÀL...!
40	BB 78 00 36 C5 37 1E 56 16 53 BF 2B 7C B9 0B 00	q̄x.6+7ΔU.Sγ+i+q̄..
50	FC AC 26 00 3D 00 74 03 26 0A 05 AA 0A C4 E2 F1	°%&C=.t.&è.τè-Γ±
60	06 1F 09 47 02 C7 07 2B 7C FB CD 13 72 67 A0 10	.vèC.¶.+i+ =rgá>
70	7C 98 F7 26 16 7C 03 06 1C 7C 03 06 0E 7C A3 3F	i+j=δ.1...+!...!ú?
80	7C A3 37 7C B0 20 00 F7 26 11 7C 0B 1E 0B 7C 03	!ú?iγ .=δ<i!YA.!
90	C3 48 F7 F3 01 06 37 7C BB 00 05 A1 3F 7C E8 9F	H=ξ...iγ..i?iδf
A0	00 B0 01 02 E8 B3 00 72 19 0B FB B9 0B 00 BE D9	.γ..δ .rvivδ...δj
B0	7D F3 A6 75 0D 0D 7F 20 BE E4 7D B9 0B 00 F3 A6	}ξ°u.ìδ δΣ}...ξ°
C0	74 18 BE 77 7D E8 6A 00 32 E4 CD 16 5E 1F 0F 04	t^jw}δj.2Σ= ^vδ.
D0	0F 44 02 CD 19 BE C4 7D EB EB A1 1C 05 33 D2 F7	δD =vδ-}δδìL 3γ°
E0	36 0B 7C FE C0 A2 3C 7C A1 37 7C A3 3D 7C BB 00	6.ìuL6<i!7iú=iγ.
F0	07 A1 37 7C E8 49 00 A1 18 7C 2A 06 3B 7C 40 3B	.i?iδi.i^i*.:iè0

A>_

Figura 16-3. Visualizzazione con le cornici chiuse

AGGIUNGERE DEI NUMERI ALLA VISUALIZZAZIONE

Avete quasi finito la seconda parte del libro. Prima di passare alla terza parte, scriverete l'ultima procedura da inserire in DISP_SEC.ASM.

Notate che nella visualizzazione del settore manca una riga di numeri nella parte superiore. Questi numeri (00, 01, 02 e così via) permetterebbero di ricercare nelle colonne gli indirizzi di ogni byte. Scrivete quindi una procedura per visualizzare questa riga di numeri. Aggiungete questa procedura, WRITE_TOP_HEX_NUMBERS, al file DISP_SEC.ASM dopo INIT_SEC_DISP:

Listato 16-10. La procedura WRITE_TOP_HEX NUMBERS in DISP_SEC.ASM

```

        EXTRN WRITE_CHAR_N_TIMES:PROC, WRITE_HEX:PROC, WRITE_CHAR:PROC
        EXTRN WRITE_HEX_DIGIT:PROC, SEND_CRLF:PROC
;-----;
; Questa procedura scrive i numeri da 0 a F nella riga superiore      ;
; della visualizzazione di mezzo settore.                             ;
;                                                                       ;
; Usa:          WRITE_CHAR_N_TIMES, WRITE_HEX, WRITE_CHAR            ;
;              WRITE_HEX_DIGIT, SEND_CRLF                            ;
;-----;
WRITE_TOP_HEX_NUMBERS      PROC
        PUSH  CX
        PUSH  DX
        MOV   DL, ' '          ;Scrive 9 spazi per il lato sinistro
        MOV   CX, 9
        CALL  WRITE_CHAR_N_TIMES
        XOR   DH, DH          ;Inizia da 0
HEX_NUMBER_LOOP:
        MOV   DL, DH
        CALL  WRITE_HEX
        MOV   DL, ' '
        CALL  WRITE_CHAR
        INC   DH
        CMP   DH, 10h        ;Finito?
        JB   HEX_NUMBER_LOOP

        MOV   DL, ' '          ;Scrive i numeri hex nella finestra ASCII
        MOV   CX, 2
        CALL  WRITE_CHAR_N_TIMES
        XOR   DL, DL
HEX_DIGIT_LOOP:
        CALL  WRITE_HEX_DIGIT
        INC   DL
        CMP   DL, 10h
        JB   HEX_DIGIT_LOOP
        CALL  SEND_CRLF
        POP   DX
        POP   CX
        RET
WRITE_TOP_HEX_NUMBERS      ENDP

```

Modificate INIT_SEC_DISP (anche in DISP_SEC.ASM) come segue, in modo da chiamare WRITE_TOP_HEX_NUMBERS prima di scrivere il resto del settore:

Listato 16-11. Modifiche a INIT_SEC_DISP in DISP_SEC.ASM

```

-----;
; Usa:  WRITE_PATTERN, SEND_CRLF, DISP_HALF_SECTOR ;
;      WRITE_TOP_HEX_NUMBERS ;
; Legge: TOP_LINE_PATTERN, BOTTOM_LINE_PATTERN ;
-----;
INIT_SEC_DISP  PROC
    PUSH  DX
    CALL  WRITE_TOP_HEX_NUMBERS
    LEA   DX, TOP_LINE_PATTERN
    CALL  WRITE_PATTERN
    CALL  SEND_CRLF
    XOR   DX, DX ;Comincia all'inizio del settore
    CALL  DISP_HALF_SECTOR
    LEA   DX, BOTTOM_LINE_PATTERN
    CALL  WRITE_PATTERN
    POP   DX
    RET
INIT_SEC_DISP  ENDP
    
```

Ora avete completato la visualizzazione di mezzo settore, come potete vedere dalla figura 16-4.

```

A>disk_io
    00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  0123456789ABCDEF
00  EB 34 90 49 42 4D 20 20 33 2E 33 00 02 04 01 00  04ÉIDM .3.3....
10  02 00 02 EF A9 F8 2B 00 11 00 00 00 11 00 00 00  ...n°+.<...<...
20  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 12  .....
30  00 00 00 00 01 00 FA 33 C0 8E D0 BC 00 7C 16 07  ....3!A.L!..
40  BB 78 00 36 C5 37 1E 56 16 53 BF 2B 7C B9 0B 00  Bx.6+7AU.S7+i!..
50  FC AC 26 00 3D 00 74 03 26 8A 05 AA 8A C4 E2 F1  %&Ç=.t.8è.-è-!±
60  06 1F 89 47 02 C7 07 2B 7C FB CD 13 72 67 A0 10  .v8G.+.i!|=rgá)
70  7C 98 F7 26 16 7C 03 06 1C 7C 03 06 0E 7C A3 3F  iÿ=&.!..-!...iú?
80  7C A3 37 7C B8 20 00 F7 26 11 7C 8B 1E 0B 7C 03  iú7!ÿ .=&<i!A.!.
90  C3 48 F7 F3 01 06 37 7C BB 00 05 A1 3F 7C E8 9F  |H=<...iÿ...i?i8f
A0  00 B8 01 02 E8 B3 00 72 19 8B FB B9 0B 00 BE D9  .ÿ..8|.rv!ÿ!..!ÿ
B0  7D F3 A6 75 0D 8D 7F 20 BE E4 7D B9 0B 00 F3 A6  }<u.ìΔ!Σ!ÿ...<u
C0  74 18 BE 77 7D E8 6A 00 32 E4 CD 16 5E 1F 8F 04  t°!w)8j.2Σ= ^vÂ.
D0  8F 44 82 CD 19 BE C4 7D EB EB A1 1C 05 33 D2 F7  ÅD ~!-)88!L 3ÿ=
E0  36 0B 7C FE C0 A2 3C 7C A1 37 7C A3 3D 7C BB 00  6.!!#6<i!7!ú=iÿ.
F0  07 A1 37 7C E8 49 00 A1 18 7C 2A 06 3B 7C 40 38  .i7!8!i.í!*.;!08
    
```

A>_

Figura 16-4. Visualizzazione completa di metà settore

Ci sono ancora alcune differenze tra questa visualizzazione e la versione finale. Modificherete `WRITE_CHAR` in modo che possa visualizzare tutti i 256 caratteri disponibili sull'IBM PC, e imparerete a centrare il contenuto dello schermo verticalmente, usando le routine del BIOS. Questo è quello che farete nei prossimi capitoli.

SOMMARIO

Avete fatto moltissimo lavoro per la costruzione del programma `Dskpatch`, aggiungendo nuove procedure, modificandone altre e spostandole da un file a un altro. Da questo momento in avanti, se perdete il controllo di quello che state facendo, fate riferimento al listato completo di `Dskpatch` nell'appendice B. Quel listato è la versione finale, ma troverete alcune somiglianze che vi aiuteranno a procedere.

La maggior parte delle modifiche fatte in questo capitolo non si sono basate su accorgimenti di programmazione, ma su duro lavoro. Avete imparato due nuove istruzioni: `LODSB` e `CLD`. `LODSB` è un'istruzione di tipo stringa che permette di usare un'istruzione per eseguire il lavoro di molte altre. Avete usato `LODSB` in `WRITE_PATTERN` per leggere dei byte consecutivi da un modello, caricando sempre un nuovo byte nel registro `AL`. `CLD`, invece, azzerava il flag di direzione, che determina la direzione dell'incremento. Ogni istruzione `LODSB` che segue `CLD` carica il byte successivo dalla memoria.

Nella prossima parte di questo libro imparerete a usare le routine del BIOS. Queste vi faranno risparmiare moltissimo tempo.

PARTE III

ROM BIOS DEL PC IBM

LE ROUTINE DELLA ROM BIOS

Nel vostro Personal Computer IBM ci sono parecchi chip o IC (Integrated Circuits), conosciuti come ROM (Read-Only Memory). Una di queste ROM contiene un numero di routine, molto simili a procedure, che forniscono tutte le routine di base per l'input e l'output verso parti differenti del vostro IBM PC. Siccome questa ROM fornisce le routine per input e output a un livello molto basso, si fa frequente riferimento come BIOS, acronimo che significa Basic Input Output System. Il DOS utilizza il BIOS per attività quali la scrittura sullo schermo o la lettura dei dati dal disco, ed è comunque possibile utilizzare tali routine all'interno dei vostri programmi.

Vi concentrerete sulle routine del BIOS che servono per Dskpatch. Una parte di queste sono dedicate alla visualizzazione a video, funzioni molto difficili da ottenere se non si lavora direttamente con l'hardware.

VIDEO_IO, LE ROUTINE DELLA ROM BIOS

Si fa riferimento agli elementi della ROM BIOS come routine per distinguerle dalle procedure. Le procedure si utilizzano con un'istruzione CALL, mentre le routine si chiamano con l'istruzione INT, e non con CALL. Si utilizzerà un'istruzione INT 21h, per esempio, per chiamare le routine di I/O del video; nello stesso modo utilizzerete un INT 21h per le routine del DOS.

INT 10h chiama le routine VIDEO_IO del BIOS. Altri numeri chiamano altre routine, ma non le considereremo; VIDEO_IO fornisce tutte le funzioni che servono al di fuori del DOS. (Per vostra informazione, in ogni caso, il DOS chiama una delle altre routine del BIOS quando richiamiamo un settore del disco).

In questo capitolo, utilizzerete le routine del BIOS per aggiungere due nuove procedure a Dskpatch: una per pulire lo schermo, e l'altra per spostare il cursore in qualsiasi posizione desiderata dello schermo. Entrambe sono funzioni molto utili, ma nessuna delle due è disponibile direttamente dal DOS. Più tardi vedrete delle cose molto interessanti che è possibile fare con queste routine, ma iniziate utilizzando INT 10h per pulire lo schermo prima di visualizzare il mezzo settore.

L'istruzione INT 10h, è l'ingresso per l'utilizzo di una serie di altre funzioni. Richiamando questa istruzione, quando utilizzate l'istruzione DOS INT 21h, si seleziona una funzione particolare mettendo il numero di quest'ultima nel registro AH. Selezionate la funzione VIDEO_IO nello stesso modo; mettendo il numero di funzione appropriato nel registro AH (una lista completa di queste funzioni può essere trovata nella Tabella 17-1).

Tabella 17-1. Funzioni INT 10h

(AH)=0 **Imposta la modalità di visualizzazione.** Il registro AL contiene il numero della modalità.

MODALITÀ TESTO

(AL)=0	40 per 25, bianco e nero
(AL)=1	40 per 25, colore
(AL)=2	80 per 28, bianco e nero
(AL)=3	80 per 25, colore
(AL)=7	80 per 25, adattatore monocromatico

MODALITÀ GRAFICHE

(AL)=4	320 per 200, colore
(AL)=5	320 per 200, bianco e nero
(AL)=6	640 per 200, bianco e nero

(AH) = 1 **Imposta la dimensione del cursore.**

(CH)	Linea di scan iniziale del cursore. La prima linea è 0 su entrambi i modi di visualizzazione monocromatico e grafico, mentre l'ultima linea è 7 per gli adattatori grafici a colori e 13 per gli adattatori monocromatici. Il range valido è tra 0 e 31.
(CL)	Ultima linea di scan del cursore.

L'impostazione all'accensione per l'adattatore grafico a colori è CH=6 e CL=7. Per l'adattatore monocromatico CH=11 e CL=12

(AH) = 2 **Imposta la posizione del cursore.**

(DH,DL)	Riga, colonna della nuova posizione del cursore; l'angolo in alto a destra è (0,0).
(BH)	Numero di pagina. Questo è il numero della pagina di visualizzazione. L'adattatore grafico a colori ha spazio per parecchie pagine di visualizzazione, ma la maggior parte dei programmi utilizzano la pagina 0.

- (AH) = 3 **Legge la posizione del cursore.**
- (BH) Numero di pagina.
In uscita (DH, DL) Riga, colonna del cursore
 (CH, CL) Dimensione del cursore
- (AH) = 4 **Legge la posizione della penna luminosa** (vedere il Tech. Ref. Man.).
- (AH) = 5 **Seleziona la pagina di visualizzazione attiva.**
- (AL) Nuovo numero pagina (da 0 a 7 per i modi 0 e 1; da 0 a 3 per i modi 2 e 3)
- (AH) = 6 **Scorrimento verso l'alto.**
- (AL) Numero di linee da cancellare nella parte bassa della finestra. Normalmente viene cancellata una sola linea. Impostate a zero per cancellare l'intera finestra.
(CH, CL) Riga, colonna dell'angolo in alto a sinistra della finestra
(DH, DL) Riga, colonna dell'angolo in basso a destra della finestra
(BH) Visualizza gli attributi da utilizzare per cancellare le righe
- (AH) = 7 **Scorrimento verso il basso.**
- Come lo scorrimento verso l'alto (funzione 6), ma si fa riferimento alle righe nella parte alta della finestra.
- (AH) = 8 **Legge l'attributo e il carattere sotto al cursore.**
- (BH) Visualizza la pagina (solo modalità testo)
(AL) Carattere da scrivere
(AH) Attributo del carattere letto (solo modalità testo)

(AH) = 9	(BX)	Visualizza la pagina (solo modalità testo)
	(CX)	Numero di volte che bisogna scrivere il carattere e l'attributo sullo schermo
	(AL)	Carattere da scrivere
	(BL)	Attributo da scrivere

(AH) = 10 **Scrive il carattere sotto al cursore** (con attributo normale).

(BH)	Visualizza la pagina
(CX)	Numero di volte da scrivere il carattere
(AL)	Carattere da scrivere

(AH)= da 11 a 13 **Varie funzioni grafiche.** (Vedere il Tech. Ref. Man. per maggiori dettagli)

(AH) = 14 **Scrittura carattere.** Scrive un carattere sullo schermo e sposta il cursore alla prossima posizione.

(AL)	Carattere da scrivere
(BL)	Colore del carattere (solo modalità grafica)
(BH)	Pagina da visualizzare (modalità testo)

(AH) = 15 **Riporta lo stato corrente del video**

(AL)	Visualizza la modalità correntemente impostata
(AH)	Numero di caratteri per linea
(BH)	Attiva le pagine da visualizzare

CANCELLARE LO SCHERMO

Utilizzerete la funzione numero 6 dell'istruzione INT 10h, `SCROLL ACTIVE PAGE UP`, per cancellare lo schermo. Momentaneamente non si vuole fare scorrere il contenuto dello schermo, ma questa funzione può essere utilizzata anche per cancellare lo schermo. Ecco la procedura; inseritela nel file `CURSOR.ASM`:

Listato 17-1. Aggiungete questa procedura a *CURSOR.ASM*

```

PUBLIC CLEAR_SCREEN
;-----;
; Questa procedura cancella l'intero schermo. ;
;-----;
CLEAR_SCREEN PROC
    PUSH AX
    PUSH BX
    PUSH CX
    PUSH DX
    XOR AL,AL ;Cancella intera finestra
    XOR CX,CX ;L'angolo superiore sinistro è a (0,0)
    MOV DH,24 ;La riga inferiore dello schermo è la 24
    MOV DL,79 ;Il limite destro è la colonna 79
    MOV BH,7 ;Utilizza l'attributo normale per spazi
    MOV AH,6 ;Richiama la funzione SCROLL-UP
    INT 10h ;Cancella la finestra
    POP DX
    POP CX
    POP BX
    POP AX
    RET
CLEAR_SCREEN ENDP

```

Sembra che alla funzione numero 6 dell'istruzione INT 10h servano molte informazioni, anche se vogliamo semplicemente cancellare lo schermo. Questa funzione è abbastanza potente: è in grado di cancellare qualsiasi parte rettangolare dello schermo (finestra, come viene generalmente chiamata). Dobbiamo impostare la finestra come intero schermo selezionando come prima e ultima riga rispettivamente 0 e 24, e impostando le colonne a 0 e 79. Le routine che utilizzate sono in grado di cancellare lo schermo in modo da renderlo tutto bianco (per utilizzarlo con i caratteri neri), o tutto nero (in modo da utilizzarlo con i caratteri bianchi). Si vuole utilizzare quest'ultimo, e questo è quello specificato con l'istruzione MOV BH,7. Quindi impostate AL a 0, il numero di linee da far scorrere, diciamo alla routine di pulire la finestra, invece che di farla scorrere.

Ora dovete modificare la procedura di test, READ_SECTOR, per chiamare CLEAR_SCREEN prima di iniziare la visualizzazione. Non mettete questa CALL in INIT_SEC_DISP, perché si vuole usare INIT_SEC_DISP per riscrivere solo la visualizzazione di mezzo settore, senza modificare il resto dello schermo.

Per modificare READ_SECTOR, inserite una dichiarazione EXTRN per CLEAR_SCREEN e inserite la CALL a CLEAR_SCREEN. Apportate le seguenti modifiche al file DISK_IO.ASM:

Listato 17-2. Cambiamenti a READ_SECTOR in DISK_IO.ASM

```

EXTRN INIT_SEC_DISP:PROC, CLEAR_SCREEN:PROC
;-----;
; Questa procedura legge il primo settore del disco A e stampa          ;
; la prima metà di questo settore.                                     ;
;-----;

READ_SECTOR      PROC
    MOV     AX,DGROUP          ;Mette il segmento dati in AX
    MOV     DS,AX              ;Imposta DS in modo che punti ai dati
    MOV     AL,0                ;Drive A (numero 0)
    MOV     CX,1                ;Legge solo 1 settore
    MOV     DX,0                ;Legge il settore numero 0
    LEA     BX,SECTOR          ;Dove salvare il settore
    INT     25h                ;Legge il settore
    POPF                    ;Scarta il flag messo nello stack dal DOS
    CALL   CLEAR_SCREEN
    CALL   INIT_SEC_DISP      ;Visualizza la prima metà

    MOV     AH,4Ch              ;Ritorna la DOS
    INT     21h
READ_SECTOR      ENDP

```

Prima di eseguire la nuova versione di Disk_io, notate dove è posizionato il cursore. Quindi, eseguite Disk_io. Lo schermo si cancellerà, e Disk_io inizierà a scrivere la prima metà del settore nella posizione in cui si trovava il cursore prima di avviare il programma - probabilmente in fondo allo schermo.

Anche se avete cancellato lo schermo, non è stato detto al programma di spostare il cursore nuovamente in cima allo schermo. In BASIC, il comando CLS cancella lo schermo in due passaggi: cancella lo schermo, quindi sposta il cursore in cima allo schermo stesso. La vostra procedura non fa ciò; dovete quindi spostare il cursore.

SPOSTARE IL CURSORE

La funzione numero 2 dell'INT 10h imposta la posizione del cursore nello stesso modo del comando LOCATE del BASIC. E' possibile utilizzare GOT_XY per spostare il cursore in qualsiasi posizione sullo schermo. Inserite questa procedura nel file CURSOR.ASM:

Listato 17-3. Aggiungete questa procedura in CURSOR.ASM

```

PUBLIC GOTO_XY
;-----;
; Questa procedura sposta il cursore                                  ;
;                                                                     ;
;     DH      Riga (Y)                                              ;
;     DL      Colonna (X)                                          ;
;-----;

```



```

GOTO_XY      PROC
    PUSH    AX
    PUSH    BX
    MOV     BH,0           ;Visualizza pagina 0
    MOV     AH,2         ;Richiama SET CURSOR POSITION
    INT    10h
    POP     BX
    POP     AX
    RET
GOTO_XY      ENDP

```

Utilizzerete GOTO_XY in una versione rivista di INIT_SEC_DISP per spostare il cursore sulla seconda linea dove in precedenza avete visualizzato la metà del settore. Ecco le modifiche a INIT_SEC_DISP in DISP_SEC.ASM

Listato 17-4. Cambiamenti a INIT_SEC_DISP in DISP_SEC.ASM

```

PUBLIC INIT_SEC_DISP
EXTRN WRITE_PATTERN:PROC, SEND_CRLF:PROC
EXTRN GOTO_XY:PROC
;-----;
; Questa procedura inizializza la visualizzazione di mezzo settore. ;
; ;
; Utilizza:    WRITE_PATTERN, SEND_CRLF, DISP_HALF_SECTOR ;
;             WRITE_TOP_HEX_NUMBERS, GOTO_XY ;
; Legge:      TOP_LINE_PATTERN, BOTTOM_LINE_PATTERN ;
;-----;
INIT_SEC_DISP  PROC
    PUSH    DX
    XOR    DL,DL           ;Sposta il cursore all'inizio della
    MOV    DH,2           ;terza riga
    CALL  GOTO_XY
    CALL    WRITE_TOP_HEX_NUMBERS
    LEA    DX,TOP_LINE_PATTERN
    .
    .
    .

```

Se provate ora, vedrete che la parte di settore visualizzata è centrata. Come si vede, è semplice lavorare con lo schermo quando si hanno a disposizione le routine del BIOS. Nel prossimo capitolo, utilizzerete un'altra routine del BIOS per modificare WRITE_CHAR, in modo che sia possibile scrivere qualsiasi carattere sullo schermo. Ma prima di continuare, fate delle altre modifiche al programma, quindi finite con una procedura chiamata WRITE_HEADER, che scriverà una linea di stato in cima allo schermo, per mostrare il drive corrente e il numero di settore.

MODIFICA DELL'USO DELLE VARIABILI

Ci sono molte cose da fare prima di creare WRITE_HEADER. Molte procedure, come si presentano ora, hanno dei numeri fissi al loro interno; per esempio, READ_SECTOR legge il settore 0 del drive A. Si vuole mettere il drive e il numero di settore in variabili di memoria, in modo che possano essere utilizzate da più di una procedura.

Dovete anche cambiare queste procedure in modo che utilizzino le variabili di memoria; iniziate a mettere tutte le variabili in un file, DSKPATCH.ASM, per rendere il lavoro più semplice. Dskpatch.asm sarà il primo file del programma dskpatch, in modo che le variabili siano facili da trovare. Ecco DSKPATCH.ASM, completo di una lunga lista di variabili di memoria:

Listato 17-5. Il nuovo file DSKPATCH.ASM

```
DOSSEG
.MODEL    SMALL

.STACK

.DATA

                PUBLIC    SECTOR_OFFSET
;-----;
; SECTOR_OFFSET è l'offset della visualizzazione           ;
; di mezzo settore nel settore intero. Deve               ;
; essere un multiplo di 16 e non maggiore di 256         ;
;-----;
SECTOR_OFFSET  DW      0

                PUBLIC CURRENT_SECTOR_NO, DISK_DRIVE_NO
CURRENT_SECTOR_NO    DW      0      ;Inizialmente settore 0
DISK_DRIVE_NO        DB      0      ;Inizialmente Drive A:

                PUBLIC  LINES_BEFORE_SECTOR, HEADER_LINE_NO
                PUBLIC  HEADER_PART_1, HEADER_PART_2
;-----;
; LINES_BEFORE_SECTOR è il numero di righe vuote         ;
; nella parte alta dello schermo prima della             ;
; visualizzazione di mezzo settore.                      ;
;-----;
LINES_BEFORE_SECTOR  DB      2
HEADER_LINE_NO       DB      0
HEADER_PART_1        DB      'Disco ',0
HEADER_PART_2        DB      '      Settore ',0

                PUBLIC SECTOR
;-----;
; L'intero settore (fino a 8192 byte) è salvato          ;
; in quest'area di memoria.                              ;
;-----;
```

```

SECTOR    DB    8192 DUP (?)

.CODE

        EXTRN  CLEAR_SCREEN:PROC, READ_SECTOR:PROC
        EXTRN  INIT_SEC_DISP:PROC
DISK_PATCH PROC
        MOV    AX,DGROUP           ;mette il segmento dati in AX
        MOV    DS,AX              ;imposta DS in modo da puntare ai dati

        CALL  CLEAR_SCREEN
        CALL  READ_SECTOR
        CALL  INIT_SEC_DISP

        MOV    AH,4Ch             ;Torna al DOS
        INT   21h
DISK_PATCH ENDP

        END    DISK_PATCH

```

La procedura principale, DISK_PATCH, chiama altre tre procedure. Sono state viste tutte in precedenza, e presto scriverete sia READ_SECTOR e INIT_SEC_DISP per utilizzare le variabili appena inserite nel segmento dati.

Prima di poter utilizzare Dskpatch, dovete modificare Disp_sec, per rimpiazzare la definizione di SECTOR con un EXTRN. Dovete anche modificare Disk_io, per cambiare READ_SECTOR in una procedura ordinaria che sia possibile chiamare da Dskpatch.

Come prima cosa prendete SECTOR. Siccome è stato messo in DSKPATCH.ASM come variabile di memoria, dovete cambiare la definizione di SECTOR in Disp_sec con una dichiarazione EXTRN. Apportate questi cambiamenti in DISP_SEC.ASM:

Listato 17-6. Cambiamenti a DISP_SEC.ASM

```

.DATA?
        EXTRN  SECTOR:BYTE
        PUBLIC  SECTOR
SECTOR  DB    8192 DUP (?)

```

Riscrivete il file DISK_IO.ASM in modo che contenga solo procedure, e READ_SECTOR utilizzi variabili di memoria (e non numeri fissi) per i numeri del drive e del settore. Ecco la nuova versione di DISK_IO.ASM

Listato 17-7. Cambiamenti a DISK_IO.ASM

```

BOSSSEG
.MODEL SMALL

.STACK

```

```

.DATA

    EXTRN    SECTOR:BYTE
    EXTRN    DISK_DRIVE_NO:BYTE
    EXTRN    CURRENT_SECTOR_NO:WORD

    . PUBLIC READ_SECTOR
    EXTRN    INIT_SEC_DISP,PROC, CLEAR_SCREEN:PROC
;-----;
; Questa procedura legge un settore (512 byte) in SECTOR. ;
; ;
; Legge:          CURRENT_SECTOR_NO, DISK_DRIVE_NO ;
; Scrive:         SECTOR ;
;-----;
READ_SECTOR PROC
    MOV     AX,DGROUP ;Mette il segmento dati in AX
    MOV     DS,AX ;Imposta DS per puntare ai dati
    PUSH   AX
    PUSH   BX
    PUSH   CX
    PUSH   DX
    MOV     AL,DISK_DRIVE_NO ;Numero drive
    MOV     CX,1 ;Legge solo 1 settore
    MOV     DX,CURRENT_SECTOR_NO ;Numero settore logico
    LEA    BX,SECTOR ;Dove memorizzare questo settore
    INT    25h ;Legge il settore
    POPF   ;Elimina flag impostati su stack dal DOS
    POP    DX
    POP    CX
    POP    BX
    POP    AX
    RET
    CALL   CLEAR_SCREEN
    CALL   INIT_SEC_DISP ;Visualizza la prima metà
    MOV    AH,4Ch ;Ritorna al DOS
    INT    21h
READ_SECTOR ENDP

END

```

Questa nuova versione di Disk_io utilizza la variabile di memoria DISK_DRIVE_NO e CURRENT_SECTOR_NO come numeri per il drive e il settore da cui leggere. Siccome queste variabili sono già definite in DSKPATCH.ASM, non dovete cambiare Disk_io quando dovete leggere dei settori differenti da altri dischi. Se utilizzate il programma Make per ricompilare DSKPATCH.COM, dovete apportare alcune modifiche al vostro Makefile.

Listato 17-8. *La nuova versione di MAKEFILE*

```
dskpatch.obj:    dskpatch.asm  
    masm dskpatch;  
  
disk_io.obj:     disk_io.asm  
    masm disk_io;  
  
disp_sec.obj:    disp_sec.asm  
    masm disp_sec  
  
video_io.obj:    video_io.asm  
    masm video_io;  
  
cursor.obj: cursor.asm  
    masm cursor;  
  
dskpatch.exe:    dskpatch.obj disk_io.obj disp_sec.obj video_io.obj cursor.obj  
    link dskpatch disk_io disp_sec video_io cursor;
```

(Ricordate che se state utilizzando il Make della Borland, le ultime due linee devono essere all'inizio del Makefile. E se state utilizzando OPTASM aggiungete le prime due linee, con la prima linea indentata e la prima linea allineata a sinistra). Se non utilizzate il Make, assicuratevi di assemblare tutti i tre file cambiati (Dskpatch, Disk_io, e Disp_sec) e di linkare i cinque file, con Dskpatch come primo della lista:

```
LINK DSKPATCH DISK_IO DISP_SEC VIDEO_IO CURSOR;
```

Sono stati apportato pochi cambiamenti, quindi controllate Dskpatch e assicuratevi che funzioni correttamente prima di continuare.

SCRIVERE L'INTESTAZIONE

Ora che avete convertito i numeri in variabili di memoria, potete scrivere la procedura WRITE_HEADER per scrivere una linea di stato, o header, nella parte superiore dello schermo. La linea di stato sarà simile a questa:

```
Disco A      Settore 0
```

WRITE_HEADER utilizzerà WRITE_DECIMAL per scrivere il settore corrente in notazione decimale. Scriverà anche due stringhe di caratteri, *Disco* e *Settore* (ognuno seguito da uno spazio bianco), la lettera del drive, come A. Mettete la procedura nel file DISP_SEC.ASM.

Per iniziare, mettete la seguente procedura in DISP_SEC.ASM

Listato 17-9. Aggiungete questa procedura a DISP_SEC.ASM

```

PUBLIC WRITE_HEADER

.DATA
    EXTRN  HEADER_LINE_NO:BYTE
    EXTRN  HEADER_PART_1:BYTE
    EXTRN  HEADER_PART_2:BYTE
    EXTRN  DISK_DRIVE_NO:BYTE
    EXTRN  CURRENT_SECTOR_NO:WORD

.CODE
    EXTRN  WRITE_STRING:PROC, WRITE_DECIMAL:PROC
    EXTRN  GOTO_XY:PROC

;-----;
; Questa procedura scrive l'header con la lettera del drive e il      ;
; numero del settore                                               ;
;                                                                     ;
; Usa:   GOTO_XY, WRITE_STRING, WRITE_CHAR, WRITE_DECIMAL          ;
; Legge: HEADER_LINE_NO, HEADER_PART_1, HEADER_PART_2,            ;
;        DISK_DRIVE_NO, CURRENT_SECTOR_NO                          ;
;-----;
WRITE_HEADER    PROC
    PUSH    DX
    XOR     DL,DL                ;Sposta il cursore sulla linea dell'header
    MOV     DH,HEADER_LINE_NO
    CALL    GOTO_XY
    LEA    DX,HEADER_PART_1
    CALL    WRITE_STRING
    MOV     DL,DISK_DRIVE_NO
    ADD    DL,'A'                ;Scrive disco A, B, ...
    CALL    WRITE_CHAR
    LEA    DX,HEADER_PART_2
    CALL    WRITE_STRING
    MOV     DX,CURRENT_SECTOR_NO
    CALL    WRITE_DECIMAL
    POP     DX
    RET
WRITE_HEADER    ENDP

```

La procedura WRITE_STRING non esiste ancora. Come potete vedere, si pensa di utilizzarla per scrivere una sequenza di caratteri sullo schermo. Le due stringhe HEADER_PART_1 e HEADER_PART_2, sono già definite in DSKPATCH.ASM. WRITE_STRING utilizzerà DS:DX come indirizzo per la stringa.

Si è pensato di utilizzare una procedura di visualizzazione della stringa che può contenere qualsiasi carattere, incluso il '\$', che non è possibile stampare con la funzione 9 del DOS. Dove il DOS utilizza il simbolo '\$' per determinare la fine di una

stringa, si utilizzerà il numero esadecimale 0. Ecco la procedura. Inseritela in VIDEO_IO.ASM:

Listato 17-10. Aggiungete questa procedura in VIDEO_IO.ASM

```

PUBLIC WRITE_STRING
;-----;
; Questa procedura scrive una stringa di caratteri sullo schermo. ;
; La stringa deve terminare con DB 0 ;
; ;
; In Ingresso: DS:DX Indirizzo della stringa ;
; ;
; Usa: WRITE_CHAR ;
;-----;
WRITE_STRING PROC
    PUSH AX
    PUSH DX
    PUSH SI
    PUSHF ;Salva flag direzione
    CLD ;Imposta direzione per incremento (avanti)
    MOV SI,DX ;Invia indirizzo a SI per LODSB
STRING_LOOP:
    LODSB ;Carica un carattere nel registro AL
    OR AL,AL ;Già arrivato a 0?
    JZ END_OF_STRING ;Sì, abbiamo finito con la stringa
    MOV DL,AL ;No, scrive carattere
    CALL WRITE_CHAR
    JMP STRING_LOOP
END_OF_STRING:
    POPF ;Ripristina flag di direzione
    POP SI
    POP DX
    POP AX
    RET
WRITE_STRING ENDP

```

Come è stata scritta adesso, WRITE_STRING scriverà i caratteri che hanno il codice ASCII inferiore a 32 (il carattere spazio) come un punto (.), perché non avete una versione di WRITE_CHAR capace di scrivere qualsiasi carattere. Si terrà conto di questo dettaglio nel prossimo capitolo, e—questo è il vantaggio del design modulare — non dovrete cambiare WRITE_STRING.

Dopo tutto il lavoro svolto in questo capitolo, è possibile mettere la ciliegina sulla torta. Cambiate DISK_PATCH in DSKPATCH.ASM per includere la CALL a WRITE_HEADER:

Listato 17-11. Cambiamenti a DISK_PATCH in DSKPATCH.ASM

```

EXTRN CLEAR_SCREEN: PROC, READ_SECTOR:PROC
EXTRN INIT_SEC_DISP:PROC, WRITE_HEADER:PROC
DISK_PATCH PROC
    MOV AC,DGROUP           ;Mette il segmento dati in AX
    MOV DS,AX               ;Imposta DS per puntare ai dati

    CALL CLEAR_SCREEN
    CALL WRITE_HEADER
    CALL READ_SECTOR
    CALL INIT_SEC_DISP

    MOV AH,4Ch              ;Ritorna al DOS
    INT 21h
DISK_PATCH ENDP
    
```

Il programma dovrebbe visualizzare una schermata di questo tipo:

Disco A Settore 0

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	0123456789ABCDEF
00	EB	34	90	49	42	4D	20	20	33	2E	33	00	02	04	01	00	04ÉIBM .3.3....
10	02	00	02	EF	A9	F8	2B	00	11	00	08	00	11	00	00	00	...n°+.<...<...
20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	123LÀ.É...
30	00	00	00	00	01	00	FA	33	C0	8E	D0	BC	00	7C	16	076+7AU.S+I...
40	BB	78	00	36	C5	37	1E	56	16	53	BF	2B	7C	B9	0B	00	¼&G=.t.&è.-è-f±
50	FC	AC	26	80	3D	00	74	03	26	8A	05	AA	8A	C4	E2	F1	.vèG. .+I+J= rgá>
60	06	1F	89	47	02	C7	07	2B	7C	FB	CD	13	72	67	A0	10	iú=&.!....ú?
70	7C	98	F7	26	16	7C	03	06	1C	7C	03	06	0E	7C	A3	3F	iú7!q .=&<IYA.!
80	7C	A3	37	7C	B8	20	00	F7	26	11	7C	8B	1E	0B	7C	03	H=&...i...i?i&f
90	C3	48	F7	F3	01	06	37	7C	BB	00	05	A1	3F	7C	E8	9F	.q...& .rvI&f...&J
A0	00	B8	01	02	E8	B3	00	72	19	8B	FB	B9	0B	00	BE	D9	}&u.i&J&J}...&^
B0	7D	F3	A6	75	0D	8D	7F	20	BE	E4	7D	B9	0B	00	F3	A6	t^Jw}J.2Σ= ^v&.!
C0	74	18	BE	77	7D	E8	6A	00	32	E4	CD	16	5E	1F	8F	04	âD =J->δ&í- 3q=&
D0	8F	44	02	CD	19	BE	C4	7D	EB	EB	A1	1C	05	33	D2	F7	6.!=&<I7!ú=i&.
E0	36	0B	7C	FE	C0	A2	3C	7C	A1	37	7C	A3	3D	7C	BB	00	.i7!&I.i^!*.;I08
F0	07	A1	37	7C	EB	49	00	A1	18	7C	2A	06	3B	7C	40	38	

A)_

Figura 17-1. Dskpatch con l'intestazione nella parte alta dello schermo

SOMMARIO

Avete finalmente incontrato le routine del BIOS del PC IBM e ne avete utilizzate due per la creazione del programma Dskpatch.

Avete imparato l'istruzione INT 10h, funzione numero 6, che avete utilizzato per cancellare lo schermo. Avete anche visto, seppur brevemente, che questa funzione offre molti altri vantaggi oltre a quelli visti in questo libro. Per esempio, è possibile utilizzarla per far scorrere zone dello schermo.

Avete quindi utilizzato la funzione 2 di INT 10h per spostare il cursore sulla terza linea dello schermo (linea numero 2), dove avete iniziato a stampare il contenuto del settore.

Per rendere il programma più facile, avete riscritto parecchie procedure in modo che utilizzino delle variabili di memoria, invece che numeri fissi. Ora siete in grado di leggere altri settori, e di cambiare il modo in cui il programma funziona, solo cambiando alcuni numeri in DSKPATCH.ASM.

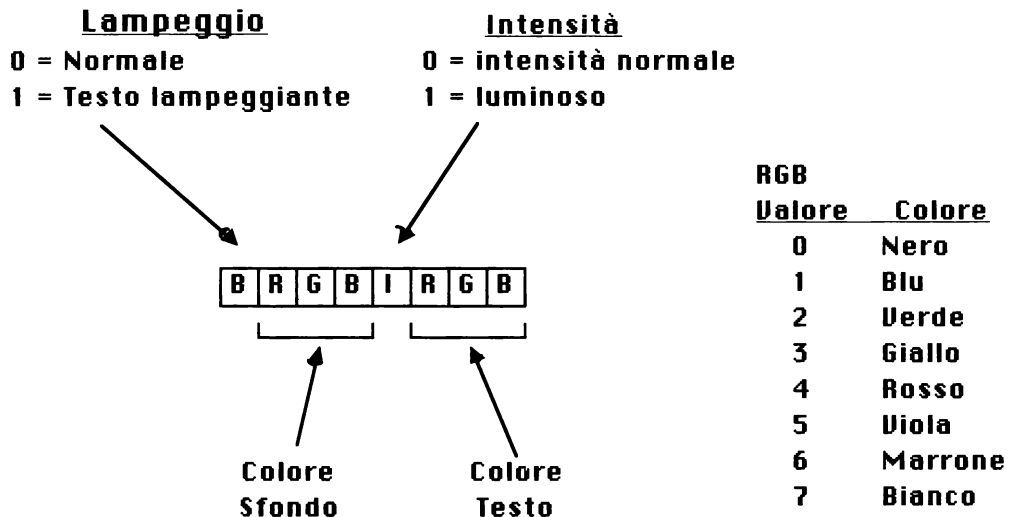
Infine, avete scritto le procedure WRITE_HEADER e WRITE_STRING, in modo da poter scrivere un'intestazione in cima allo schermo. Come già detto, scriverete una versione più completa di WRITE_CHAR nel prossimo capitolo, rimpiazzando i punti nella finestra ASCII con i rispettivi caratteri grafici. E grazie al design modulare, potete fare questo senza cambiare le procedure che utilizzano WRITE_CHAR.

WRITE_CHAR

Nell'ultimo capitolo avete fatto un buon uso delle routine del BIOS per cancellare lo schermo e spostare il cursore. Ma ci sono molti altri utilizzi, alcuni dei quali saranno presentati in questo capitolo.

Utilizzando il DOS da solo, non siete stati in grado di visualizzare tutti i 256 caratteri che il PC IBM può utilizzare. Così, in questo capitolo, presentiamo una nuova versione di WRITE_CHAR in grado di svolgere questo compito, grazie a un'altra funzione VIDEO_IO.

Quindi aggiungerete un'altra procedura utile, chiamata CLEAR_TO_END_OF_LINE, che cancella la linea dalla posizione del cursore fino al limite destro dello schermo. La utilizzerete in WRITE_HEADER, in modo che cancelli il resto della linea.



Attributo = colore di sfondo * 16 + colore del testo

Figura 18-1. Tabella dei colori

Supponete che ci si voglia spostare dal settore numero 10 (due cifre) al settore numero 9. Uno 0 sarebbe lasciato sullo schermo dopo la chiamata a `WRITE_HEADER` con il numero di settore impostato a 9. `CLEAR_TO_END_OF_LINE` cancellerà questo zero e tutto quello che rimane fino al margine destro.

LA NUOVA WRITE_CHAR

La funzione 9 della ROM BIOS per INT 10h scrive un carattere e il relativo *attributo* nella posizione corrente del cursore. L'attributo controlla alcune caratteristiche come il sottolineato, il lampeggio, e il colore (vedere figura 18-1). Utilizzeremo solo due attributi per Dskpatch: l'attributo 7, che è quello normale, e l'attributo 70h, che è un primo piano di colore zero e uno sfondo di colore sette (caratteri neri su sfondo bianco). E' possibile impostare gli attributi individualmente per ogni carattere, e lo faremo in seguito per creare il cursore in inverso (conosciuto anche come cursore *phantom*). Per ora utilizzeremo gli attributi normali quando scriveremo un carattere. La funzione 9 di INT 10h, scrive il carattere e l'attributo nella posizione corrente del cursore ma, a differenza del DOS, non porta il cursore nella posizione successiva a meno che non debba scrivere più di una volta il carattere. Utilizzerete questo successivamente, in una procedura differente; ma ora, dato che vogliamo una sola copia di ogni carattere, sposteremo il cursore da soli.

Ecco la nuova versione di `WRITE_CHAR` in `VIDEO_IO.ASM`:

Listato 18-1. Cambiamenti a `WRITE_CHAR` in `VIDEO_IO.ASM`

```

PUBLIC  WRITE_CHAR
EXTRN  CURSOR_RIGHT:PROC
;-----;
; Questa procedura visualizza un carattere sullo schermo utilizzando le      ;
; routine della ROM BIOS, in modo che caratteri come backspace siano      ;
; trattati come qualsiasi altro carattere da visualizzare.                 ;
; Questa procedura deve effettuare parecchie operazioni per aggiornare     ;
; la posizione del cursore.                                                ;
;                                                                           ;
; Inserimento: DL      Byte da stampare sullo schermo                    ;
;                                                                           ;
; Usa:      CURSOR_RIGHT
;-----;
WRITE_CHAR  PROC
            PUSH  AX
            PUSH  BX
            PUSH  CX
            PUSH  DX
            CMP   DL,32      ;Il carattere è prima di spazio?
            JAE   IS_PRINTABLE ;No, quindi si stampi così
            MOV   DL, '.'    ;Sì, sostituisilo con un punto (.)

```

```

IS_PRINTABLE
    MOV     AH,2
    INT     21H           ;Manda il carattere nel registro DL
    MOV     AH,9         ;Richiede l'output del carattere/attributo
    MOV     BH,0         ;Imposta su visualizzazione pagina 0
    MOV     CX,1         ;Scrive un solo carattere
    MOV     AL,DL        ;Carattere da scrivere
    MOV     BL,7         ;Attributo normale
    INT     10h         ;Scrive carattere e attributo
    CALL    CURSOR_RIGHT ;Passa alla successiva posizione del cursore
    POP     DX
    POP     CX
    POP     BX
    POP     AX
    RET
WRITE_CHAR     ENDP

```

Nel leggere questa procedura, vi sarete chiesti perché abbiamo incluso l'istruzione MOV BH,0. Se avete un adattatore per grafici, quest'ultimo ha quattro pagine di testo in modalità normale. Utilizzeremo solo la prima pagina, pagina 0; da qui l'istruzione MOV.

Per il cursore, WRITE_CHAR utilizza la procedura CURSOR_RIGHT per spostare il cursore a destra di un carattere o all'inizio della riga successiva, se il movimento porta il cursore oltre la colonna 79. Inserite le seguenti procedure in CURSOR.ASM:

Listato 18-2. Aggiungete questa procedura a CURSOR.ASM

```

PUBLIC CURSOR_RIGHT
;-----;
; Questa procedura sposta il cursore a destra di una posizione o alla ;
; riga successiva se il cursore si trova a fine riga. ;
; ;
; Usa: SEND_CRLF ;
;-----;
CURSOR_RIGHT PROC
    PUSH AX
    PUSH BX
    PUSH CX
    PUSH DX
    MOV AH,3 ;Legge la posizione corrente del cursore
    MOV BH,0 ;A pagina 0
    INT 10h ;Legge la posizione del cursore
    MOV AH,2 ;Imposta nuova posizione del cursore
    INC DL ;Imposta colonna sulla posizione successiva
    CMP DL,79 ;Si assicura che la colonna sia <= 79
    JBE OK
    CALL SEND_CRLF ;Va alla riga successiva
    JMP DONE
OK: INT 10h

```

```

DONE:      POP     DX
           POP     CX
           POP     BX
           POP     AX
           RET
CURSOR_RIGHT  ENDP
    
```

CURSOR_RIGHT utilizza due nuove funzioni di INT 10h. La funzione 3 legge la posizione del cursore, e la funzione 2 cambia la posizione. La prima procedura utilizza la funzione 3 per trovare la posizione del cursore; questa viene restituita in due byte, il numero della colonna in DL, e il numero della riga in DH. Quindi CURSOR_RIGHT incrementa il numero di colonna (in DL) e sposta il cursore. Se DL fosse stato nell'ultima colonna (79), la procedura avrebbe inviato un CR/LF per spostare il cursore alla posizione successiva. In Dskpatch non serve il controllo della colonna 79, ma lo includiamo in CURSOR_RIGHT per rendere la procedura di uso generale.

Con questi cambiamenti, Dskpatch dovrebbe ora visualizzare tutti i 256 caratteri come mostrato in figura 18-2.

Potete verificarlo cercando un byte con valore inferiore a 20h e vedendo gli strani caratteri che rimpiazzano i punti (.) nella finestra ASCII.

Ora facciamo qualcosa di più interessante: scriviamo una procedura per cancellare una riga a partire dalla posizione del cursore fino alla fine.

Disco A Settore 0

00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	0123456789ABCDEF
00 EB 34 90 49 42 4D 20 20 33 2E 33 00 02 04 01 00	δ4ÉIBM 3.3
10 02 00 02 EF A9 F8 2B 00 11 00 08 00 11 00 00 00	n r ° + < . <
20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 12	
30 00 00 00 00 01 00 FA 33 C0 0E D0 BC 00 7C 16 07	.3L&L !
40 BB 78 00 36 C5 37 1E 56 16 53 BF 2B 7C B9 0B 00	x 6+7AU S1+i
50 FC AC 26 00 3D 00 74 03 26 8A 05 AA 8A C4 E2 F1	%&C= t &è -è-Γ±
60 06 1F 09 47 02 C7 07 2B 7C FB CD 13 72 67 A0 10	vèG +iJ= rgá)
70 7C 98 F7 26 16 7C 03 06 1C 7C 03 06 0E 7C A3 3F	iÿ=& ! L i iú?
80 7C A3 37 7C B8 20 00 F7 26 11 7C 0B 1E 0B 7C 03	iú7iÿ =&<iYA !
90 C3 48 F7 F3 01 06 37 7C BB 00 05 A1 3F 7C E8 9F	H=¿ 7iÿ i?i&f
A0 00 B8 01 02 E8 B3 00 72 19 0B FB B9 0B 00 BE D9	ÿ rviJ dJ
B0 7D F3 A6 75 0D 0D 7F 20 BE E4 7D B9 0B 00 F3 A6	}¿&u i& dΣ} ¿&
C0 74 18 BE 77 7D E8 6A 00 32 E4 CD 16 5E 1F 8F 04	t^Jw}dJ ZΣ= ^VA
D0 8F 44 02 CD 19 BE C4 7D EB EB A1 1C 05 33 D2 F7	âD =vJ -}d&iL 3ÿ#
E0 36 0B 7C FE C0 A2 3C 7C A1 37 7C A3 3D 7C BB 00	6 !uL6<i i7iú=iÿ
F0 07 A1 37 7C E8 49 00 A1 18 7C 2A 06 3B 7C 40 38	i7i&i i^! * ;i&

A>_

Figura 18-2. Dskpatch con WRITE_CHAR

CANCELLARE FINO ALLA FINE DELLA RIGA

Nell'ultimo capitolo, abbiamo utilizzato la funzione 6 di INT 10h, per cancellare lo schermo con la procedura CLEAR_SCREEN. Avevamo anche detto che la funzione 6 è in grado di cancellare qualsiasi finestra rettangolare. Questa capacità è valida anche se la finestra è alta solo una riga e lunga meno di una riga; è possibile quindi utilizzare la funzione 6 per cancellare parti di righe.

La parte sinistra della finestra, in questo caso, è il numero di colonna in cui si trova il cursore, che è possibile ottenere con la funzione 3 (utilizzata anche da CURSOR_RIGHT). La parte destra della finestra è sempre alla colonna 79. E' possibile vedere i dettagli in CLEAR_TO_END_OF_LINE; mettete questa procedura in CURSOR.ASM:

Listato 18-3 Aggiungete questa procedura a CURSOR.ASM

```

PUBLIC CLEAR_TO_END_OF_LINE
;-----;
; Questa procedura cancella la riga dalla posizione corrente del cursore ;
; alla fine della riga stessa. ;
;-----;
CLEAR_TO_END_OF_LINE PROC
    PUSH AX
    PUSH BX
    PUSH CX
    PUSH DX
    MOV AH,3 ;Legge la posizione corrente del cursore a pag. 0
    XOR BH,BH
    INT 10h ;Si hanno ora (X,Y) in DL, DH
    MOV AH,6 ;Imposta per cancellare fino alla fine riga
    XOR AL,AL ;Cancella finestra
    MOV CH,DH ;Tutto sulla stessa riga
    MOV CL,DL ;Inizia dalla posizione del cursore
    MOV DL,79 ;E termina alla fine della riga
    MOV BH,7 ;Usa gli attributi normali
    INT 10h
    POP DX
    POP CX
    POP BX
    POP AX
    RET
CLEAR_TO_END_OF_LINE ENDP

```

Utilizzerete questa procedura in WRITE_HEADER, per cancellare il resto della riga quando si inizia a leggere un altro settore (lo faremo molto presto). Non c'è nessun modo per vedere CLEAR_TO_END_OF_LINE funzionare con WRITE_HEADER finché non vengono aggiunte le procedure che permettono di leggere un settore differente e di aggiornare lo schermo. Apportate i seguenti cambiamenti a WRITE_HEADER in

VIDEO_IO.ASM, per richiamare CLEAR_TO_END_OF_LINE alla fine della procedura:

Listato 18-4. Cambiamenti a WRITE_HEADER in VIDEO_IO.ASM

```

PUBLIC WRITE_HEADER
DATA_SEG SEGMENT PUBLIC
EXTRN HEADER_LINE_NO:BYTE
EXTRN HEADER_PART_1:BYTE
EXTRN HEADER_PART_2:BYTE
EXTRN DISK_DRIVE_NO:BYTE
EXTRN CURRENT_SECTOR_NO:WORD
DATA_SEG ENDS
EXTRN GOTO_XY:NEAR, CLEAR_TO_END_OF_LINE:NEAR
;-----;
; Questa procedura scrive l'intestazione con il numero di settore e
; l'identificativo del drive.
;
; Usa:          GOTO_XY, WRITE_STRING, WRITE_CHAR, WRITE_DECIMAL
;              CLEAR_TO_END_OF_LINE
; Legge:       HEADER_LINE_NO, HEADER_PART_1, HEADER_PART_2
;              DISK_DRIVE_NO, CURRENT_SECTOR_NO
;-----;
WRITE_HEADER PROC NEAR
    PUSH DX
    XOR DL,DL ;Sposta il cursore sulla riga
d'intestazione numero
    MOV DH,HEADER_LINE_NO
    CALL GOTO_XY
    LEA DX,HEADER_PART_1
    CALL WRITE_STRING
    MOV DL,DISK_DRIVE_NO
    ADD DL,'A' ;Scrive drive A, B, ...
    CALL WRITE_CHAR
    LEA DX,HEADER_PART_2
    CALL WRITE_STRING
    MOV DX,CURRENT_SECTOR_NO
    CALL WRITE_DECIMAL
    CALL CLEAR_TO_END_OF_LINE ;Cancella numero settore rimasto
    POP DX
    RET
WRITE_HEADER ENDP

```

Questa revisione segna sia la versione finale di WRITE_HEADER sia il completamento del file CURSOR.ASM. In ogni caso, mancano sempre delle parti importanti del file Dskpatch. Nel prossimo capitolo continueremo e aggiungeremo a Dispatch i comandi per la tastiera in modo da poter premere F3 e F4 per leggere altri settori del disco.

SOMMARIO

Questo capitolo è stato relativamente facile, senza troppe informazioni nuove. Avete imparato come utilizzare la funzione numero 9 di INT 10h, per scrivere qualsiasi carattere sullo schermo.

Durante questo processo, avete visto come leggere la posizione del cursore con la funzione 3 di INT 10h, in modo da poter spostare il cursore a destra di una posizione dopo aver scritto un carattere. La ragione di questa operazione è dovuta al fatto che INT 10h funzione 9 non sposta il cursore dopo aver scritto un carattere, a meno che non venga effettuata una copia del carattere. Alla fine avete utilizzato la funzione 6 di INT 10h per cancellare una parte di una riga.

Nel prossimo capitolo, ci metteremo ancora al lavoro per costruire la parte centrale di Dispatch.

LE ROUTINE DI SMISTAMENTO

In qualsiasi linguaggio è importante avere un programma ben scritto, ma è molto più importante renderlo interattivo con l'utente. E' nella natura umana poter dire "Se io faccio questo, tu fai quello", quindi utilizzeremo questo capitolo per aggiungere un po' di interattività a Dskpatch.

Scriverete una semplice procedura per l'input da tastiera. Il lavoro delle routine di smistamento (dispatcher) sarà quello di chiamare la procedura corretta per ogni tasto premuto. Per esempio, quando si preme F3 per leggere e visualizzare il settore precedente, dispatcher chiamerà una procedura chiamata PREVIOUS_SECTOR. Per fare questo, dovrete apportare molti cambiamenti a Dskpatch. Inizierete creando DISPATCHER e altre procedure per formattare la visualizzazione. Aggiungerete quindi altre due procedure, PREVIOUS_SECTOR e NEXT_SECTOR, che saranno chiamate da DISPATCHER.

LE ROUTINE DI SMISTAMENTO

Le routine di smistamento funzioneranno come controllo centrale per Dskpatch; tutti gli input da tastiera passeranno quindi attraverso questo. Il lavoro di DISPATCHER sarà quello di leggere i caratteri e di chiamare le procedure opportune. Presto vedrete come funziona, ma prima vediamo come metterlo in Dskpatch.

DISPATCHER avrà la sua linea di input, proprio sotto la visualizzazione del contenuto del settore, dove il cursore attende l'inserimento da tastiera. Non sarete in grado di inserire i numeri in esadecimale in questa prima versione della procedura, ma lo farete successivamente. Ecco la prima modifica a DSKPATCH.ASM; questa modifica aggiunge dei dati sulla riga di inserimento:

Listato 19-1. Aggiunte a DATA_SEG in DSKPATCH.ASM

```

HEADER_LINE_NO          DB          0
HEADER_PART_1           DB          'Disco ',0
HEADER_PART_2           DB          '          Settore ',0
PUBLIC PROMPT_LINE_NO, EDITOR_PROMPT
PROMPT_LINE_NO          DB          21
EDITOR_PROMPT           DB          'Premere un tasto funzione o introdurre'
                        DB          ' un carattere o byte esadecimale: ',0

```

Aggiungerete altre richieste, come l'input di un numero di settore, più tardi; in questo modo renderete il lavoro più semplice utilizzando una procedura, `WRITE_PROMPT_LINE`, per scrivere la riga voluta. Ogni procedura che utilizza `WRITE_PROMPT_LINE` fornirà l'indirizzo del prompt (qui l'indirizzo di `EDITOR_PROMPT`), e quindi scriverà il prompt sulla riga 21 (dal momento che `PROMPT_LINE_NO` è 21). Per esempio, questa nuova versione di `DISK_PATCH` (in `DSKPATCH.ASM`) utilizza `WRITE_PROMPT_LINE` prima di chiamare `DISPATCHER`:

Listato 19-2. Aggiunte a `DISK_PATCH` in `DSKPATCH.ASM`

```

EXTRN CLEAR_SCREEN:PROC, READ_SECTOR:PROC
EXTRN INIT_SEC_DISP:PROC, WRITE_HEADER:PROC
EXTRN WRITE_PROMPT_LINE:PROC, DISPATCHER:PROC

DISK_PATCH      PROC
MOV      AX,DGROUP      ;Mette il segmento dati in AX
MOV      DS,AX          ;Imposta DS per puntare ai dati
CALL     CLEAR_SCREEN
CALL     WRITE_HEADER
CALL     READ_SECTOR
CALL     INIT_SEC_DISP
LEA     DX,EDITOR_PROMPT
CALL     WRITE_PROMPT_LINE
CALL     DISPATCHER

MOV      AH,4C          ;Ritorna al DOS
INT      21h
DISK_PATCH      ENDP

```

Le routine di smistamento formano un programma relativamente semplice, ma bisogna utilizzare dei nuovi trucchi. Il listato seguente è la prima versione del file `DISPATCH.ASM`:

Listato 19-3. Il nuovo file `DISPATCH.ASM`.

```

.MODEL SMALL

.CODE

EXTRN NEXT_SECTOR:NEAR      ;In DISK_IO.ASM
EXTRN PREVIOUS_SECTOR:NEAR ;In DISK_IO.ASM

.DATA
;-----;
; Questa tabella contiene i tasti estesi ASCII ammessi e gli indirizzi ;
; delle procedure che devono essere richiamati alla pressione di ogni tasto. ;
; ;

```

```

; Il formato della tabella è
;          DB      72          ;Codice esteso per cursore alto ;
;          DW      OFFSET CGROUP:PHANTOM_UP
;-----;
DISPATCH_TABLE LABEL          BYTE
                DB      61          ;F3
                DW      OFFSET CGROUP:PREVIOUS_SECTOR
                DB      62          ;F4
                DW      OFFSET CGROUP:NEXT_SECTOR
                DB      0           ;Fine della tabella

.CODE

                PUBLIC DISPATCHER
                EXTRN READ_BYTE:PROC
;-----;
; Questa è la routine di smistamento principale. Durante le normali
; operazioni di editing e di visualizzazione questa procedura legge i
; caratteri dalla tastiera e, se il carattere è un tasto di comando
; (come ad esempio un tasto cursore), DISPATCHER chiama le procedure
; che effettuano il lavoro. Questo smistamento e' effettuato attraverso
; tutti i tasti elencati nella tabella DISPATCH_TABLE, dove gli
; indirizzi delle procedure sono memorizzati subito dopo i nomi dei
; tasti.
; Se il carattere non è un tasto speciale, dovrà essere introdotto
; direttamente nel buffer di settore (modalità di editing).
;
; Usa:          READ_BYTE
;-----;
DISPATCHER    PROC
                PUSH    AX
                PUSH    BX
DISPATCH_LOOP:
                CALL    READ_BYTE          ;Legge carattere in AX
                OR      AH,AH              ;AX = -10 se nessun carattere letto, 1
                                                ;per un codice esteso.
                JZ      DISPATCH_LOOP     ;Nessun carattere letto, riprova
                JS      SPECIAL_KEY       ;Letto codice esteso
                                                ;per ora non fare nessuna operazione sul carattere
                JMP     DISPATCH_LOOP     ;Legge un altro carattere
SPECIAL_KEY:
                CMP     AL,68              ;F10-uscita?
                JE      END_DISPATCH      ;Si, esci
                                                ;Usa BX per consultare tabella
                LEA    BX,DISPATCH_TABLE
SPECIAL_LOOP:
                CMP     BYTE PTR [BX],0    ;Fine tabella?
                JE      NOT_IN_TABLE      ;Si, il tasto non era presente in tabella
                CMP     AL,[BX]           ;Corrisponde a questo elemento di tabella?
                JE      DISPATCH          ;Si, allora smista
                ADD     BX,3               ;No, prova prossimo elemento
                JMP     SPECIAL_LOOP       ;Controlla elemento successivo in tabella

```

```

DISPATCH:
    INC    BX                ;Punta a indirizzo di procedura
    CALL  WORD PTR [BX]    ;Chiama procedura
    JMP   DISPATCH_LOOP    ;Attende altro tasto

NOT_IN_TABLE:                ;Non produce nulla legge il carattere successivo
    JMP   DISPATCH_LOOP

END_DISPATCH:
    POP   BX
    POP   AX
    RET
DISPATCHER    ENDP

END

```

DISPATCH_TABLE contiene i codici ASCII estesi da utilizzare per i tasti F3 e F4. Ogni codice è seguito dall'indirizzo che la procedura DISPATCHER dovrebbe chiamare quando legge quel particolare codice esteso. Per esempio, quando READ_BYTE, che è chiamato da DISPATCHER, legge il tasto F3 (codice esteso 61), DISPATCHER chiama la procedura PREVIOUS_SECTOR.

Gli indirizzi delle procedure che si vogliono far chiamare da DISPATCHER si trovano nella tabella delle routine di smistamento; per ottenere questi indirizzi è necessario utilizzare una nuova direttiva, OFFSET. La riga

```
DW    OFFSET _TEXT:PREVIOUS_SECTOR
```

per esempio, indica all'assemblatore di utilizzare l'*offset* della procedura PREVIOUS_SECTOR (questo offset viene calcolato in relazione all'inizio del segmento di codice _TEXT); questo è il motivo per cui è stato utilizzato _TEXT: prima del nome della procedura. (In questo contesto, _TEXT: non è necessario. In ogni caso, per chiarezza, scriveremo sempre OFFSET _TEXT:).

Notate che DISPATCH_TABLE contiene sia byte che parole. Questo può sollevare alcune considerazioni. In passato, avete sempre lavorato con tabelle di un tipo o di un altro: o tutte parole o tutti byte. Ma qui si possono trovare entrambi i tipi; bisogna quindi indicare all'assemblatore quali dati vengono utilizzati con un'istruzione CMP o CALL. Nel caso di un'istruzione scritta in questo modo:

```
CMP   [BX],0
```

l'assemblatore non capisce se si vuole fare un confronto sui byte o sulle parole. Ma scrivendo l'istruzione in questo modo:

```
CMP   BYTE PTR [BX],0
```

si indica all'assembler che BX punta ad un byte, e che si desidera confrontare dei byte.

Nello stesso modo, l'istruzione `CMP WORD PTR [BX],0` farà un confronto di parole. In altri casi, un'istruzione come `CMP AL,[BX]` non causa nessun problema dal momento che `AL` è un registro di un byte, e l'assemblatore ne è a conoscenza. Quindi, ricordate che un'istruzione `CALL` può essere sia `NEAR` sia `FAR`. A una `NEAR CALL` serve una parola per l'indirizzo, mentre a una `FAR CALL` ne servono due. L'istruzione:

```
CALL WORD PTR [BX]
```

indica all'assemblatore, con `WORD PTR`, che `[BX]` punta ad una parola, quindi dovrebbe generare una `NEAR CALL` e utilizzare la parola puntata da `[BX]` come indirizzo, che diventa l'indirizzo salvato in `DISPATCH_TABLE`. (Per una `FAR CALL`, che utilizza un indirizzo a due parole, si utilizzerà l'istruzione `CALL DWORD PTR [BX]`. `DWORD` significa *Double Word*, o due parole).

Come vedrete nel Capitolo 22, si potranno aggiungere altri comandi a `Dskpatch` semplicemente aggiungendo altre procedure e inserendo dei nuovi dati in `DISPATCH_TABLE`. Bisogna ora aggiungere quattro nuove procedure prima di poter verificare la nuova versione di `Dskpatch`: `READ_BYTE`, `WRITE_PROMPT_LINE`, `PREVIOUS_SECTOR`, e `NEXT_SECTOR`.

`READ_BYTE` è una procedura per leggere i codici estesi dei caratteri dalla tastiera. La versione finale sarà in grado di leggere dei tasti speciali (come i tasti per lo spostamento del cursore e i tasti funzione), i caratteri ASCII, e numeri esadecimali a due cifre. A questo punto, scriverete una semplice versione di `READ_BYTE`, per leggere sia un carattere che un tasto speciale. Ecco la prima versione di `KBD_IO.ASM`, che è il file in cui salverete le procedure per leggere i caratteri dalla tastiera:

Listato 19-4. Il nuovo file `KBD_IO.ASM`

```
.MODEL SMALL
.CODE

        PUBLIC  READ_BYTE

;-----;
; Questa procedura legge un carattere ASCII. Questa e' solo una          ;
; versione di prova di READ_BYTE.                                       ;
;                                                                           ;
; Ritorna:          AL          Codice carattere (ad eccezione di AH=0)   ;
;                  AH          0 se legge un carattere ASCII             ;
;                  AH          1 se legge un tasto speciale              ;
;-----;

READ_BYTE PROC
        XOR  AH,AH                ;Richiede la funzione per leggere da tastiera
        INT  16h                 ;Legge un carattere/scan code dalla tastiera
        OR   AL,AL                ;E' un codice esteso?
        JZ   EXTENDED_CODE       ;Si
NOT_EXTENDED:
        XOR  AH,AH                ;Ritorna solo il codice ASCII
DONE_READING:
```

```

RET

EXTENDED_CODE:
MOV    AL,AH          ;Mette lo scan code in AL
MOV    AH,1          ;Segnala un codice esteso
JMP    DONE_READING
READ_BYTE ENDP

END

```

READ_BYTE utilizza un nuovo interrupt, INT 16h, che è un interrupt che fornisce l'accesso ai servizi della tastiera nella ROM BIOS. La funzione 0 legge un carattere dalla tastiera senza visualizzarlo sullo schermo. Ritorna il codice in AL, e lo *scan code* nel registro AH.

Lo scan code è il codice assegnato ad ogni tasto della tastiera. Certi tasti, come F3 non hanno un codice ASCII (il che significa che AL sarà 0), ma hanno uno scan code (troverete una tabella degli scan code nell'Appendice D). READ_BYTE mette questo scan code nel registro AL per i tasti speciali, e imposta AH a 1.

Aggiungete ora la nuova procedura WRITE_PROMPT_LINE a DISP_SEC.ASM:

Listato 19-5. Aggiungete questa procedura a VIDEO_IO.ASM

```

PUBLIC WRITE_PROMPT_LINE
EXTRN CLEAR_TO_END_OF_LINE:PROC, WRITE_STRING:PROC
EXTRN GOTO_XY:PROC

.DATA
    EXTRN PROMPT_LINE_NO:BYTE

.CODE
;-----;
; Questa procedura scrive la riga di messaggio sullo schermo e cancella ;
; la parte restante della riga. ;
; ;
; Inserimento: DS:DX Indirizzo della riga di messaggio ;
; ;
; Usa: WRITE_STRING, CLEAR_TO_END_OF_LINE, GOTO_XY ;
; Legge: PROMPT_LINE_NO ;
;-----;
WRITE_PROMPT_LINE PROC
    PUSH    DX
    XOR     DL,DL          ; Scrive la riga di messaggio e
    MOV     DH,PROMPT_LINE_NO ; sposta il cursore su quella riga
    CALL    GOTO_XY
    POP     DX
    CALL    WRITE_STRING
    CALL    CLEAR_TO_END_OF_LINE
    RET
WRITE_PROMPT_LINE ENDP

```


Non c'è molto in questa procedura. Il cursore viene spostato all'inizio della riga di inserimento, che è stata impostata come riga 21 (in DSKPATCH.ASM), e la riga di inserimento viene quindi scritta e viene cancellato il resto della riga. Il cursore sarà nella giusta posizione una volta completata WRITE_PROMPT_LINE, e il resto della riga verrà cancellata da CLEAR_TO_END_OF_FILE.

LEGGERE ALTRI SETTORI

A questo punto servono due procedure, PREVIOUS_SECTOR e NEXT_SECTOR, per leggere e visualizzare nuovamente il settore precedente o successivo. Aggiungete queste due procedure a DISK_IO.ASM:

Listato 19-6. Aggiungete queste procedure a DISK_IO.ASM

```

PUBLIC PREVIOUS_SECTOR
EXTRN INIT_SEC_DISP:PROC, WRITE_HEADER:PROC
EXTRN WRITE_PROMPT_LINE:PROC

.DATA
    EXTRN CURRENT_SECTOR_NO:WORD, EDITOR_PROMPT:BYTE

.CODE
;-----;
; Questa procedura legge il settore precedente, se possibile. ;
; ;
; Usa:          WRITE_HEADER, READ_SECTOR, INIT_SEC_DISP ;
;              WRITE_PROMPT_LINE ;
; Legge:       CURRENT_SECTOR_NO, EDITOR_PROMPT ;
; Scrive:      CURRENT_SECTOR_NO ;
;-----;
PREVIOUS_SECTOR PROC
    PUSH  AX
    PUSH  DX
    MOV   AX,CURRENT_SECTOR_NO    ;Rileva il numero del settore corrente
    OR    AX,AX                    ;Non decrementa se già 0
    JZ    DONT_DECREMENT_SECTOR
    DEC   AX
    MOV   CURRENT_SECTOR_NO,AX    ;Salva nuovo numero di settore
    CALL  WRITE_HEADER
    CALL  READ_SECTOR
    CALL  INIT_SEC_DISP            ;Visualizza nuovo settore
    LEA  DX,EDITOR_PROMPT
    CALL  WRITE_PROMPT_LINE
DONT_DECREMENT_SECTOR:
    POP   DX
    POP   AX
    RET
PREVIOUS_SECTOR ENDP

PUBLIC NEXT_SECTOR

```

```

        EXTRN  INIT_SEC_DISP:PROC, WRITE_HEADER:PROC
        EXTRN  WRITE_PROMPT_LINE:PROC

.DATA
        EXTRN  CURRENT_SECTOR_NO:WORD, EDITOR_PROMPT:BYTE

.CODE
;-----;
; Legge il settore successivo.                                     ;
;                                                                 ;
; Usa:          WRITE_HEADER, READ_SECTOR, INIT_SEC_DISP        ;
;              WRITE_PROMPT_LINE                               ;
; Legge:        CURRENT_SECTOR_NO, EDITOR_PROMPT               ;
; Scrive:       CURRENT_SECTOR_NO                              ;
;-----;
NEXT_SECTOR    PROC
                PUSH  AX
                PUSH  DX
                MOV   AX,CURRENT_SECTOR_NO
                INC   AX                    ;Passa al settore successivo
                MOV   CURRENT_SECTOR_NO,AX
                CALL  WRITE_HEADER
                CALL  READ_SECTOR
                CALL  INIT_SEC_DISP        ;Visualizza il nuovo settore
                LEA  DX,EDITOR_PROMPT
                CALL  WRITE_PROMPT_LINE
                POP   DX
                POP   AX
                RET
NEXT_SECTOR    ENDP

```

Ora siete pronti per assemblare tutti i file che avete creato o cambiato: Dskpatch, Video_io, Kbd_io, Dispatch, e Disk_io. Quando collegate i file, ricordate che sono sette: Dskpatch, Disp_sec, Disk_io, Video_io, Kbd_io, Dispatch, e Cursor. Se utilizzate un Make, ecco le aggiunte che dovete fare al Makefile (la barra rovesciata alla fine della terza riga dal basso, indica a Make che la lista continua sulla riga successiva):

Listato 19-7. Cambiamenti a MAKEFILE

```

cursor.obj:    cursor.asm
               masm cursor;
dispatch.obj:  dispatch.asm
               masm dispatch;
Kbd_io.obj:    kbd_io.asm
               masm kbd_io;
dskpatch.exe:  dskpatch.obj disk_io.obj disp_sec.obj video_io.obj cursor.obj \
               dispatch.obj kbd_io.obj
               link dskpatch disk_io disp_sec video_io cursor dispatch kbd_io;

```

Disco A

Settore 0

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	0123456789ABCDEF
00	EB	34	90	49	42	4D	20	20	33	2E	33	00	02	04	01	00	64E1BM 3.3
10	02	00	02	EF	A9	F8	2B	00	11	00	00	00	11	00	00	00	n-°+ < . <
20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	12	
30	00	00	00	00	01	00	FA	33	C0	0E	D0	BC	00	7C	16	07	.3LxL ;
40	BB	78	00	36	C5	37	1E	56	16	53	BF	2B	7C	B9	0B	00	x 6+7AV S1+i
50	FC	AC	26	00	3D	00	74	03	26	8A	05	AA	8A	C4	E2	F1	%&C= t &è -è-Γ±
60	06	1F	09	47	02	C7	07	2B	7C	FB	CD	13	72	67	A0	10	vèG +iJ= rgá>
70	7C	98	F7	26	16	7C	03	06	1C	7C	03	06	0E	7C	A3	3F	iù?i? =&<i?A !
80	7C	A3	37	7C	B8	20	00	F7	26	11	7C	0B	1E	0B	7C	03	H=≤ 7i? i?i?f
90	C3	48	F7	F3	01	06	37	7C	BB	00	05	A1	3F	7C	E8	9F	7 0 rvi? dJ
A0	00	B8	01	02	E8	B3	00	72	19	0B	FB	B9	0B	00	BE	D9	}≤u iΔ JΣ} ≤
B0	7D	F3	A6	75	0D	0D	7F	20	BE	E4	7D	B9	0B	00	F3	A6	t^Jw}0j 2Σ= ^vA
C0	74	18	BE	77	7D	E8	6A	00	32	E4	CD	16	5E	1F	8F	04	AD =vJ-}δδiL 3T=
D0	8F	44	02	CD	19	BE	C4	7D	EB	EB	A1	1C	05	33	D2	F7	6 !L6<i!7!ú=iq
E0	36	0B	7C	FE	C0	A2	3C	7C	A1	37	7C	A3	3D	7C	BB	00	i?i?i i^!* ;!08
F0	07	A1	37	7C	E8	49	00	A1	18	7C	2A	06	3B	7C	40	38	

Premere un tasto funzione o introdurre carattere o byte esadecimale: _

Figura 19-1. Dskpatch con la linea del Prompt

(Ricordate che le ultime tre righe devono essere messe all'inizio del file se utilizzate il Make della Borland. Se utilizzate OPTASM, dovete aggiungere quattro righe per assemblare dispatch e kbd_io). Se non avete un Make, potreste scrivere un batch file come il seguente per creare il file .EXE:

```
LINK DSKPATCH DISK_IO DISP_SEC VIDEO_IO CURSOR DISPATCH KBD_IO;
```

Se aggiungete altri file, dovrete solo cambiare il file batch invece che riscrivere tutto il comando per creare il programma .EXE.

Questa versione di Dskpatch ha tre tasti attivi: F3 legge e visualizza il settore precedente, fermandosi al settore 0; F4 legge il settore successivo; F10 esce da Dskpatch. Ora, se avviate il programma, dovrete vederlo come in Figura 19-1.

FILOSOFIA DEI CAPITOLI SUCCESSIVI

In questo capitolo avete imparato molto di più rispetto ai precedenti. Questa sarà la filosofia dei capitoli che vanno dal 20 al 27. Da questo punto in avanti, vi saranno presentati molti esempi su come scrivere grossi programmi. Troverete inoltre più procedure di quelle che utilizzerete nei programmi.

Nei capitoli della Parte IV tornerete ad imparare cose nuove, quindi abbiate pazienza,

o (se volete) saltate i rimanenti capitoli su Dskpatch fino a quando non siete pronti per scrivere i vostri programmi.

Nei prossimi capitoli troverete tantissimi aiuti, e una grossa quantità di dettagli per scrivere le procedure.

Dal Capitolo 21, vi saranno presentate molte procedure e vi sarà dato modo di scoprire come funzionano. Perché? Ci sono due ragioni, entrambe legate al fatto di farvi imparare la programmazione in linguaggio assembly. Innanzitutto si creerà una libreria di procedure che potrà essere utilizzata in altri programmi. In secondo luogo, presentandovi questo grosso esempio di programmazione, non si vuole solo mostrare come si scrivono grossi programmi, ma anche darvi qualche suggerimento per farlo. Quindi seguite la parte seguente di questo libro nel modo migliore. Il Capitolo 20 è per quelle persone che vogliono scrivere dei programmi. Nel Capitolo 21, torneremo a Dskpatch e scriveremo le procedure per scrivere e spostare quello che viene chiamato cursore fantasma: un cursore in inverso per la visualizzazione ASCII ed esadecimale.

UNA SFIDA ALLA PROGRAMMAZIONE

Questo libro contiene sei capitoli di procedure. Se volete provare a crearne una, leggete questo capitolo in cui sarà progettata una procedura che sarà poi scritta nei capitoli 21 e 22. Potrete quindi provare a scrivere le procedure di ogni capitolo prima di leggerlo. Se non volete scrivere un altro pezzo di Dskpatch, per ora potete saltare questo capitolo.

Se decidete di leggerlo, ecco un suggerimento su come procedere: leggete una sezione e provate ad apportare i cambiamenti suggeriti a Dskpatch. Quando pensate di aver finito, leggete il capitolo con lo stesso titolo del paragrafo. Quando avete terminato di leggere il capitolo corrispondente potete proseguire con il paragrafo successivo.

Nota: Dovreste fare una copia di tutti i file prima di fare i cambiamenti. In questo modo, quando arriverete al Capitolo 21, potrete scegliere se proseguire con la vostra versione o apportare le modifiche suggerite.

IL CURSORE FANTASMA

Nel Capitolo 21, si utilizzeranno due cursori fantasma sullo schermo: uno nella finestra esadecimale, l'altro nella finestra ASCII. Un cursore fantasma è simile ad un cursore normale, ma non lampeggia e lo sfondo diventa nero con i caratteri in bianco, come potete vedere nella Figura 20-1.

Il cursore nella finestra esadecimale è largo quattro caratteri, mentre quello nella finestra ASCII solo uno.

Come creare un cursore fantasma? Ogni carattere sullo schermo ha un byte che identifica l'*attributo*. Questo byte dice al PC come visualizzare ogni carattere. Un byte uguale a 7h visualizza il carattere normalmente, mentre 70h visualizza un carattere in video inverso. L'ultimo è proprio quello che vogliamo per costruire il cursore fantasma. Ecco quindi il problema: come è possibile cambiare l'attributo dei caratteri per farlo diventare 70h?

INT 10h funzione 9, scrive sullo schermo sia il carattere che l'attributo, e INT 10h funzione 8 legge il codice del carattere nella posizione attuale del cursore.

E' possibile creare un cursore fantasma nella finestra esadecimale seguendo questi passaggi:

- Salvate la posizione del cursore reale (INT 10h funzione 3 per leggere la posizione del cursore e salvarla in una variabile).
- Spostate il cursore vero all'inizio del cursore fantasma nella finestra esadecimale.
- Per i quattro caratteri successivi, leggete il codice del carattere (funzione 8) e scrivete sia il carattere che il suo attributo (impostando l'attributo a 70h).
- Alla fine, ripristinate la vecchia posizione del cursore.

Per scrivere il cursore nella finestra ASCII si segue praticamente lo stesso procedimento. Una volta che avete il cursore nella finestra esadecimale, potete aggiungere i codici extra per la finestra ASCII.

Ricordate che il vostro primo tentativo è solo temporaneo. Una volta che avete un programma funzionante per il cursore fantasma, potete tornare indietro e riscrivere i cambiamenti, in modo da aver un certo numero di piccole procedure che fanno il lavoro. Date un'occhiata alle procedure del Capitolo 21 quando avete finito.

UN SEMPLICE EDITING

Una volta che avete creato il cursore fantasma, dovrete fare in modo che sia spostabile sullo schermo. Dovete fare attenzione a determinate condizioni per mantenere i cursori fantasma all'interno di ogni finestra. Bisogna anche che i due cursori si

```

Disco A           Settore 0

00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  0123456789ABCDEF
00 EB 34 98 49 42 4D 28 28 33 2E 33 00 02 04 01 00  04ÉIBM 3.3
10 02 00 02 EF A9 F8 2B 00 11 00 08 00 11 00 00 00  nr"+ < . <
20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 12
30 00 00 00 00 01 00 FA 33 C0 0E D0 BC 00 7C 16 07      .3LÄL!
40 BB 78 00 36 C5 37 1E 56 16 53 BF 2B 7C B9 0B 00  x 6+7AU S1+i
50 FC AC 26 00 3D 00 74 03 26 8A 05 AA 8A C4 E2 F1  %&C= t àè -è-f±
60 06 1F 89 47 02 C7 07 2B 7C FB CD 13 72 67 A0 10  vëG | +i|= rgá)
70 7C 98 F7 26 16 7C 03 06 1C 7C 03 06 0E 7C A3 3F  iÿ=& ! |! iú?
80 7C A3 37 7C B8 20 00 F7 26 11 7C 0B 1E 0B 7C 03  iú7!q =&<i!A !
90 C3 48 F7 F3 01 06 37 7C BB 00 05 A1 3F 7C E8 9F  |H=≤ 7!q i?i&f
A0 00 00 01 02 E8 B3 00 72 19 0B FB B9 0B 00 BE D9  q 0| rvÿ!j dj
B0 7D F3 A6 75 0D 8D 7F 28 BE E4 7D B9 0B 00 F3 A6  }≤u ià dΣ}+ ≤≤
C0 74 18 BE 77 7D E8 6A 00 32 E4 CD 16 5E 1F 8F 04  t^jw)ÿj ΣΣ= ^vâ
D0 8F 44 02 CD 19 BE C4 7D EB EB A1 1C 05 33 D2 F7  ÅD =v-}δδiL 3q=
E0 36 0B 7C FE C8 A2 3C 7C A1 37 7C A3 3D 7C BB 00  6 !#Ló<i!7!ú=iq
F0 07 A1 37 7C E8 49 00 A1 18 7C 2A 06 3B 7C 40 38  i7!0I i^!*:i08
    
```

Premere un tasto funzione o introdurre carattere o byte esadecimale: _

Figura 20-1. Una schermata con il cursore fantasma.

spostino contemporaneamente, dal momento che la finestra esadecimale e quella ASCII rappresentano la stessa cosa.

Come è possibile spostare i cursori fantasma? Ogni tasto per il movimento del cursore invia un numero speciale: 72 per il cursore verso l'alto, 80 verso il basso, 75 verso sinistra, e 77 verso destra. Questi sono i numeri che dobbiamo aggiungere a DISPATCH_TABLE, con l'indirizzo delle quattro procedure per spostare i cursori fantasma.

Per spostare il cursore, bisogna cancellarlo, cambiare le due coordinate, e riscriverlo nuovamente. Se siete stati attenti a come fare per creare il cursore fantasma, non sarà difficile scrivere le quattro procedure per il relativo movimento.

Quando digitate un carattere sulla tastiera, Dskpatch dovrebbe leggere questo carattere e rimpiazzare il byte sotto al cursore fantasma con il carattere appena letto. Ecco i passaggi da seguire:

- Leggere un carattere dalla tastiera.
- Cambiare il numero esadecimale nella relativa finestra, e il carattere nella finestra ASCII, in modo che combacino con quelli appena letti dalla tastiera.
- Cambiare il byte nel buffer di settore, SECTOR.

Ecco anche un semplice suggerimento: non dovete apportare tanti cambiamenti per aggiungere l'editing. Dispatch richiede qualcosa in più che richiamare una nuova procedura (chiamata EDIT_BYTE) che svolga la maggior parte del lavoro. EDIT_BYTE è responsabile per i cambiamenti sia dello schermo che del SECTOR.

AGGIUNTE E CAMBIAMENTI A DSKPATCH

Dal Capitolo 23 al Capitolo 27, i cambiamenti diventeranno più complessi. Se siete sempre interessati a scrivere la vostra versione considerate questo fatto: cosa vorreste che faccia Dispatch in più rispetto a quello che fa adesso? Nei capitoli successivi sono state utilizzate queste idee.

Si creerà una nuova versione di READ_BYTE che legge sia un carattere che un numero esadecimale a due cifre, e aspetta la pressione del tasto Invio prima di fornire il carattere a Dispatch. Questa parte delle "cose da fare" non è così semplice come sembra, e saranno utilizzati due capitoli (Capitoli 23 e 24) per lavorare su questo problema.

Nel Capitolo 25, andrete alla ricerca di eventuali errori, mentre nel Capitolo 26 imparerete come riscrivere su disco, grazie alla funzione DOS INT 26h, un settore modificato; la funzione INT 26h è analoga a INT 25h che è stata usata per leggere un settore da disco. (Nel Capitolo 26 non saranno controllati gli errori di lettura, ma troverete questi controlli nella versione su disco che è disponibile con questo libro). Alla fine, nel Capitolo 27, saranno apportati alcuni cambiamenti a Dskpatch in modo da poter visualizzare anche l'altra metà del settore. Questi cambiamenti non permetteranno di scorrere liberamente all'interno del settore; ancora una volta, questo tipo di modifiche, saranno disponibili nel programma su disco.

I CURSORI FANTASMA

In questo capitolo saranno create le procedure per scrivere e cancellare un cursore fantasma nella finestra esadecimale, ed un altro nella finestra ASCII. Un cursore fantasma è chiamato in questo modo perché non è il cursore standard del PC; è un cursore inverso, ovvero un carattere nero su sfondo bianco. Nella finestra esadecimale c'è spazio a sufficienza per creare il cursore largo quattro caratteri, mentre nella finestra ASCII il cursore sarà largo un solo carattere, perché non c'è spazio tra i caratteri.

Ci sono parecchie procedure da scrivere ora; queste saranno quindi descritte solo brevemente.

I CURSORI FANTASMA

INIT_SEC_DISP è la sola procedura in grado di cambiare la visualizzazione del settore. Una nuova schermata appare quando avviamo Dskpatch e ogni volta che leggiamo un nuovo settore. Siccome il cursore fantasma sarà nell'area destinata alla visualizzazione del settore, inizierete il lavoro scrivendo una chiamata a WRITE_PHANTOM in INIT_SEC_DISP. In questo modo, il cursore fantasma sarà scritto ogni volta che verrà visualizzato un nuovo settore.

Ecco la versione revisionata (e finale) di INIT_SEC_DISP in DISP_SEC.ASM:

Listato 21-1. *Cambiamenti a INIT_SEC_DISP in DISP_SEC.ASM*

```

PUBLIC INIT_SEC_DISP
EXTRN WRITE_PATTERN:PROC, SEND_CRLF:PROC
EXTRN GOTO_XY:PROC, WRITE_PHANTOM:PROC

.DATA
EXTRN LINES_BEFORE_SECTOR:BYTE
EXTRN SECTOR_OFFSET:WORD

.CODE

;-----;
; Questa procedura inizializza la visualizzazione di mezzo settore. ;
; ;
; Usa: WRITE_PATTERN, SEND_CRLF, DISP_HALF_SECTOR ;
; WRITE_TOP_HEX_NUMBERS, GOTO_XY, WRITE_PHANTOM ;
; Legge: TOP_LINE_PATTERN, BOTTOM_LINE_PATTERN ;
; LINES_BEFORE_SECTOR ;

```

```

; Scrive:      SECTOR_OFFSET      ;
;-----;
INIT_SEC_DISP  PROC
    PUSH  DX
    XOR   DL,DL                ;Sposta il cursore all'inizio
    MOV   DH,LINES_BEFORE_SECTOR
    CALL  GOTO_XY
    CALL  WRITE_TOP_HEX_NUMBERS
    LEA  DX,TOP_LINE_PATTERN
    CALL  WRITE_PATTERN
    CALL  SEND_CRLF
    XOR   DX,DX                ;Comincia all'inizio del settore
    MOV   SECTOR_OFFSET,DX     ;Imposta l'offset del settore a 0
    CALL  DISP_HALF_SECTOR
    LEA  DX,BOTTOM_LINE_PATTERN
    CALL  WRITE_PATTERN
    CALL  WRITE_PHANTOM        ;Scrive il cursore fantasma
    POP  DX
    RET
INIT_SEC_DISP  ENDP

```

Notate che è anche stato aggiornato INIT_SEC_DISP per utilizzare e inizializzare le variabili. SECTOR_OFFSET è stato impostato a zero per visualizzare la prima metà del settore.

Ora concentrate il lavoro su WRITE_PHANTOM. Dovete scrivere sei procedure, e l'idea è molto semplice. Come prima cosa bisogna spostare il vero cursore nella posizione del cursore fantasma nella finestra esadecimale e cambiare l'attributo dei quattro caratteri successivi (attributo 70h). Questo crea un blocco bianco, largo quattro caratteri, con i numeri esadecimali in nero. Poi si farà lo stesso con la finestra ASCII, ma per un singolo carattere. Alla fine il vero cursore sarà riportato alla sua posizione originale. Tutte le procedure per i cursori fantasma saranno scritte in PHANTOM.ASM, eccetto WRITE_ATTRIBUTE_N_TIMES, la procedura che imposta gli attributi dei caratteri.

Inserite le seguenti procedure nel file PHANTOM.ASM

Listato 21-2. Il nuovo file PHANTOM.ASM

```

.MODEL  SMALL

.DATA

REAL_CURSOR_X  DB  0
REAL_CURSOR_Y  DB  0
    PUBLIC  PHANTOM_CURSOR_X, PHANTOM_CURSOR_Y
PHANTOM_CURSOR_X  DB  0
PHANTOM_CURSOR_Y  DB  0

.CODE

```

```

PUBLIC MOV_TO_HEX_POSITION
EXTRN GOTO_XY:PROC

.DATA
    EXTRN LINES_BEFORE_SECTOR:BYTE

.CODE
;-----;
; Questa procedura sposta il cursore reale nella posizione del cursore
; fantasma nella finestra esadecimale.
;
; Usa:          GOTO_XY
; Legge:       LINES_BEFORE_SECTOR, PHANTOM_CURSOR_X, PHANTOM_CURSOR_Y
;-----;
MOV_TO_HEX_POSITION    PROC
    PUSH    AX
    PUSH    CX
    PUSH    DX
    MOV     DH,LINES_BEFORE_SECTOR    ;Trova la riga del cursore fantasma (0,0)
    ADD     DH,2                      ;Più la riga di hex e barra orizzontale
    ADD     DH,PHANTOM_CURSOR_Y      ;DH = riga del cursore fantasma
    MOV     DL,8                      ;Rientro a sinistra
    MOV     CL,3                      ;Ogni colonna usa 3 caratteri, quindi
    MOV     AL,PHANTOM_CURSOR_X      ;dobbiamo moltiplicare CURSOR_X per 3
    MUL     CL
    ADD     DL,AL                    ;E aggiungerlo al rientro per avere la colonna
    CALL   GOTO_XY                  ; del cursore fantasma
    POP     DX
    POP     CX
    POP     AX
    RET
MOV_TO_HEX_POSITION    ENDP

PUBLIC MOV_TO_ASCII_POSITION
EXTRN GOTO_XY:PROC

.DATA
    EXTRN LINES_BEFORE_SECTOR:BYTE

.CODE
;-----;
; Questa procedura sposta il cursore reale all'inizio del cursore
; fantasma nella finestra ASCII.
;
; Usa:          GOTO_XY
; Legge:       LINES_BEFORE_SECTOR, PHANTOM_CURSOR_X, PHANTOM_CURSOR_Y
;-----;
MOV_TO_ASCII_POSITION  PROC
    PUSH    AX
    PUSH    DX
    MOV     DH,LINES_BEFORE_SECTOR    ;Trova la riga del cursore fantasma (0,0)
    ADD     DH,2                      ;Più la riga di hex e barra orizzontale
    ADD     DH,PHANTOM_CURSOR_Y      ;DH = riga del cursore fantasma
    MOV     DL,59                    ;Rientro a sinistra
    ADD     DL,PHANTOM_CURSOR_X      ;Aggiunge CURSOR_X per ottenere posizione
    CALL   GOTO_XY                  ; X per cursore fantasma

```

```

        POP    DX
        POP    AX
        RET
MOV_TO_ASCII_POSITION      ENDP

        PUBLIC SAVE_REAL_CURSOR
;-----;
; Questa procedura salva la posizione del cursore reale nelle due ;
; variabili REAL_CURSOR_X e REAL_CURSOR_Y. ;
; ;
; Scrive:      REAL_CURSOR_X, REAL_CURSOR_Y ;
;-----;
SAVE_REAL_CURSOR PROC
        PUSH  AX
        PUSH  BX
        PUSH  CX
        PUSH  DX
        MOV   AH,3                ;Legge la posizione del cursore
        XOR   BH,BH                ;a pagina 0
        INT  10h                  ;E la riporta in DL,DH
        MOV   REAL_CURSOR_Y,DL    ;Salva la posizione
        MOV   REAL_CURSOR_X,DH
        POP   DX
        POP   CX
        POP   BX
        POP   AX
        RET
SAVE_REAL_CURSOR ENDP

        PUBLIC RESTORE_REAL_CURSOR
        EXTRN GOTO_XY:PROC
;-----;
; Questa procedura riporta il cursore reale nella vecchia posizione, ;
; salvata in REAL_CURSOR_X e REAL_CURSOR_Y. ;
; ;
; Usa:        GOTO_XY ;
; Legge:      REAL_CURSOR_X, REAL_CURSOR_Y ;
;-----;
RESTORE_REAL_CURSOR      PROC
        PUSH  DX
        MOV   DL,REAL_CURSOR_Y
        MOV   DH,REAL_CURSOR_X
        CALL  GOTO_XY
        POP   DX
        RET
RESTORE_REAL_CURSOR      ENDP

        PUBLIC WRITE_PHANTOM
        EXTRN WRITE_ATTRIBUTE_N_TIMES:PROC

```

```

-----;
; Questa procedura usa CURSOR_X e CURSOR_Y, tramite MOV_TO_..., come      ;
; coordinate per il cursore fantasma. WRITE_PHANTOM scrive il cursore    ;
; fantasma.                                                                ;
;                                                                           ;
; Usa:        WRITE_ATTRIBUTE_N_TIMES, SAVE_REAL_CURSOR                    ;
;             RESTORE_REAL_CURSOR, MOV_TO_HEX_POSITION                    ;
;             MOV_TO_ASCII_POSITION                                       ;
-----;
WRITE_PHANTOM  PROC
    PUSH  CX
    PUSH  DX
    CALL  SAVE_REAL_CURSOR
    CALL  MOV_TO_HEX_POSITION      ;Coordinate del cursore nella finestra hex
    MOV   CX,4                     ;Rende il cursore fantasma largo 4 caratteri
    MOV   DL,70h
    CALL  WRITE_ATTRIBUTE_N_TIMES
    CALL  MOV_TO_ASCII_POSITION    ;Coordinate del cursore nella finestra
ASCII
    MOV   CX,1                     ;Qui il cursore è largo un carattere
    CALL  WRITE_ATTRIBUTE_N_TIMES
    CALL  RESTORE_REAL_CURSOR
    POP   DX
    POP   CX
    RET
WRITE_PHANTOM  ENDP

    PUBLIC  ERASE_PHANTOM
    EXTRN  WRITE_ATTRIBUTE_N_TIMES:PROC

-----;
; Questa procedura cancella il cursore fantasma; funziona in modo      ;
; contrario a WRITE_PHANTOM                                             ;
;                                                                           ;
; Usa:        WRITE_ATTRIBUTE_N_TIMES, SAVE_REAL_CURSOR                    ;
;             RESTORE_REAL_CURSOR, MOV_TO_HEX_POSITION                    ;
;             MOV_TO_ASCII_POSITION                                       ;
-----;
ERASE_PHANTOM  PROC
    PUSH  CX
    PUSH  DX
    CALL  SAVE_REAL_CURSOR
    CALL  MOV_TO_HEX_POSITION      ;Coordinate cursore nella finestra hex
    MOV   CX,4                     ;Riporta a bianco su nero
    MOV   DL,7
    CALL  WRITE_ATTRIBUTE_N_TIMES
    CALL  MOV_TO_ASCII_POSITION
    MOV   CX,1
    CALL  WRITE_ATTRIBUTE_N_TIMES
    CALL  RESTORE_REAL_CURSOR
    POP   DX

```

```

POP      CX
RET
ERASE_PHANTOM  ENDP

END
    
```

WRITE_PHANTOM e ERASE_PHANTOM sono praticamente lo stesso. Infatti la sola differenza è nell'attributo utilizzato: WRITE_PHANTOM imposta l'attributo a 70h per il video inverso, mentre ERASE_PHANTOM riporta l'attributo a normale (7).

Entrambe queste procedure salvano la posizione del cursore vero con SAVE_REAL_CURSOR, che utilizza la funzione numero 3 di INT 10h per leggere la posizione del cursore e per salvarla nei due byte REAL_CURSOR_X e REAL_CURSOR_Y. Dopo aver salvato la vera posizione del cursore, sia WRITE_PHANTOM che ERASE_PHANTOM chiamano MOV_TO_HEX_POSITION, che sposta il cursore all'inizio del cursore fantasma nella finestra esadecimale. Quindi WRITE_ATTRIBUTE_N_TIMES scrive i quattro caratteri in video inverso, partendo dalla posizione del cursore e spostandosi a destra. Questa scrive il cursore fantasma nella finestra esadecimale. Praticamente nello stesso modo, WRITE_PHANTOM scrive il cursore fantasma nella finestra ASCII. Alla fine, RESTORE_REAL_CURSOR riporta il cursore reale alla posizione originale prima della chiamata a WRITE_PHANTOM.

La sola procedura che non è stata scritta è WRITE_ATTRIBUTE_N_TIMES; fatelo ora.

Disco A Settore 0

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	0123456789ABCDEF	
00	EB	34	90	49	42	4D	20	20	33	2E	33	00	02	04	01	00	04ÉIBM 3.3
10	02	00	02	EF	A9	F8	2B	00	11	00	08	00	11	00	00	00	nr°+ < . <
20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	12	
30	00	00	00	00	01	00	FA	33	C0	8E	D0	BC	00	7C	16	07	.3LxL!
40	BB	78	00	36	C5	37	1E	56	16	53	BF	2B	7C	B9	0B	00	x 6+7AU S1+i
50	FC	AC	26	00	3D	00	74	03	26	8A	05	AA	8A	C4	E2	F1	4&C= t àè -è-f±
60	06	1F	89	47	02	C7	07	2B	7C	FB	CD	13	72	67	A0	10	vèG +i = rgá)
70	7C	98	F7	26	16	7C	03	06	1C	7C	03	06	0E	7C	A3	3F	iù=& ! -! iú?
80	7C	A3	37	7C	B8	20	00	F7	26	11	7C	0B	1E	0B	7C	03	iú7!q =&<i!A !
90	C3	48	F7	F3	01	06	37	7C	BB	00	05	A1	3F	7C	E8	9F	H=< 7!q i?i&f
A0	00	B8	01	02	E8	B3	00	72	19	0B	FB	B9	0B	00	BE	D9	q rvi!f d
B0	7D	F3	A6	75	0D	0D	7F	20	BE	E4	7D	B9	0B	00	F3	A6	}<u i& dΣ} <è
C0	74	18	BE	77	7D	E8	6A	00	32	E4	CD	16	5E	1F	0F	04	t^jw)èj 2Σ= ^vÁ
D0	0F	44	02	CD	19	BE	C4	7D	EB	EB	A1	1C	05	33	D2	F7	ÁD =j-}òòil 3p=
E0	36	0B	7C	FE	C0	A2	3C	7C	A1	37	7C	A3	3D	7C	BB	00	6 !#L<i!7!ú=iq
F0	07	A1	37	7C	EB	49	00	A1	18	7C	2A	06	3B	7C	40	38	i7!èi i^!* ;!08

Premere un tasto funzione o introdurre carattere o byte esadecimale: _

Figura 21-1. La schermata con il cursore fantasma.

CAMBIARE GLI ATTRIBUTI DEL CARATTERE

Dobbiamo utilizzare `WRITE_ATTRIBUTE_N_TIMES` per fare tre cose. Come prima cosa bisogna leggere il carattere sotto alla posizione del cursore. Bisogna fare ciò perché la funzione numero 9 di INT 10h scrive il carattere e l'attributo nella posizione del cursore. Quindi `WRITE_ATTRIBUTE_N_TIMES` cambierà l'attributo scrivendo il carattere con il nuovo attributo. Alla fine la procedura sposterà il cursore a destra nella posizione del carattere successivo, in modo da poter ripetere l'intero processo N volte. Potete vedere i dettagli nella procedura stessa; mettete `WRITE_ATTRIBUTE_N_TIMES` nel file `VIDEO_IO.ASM`:

Listato 21-3. Aggiungete questa procedura a `VIDEO_IO.ASM`

```

PUBLIC WRITE_ATTRIBUTE_N_TIMES
EXTRN CURSOR_RIGHT:PROC
;-----;
; Questa procedura imposta l'attributo per N caratteri, iniziando dalla ;
; posizione corrente del cursore ;
; ;
; Inserimento: CX Numero di caratteri per cui impostare attributo ;
; DL Nuovo attributo per i caratteri ;
; ;
; Usa: CURSOR_RIGHT ;
;-----;
WRITE_ATTRIBUTE_N_TIMES PROC
    PUSH AX
    PUSH BX
    PUSH CX
    PUSH DX
    MOV BL,DL ;Imposta il nuovo attributo
    XOR BH,BH ;Imposta pagina visualizzazione a 0
    MOV DX,CX ;CX è utilizzato dalle routine del BIOS
    MOV CX,1 ;Imposta attributo per un carattere
ATTR_LOOP:
    MOV AH,8 ;Legge carattere nella posizione del cursore
    INT 10h
    MOV AH,9 ;Scrive attributo/carattere
    INT 10h
    CALL CURSOR_RIGHT
    DEC DX ;Imposta l'attributo per N caratteri?
    JNZ ATTR_LOOP ;No, continua
    POP DX
    POP CX
    POP BX
    POP AX
    RET
WRITE_ATTRIBUTE_N_TIMES ENDP

```

Questa è la prima (e definitiva) versione di `WRITE_ATTRIBUTE_N_TIMES`. Con essa è stata creata anche la versione finale di `VIDEO_IO.ASM`.

SOMMARIO

Ora avete otto file da collegare, sempre con la procedura principale in Dskpatch. Di questi, sono stati cambiati due file, Disp_sec e Video_io, e ne è stato creato uno, Phantom. Se state utilizzando il Make o il file batch consigliato, ricordate di aggiungere il nuovo file, Phantom, alla lista.

Quando eseguite Dskpatch, lo vedrete visualizzare il settore come in precedenza, ma con i due caratteri fantasma. (Vedere Figura 21-1). Notate che il cursore reale è nella posizione in cui si dovrebbe trovare.

Nel prossimo capitolo aggiungerete delle procedure per spostare il cursore fantasma, e aggiungerete anche una semplice procedura di editing per permettere all'utente di cambiare il byte sotto al cursore fantasma.

UN SEMPLICE EDITING

Avete finalmente raggiunto il punto in cui è possibile iniziare a modificare i settori visualizzati. Aggiungerete presto delle procedure molto semplici per cambiare i byte visualizzati, ma prima di farlo, dovete spostare i cursori fantasma sulla parte interessata. Questo è relativamente semplice, ora che avete a disposizione le due procedure ERASE_PHANTOM e WRITE_PHANTOM.

SPOSTARE I CURSORI FANTASMA

Spostare i cursori fantasma in una direzione qualsiasi si basa su tre passaggi fondamentali: cancellare il cursore fantasma nella posizione corrente, cambiare la posizione del cursore modificando una delle variabili, PHANTOM_CURSOR_X o PHANTOM_CURSOR_Y, e utilizzando WRITE_PHANTOM per scrivere il cursore fantasma nella nuova posizione. Durante queste operazioni, bisogna fare attenzione a non spostare il cursore fuori dalla finestra, che è larga e alta 16 byte.

Per spostare il cursore fantasma, servono quattro nuove procedure, una per ogni tasto di direzione presente sulla tastiera. DISPATCHER non deve essere modificato, perché tutte le informazioni sulle procedure e sui codici estesi sono nella tabella DISPATCH_TABLE. Dobbiamo solo aggiungere i codici ASCII estesi e gli indirizzi delle procedure per ogni tasto di direzione. Ecco le modifiche da apportare a DISPATCH.ASM:

Listato 22-1. Cambiamenti a DISPATCH.ASM

```
.MODEL    SMALL
.CODE
    EXTRN NEXT_SECTOR:PROC                ;In DISK_IO.ASM
    EXTRN PREVIOUS_SECTOR:PROC           ;In DISK_IO.ASM
    EXTRN PHANTOM_UP:PROC, PHANTOM_DOWN:PROC ;In PHANTOM.ASM
    EXTRN PHANTOM_LEFT:PROC, PHANTOM_RIGHT:PROC

.DATA
;-----;
; Questa tabella contiene i tasti estesi ASCII ammessi e gli indirizzi ;
; delle procedure che devono essere richiamate alla pressione di ogni ;
; tasto. ;
; Il formato della tabella è ;
; DB 72 ;Codice esteso per il cursore verso l'alto ;
; DW OFFSET _TEXT:PHANTOM_UP ;
;-----;
```

```

DISPATCH_TABLE LABEL          BYTE
                DB      61                      ;F3
                DW      OFFSET _TEXT:PREVIOUS_SECTOR
                DB      62                      ;F4
                DW      OFFSET _TEXT:NEXT_SECTOR
                DB      72                      ;Cursore verso l'alto
                DW      OFFSET _TEXT:PHANTOM_UP
                DB      80                      ;Cursore verso il basso
                DW      OFFSET _TEXT:PHANTOM_DOWN
                DB      75                      ;Cursore a sinistra
                DW      OFFSET _TEXT:PHANTOM_LEFT
                DB      77                      ;Cursore a destra
                DW      OFFSET _TEXT:PHANTOM_RIGHT
                DB      0                        ;Fine della tabella

```

Come è possibile vedere, è semplice aggiungere dei comandi a Dskpatch: si mettono i nomi delle procedure in DISPATCH_TABLE e si scrivono le procedure.

Le procedure PHANTOM_UP, PHANTOM_DOWN e così via, sono abbastanza semplici. E' già stato detto come funzionano; potete quindi provare a scriverle da soli, nel file PHANTOM.ASM, prima di leggerle.

Ecco una versione delle procedure per spostare i cursori fantasma:

Listato 22-2. Aggiungete queste procedure a PHANTOM.ASM

```

;-----;
; Queste quattro procedure spostano i cursori fantasma. ;
; ;
; Usa:          ERASE_PHANTOM, WRITE_PHANTOM ;
; Legge:       PHANTOM_CURSOR_X, PHANTOM_CURSOR_Y ;
; Scrive:      PHANTOM_CURSOR_X, PHANTOM_CURSOR_Y ;
;-----;

                PUBLIC  PHANTOM_UP
PHANTOM_UP     PROC
                CALL   ERASE_PHANTOM          ;Cancella alla posizione corrente
                DEC    PHANTOM_CURSOR_Y      ;Sposta il cursore verso l'alto di una riga
                JNS    WASNT_AT_TOP          ;Non era al limite superiore, scrive cursore
                MOV    PHANTOM_CURSOR_Y,0    ;Era al limite superiore, riscriverlo lì
WASNT_AT_TOP:
                CALL   WRITE_PHANTOM         ;Scrive cursore fantasma nella nuova posizione
                RET
PHANTOM_UP     ENDP

                PUBLIC  PHANTOM_DOWN
PHANTOM_DOWN   PROC
                CALL   ERASE_PHANTOM         ;Cancella alla posizione corrente
                INC    PHANTOM_CURSOR_Y      ;Sposta il cursore verso il basso di una riga
                CMP    PHANTOM_CURSOR_Y,16   ;E' al limite inferiore?
                JB     WASNT_AT_BOTTOM       ;No, scrive quindi il cursore fantasma
                MOV    PHANTOM_CURSOR_Y,15   ;Si, riscriverlo a quella posizione

```

```
WASNT_AT_BOTTOM:
    CALL WRITE_PHANTOM      ;Scrivo il cursore fantasma
    RET
PHANTOM_DOWN    ENDP

    PUBLIC PHANTOM_LEFT
PHANTOM_LEFT    PROC
    CALL ERASE_PHANTOM      ;Cancella alla posizione corrente
    DEC PHANTOM_CURSOR_X    ;Sposta il cursore a sinistra di 1 posizione
    JNS WASNT_AT_LEFT      ;Non era al limite sinistro, scrivo cursore
    MOV PHANTOM_CURSOR_X,0 ;Era al limite sinistro, riscriverlo lì
WASNT_AT_LEFT:
    CALL WRITE_PHANTOM      ;Scrivo cursore fantasma
    RET
PHANTOM_LEFT    ENDP

    PUBLIC PHANTOM_RIGHT
PHANTOM_RIGHT    PROC
    CALL ERASE_PHANTOM      ;Cancella alla posizione corrente
    INC PHANTOM_CURSOR_X    ;Sposta il cursore a destra di una posizione
    CMP PHANTOM_CURSOR_X,16 ;Era al limite destro?
    JB WASNT_AT_RIGHT
    MOV PHANTOM_CURSOR_X,15 ;Era al limite destro, riscriverlo lì
WASNT_AT_RIGHT:
    CALL WRITE_PHANTOM      ;Scrivo cursore fantasma
    RET
PHANTOM_RIGHT    ENDP
```

PHANTOM_LEFT e PHANTOM_RIGHT sono in versione finale, ma dovrete cambiare PHANTOM_UP e PHANTOM_DOWN quando inizierete a far scorrere la schermata. Provate Dskpatch ora per vedere se potete spostare i cursori fantasma sullo schermo. Dovrebbero muoversi insieme e dovrebbero stare nella loro rispettiva finestra. A questo punto, è possibile vedere solo metà settore. Nel Capitolo 27, farete delle modifiche e delle aggiunte a Dskpatch in modo da poter far scorrere il settore per vedere anche la parte restante. A questo punto, cambierete sia PHANTOM_UP che PHANTOM_DOWN per far scorrere lo schermo quando si prova a spostare il cursore oltre il limite della finestra. Per esempio, quando il cursore è alla fine della prima metà del settore, premendo il tasto di direzione verso il basso, la schermata dovrà scendere di una riga, in modo da visualizzare i 16 byte successivi. Nel Capitolo 26, svilupperete la sezione di editing e di input da tastiera di Dskpatch. Ora aggiungete la procedura di editing in modo da poter cambiare il contenuto dei settori.

UN SEMPLICE EDITING

Avete già una semplice procedura per l'editing da tastiera, `READ_BYTE`, che legge un solo carattere dalla tastiera senza aspettare la pressione del tasto Invio. Utilizzerete questa vecchia versione di `READ_BYTE` per sviluppare l'editing. Quindi, nel prossimo capitolo, scriverete una versione più sofisticata della procedura che aspetta la pressione del tasto Invio o di un tasto speciale, come un tasto funzione o un tasto per lo spostamento del cursore.

La procedura sarà chiamata `EDIT_BYTE`, e cambierà un byte sia sullo schermo che in memoria (`SECTOR`). `EDIT_BYTE` porterà il carattere nel registro `DL`, lo scriverà nella locazione di memoria in `SECTOR` che è attualmente puntata dal cursore fantasma, e cambierà quindi la visualizzazione.

`DISPATCHER` è già predisposto per l'inserimento della `CALL` a `EDIT_BYTE`. Ecco la nuova versione di `DISPATCHER` in `DISPATCH.ASM`, con la `CALL` a `EDIT_BYTE`:

Listato 22-3. Cambiamenti a `DISPATCHER` in `DISPATCH.ASM`

```

PUBLIC DISPATCHER
EXTRN READ_BYTE:PROC, EDIT_BYTE:PROC
;-----;
; Questa è la routine di smistamento centrale. Durante le normali ;
; operazioni di editing e di visualizzazione questa procedura legge i ;
; caratteri dalla tastiera e, se il carattere è un tasto di comando ;
; (come ad esempio un tasto cursore), DISPATCHER richiama le procedure ;
; che effettuano il lavoro relativo. Lo smistamento è effettuato per ;
; tutti i tasti speciali elencati nella tabella DISPATCH_TABLE, dove gli ;
; indirizzi delle procedure sono memorizzati subito dopo i nomi dei ;
; tasti. ;
; Se il carattere non è un tasto speciale, dovrà essere introdotto ;
; direttamente nel buffer di settore - modalità di editing). ;
; ;
; Usa: READ_BYTE, EDIT_BYTE ;
;-----;
DISPATCHER PROC
    PUSH AX
    PUSH BX
    PUSH DX
DISPATCH_LOOP:
    CALL READ_BYTE ;Legge il carattere in AL
    OR AH,AH ;AX=-1 nessun carattere letto, 1 per codice esteso.
    JS DISPATCH_LOOP ;Nessun carattere letto, riprovare
    JNZ SPECIAL_KEY ;Legge codice esteso
    MOV DL,AL
    CALL EDIT_BYTE ;Carattere normale, modifica byte
    JMP DISPATCH_LOOP ;Legge un altro carattere
SPECIAL_KEY:
    CMP AL,68 ;F10-uscita?
    JE END_DISPATCH ;Sì, esci
;Usa BX per consultare la tabella

```

```

        LEA    BX,DISPATCH_TABLE
SPECIAL_LOOP:
    CMP    BYTE PTR [BX],0    ;Fine della tabella?
    JE     NOT_IN_TABLE      ;Si, il tasto non è presente in tabella
    CMP    AL,[BX]           ;Corrisponde a questo elemento della tabella?
    JE     DISPATCH         ;Si, allora smista
    ADD    BX,3              ;No, prova il prossimo elemento
    JMP    SPECIAL_LOOP      ;Controlla il successivo elemento della tabella

DISPATCH:
    INC    BX                ;Punta all' indirizzo della procedura
    CALL   WORD PTR [BX]     ;Richiama procedura
    JMP    DISPATCH_LOOP     ;Attende un altro tasto

NOT_IN_TABLE:
                                ;Non produce nulla, legge il carattere successivo
    JMP    DISPATCH_LOOP

END_DISPATCH:
    POP    DX
    POP    BX
    POP    AX
    RET

DISPATCHER    ENDP

```

La procedura EDIT_BYTE fa parecchio lavoro, anche richiamando altre procedure; come già detto in precedenza, questa è una caratteristica del design modulare. Con questo tipo di progettazione è spesso possibile creare delle procedure molto complesse, fornendo semplicemente una lista di CALL ad altre procedure che facciano il lavoro. Molte procedure presenti in EDIT_BYTE lavorano con il carattere nel registro DL, ma questo è già impostato quando si richiama EDIT_BYTE; l'unica istruzione diversa da CALL (o PUSH, POP), quindi, è l'istruzione LEA che imposta l'indirizzo del prompt per WRITE_PROMPT_LINE. La maggior parte delle procedure chiamate in EDIT_BYTE servono per l'aggiornamento del video quando si modifica un byte.

Siccome EDIT_BYTE cambia il byte sullo schermo, serve un'altra procedura, WRITE_TO_MEMORY, per cambiare il byte in SECTOR. WRITE_TO_MEMORY utilizza le coordinate presenti in PHANTOM_CURSOR_X e PHANTOM_CURSOR_Y per calcolare l'offset in SECTOR del cursore fantasma, quindi scrive il byte (carattere) nel registro DL con il byte corretto presente in SECTOR.

Ecco il nuovo file, EDITOR.ASM, che contiene le versioni finali di EDIT_BYTE e WRITE_TO_MEMORY.

Listato 22-4. Il Nuovo File EDITOR.ASM

```

.MODEL    SMALL

.CODE

.DATA

        EXTRN  SECTOR:BYTE
        EXTRN  SECTOR_OFFSET:WORD
        EXTRN  PHANTOM_CURSOR_X:BYTE
        EXTRN  PHANTOM_CURSOR_Y:BYTE

.CODE

;-----;
; Questa procedura scrive un byte su SECTOR, alla locazione di memoria ;
; puntata dal cursore fantasma. ;
; ;
;      DL      Byte da scrivere su SECTOR ;
; ;
; L'offset è calcolato da ;
;  OFFSET = SECTOR_OFFSET + (16 * PHANTOM_CURSOR_Y) + PHANTOM_CURSOR_X ;
; ;
; Legge:      PHANTOM_CURSOR_X, PHANTOM_CURSOR_Y, SECTOR_OFFSET ;
; Scrive:     SECTOR ;
;-----;
WRITE_TO_MEMORY PROC
        PUSH  AX
        PUSH  BX
        PUSH  CX
        MOV   BX,SECTOR_OFFSET
        MOV   AL,PHANTOM_CURSOR_Y
        XOR   AH,AH
        MOV   CL,4 ;Moltiplica PHANTOM_CURSOR_Y per 16
        SHL  AX,CL
        ADD  BX,AX ;BX = SECTOR_OFFSET + (16 * Y)
        MOV  AL,PHANTOM_CURSOR_X
        XOR  AH,AH
        ADD  BX,AX ;Questo è l'indirizzo!
        MOV  SECTOR[BX],DL ;Ora, memorizza il byte
        POP  CX
        POP  BX
        POP  AX
        RET
WRITE_TO_MEMORY ENDP

        PUBLIC  EDIT_BYTE
        EXTRN  SAVE_REAL_CURSOR:PROC, RESTORE_REAL_CURSOR:PROC
        EXTRN  MOV_TO_HEX_POSITION:PROC, MOV_TO_ASCII_POSITION:PROC
        EXTRN  WRITE_PHANTOM:PROC, WRITE_PROMPT_LINE:PROC
        EXTRN  CURSOR_RIGHT:PROC, WRITE_HEX:PROC, WRITE_CHAR:PROC

.DATA

        EXTRN  EDITOR_PROMPT:BYTE

.CODE

```

```

;-----;
; Questa procedura modifica un byte in memoria e sullo schermo. ;
; ;
; Inserimento: DL Byte da scrivere su SECTOR, e modificare sullo schermo ;
; ;
; Usa:          SAVE_REAL_CURSOR, RESTORE_REAL_CURSOR ;
;              MOV_TO_HEX_POSITION, MOV_TO_ASCII_POSITION ;
;              WRITE_PHANTOM, WRITE_PROMPT_LINE, CURSOR_RIGHT ;
;              WRITE_HEX, WRITE_CHAR, WRITE_TO_MEMORY ;
; Legge:       EDITOR_PROMPT ;
;-----;
EDIT_BYTE PROC
    PUSH    DX
    CALL    SAVE_REAL_CURSOR
    CALL    MOV_TO_HEX_POSITION      ;Porta sul numero esadecimale nella
    CALL    CURSOR_RIGHT            ;finestra esadecimale
    CALL    WRITE_HEX               ;Scrive il nuovo numero
    CALL    MOV_TO_ASCII_POSITION   ;Si porta sul carattere nella finestra ASCII
    CALL    WRITE_CHAR              ;Scrive il nuovo carattere
    CALL    RESTORE_REAL_CURSOR     ;Riporta il cursore alla posizione iniziale
    CALL    WRITE_PHANTOM          ;Riscrive il cursore fantasma
    CALL    WRITE_TO_MEMORY        ;Salva questo nuovo byte in SECTOR
    LEA    DX,EDITOR_PROMPT
    CALL    WRITE_PROMPT_LINE
    POP    DX
    RET
EDIT_BYTE ENDP

END

```

SOMMARIO

Dskpatch ora è costituito da nove file: Dskpatch, Dispatch, Disp_sec, Disk_io, Video_io, Kbd_io, Phantom, Cursor, e Editor. In questo capitolo, avete modificato Dispatch e aggiunto il nuovo file Editor. I file non sono lunghi per cui non serve molto tempo per assemblarli. In ogni caso è possibile apportare delle modifiche abbastanza agilmente cambiando i file in questione, riassemblendoli e collegandoli ancora in un unico file.

Nella versione corrente di Dskpatch, quando premete un tasto qualsiasi, vedete cambiare i caratteri sotto ai cursori fantasma. L'editing funziona, ma non è ancora sicuro, dal momento che è possibile cambiare un byte premendo un tasto qualsiasi. Dovete creare un certo tipo di salvaguardia, come la pressione del tasto Invio per confermare il cambiamento del byte.

Oltre tutto, la versione corrente di READ_BYTE non permette di inserire un numero esadecimale per cambiare un byte. Nel Capitolo 24, riscriverete READ_BYTE, in modo da renderla più completa e più sicura. Come prima cosa, tuttavia, bisogna scrivere una procedura per l'input esadecimale. Nel prossimo capitolo scriverete due procedure: una per l'input decimale, e l'altra per quello esadecimale.

INPUT ESADECIMALE E DECIMALE

In questo capitolo incontrerete due nuove procedure per l'inserimento da tastiera: una procedura per leggere un byte leggendo sia un numero esadecimale a due cifre sia un singolo carattere, ed un'altra per leggere una parola leggendo i caratteri come numeri decimali. Queste saranno le vostre procedure decimale ed esadecimale. Entrambe le procedure sono un po' difficili, quindi si utilizzerà un programma di verifica prima di collegarle a Dskpatch. Lavorerete con READ_BYTE, e la procedura di verifica sarà particolarmente importante, dal momento che essa perderà (momentaneamente) l'abilità di leggere i tasti speciali. Siccome Dskpatch fa uso di questi tasti, non sarete in grado di utilizzare READ_BYTE con Dskpatch. Vi renderete anche conto che non sarà possibile leggere i caratteri speciali con la procedura READ_BYTE sviluppata in questo capitolo; nel prossimo saranno apportate le modifiche necessarie per risolvere questo problema.

INPUT ESADECIMALE

Iniziate riscrivendo READ_BYTE. Nell'ultimo capitolo, READ_BYTE leggeva sia un carattere normale sia un tasto speciale e riportava un byte a Dispatch. Dispatch chiamava quindi l'Editor se READ_BYTE leggeva un carattere normale, e EDIT_BYTE modificava il byte puntato dal cursore fantasma. In altre parole, Dispatch guarda i tasti speciali in DISPATCH_TABLE per vedere se il byte è nella tabella; se è così, Dispatch chiama la procedura della tabella.

Ma, come detto nell'ultimo capitolo, con la vecchia versione di READ_BYTE è molto facile cambiare un byte accidentalmente. Se, non intenzionalmente, toccate un tasto della tastiera (diverso dai tasti speciali), EDIT_BYTE cambierà il byte sotto al cursore fantasma.

Cambierete READ_BYTE in modo che il carattere digitato non sia modificato fino a quando non premete Invio. Si potrà fare ciò grazie alla funzione 0Ah di INT 21h, per la lettura di una stringa di caratteri. Il DOS ritorna questa stringa solo dopo la pressione del tasto Invio ma, purtroppo, perde il controllo sui tasti speciali, come vedrete in seguito.

Per vedere esattamente come i cambiamenti influiscono su READ_BYTE, dovete scrivere un programma di verifica per controllare il funzionamento di READ_BYTE in isolamento. In questo modo, se qualcosa non dovesse funzionare, sapremo che è READ_BYTE e non altre parti di Dskpatch. La scrittura di questa procedura di verifica

sarà semplificata se utilizzerete alcune procedure di Kbd_io, Video_io, e Cursor per stampare le informazioni di READ_BYTE. Dovrete utilizzare procedure tipo WRITE_HEX e WRITE_DECIMAL per stampare il codice del carattere ritornato e il numero di caratteri letti. I dettagli sono indicati di seguito, in TEST.ASM:

Listato 23-1. Il programma TEST.ASM

```
.MODEL    SMALL

.STACK

.DATA
ENTER_PROMPT          DB          'Digitare i caratteri: ',0
CHARACTER_PROMPT DB          'Codice carattere',0
SPECIAL_CHAR_PROMPT  DB          'Carattere speciale letto: ',0

.CODE

EXTRN WRITE_HEX:PROC, WRITE_DECIMAL:PROC
EXTRN WRITE_STRING:PROC, SEND_CRLF:PROC
EXTRN READ_BYTE:PROC

TEST_READ_BYTE  PROC
    MOV     AX,DGROUP
    MOV     DS,AX

    LEA    DX,ENTER_PROMPT
    CALL   WRITE_STRING
    CALL   READ_BYTE
    CALL   SEND_CRLF
    LEA    DX,CHARACTER_PROMPT
    CALL   WRITE_STRING
    MOV    DL,AL
    CALL   WRITE_HEX
    CALL   SEND_CRLF
    LEA    DX,SPECIAL_CHAR_PROMPT
    CALL   WRITE_STRING
    MOV    DL,AH
    XOR    DH,DH
    CALL   WRITE_DECIMAL
    CALL   SEND_CRLF

    MOV    AH,4Ch          ;Ritorna al DOS
    INT    21H
TEST_READ_BYTE  ENDP

END    TEST_READ_BYTE
```

Provate a collegare questo programma a Kbd_io, Video_io, e Cursor (mettete Test come primo file del comando LINK). Se premete un tasto funzione speciale, Test visualizzerà lo scan code (codice di scansione), e un 1 per indicare che avete premuto

un tasto speciale. Altrimenti visualizzerà 0 (nessun tasto speciale).

Le istruzioni in TEST.ASM riguardano la formattazione: permettono cioè di ottenere uno schermo più ordinato.

Potreste aver notato che sono state utilizzate le procedure di kbd_io, video_io e cursor, senza badare agli altri file del progetto. E' stato possibile agire in questo modo dal momento che avete utilizzato solo delle procedure di uso generale in questi file. In altre parole, kbd_io, video_io, e cursor sono state create per poter essere utilizzate con qualsiasi programma. In generale, è una buona idea suddividere le procedure nei sorgenti in procedure specifiche e di uso generale, in modo da poterle utilizzare in altri programmi.

Ora bisogna riscrivere READ_BYTE per fare in modo che accetti una stringa di caratteri. Non solo vi salverà quando utilizzerete Dskpatch, ma vi permetterà anche di utilizzare il tasto Backspace per cancellare dei caratteri appena digitati (un'altra caratteristica utile dal momento che è facile compiere errori). READ_BYTE utilizzerà la procedura READ_STRING per leggere una stringa di caratteri.

READ_STRING è veramente semplice, ma conviene metterla in una procedura separata in modo da poterla riscrivere nel prossimo capitolo quando servirà per leggere i tasti funzione senza premere il tasto Invio. Per risparmiare tempo, aggiungerete altre tre procedure che utilizzeranno READ_BYTE: STRING_TO_UPPER, CONVERT_HEX_DIGIT, e HEX_TO_BYTE.

STRING_TO_UPPER e HEX_TO_BYTE lavorano sulle stringhe. STRING_TO_UPPER converte le lettere minuscole in lettere maiuscole. Questo significa che è possibile digitare f3 o F3 come numero esadecimale F3h. Permettendo che i numeri esadecimali possano essere digitati sia in maiuscolo che in minuscolo, avete aggiunto un aspetto amichevole al programma!

HEX_TO_BYTE prende la stringa letta dal DOS, dopo aver chiamato STRING_TO_UPPER, e converte la stringa esadecimale a due numeri in un numero di un singolo byte. HEX_TO_BYTE si serve di CONVERT_HEX_DIGIT per convertire ogni cifra esadecimale in un numero di quattro bit.

Come è possibile essere sicuri che il DOS non legga più di due cifre esadecimali? La funzione DOS 0Ah legge l'intera stringa di caratteri in un'area di memoria definita nel modo seguente:

```
CHAR_NUM_LIMIT    DB    0
NUM_CHARS_READ    DB    0
STRINGS           DB    80 DUP (0)
```

Il primo byte assicura che non vengano letti troppi caratteri. CHAR_NUM_LIMIT indica al DOS quanti caratteri, al massimo, leggere. Se lo impostate a tre, il DOS ne leggerà due più il ritorno a capo (che viene sempre contato). Qualsiasi carattere digitato successivamente sarà scartato, e per ogni carattere extra il DOS emetterà un segnale acustico per avvertirvi che avete superato il limite. Quando premete il tasto Invio, il DOS imposta il secondo byte, NUM_CHARS_READ, al numero di caratteri attualmente letti, senza includere il ritorno a capo.

STRING_TO_UPPER, READ_BYTE, e STRING_TO_UPPER utilizzano tutti NUM_CHARS_READ. Per esempio, READ_BYTE controlla NUM_CHARS_READ per

cercare se avete digitato un singolo carattere o un numero esadecimale a due cifre. Se NUM_CHARS_READ è stato impostato a uno, READ_BYTE ritorna un singolo carattere nel registro AL. Se NUM_CHARS_READ è stato impostato a due, READ_BYTE utilizza HEX_TO_BYTE per convertire la stringa esadecimale in un byte. Ecco quindi il nuovo file KBD_IO.ASM, con le quattro nuove procedure (notate che è stato tenuta la vecchia READ_BYTE, rinominandola READ_KEY, che sarà utilizzata nel prossimo capitolo):

Listato 23-2. La nuova versione di KBD_IO.ASM

```
.MODEL    SMALL

.DATA

KEYBOARD_INPUT LABEL    BYTE
CHAR_NUM_LIMIT DB      0           ;Lunghezza del buffer di input
NUM_CHARS_READ DB      0           ;Numero di caratteri letti
CHARS        DB          80 DUP (0) ;Buffer per input da tastiera

.CODE

        PUBLIC  STRING_TO_UPPER
;-----;
; Questa procedura converte i caratteri della stringa, usando il formato ;
; del DOS per le stringhe, in tutte lettere maiuscole. ;
; ;
; DS:DX Indirizzo del buffer di stringa ;
;-----;
STRING_TO_UPPER PROC
        PUSH  AX
        PUSH  BX
        PUSH  CX
        MOV   BX,DX
        INC   BX           ;Punta contatore di carattere
        MOV   CL,[BX]     ;Conteggio caratteri nel 2 ° byte del buffer
        XOR   CH,CH       ;Azzerà byte superiore del contatore
UPPER_LOOP:
        INCA  BX           ;Punta al carattere successivo nel buffer
        MOV   AL,[BX]
        CMP   AL,'a'       ;Controlla se si tratta di lettera minuscola
        JB   NOT_LOWER    ;No
        CMP   AL,'z'
        JA   NOT_LOWER
        ADD   AL,'A'-'a'   ;Converte in lettera maiuscola
        MOV   [BX],AL
NOT_LOWER:
        LOOP UPPER_LOOP
        POP   CX
        POP   BX
        POP   AX
        RET
```

STRING_TO_UPPER ENDP

```

;-----;
; Questa procedura converte un carattere da ASCII (esadecimale) a un ;
; nibble (4 bit). ;
; ;
; AL Carattere da convertire ;
; Riporta: AL nibble ;
; DF Impostato in caso di errore, altrimenti azzerato ;
;-----;

```

```

CONVERT_HEX_DIGIT PROC
    CMP AL, '0' ;E' una cifra ammessa?
    JB BAD_DIGIT ;No
    CMP AL, '9' ;Non è ancora sicuro
    JA TRY_HEX ;Potrebbe essere una cifra esadecimale
    SUB AL, '0' ;E' decimale, converti in nibble
    CLC ;Azzerare il riporto, nessun errore
    RET

```

```

TRY_HEX:
    CMP AL, 'A' ;Non è ancora sicuro
    JB BAD_DIGIT ;Non esadecimale
    CMP AL, 'F' ;Non è ancora sicuro
    JA BAD_DIGIT ;Non esadecimale
    SUB AL, 'A'-10 ;E' esadecimale, converti in nibble
    CLC ;Azzerare il riporto, nessun errore
    RET

```

```

BAD_DIGIT:
    STC ;Impostare il riporto, errore
    RET

```

CONVERT_HEX_DIGIT ENDP

PUBLIC HEX_TO_BYTE

```

;-----;
; Questa procedura converte i due caratteri in DS:DX da esadecimale a un ;
; byte. ;
; ;
; DS:DX Indirizzo dei due caratteri del numero esadecimale ;
; Riporto: AL Byte ;
; CF Impostato in caso di errore, altrimenti azzerato ;
;-----;

```

```

HEX_TO_BYTE PROC
    PUSH BX
    PUSH CX
    MOV BX,DX ;Invia indirizzo in BX per indirizzamento indiretto
    MOV AL,[BX] ;Preleva la prima cifra
    CALL CONVERT_HEX_DIGIT
    JC BAD_HEX ;Se il riporto è impostato, la cifra hex è errata
    MOV CX,4 ;Ora moltiplica per 16
    SHL AL,CL
    MOV AH,AL ;Ne tiene una copia
    INC BX ;Preleva la seconda cifra
    MOV AL,[BX]
    CALL CONVERT_HEX_DIGIT

```

```

        JC      BAD_HEX      ;Se il riporto è impostato, la cifra hex è errata
        OR      AL,AH        ;Unisce due nibble
        CLC          ;Azzera il riporto per assenza dell'errore
DONE_HEX:
        POP     CX
        POP     BX
        RET

BAD_HEX:
        STC          ;Imposta il riporto per presenza dell'errore
        JMP     DONE_HEX
HEX_TO_BYTE ENDP

;-----;
; Questa è una semplice versione di READ_STRING. ;
; ; ;
; DS:DX Indirizzo dell'area stringa ;
;-----;
READ_STRING PROC
        PUSH    AX
        MOV     AH,0Ah      ;Richiama input da tastiera bufferizzato
        INT     21h        ;Richiama funzione DOS per input bufferizzato
        POP     AX
        RET
READ_STRING ENDP

        PUBLIC  READ_BYTE

;-----;
; Questa procedura legge o un singolo carattere ASCII o un numero ;
; esadecimale a due cifre. Questa è solo una versione di prova di ;
; READ_BYTE. ;
; ; ;
; Ritorno AL Codice carattere (ad eccezione di AH=0) ;
; AH 0 se legge un carattere ASCII ;
; 1 se legge un tasto speciale ;
; -1 se non viene letto nessun carattere ;
; ; ;
; Usa: HEX_TO_BYTE, STRING_TO_UPPER, READ_STRING ;
; Legge: KEYBOARD_INPUT, etc. ;
; Scrive: KEYBOARD_INPUT, etc. ;
;-----;
READ_BYTE PROC
        PUSH    DX
        MOV     CHAR_NUM_LIMIT,3 ;Ammette solo due caratteri (più RETURN)
        LEA    DX,KEYBOARD_INPUT
        CALL   READ_STRING
        CMP    NUM_CHARS_READ,1 ;Vede quanti caratteri
        JE     ASCII_INPUT      ;Solo uno, lo tratta come carattere ASCII
        JB     NO_CHARACTERS    ;Premuto solo RETURN
        CALL   STRING_TO_UPPER  ;No, converte stringa in maiuscole
        LEA    DX,CHARS         ;Indirizzo della stringa da convertire
        CALL   HEX_TO_BYTE      ;Converte stringa da esadecimale a byte
        JC     NO_CHARACTERS    ;Errore, segnala quindi 'nessun carattere letto'
        XOR    AH,AH            ;Segnala lettura di un carattere

```

```

DONE_READ:
    POP    DX
    RET

NO_CHARACTERS:
    XOR    AH,AH            ;Imposta su 'nessun carattere letto'
    NOT    AH              ;Ritorna -1 in AH

ASCII_INPUT:
    MOV    AL,CHARS        ;Carica il carattere letto
    MOV    AH,1            ;Segnala la lettura di un carattere
    JMP    DONE_READ

READ_BYTE ENDP

        PUBLIC  READ_KEY

;-----;
; Questa procedura legge un carattere dalla tastiera. ;
; ; ;
; Ritorna il byte in  AL      Codice carattere (ad eccezione di AH=1) ;
; ; AH      0 se legge carattere ASCII ;
; ; ; 1 se legge un carattere speciale ;
;-----;

READ_KEY PROC
    XOR    AH,AH
    INT    16h            ;Legge il carattere/scan code dalla tastiera
    OR     AL,AL          ;E' un codice esteso?
    JZ     EXTENDED_CODE ;Sì

NOT_EXTENDED:
    XOR    AH,AH          ;Ritorna solo il codice ASCII

DONE_READING:
    RET

EXTENDED_CODE:
    MOV    AL,AH          ;Mette lo scan code in AL
    MOV    AH,1          ;Segnala codice esteso
    JMP    DONE_READING

READ_KEY ENDP

    END

```

Riassemblate Kbd_io e usate LINK sui quattro file Test, Kbd_io, Video_io, e Cursor per provare questa versione di READ_BYTE.

A questo punto ci sono due problemi con READ_BYTE. Ricordate i tasti funzione speciali? Non è possibile leggerli con la funzione DOS 0Ah; non funzionerà. Provate a premere un tasto funzione quando eseguite Test. Il DOS non ritornerà due byte, con il primo impostato a zero, come pensavate. Il programma Test riporta 255 per i tasti speciali (1 in AH), che significa che READ_BYTE non ha letto nessun carattere.

Non c'è nessun modo di leggere i codici estesi con l'input bufferizzato del DOS, utilizzando la funzione 0Ah. Abbiamo usato questa funzione in modo da poter utilizzare il tasto Backspace per cancellare i caratteri prima di premere Invio per confermare la scelta. Ma ora, siccome non è possibile leggere i tasti speciali, dovrete

scrivere la vostra procedura `READ_STRING`; bisognerà rimpiazzare la funzione `0Ah` per essere sicuri di poter premere un tasto funzione senza premere Invio.

L'altro problema con la funzione `DOS 0Ah` concerne il carattere di line-feed. Premete `Control-Invio` (line feed) dopo aver premuto un carattere e quindi provate il tasto `Backspace`. Vi accorgete che siete sulla riga successiva, senza nessuna possibilità di tornare sulla riga precedente. La nuova versione di `Kbd_io`, nel nuovo capitolo, tratterà il carattere di line feed (`Control-Invio`) come un carattere qualsiasi; quindi, premendo line feed il cursore non si sposterà sulla riga successiva.

Ma prima di procedere con la soluzione di questo problema per `READ_BYTE` e `READ_STRING`, bisogna scrivere una procedura per leggere un numero decimale senza segno. Non utilizzerete questa procedura in questo libro, ma la versione di `Dskpatch` sul disco la utilizza in modo che sia possibile, per esempio, chiedere al programma di visualizzare il settore numero 567.

INPUT DECIMALE

Ricordate che il numero decimale senza segno più grande che è possibile porre in una singola parola è 65536. Quando utilizzate `READ_STRING` per leggere una stringa di numeri decimali, bisogna indicare al `DOS` di non leggere più di sei caratteri (cinque numeri e il `Return` alla fine). Naturalmente questo significa che `READ_DECIMAL` sarà sempre in grado di leggere i numeri tra 65536 e 99999, anche se questi numeri non stanno in una parola. Bisogna tener d'occhio questi numeri e fornire un codice d'errore se `READ_DECIMAL` prova a leggere un numero maggiore di 65535, o se prova a leggere un numero non compreso tra zero e nove.

Per convertire le stringhe di cinque caratteri in una parola, utilizzerete la moltiplicazione come nel Capitolo 1: prendete la prima cifra, la moltiplicate per dieci, prendete la seconda cifra, moltiplicatela per dieci, e così via. Utilizzando questo metodo, è possibile, per esempio, scrivere 49856 come:

$$4*10^4 + 9*10^3 + 8*10^2 + 5*10^1 + 6*10^0$$

o, se preferite:

$$10*(10*(10*(10*4+9)+8)+5)+6$$

Naturalmente bisogna stare attenti a non commettere errori quando si fanno queste moltiplicazioni. Ma come si fa a sapere quando si sta leggendo un numero maggiore di 65535? Con i numeri molto grandi, l'ultima `MUL` causerà un overflow nel registro `DX`. Il flag `CF` è impostato quando `DX` non è zero dopo una parola `MUL`, quindi è possibile usare un'istruzione `JC` (*Jump if Carry set*) per manipolare un errore. Di seguito trovate `READ_DECIMAL`, che controlla anche ogni cifra (che sia tra 0 e 9). Mettete questa procedura nel file `KBD_IO.ASM`:

Listato 23-3. Aggiungete questa procedura in KBD_IO.ASM

```

PUBLIC READ_DECIMAL
;-----;
; Questa procedura preleva il buffer dell'output di READ_STRING e ;
; converte la stringa di cifre decimali in una parola. ;
; ;
; AX Parola convertita da decimale ;
; CF Impostato in caso di errore, altrimenti azzerato ;
; ;
; Usa: READ_STRING ;
; Legge: KEYBOARD_INPUT, etc. ;
; Scrive: KEYBOARD_INPUT, etc. ;
;-----;
READ_DECIMAL PROC
    PUSH BX
    PUSH CX
    PUSH DX
    MOV CHAR_NUM_LIMIT,6 ;Il numero massimo è di 5 cifre (65535)
    LEA DX,KEYBOARD_INPUT
    CALL READ_STRING
    MOV CL,NUM_CHARS_READ ;Rileva il numero di caratteri letti
    XOR CH,CH ;Imposta il byte superiore del contatore a 0
    CMP CL,0 ;Riporta errore se non è letto un carattere
    JLE BAD_DECIMAL_DIGIT ;Nessun carattere letto, segnala errore
    XOR AX,AX ;Inizia con il numero impostato a 0
    XOR BX,BX ;Comincia dall'inizio della stringa
CONVERT_DIGIT:
    MOV DX,10 ;Moltiplica il numero per 10
    MUL DX ;Moltiplica AX per 10
    JC BAD_DECIMAL_DIGIT ;CF impostato se MUL supera una parola
    MOV DL,CHARS[BX] ;Rileva la cifra successiva
    SUB DL,'0' ;E la converte in un semibyte (4 bit)
    JS BAD_DECIMAL_DIGIT ;Cifra errata se < 0
    CMP DL,9 ;E' una cifra errata?
    JA BAD_DECIMAL_DIGIT ;Si
    ADD AX,DX ;No, allora aggiungila al numero
    INC BX ;Punta al carattere successivo
    LOOP CONVERT_DIGIT ;Preleva la cifra successiva
DONE_DECIMAL:
    POP DX
    POP CX
    POP BX
    RET
BAD_DECIMAL_DIGIT:
    STC ;Imposta il riporto per segnalare l'errore
    JMP DONE_DECIMAL
READ_DECIMAL ENDP

```

Per essere sicuri che funzioni correttamente, bisogna verificare questa procedura con tutte le condizioni. Ecco un semplice programma di verifica per READ_DECIMAL che utilizza lo stesso approccio usato per verificare READ_BYTE:

Listato 23-4. Cambiamenti a TEST.ASM

```
.MODEL    SMALL

.STACK

.DATA
ENTER_PROMPT      DB      'Digitare i caratteri: ',0
NUMBER_READ_PROMPT DB      'Numero letto: ',0
CHARACTER_PROMPT  DB      'Codice carattere',0
SPECIAL_CHAR_PROMPT DB     'Carattere speciale letto: ',0

.CODE

        EXTRN WRITE_HEX:PROC, WRITE_DECIMAL:PROC
        EXTRN WRITE_STRING:PROC, SEND_CRLF:PROC
        EXTRN READ_DECIMAL:PROC

TEST_READ_DECIMAL      PROC
        MOV     AX,DGROUP
        MOV     DS,AX
        LEA    DX,ENTER_PROMPT
        CALL   WRITE_STRING
        CALL   READ_DECIMAL
        JC     ERROR
        CALL   SEND_CRLF
        LEA    DX,NUMBER_READ_PROMPT
        CALL   WRITE_STRING
        MOV     DX,AX
        CALL   WRITE_DECIMAL
ERROR:    CALL   SEND_CRLF
        LEA    DX,SPECIAL_CHAR_PROMPT
        CALL   WRITE_STRING
        MOV     DI,7AH
        XOR    DI,DI
        CALL   WRITE_DECIMAL
        CALL   SEND_CRLF

        MOV     AH,4Ch                ;Ritorna al DOS
        INT    21H
TEST_READ_DECIMAL      ENDP

        END    TEST_READ_DECIMAL
```

Ora dovete ancora collegare con il programma LINK i quattro file: Test (il file precedente), Kbd_io, Video_io, e Cursor. Provate le condizioni di limite, utilizzando

sia delle cifre valide che non valide (come A, che non è una cifra decimale valida), e con numeri tipo 0, 65535, e 65536.

SOMMARIO

Tornerete sulle due procedure in seguito, quando si discuteranno le tecniche per scrivere i programmi. Vedrete come sia possibile utilizzare una versione avanzata di TEST.ASM per scrivere un programma che converta i numeri tra esadecimale e decimale.

Ma ora, siete pronti per passare al capitolo successivo, dove scriverete una versione avanzata di READ_BYTE e READ_STRING.

MIGLIORAMENTO DELL'INPUT DI TASTIERA

In questo capitolo, scriverete una nuova versione di `READ_BYTE`, che inserirà un piccolo errore in `Dskpatch`. Nel prossimo capitolo, sarà spiegato il modo di sconfiggere questo piccolo baco; tuttavia potete provare a trovarlo da soli. (Un piccolo aiuto: controllate attentamente le condizioni per `READ_BYTE` quando lo inserite in `Dskpatch`).

UNA NUOVA `READ_STRING`

La nuova versione di `READ_STRING` è un esempio della progettazione modulare: procedure molto corte in modo che non siano difficili da capire. Questa dovrebbe essere riscritta con un numero maggiore di procedure: provate a farlo voi. In questo capitolo vi sarà fornita una nuova procedura `BACK_SPACE` per emulare la funzione del tasto Backspace (funzione `0Ah` del DOS). Quando si preme il tasto Backspace, `BACK_SPACE` cancellerà l'ultimo carattere digitato, sia dallo schermo che dalla stringa in memoria.

Sullo schermo, `BACK_SPACE` cancellerà il carattere spostando il cursore a sinistra di un carattere, scrivendo uno spazio sopra di esso, e quindi spostandosi a destra ancora di un carattere. Questa sequenza esegue lo stesso tipo di cancellazione del Backspace del DOS.

Nel buffer, `BACK_SPACE` cancellerà un carattere cambiando il puntatore del buffer, `DS:SI+BX`, in modo da farlo puntare al byte precedente in memoria. In altre parole, `BACK_SPACE` decrementa semplicemente `BX:(BX=BX-1)`. Il carattere sarà sempre nel buffer, ma il programma non lo vedrà. Perché no? `READ_STRING` fornisce il numero di caratteri letti. Se provate a leggerne di più, vedrete anche i caratteri cancellati.

Bisogna stare attenti a non cancellare nessun carattere quando il buffer è vuoto. Ricordate che l'area dei dati assomiglia a questa:

```
CHAR_NUM_LIMIT    DB    0
NUM_CHARS_READ    DB    0
STRING            DB    80 DUP (0)
```

Il buffer della stringa parte dal secondo byte dell'area di dati, o con uno *scarto* di 2 dall'inizio. Quindi, `BACK_SPACE` non cancellerà il carattere se `BX` è impostato a 2 (vale a dire l'inizio del buffer della stringa); il buffer è infatti vuoto quando `BX` è uguale a 2.

Ecco BACK_SPACE; inseritela in KBD_IO.ASM:

Listato 24-1. Aggiungete questa procedura a KBD_IO.ASM

```

PUBLIC  BACK_SPACE
EXTRN  WRITE_CHAR:PROC

;-----;
; Questa procedura cancella i caratteri, uno alla volta, dal buffer e      ;
; dallo schermo quando il buffer non è vuoto. BACK_SPACE ritorna          ;
; quando il buffer è vuoto.                                              ;
;                                                                           ;
; Inserimento:  DS:SI+BX  Il carattere più recente ancora nel buffer      ;
; Ritorna:      DS:SI+BX  Punta al carattere successivo più recente      ;
;                                                                           ;
; Usa:          WRITE_CHAR                                               ;
;-----;
BACK_SPACE  PROC                ;Cancella un carattere
            PUSH  AX
            PUSH  DX
            CMP   BX,2           ;Il buffer è vuoto?
            JE    END_BS        ;Sì, legge il carattere successivo
            DEC   BX            ;Cancella un carattere dal buffer
            MOV   AH,2          ;Cancella un carattere dallo schermo
            MOV   DL,BS
            INT   21h
            MOV   DL,20h        ;Scrive uno spazio in quella posizione
            CALL  WRITE_CHAR
            MOV   DL,BS         ;Ripetizione
            INT   21h
END_BS:    POP   DX
            POP   AX
            RET
BACK_SPACE  ENDP

```

Ed ora la nuova versione di READ_STRING. Sarà una procedura molto lunga, ma non complicata. E' molto lunga perché deve tener conto di parecchie condizioni. Sono state aggiunte alcune nuove funzioni. Se premete il tasto Escape, READ_STRING cancellerà il buffer e rimuoverà tutti i caratteri dallo schermo. Anche il DOS cancella tutti i caratteri dal buffer quando premete il tasto Escape, ma non cancella nessun carattere dallo schermo. Scrive semplicemente una barra rovesciata (\) alla fine della riga e si sposta sulla riga successiva. La vostra versione di READ_STRING sarà molto più versatile della funzione READ_STRING del DOS;

READ_STRING utilizza tre tasti speciali: Backspace, Escape, e Invio. Potete scrivere i codici ASCII per ognuno di questi tasti in READ_STRING ogni volta che vi servono, ma potete anche aggiungere qualche definizione all'inizio di KBD_IO.ASM per rendere READ_STRING più leggibile. Ecco le definizioni:

Listato 24-2. Aggiunte a KBD_IO.ASM

```
.MODEL    SMALL

BS        EQU    8           ;Backspace
CR        EQU    13        ;Invio (Carriage Return)
ESCAPE    EQU    27        ;Escape

.DATA
```

Ecco READ_STRING. Anche se è abbastanza lunga, potete vedere dal listato che non è per niente complicata. Rimpiazzate la vecchia versione di READ_STRING in KBD_IO.ASM con questa nuova versione:

Listato 24-3. La nuova READ_STRING in KBD_IO.ASM

```

PUBLIC  READ_STRING
EXTRN  WRITE_CHAR:NEAR

;-----;
; Questa procedura svolge una funzione molto simile alla funzione 0Ah ;
; del DOS. Questa funzione però riporterà un carattere speciale se ;
; viene premuto un tasto funzione o un tasto speciale (dopo questi tasti ;
; non premere INVIO). ESCAPE cancellerà l'input e permetterà di ;
; ricominciare. ;
; ;
; DS:DX  Indirizzo del buffer di tastiera. Il primo byte deve ;
;        contenere il numero massimo di caratteri da leggere ;
;        (più uno per INVIO). Il secondo byte verrà utilizzato ;
;        da questa procedura per riportare il numero di caratteri ;
;        letti effettivamente. ;
;        0      Nessun carattere letto ;
;        -1     Letto un carattere speciale ;
;        altrimenti il numero letto effettivamente ;
;        (escluso tasto INVIO) ;
; ;
; Usa:   BACK_SPACE, WRITE_CHAR, READ_KEY ;
;-----;
READ_STRING  PROC          PROC
            PUSH  AX
            PUSH  BX
            PUSH  SI
            MOV   SI,DX          ;Usa SI per il registro indice e
START_OVER:
            MOV   BX,2          ;BX per l'offset dall'inizio del buffer
            CALL  READ_KEY      ;Legge un carattere dalla tastiera
            OR    AH,AH         ;E' un carattere ASCII esteso?
            JZ    EXTENDED     ;Sì, legge carattere esteso
STRING_NOT_EXTENDED:          ;Il carattere esteso è errore se buffer è pieno

```



```

;-----;
; Il buffer era pieno, quindi non è possibile ;
; leggere un altro carattere. Invia un segnale ;
; acustico per avvisare l'utente della condizione ;
; di buffer pieno. ;
;-----;
BUFFER_FULL:
    JMP     SHORT SIGNAL_ERROR      ;Se il buffer è pieno, invia segnale acustico

;-----;
; Legge il codice ASCII esteso e lo introduce ;
; nel buffer come carattere individuale, quindi ;
; riporta -1 come numero di caratteri letti. ;
;-----;
EXTENDED:
    MOV     [SI+2],AL              ;Legge un codice ASCII esteso
    MOV     BL,OFFh                ;Introduce questo carattere nel buffer
    MOV     BL,OFFh                ;Numero dei caratteri speciali letti
    JMP     SHORT END_STRING

;-----;
; Salva il conteggio dei caratteri letti e ;
; rientra. ;
;-----;
END_INPUT:
    SUB     BL,2                   ;Input terminato
    ;Conteggio dei caratteri letti
END_STRING:
    MOV     [SI+1],BL              ;Riporta il numero di caratteri letti
    POP     SI
    POP     BX
    POP     AX
    RET
READ_STRING ENDP

```

Scorrendo la procedura, è possibile osservare che READ_STRING come prima cosa controlla se è stato premuto un tasto funzione speciale. E' possibile far questo solo quando la stringa è vuota. Per esempio, se avete premuto F3 dopo aver premuto il tasto *a*, READ_STRING ignorerà il tasto F3 ed emetterà un segnale acustico per avvisare che è stato premuto un tasto speciale in un momento sbagliato. E' comunque possibile premere Escape, quindi F3, dal momento che la pressione del tasto Escape fa in modo che READ_STRING cancelli il buffer di stringa.

Se READ_STRING legge un Carriage Return (CR), inserisce il numero di caratteri letti nel secondo byte dell'area di stringa e rientra. La nuova versione di READ_BYTE controllerà questo byte per vedere quanti caratteri sono stati letti da READ_STRING. A questo punto READ_STRING controlla per vedere se è stato premuto il tasto backspace. Se è così, chiama (CALL) BACK_SPACE per cancellare un carattere. Se il buffer diventa vuoto (BX diventa uguale a 2 - l'inizio del buffer di stringa), allora READ_STRING torna all'inizio, dove può leggere un tasto speciale. Altrimenti legge il carattere successivo.

Alla fine, READ_STRING controlla il carattere ESC. BACK_SPACE cancella i caratteri solo quando sono nel buffer, quindi è possibile cancellare il buffer di stringa chiamando la procedura BACK_SPACE per un numero di volte uguale a CHAR_NUM_LIMIT, dal momento che READ_STRING non è in grado di leggere un numero di caratteri maggiore di CHAR_NUM_LIMIT. Qualsiasi altro carattere viene salvato nel buffer di stringa e visualizzato sullo schermo con WRITE_CHAR, a meno che il buffer non sia pieno.

Nell'ultimo capitolo, cambierete READ_BYTE in modo che non sia in grado di leggere i tasti funzione speciali. Ecco i cambiamenti da fare a READ_BYTE in KBD_IO.ASM:

Listato 24-4. Cambiamenti a READ_BYTE in KBD_IO.ASM.

```

PUBLIC READ_BYTE
;-----;
; Questa procedura legge o un singolo carattere ASCII o un numero ;
; esadecimale a due cifre. Questa e' solo una versione di prova di ;
; READ_BYTE. ;
; ;
; Riporta byte in AL Codice carattere (ad eccezione di AH=0) ;
; AH 0 se legge un carattere ASCII o numero esa. ;
; 1 se legge un tasto speciale ;
; -1 se non legge nessun carattere ;
; ;
; Usa: HEX_TO_BYTE, STRING_TO_UPPER, READ_STRING ;
; Legge: KEYBOARD_INPUT, etc. ;
; Scrive: KEYBOARD_INPUT, etc. ;
;-----;
READ_BYTE PROC
PUSH DX
MOV CHAR_NUM_LIMIT,3 ;Ammette solo due caratteri (più INVIO)
LEA DX,KEYBOARD_INPUT
CALL READ_STRING
CMP NUM_CHARS_READ,1 ;Vede quanti caratteri
JE ASCII_INPUT ;Solo uno, lo tratta come carattere ASCII
JB NO_CHARACTERS ;Premuto solo INVIO
CMP BYTE PTR NUM_CHARS_READ,0FFh ;Tasto funzione speciale?
JE SPECIAL_KEY ;Sì
CALL STRING_TO_UPPER ;No, converte la stringa in maiuscole
LEA DX,CHARS ;Indirizzo della stringa da convertire
CALL HEX_TO_BYTE ;Converte la stringa da esadecimale a byte
JC NO_CHARACTERS ;Errore, segnala 'nessun carattere letto'
XOR AH,AH ;Segnala lettura di un carattere
DONE_READ:
POP DX
RET
NO_CHARACTERS:
XOR AH,AH ;Imposta su 'nessun carattere letto'
NOT AH ;Ritorna -1 in AH
JMP DONE_READ
ASCII_INPUT:
MOV AL,CHARS ;Carica il carattere letto

```

```
XOR    AH,AH
JMP    DONE_READ
SPECIAL_KEY:
MOV    AL,CHARS[0]        ;Riporta il codice di scansione
MOV    AH,1                ;Segnala tasto speciale con 1
JMP    DONE_READ
READ_BYTE ENDP
```

Dskpatch, con la nuova versione di READ_BYTE e READ_STRING, dovrebbe essere più facile da utilizzare. Ma c'è un piccolo errore, come è già stato detto. Per trovarlo, avviate Dskpatch e provate tutte le condizioni per READ_BYTE e HEX_TO_BYTE. (Ricordate che ci sono nove file che devono essere processati con LINK per costruire il file eseguibile: Dskpatch, Dispatch, Disp_sec, Disk_io, Video_io, Kbd_io, Phantom, Cursor, e Editor).

SEMPLICE PER L'UTENTE O SEMPLICE PER IL PROGRAMMATTORE

E' stata presa una decisione in READ_STRING che ha reso Dskpatch più facile da scrivere, ma non molto amichevole con l'utente. Avviate Dskpatch e provate a fare questo: digitate una lettera, come *f*, quindi premete un tasto per lo spostamento del cursore. Dskpatch emetterà un suono. Perché?

Come programmatori di Dskpatch dovrete sapere esattamente il perché: la procedura READ_STRING non vi restituisce il controllo fino a quando non viene premuto il tasto Escape o Invio. Ma l'utente non può saperlo; questo è il primo problema. Il secondo problema è che l'utente tende ad irritarsi quando il programma emette un segnale acustico senza una ragione apparente; dopo tutto l'utilizzo dei tasti per il movimento del cursore sono universalmente riconosciuti.

I programmi amichevoli, richiedono spesso uno sforzo di programmazione per renderli semplici e naturali. Ecco alcune considerazioni per scrivere dei programmi user-friendly:

- Evitate i segnali acustici a meno che non si tratti di un errore critico (come un errore di disco). Nella maggior parte dei casi non conviene emettere un segnale acustico se premete un tasto non permesso.
- Tenete presente cosa vuole l'utente, e non cos'è più facile da scrivere. Molto spesso le due cose coincidono, ma altre volte no e dovrete fare un grosso sforzo per sviluppare dei programmi 'amichevoli'.
- Provate a scrivere dei programmi modulari. Facendo ciò eliminerete molte condizioni d'errore come quella appena incontrata in READ_STRING.
- Esponete sempre le vostre idee a degli utenti, e non ad altri programmatori. Gli utenti non vogliono sapere come scriverete il programma, ma vogliono solo che il programma sia 'ovvio'. E se un utente ha dei problemi nell'eseguire i vostri programmi, cercate di capire il perché.

Tenete comunque presente che ci sono parecchi libri che trattano esclusivamente la progettazione dei programmi.

SOMMARIO

Avete scritto una nuova versione di `READ_STRING` che permette di leggere i caratteri speciali, oltre alle stringhe. E, ad eccezione del piccolo baco che troverete e risolverete nel prossimo capitolo, `READ_STRING` funziona come previsto.

Avete poi affrontato altri problemi riguardo `READ_STRING`. Primo di tutti, è una procedura troppo lunga e complicata. Bisogna riscriverla in modo che sia più modulare.

Alla fine avete visto che `READ_STRING` non è molto user-friendly dal momento che emette dei suoni quando l'utente tenta di premere i tasti per lo spostamento del cursore, dopo aver iniziato a digitare un numero esadecimale. Questo problema non sarà risolto in questo libro, ma potreste farlo da soli.

Ora passiamo al prossimo capitolo dove cercheremo l'errore che affligge `Dskpatch`.

ALLA RICERCA DEGLI ERRORI

Se provate la nuova versione di Dskpatch con *ag*, che non è un numero esadecimale valido, noterete che Dskpatch non fa nulla quando premete il tasto Invio. Siccome la stringa *ag* non è un numero esadecimale, non c'è nulla di strano nel fatto che venga ignorata, ma il programma dovrebbe, almeno, cancellarla dallo schermo.

Questo errore può essere trovato solo quando si controllano le condizioni del programma. Questo non è un problema di READ_BYTE, anche se è apparso quando avete riscritto la procedura. Al contrario, il problema è da ricercare nel modo in cui sono stati scritti DISPATCHER e EDIT_BYTE.

EDIT_BYTE richiama WRITE_PROMPT_LINE per riscrivere la linea di prompt e cancellare il resto della riga. Questo cancella ogni carattere digitato. Ma se si digita una stringa come *ag*, READ_BYTE riporta di aver letto una stringa di lunghezza zero, e DISPATCH non chiama EDIT_BYTE. Qual è la soluzione?

RISOLVERE I PROBLEMI DI DISPATCHER

In questo momento ci sono due modi per risolvere questo problema. La soluzione migliore sarebbe quella di riscrivere Dskpatch e DISPATCHER. Ma ora non sarà fatto questo. Ricordate: i programmi non sono mai completi, ma bisogna fermarsi in qualche momento. Saranno invece fissati i problemi a DISPATCHER in modo che venga riscritta la linea di prompt anche se READ_BYTE legge una stringa di zero caratteri.

Ecco le modifiche a DISPATCHER (in DISPATCH.ASM) per risolvere il problema:

Listato 25-1. Cambiamenti a DISPATCHER in DISPATCH.ASM

```

PUBLIC DISPATCHER
EXTRN READ_BYTE:PROC, EDIT_BYTE:PROC
EXTRN WRITE_PROMPT_LINE:PROC
.DATA
EXTRN EDITOR_PROMPT:BYTE
.CODE
;-----;
; Questa è la routine di smistamento principale. Durante le normali ;
; operazioni di editing e di visualizzazione questa procedura legge i ;
; caratteri dalla tastiera e, se il carattere è un tasto di comando ;

```

```

; (come ad esempio un tasto cursore), DISPATCHER richiama le procedure      ;
; che effettuano il lavoro effettivo. Lo smistamento è effettuato per      ;
; tutti i tasti elencati nella tabella DISPATCH_TABLE, dove gli            ;
; indirizzi delle procedure sono memorizzati subito dopo i nomi dei        ;
; tasti.                                                                     ;
; Se il carattere non è un tasto speciale, dovrà essere introdotto         ;
; direttamente nel buffer di settore (modalità di editing).                ;
;                                                                             ;
; Usa:          READ_BYTE, EDIT_BYTE, WRITE_PROMPT_LINE                    ;
; Legge:        EDITOR_PROMPT                                             ;
;-----;
DISPATCHER      PROC
    PUSH  AX
    PUSH  BX
    PUSH  DX
DISPATCH_LOOP:
    CALL  READ_BYTE          ;Legge carattere in AX
    OR    AH,AH             ;AX = -1 se nessun carattere letto, 1
                                ; per un codice esteso.
    JS    NO_CHARS_READ     ;Nessun carattere letto, riprova
    JNZ   SPECIAL_KEY      ;Letto un codice esteso
    MOV   DL,AL
    CALL  EDIT_BYTE         ;Era un carattere normale, modifica byte
    JMP   DISPATCH_LOOP    ;Legge un altro carattere
SPECIAL_KEY:
    CMP   AL,68             ;F10-uscita?
    JE    END_DISPATCH     ;Si, esci
                                ;Usa BX per consultare la tabella
    LEA  BX,DISPATCH_TABLE
SPECIAL_LOOP:
    CMP  BYTE PTR [BX],0   ;Fine tabella?
    JE   NOT_IN_TABLE     ;Si, tasto non presente in tabella
    CMP  AL,[BX]          ;Corrisponde a questo elemento di tabella?
    JE   DISPATCH        ;Si, allora smista
    ADD  BX,3             ;No, prova il prossimo elemento
    JMP  SPECIAL_LOOP     ;Controlla il successivo elemento nella tabella

DISPATCH:
    INC  BX               ;Punta un indirizzo di procedura
    CALL WORD PTR [BX]   ;Richiama la procedura
    JMP  DISPATCH_LOOP   ;Attende un altro tasto

NOT_IN_TABLE:
                                ;Non produce nulla, legge solo il carattere
                                ; successivo
    JMP  DISPATCH_LOOP

NO_CHARS_READ:
    LEA  DX,EDITOR_PROMPT
    CALL WRITE_PROMPT_LINE ;Cancella i caratteri non validi
    JMP  DISPATCH_LOOP    ;Riprova

```

```
END_DISPATCH:
    POP    DX
    POP    BX
    POP    AX
    RET
DISPATCHER    ENDP
```

Questa soluzione non crea nessun problema, ma rende DISPATCHER molto meno elegante. L'eleganza e la chiarezza vanno spesso mano nella mano, e la progettazione modulare deve essere animata da entrambe.

SOMMARIO

DISPATCHER è elegante perché è come una semplice soluzione di un problema. Invece di fare dei confronti sui vari caratteri digitati, avete creato una tabella in cui cercare. Facendo ciò avete reso DISPATCHER semplice e sicuro, rispetto ad un programma che contiene parecchie istruzioni per controllare decine e decine di condizioni. Risolvendo il piccolo problema, avete complicato DISPATCHER (non molto per la verità in questo caso, ma certi errori richiedono la creazione di procedure più complesse).

Se dovete modificare alcune delle vostre procedure per risolvere dei problemi, vi conviene riscrivere l'intera procedura. Risparmierete molte ore di lavoro!

Non si vuole, in questa sede, enfatizzare l'importanza del controllo di tutte le condizioni e della progettazione modulare. Entrambe le tecniche rendono i programmi più sicuri. Nel prossimo capitolo, troverete un altro metodo per collaudare i programmi.

SCRIVERE I SETTORI MODIFICATI

A questo punto avete un Dskpatch funzionante. In questo capitolo, creerete la procedura per scrivere sul disco un settore modificato, e nel prossimo capitolo, scriverete la procedura per visualizzare l'altra metà del settore.

SCRIVERE SUL DISCO

Scrivere sul disco un settore modificato può essere disastroso se non viene fatto intenzionalmente. Tutte le funzioni di Dskpatch vengono attivate dai tasti funzione F3, F4, e F10 e dai tasti cursore. Ma ognuno di questi tasti può essere premuto accidentalmente. Ma, fortunatamente, non è possibile premere accidentalmente una combinazione di tasti con lo shift, quindi utilizzerete Shift-F2 per scrivere un settore sul disco. Questo evita la possibilità di scrivere un settore accidentalmente. Apportate i seguenti cambiamenti a DISPATCH.ASM, per aggiungere WRITE_SECTOR alla tabella:

Listato 26-1. Cambiamenti a DISPATCH.ASM

```
.CODE
EXTRN NEXT_SECTOR:PROC                ;In DISK_IO.ASM
EXTRN PREVIOUS_SECTOR:PROC            ;In DISK_IO.ASM
EXTRN PHANTOM_UP:PROC, PHANTOM_DOWN:PROC ;In PHANTOM.ASM
EXTRN PHANTOM_LEFT:PROC, PHANTOM_RIGHT:PROC
EXTRN WRITE_SECTOR:PROC                ;In DISK_IO.ASM

.DATA
;-----;
; Questa tabella contiene i tasti estesi ASCII ammessi e gli indirizzi ;
; delle procedure che devono essere richiamati alla pressione di ogni ;
; tasto. ;
; Il formato della tabella è ;
; DB 72 ;Codice esteso per cursore verso l'alto ;
; DW OFFSET PHANTOM_UP ;
;-----;
DISPATCH_TABLE LABEL BYTE
DB 61 ;F3
DW OFFSET _TEXT:PREVIOUS_SECTOR
DB 62 ;F4
DW OFFSET _TEXT:NEXT_SECTOR
```

```

DB      72                                ;Cursore verso l'alto
DW      OFFSET _TEXT:PHANTOM_UP
DB      80                                ;Cursore verso il basso
DW      OFFSET _TEXT:PHANTOM_DOWN
DB      75                                ;Cursore a sinistra
DW      OFFSET _TEXT:PHANTOM_LEFT
DB      77                                ;Cursore a destra
DW      OFFSET _TEXT:PHANTOM_RIGHT
DB      85                                ;SHIFT-F2
DW      OFFSET _TEXT:WRITE_SECTOR
DB      0                                  ;Fine della tabella

```

WRITE_SECTOR è praticamente identica a READ_SECTOR. Il solo cambiamento, ovviamente, è che si vuole scrivere un settore e non leggerlo. Dal momento che INT 25h chiede al DOS di leggere un settore, la funzione gemella INT 26h, chiede al DOS di scrivere il settore sul disco. Ecco WRITE_SECTOR; mettetelo in DISK_IO.ASM:

Listato 26-2. Aggiungete questa procedura a DISK_IO.ASM

```

PUBLIC WRITE_SECTOR
;-----;
; Questa procedura riscrive sul disco il settore. ;
; ;
; Legge:      DISK_DRIVE_NO, CURRENT_SECTOR_NO, SECTOR ;
;-----;
WRITE_SECTOR PROC
    PUSH AX
    PUSH BX
    PUSH CX
    PUSH DX
    MOV AL,DISK_DRIVE_NO      ;Numero del drive
    MOV CX,1                  ;Scrive 1 settore
    MOV DX,CURRENT_SECTOR_NO ;Settore logico
    LEA BX,SECTOR
    INT 26h                   ;Scrive il settore su disco
    POPF                      ;Elimina il flag delle informazioni
    POP DX
    POP CX
    POP BX
    POP AX
    RET
WRITE_SECTOR ENDP

```

Ora riassemblete sia Dispatch che Disk_io, ma non provate la nuova funzione di Dskpatch. Prendete un dischetto che non contiene dati importanti che volete conservare e mettetelo nel drive A. Avviate Dskpatch dal drive C (o da un altro drive), in modo che Dskpatch legga il primo settore dal disco nel drive A. Prima di continuare assicuratevi che il disco nel drive A non vi serva più.

Cambiate un byte a caso e prendetene nota. Quindi, premete il tasto Shift-F2. Vedrete la luce rossa del drive accendersi: il settore modificato è stato scritto sul disco nel drive A.

Ora, premete F4 per leggere il settore successivo (setteore 1), quindi F3 per leggere il settore precedente (il settore 0, appena modificato). Dovreste vedere ancora il settore ma questa volta con il byte che avete cambiato.

ALTRE TECNICHE DI COLLAUDO

Come possiamo verificare la presenza di piccoli errori nel nostro programma? Dskpatch è un programma troppo esteso per poter utilizzare Debug per cercare eventuali errori. Dskpatch è composto da nove file differenti che devono essere processati con LINK per creare DSKPATCH.EXE. Come è possibile trovare le procedure in un programma così ampio senza tracciarlo lentamente? Come potrete vedere in questo capitolo, ci sono due possibilità per trovare le procedure: utilizzando una mappa che è possibile ottenere da LINK, o utilizzando un collaudatore a livello sorgente, come Microsoft CodeView o Borland Turbo Debugger.

Quando gli autori di questo libro hanno scritto Dskpatch, è successo qualcosa quando hanno aggiunto WRITE_SECTOR: premendo Shift-F2 la macchina si fermava. Non sembrava esserci nessun problema in WRITE_SECTOR e in DISPATCH_TABLE. Tutto sembrava corretto.

Alla fine il problema è stato trovato: era nella tabella DISPATCH_TABLE per WRITE_SECTOR. Per errore, nella tabella era stato digitato DW invece che DB, quindi WRITE_SECTOR veniva salvato in memoria in un byte più alto. Potete vedere l'errore evidenziato nella zona in grassetto:

```
DISPATCH_TABLE LABEL BYTE
                .
                .
                .
                DB 77 ;Cursore a destra
                DW OFFSET _TEXT:PHANTOM_RIGHT
                DW 85 ;SHIFT-F2
                DW OFFSET _TEXT:WRITE_SECTOR
                DB 0 ;Fine della tabella
DATA_SEG ENDS
```

Come esercizio, apportate questi cambiamenti al file DISPATCH.ASM, e seguite le istruzioni della sezione successiva.

COSTRUIRE UNA MAPPA

Utilizzando il LINK è possibile creare una mappa di Dskpatch. Questa mappa vi aiuterà a trovare le procedure e le variabili in memoria.

Il comando LINK utilizzato fino ad ora è molto lungo:

```
LINK DSKPATCH DISK_IO DISP_SEC VIDEO_IO CURSOR DISPATCH KBD_IO PHANTOM EDITOR;
```

e si allungherà ulteriormente. Fortunatamente il comando LINK permette di fornire un *file* che contiene tutte le informazioni. Con questo tipo di file, che chiamerete linkinfo, potrete digitare semplicemente:

```
LINK @LINKINFO
```

e LINK leggerà tutte le informazioni da questo file.

Con i nomi dei file elencati in precedenza, linkinfo sarà più o meno così:

```
DSKPATCH DISK_IO DISP_SEC VIDEO_IO CURSOR +
DISPATCH KBD_IO PHANTOM EDITOR
```

Il segno più (+) alla fine della prima riga, indica a LINK di continuare a leggere i nomi sulla riga successiva.

E' possibile aggiungere altre informazioni che indicano a LINK di creare una mappa delle procedure e delle variabili del programma. Ecco il file LINKINFO completo:

```
DSKPATCH DISK_IO DISP_SEC VIDEO_IO CURSOR +
DISPATCH KBD_IO PHANTOM EDITOR
DSKPATCH
DSKPATCH /MAP;
```

Le ultime due righe rappresentano i nuovi parametri. Il primo, *dskpatch*, indica a LINK che il nome del file eseguibile deve essere DSKPATCH.EXE; la seconda nuova riga indica a LINK di creare una lista dei file chiamata DSKPATCH.MAP (per creare la mappa). Il parametro */map* richiede a LINK di fornire una lista di tutte le procedure e le variabili che sono state dichiarate pubbliche.

Create il file mappa usando nuovamente LINK per creare Dskpatch. Il file mappa prodotto dal linker è lungo circa 130 linee. E' troppo lungo per poter essere riprodotto interamente, quindi vi saranno proposte solo le parti di maggior interesse. Ecco una parte del file mappa, DSKPATCH.MAP:

Start	Stop	Length	Name	Class
00000H	005C9H	005CAH	_TEXT	CODE
005CAH	006BBH	000F2H	_DATA	DATA
006BCH	026BBH	02000H	_BSS	BBS
026C0H	02ABFH	00400H	STACK	STACK

Origin Group
 005c:0 DGROUP

Address	Publics by Name
0000:03EA	BACK_SPACE
0000:027E	CLEAR_SCREEN
0000:02C0	CLEAR_TO_END_OF_LINE
0000:000C	CURRENT_SECTOR_NO
0000:02A0	CURSOR_RIGHT
0000:000E	DISK_DRIVE_NO
0000:02EC	DISPATCHER
0000:0131	DISP_HALF_SECTOR
.	
.	
.	
0000:01EB	WRITE_HEX_DIGIT
0000:025F	WRITE_PATTERN
0000:0546	WRITE_PHANTOM
0000:01A9	WRITE_PROMPT_LINE
0000:0086	WRITE_SECTOR
0000:01BC	WRITE_STRING
005C:00FC	_edata
005C:2100	_end

Address	Publics by Value
0000:0030	PREVIOUS_SECTOR
0000:0050	NEXT_SECTOR
0000:006C	READ_SECTOR
0000:0086	WRITE_SECTOR
0000:00A0	INIT_SEC_DISP
0000:00CC	WRITE_HEADER
0000:0131	DISP_HALF_SECTOR
.	
.	
.	
005C:000F	LINES_BEFORE_SECTOR
005C:0010	HEADER_LINE_NO
005C:0011	HEADER_PART_1
005C:0017	HEADER_PART_2
005C:0028	PROMPT_LINE_NO
005C:0029	EDITOR_PROMPT
005C:00FA	PHANTOM_CURSOR_X
005C:00FB	PHANTOM_CURSOR_Y
005C:00FC	_edata

```
005C:00FC    SECTOR
005C:2100    _end
```

Program entry point at 0000:0010

Ci sono tre parti principali per questa *mappa* (chiamata in questo modo perché indica dove sono caricate in memoria le procedure). La prima parte mostra una lista di segmenti del programma. Dskpatch ha parecchi segmenti: `_TEXT` (che contiene tutto il codice) e `_DATA`, `_BSS` e `STACK` che sono raggruppati insieme in `DGROUP`, e contengono tutti i dati. Per coloro che sono interessati a maggiori dettagli, `_DATA` contiene tutte le variabili di memoria definite nel segmento `.DATA?` (così come `SECTOR`), e `STACK` lo stack definito in `.STACK`.

Nota: Potreste vedere dei numeri differenti se le vostre procedure sono in un ordine diverso da quello del libro (potete controllare tale ordine nell'Appendice B).

La parte successiva della mappa, mostra le procedure pubbliche e le variabili, elencate in ordine alfabetico. `LINK` elenca solo le procedure e le variabili che sono state dichiarate `PUBLIC`. Se state collaudando un programma lungo, potreste dichiarare *tutte* le variabili e le procedure pubbliche, in modo da poterle ritrovare nella mappa.

La sezione finale della mappa, elenca tutte le procedure e le variabili di memoria, ma questa volta nell'ordine in cui appaiono in memoria.

Entrambe le liste includono gli indirizzi di memoria per ogni procedura e per ogni variabile dichiarata `PUBLIC`. Se controllate questa lista, troverete che la procedura `DISPATCHER` inizia all'indirizzo `2ECh`. Ora utilizzerete questo indirizzo, per trovare il baco in Dskpatch.

TRACCIARE GLI ERRORI

Se provate ad eseguire l'attuale versione di Dskpatch con il baco, troverete che tutto funziona, ad eccezione della combinazione di tasti `Shift-F2`, che causa un blocco della macchina.

Siccome tutto funziona, eccetto `Shift-F2`, il primo pensiero è che l'errore sia stato introdotto in `WRITE_SECTOR`. Per trovare questo baco, è possibile iniziare a collaudare Dskpatch attraverso `WRITE_SECTOR`, ma conviene iniziare con un altro approccio. Si sa che `DISPATCHER` funziona correttamente, perché i tasti `F3`, `F4`, `F10` e i tasti cursore funzionano senza problemi. Questo significa che `DISPATCHER` è un buon punto d'inizio per cercare l'errore in Dskpatch. In altre parole, iniziate la verifica dalla parte di codice che *sicuramente* funziona.

Se guardate le istruzioni di `DISPATCHER` (nel Capitolo 25), vedrete che l'istruzione:

```
CALL WORD PTR [BX]
```

è il cuore di DISPATCHER, dal momento che richiama tutte le altre routine. In particolare questa istruzione richiama WRITE_SECTOR quando si preme il tasto Shift-F2. Iniziate la vostra ricerca da qui.

Si deve utilizzare Debug per avviare Dskpatch con un'interruzione su questa istruzione. Naturalmente bisogna conoscere l'indirizzo di questa istruzione, e potete trovarlo disassemblando DISPATCHER, che inizia a 2ECh. Dopo un comando U 2EC, seguito da un altro comando U, dovrete vedere il comando CALL:

```

.
.
3E05:0313    EBF2          JMP    0307
3E05:0315    43           INC    BX
3E05:0316    FF17        CALL  [BX]
3E05:0318    EBD5        JMP    02EF
.
.
.
```

Ora che sapete che l'istruzione CALL è all'indirizzo 316h, potete mettere un'interruzione a quest'indirizzo, quindi eseguire WRITE_SECTOR passo a passo.

Come prima cosa, utilizzate il comando G 316 per eseguire Dskpatch fino a questa istruzione. Vedrete Dskpatch avviarsi, quindi aspettare un comando. Premete Shift-F2, dal momento che è questo il comando che causa il problema, e vedrete ciò che segue:

-G 316

```
AX=0155 BX=00A3 CX=06BC DX=0029 SP=03F8 BP=75F0 SI=0000 DI=0F8A
DS=3E61 ES=3DF5 SS=4071 CS=3E05 IP=0316 NV UP EI PL NZ NA PE NC
3E05:0316 FF17          CALL  [BX]                      DS:00A3=0086
```

A questo punto il registro BX punta ad una parola che dovrebbe contenere l'indirizzo di WRITE_SECTOR. Vediamo se è così:

-D A3 L 2

```
3E61:00A0      00 86
-
```

In altre parole, si sta provando a chiamare (CALL) la procedura in 8600h (ricordate che il byte basso è visualizzato per primo). Ma se guardate la mappa della memoria, vedete che WRITE_SECTOR dovrebbe essere a 86h; da questa mappa è possibile vedere che non ci sono procedure in 8600h. L'indirizzo è completamente sbagliato!

COLLAUDO A LIVELLO SORGENTE

Sia Microsoft che Borland hanno lavorato duramente per fornire degli strumenti di programmazione. Sia CodeView di Microsoft che Turbo Debugger di Borland sono dei collaudatori chiamati *Source-Level* (Livello Sorgente). In altre parole, dove Debug mostra solo l'indirizzo in CALL e JMP, questi due programmi mostrano il codice sorgente.

Le prossime due sezioni trattano questi due collaudatori. Potreste leggerne solo una delle due, dal momento che ci sono alcune ripetizioni.

MICROSOFT CODEVIEW

CodeView, il più vecchio dei due, è stato introdotto nel 1986, circa due anni prima del Turbo Debugger di Borland. E' incluso in ogni pacchetto Microsoft Macro Assembler e in altri prodotti della stessa Microsoft. Come potrete vedere in questa sezione, CodeView è così utile che potreste prendere in considerazione di aggiornare il vostro macro assembler, se non disponete dell'ultima versione.

CodeView condivide alcune caratteristiche con Debug, anche perché Microsoft ha scritto entrambi i programmi. Ma ci sono più differenze che somiglianze. Utilizzerete due nuove caratteristiche: source-level debugging e screen-swapping.

Source-level debugging (collaudo a livello sorgente) permette di vedere l'intero codice, completo di commenti, al posto delle sole istruzioni con i relativi indirizzi. Per esempio se utilizzate Debug per disassemblare la prima linea di Dskpatch, vedrete:

```
2C14:0100      E88C03          CALL 048F
```

Con CodeView vedrete invece (come nella figura 26-1):

```
CALL    CLEAR_SCREEN
```

Qual è la più facile da leggere?

La seconda nuova caratteristica, lo screen swapping, è utile per il collaudo di Dskpatch. Dskpatch sposta il cursore sullo schermo, scrivendo in punti differenti. Nell'ultima sezione, dove è stato utilizzato Debug, quest'ultimo ha iniziato a scrivere sullo stesso schermo in cui era presente Dskpatch.

CodeView, invece, mantiene due schermi separati: uno per Dskpatch e uno per se stesso. Quando è attivo Dskpatch vedrete la schermata di Dskpatch, mentre quando è attivo CodeView vedrete la schermata di CodeView. Vi potete fare un'idea dello screen swapping seguendo gli esempi seguenti.

Prima di poter utilizzare le caratteristiche di collaudo simbolico di CodeView, bisogna indicare all'assemblatore e al linker di salvare le informazioni di questo processo. Per l'assemblatore bisogna utilizzare il parametro /Zi, mentre per il linker il parametro /CO (Codeview).

Modificate tutte le linee del Makefile (e riassemblete ogni file) in modo che abbia il parametro /Zi prima di ogni punto e virgola, e modificate anche il file per il LINK:

Listato 26-3. Apportate questi cambiamenti a MAKEFILE

```
dskpatch.obj:    dskpatch.asm
                masm dskpatch /Zi;

disk_io.obj:     disk_io.asm
                masm disk_io /Zi;
                .
                .
                .

dskpatch.exe:   dskpatch.obj disk_io.obj disp_sec.obj video_io.obj cursor.obj \
                dispatch.obj kbd_io.obj phantom.obj editor.obj
                link @linkinfo
```

Cambiate il file LINKINFO in questo modo:

Listato 26-4. Cambiamenti a LINKINFO

```
dskpatch disk_io disp_sec video_io cursor +
dispatch kbd_io phantom editor
dskpatch
dskpatch /map /CO;
```

Cancellate tutto i file .OBJ (DEL *.OBJ) e ricostruite Dskpatch.exe. Ora siete pronti per avviare CodeView. Digitate:

```
C:\> CV DSKPATCH
```

e dovrete vedere una schermata come quella della figura 26-1. Notate che state esaminando il *file sorgente*. Questo perché CodeView è conosciuto come collaudatore a livello di sorgente.

Ora che siete in CodeView, potete analizzare la procedura DISPATCHER senza sapere dove si trova. Premete Alt-S (per aprire il menu Search), quindi L (Label) per cercare un'etichetta. Ora digitate *dispatcher* nel box di dialogo che appare e premete Invio per vedere il codice di DISPATCHER. Utilizzate quindi i tasti cursore per scorrere l'istruzione CALL WORD PTR [BX] nella seconda pagina.

Quando avete il cursore sulla linea dell'istruzione CALL WORD PTR [BX], premete F7 (che esegue il programma fino a quando non incontra CALL). Vedrete Dskpatch sullo schermo, e quando premete Shift-F2 tornerà CodeView. Premendo il tasto F4 tornerete allo schermo di Dskpatch. Con un tasto qualsiasi si può tornare a CodeView. Se guardate nella parte bassa della schermata in figura 26-2, vedrete le due linee seguenti:

```
DS:00A3
      8600
```

```

File View Search Run Watch Options Language Calls Help | F8=Trace F5=G
-----|-----|-----|-----|-----|-----|-----|-----|
dskpatch.ASM
45: .CODE
46:
47:     EXTRN  CLEAR_SCREEN:PROC, READ_SECTOR:PROC
48:     EXTRN  INIT_SEC_DISP:PROC, WRITE_HEADER:PROC
49:     EXTRN  WRITE_PROMPT_LINE:PROC, DISPATCHER:PROC
50: DISK_PATCH PROC
51:     MOV    AX,DGROUP      ;Mette il segmento da
52:     MOV    DS,AX          ;Imposta DS per punta
53:
54:     CALL   CLEAR_SCREEN
55:     CALL   WRITE_HEADER
56:     CALL   READ_SECTOR
57:     CALL   INIT_SEC_DISP
58:     LEA   DX,EDITOR_PROMPT
59:     CALL   WRITE_PROMPT_LINE
60:     JMP   DISPATCHER
61:
62:     MOV   AH,4Ch          ;Torna al DOS
-----|-----|-----|-----|-----|-----|-----|
Microsoft (R) CodeView (R) Version 2.2
(C) Copyright Microsoft Corp. 1986-1988. All rights reserved.

```

Figura 26-1. La vista iniziale di Dskpatch.exe in CodeView.

Questa parte del video è utilizzata per mostrare il valore in memoria puntato dall'istruzione corrente, che è l'istruzione CALL sotto al cursore in video inverso. In questo caso, è il valore alla locazione della memoria [BX]. Come potete chiaramente vedere, 8600 è l'esatto valore trovato utilizzando Debug. Ma qui il valore è stato trovato molto più velocemente.

Digitate Alt-F (per richiamare il menu File) e X (eXit) per uscire da CodeView. Potreste saltare la prossima sezione e passare direttamente al sommario, ma non dimenticate di cambiare l'istruzione DW in DB nel file Dispatch.asm.

Potreste cambiare anche il file linkinfo. Avevate aggiunto il parametro /CO in modo che LINK aggiungesse le informazioni di debug al file eseguibile. Ma queste informazioni di debug rendono il file un po' più grande. In qualsiasi caso, potreste voler rimuovere il parametro /CO prima di dare il programma ad altre persone.

BORLAND TURBO DEBUGGER

Il Turbo Debugger ha molte familiarità con il Debug. Come vedrete in questa sezione, Turbo Debugger utilizza lo stile multifinestra della Borland, al posto dell'interfaccia poco 'friendly' di Debug.

```

File View Search Run Watch Options Language Calls Help | F8=Trace F5=G
-----|-----
dispatch.ASM
80:      JE      DISPATCH      ;Si, quindi dispatch
81:      ADD     BX,3           ;No, allora prova anc
82:      JMP     SPECIAL_LOOP   ;Controlla la tabella
83:
84:  DISPATCH:
85:      INC     BX              ;Punta all'indirizzo
86:      CALL    WORD PTR [BX]   ;Chiama la procedura
87:      JMP     DISPATCH_LOOP   ;Aspetta un altro tas
88:
89:  NOT_IN_TABLE:
90:      JMP     DISPATCH_LOOP   ;Non fare nulla, legg
91:
92:  NO_CHARS_READ:
93:      LEA     DX,EDITOR_PROMPT
94:      CALL    WRITE_PROMPT_LINE ;Cancella qualsiasi c
95:      JMP     DISPATCH_LOOP   ;Prova ancora
96:
97:  END_DISPATCH:
-----|-----
Microsoft (R) CodeView (R) Version 2.2
(C) Copyright Microsoft Corp. 1986-1988. All rights reserved.

```

Figura 26-2. CodeView dopo il comando F7 (Go)

Utilizzerete due nuove caratteristiche: source-level debugging e screen-swapping. Source-level debugging (collaudo a livello sorgente) permette di vedere l'intero codice, completo di commenti, al posto delle sole istruzioni con i relativi indirizzi. Per esempio se utilizzate Debug per disassemblare la prima linea di Dskpatch, vedrete:

```
2C14:0100      E88C03      CALL 048F
```

Con Turbo Debugger vedrete invece (come nella figura 26-3):

```
CALL CLEAR_SCREEN
```

Qual è la più facile da leggere?

La seconda nuova caratteristica, lo screen swapping, è utile per il collaudo di Dskpatch. Dskpatch sposta il cursore sullo schermo, scrivendo in punti differenti. Nell'ultima sezione, dove è stato utilizzato Debug, quest'ultimo ha iniziato a scrivere sullo stesso schermo in cui era presente Dskpatch.

Turbo Debugger, invece, mantiene due schermi separati: uno per Dskpatch e uno per se stesso. Quando è attivo Dskpatch vedrete la schermata di Dskpatch, mentre quando è attivo Turbo Debugger vedrete la schermata di Turbo Debugger. Vi potete fare un'idea dello screen swapping seguendo gli esempi seguenti.

Prima di poter utilizzare le caratteristiche di collaudo simbolico di Turbo Debugger, bisogna indicare all'assemblatore e al linker di salvare le informazioni di questo

processo. Per l'assemblatore bisogna utilizzare il parametro /zi, mentre per il linker il parametro /v.

Modificate tutte le linee del Makefile (e riassemblete ogni file) in modo che abbia il parametro /zi prima di ogni punto e virgola, e modificate anche il file per il TLINK:

Listato 26-5. Apportate questi cambiamenti a MAKEFILE

```
dskpatch.exe:    dskpatch.obj disk_io.obj disp_sec.obj video_io.obj cursor.obj \
                 dispatch.obj kbd_io.obj phantom.obj editor.obj
tlink @linkinfo

dskpatch.obj:   dskpatch.asm
tasm dskpatch /zi;

disk_io.obj:    disk_io.asm
tasm disk_io /zi;
.
.
.
```

Cambiate il file LINKINFO in questo modo:

Listato 26-4. Cambiamenti a LINKINFO

```
dskpatch disk_io disp_sec video_io cursor +
dispatch kbd_io phantom editor
dskpatch
dskpatch /map /v;
```

Cancellate tutto i file .OBJ (DEL *.OBJ) e ricostruite Dskpatch.exe. Ora siete pronti per avviare Turbo Debugger. Digitate:

```
C:\>TD DSKPATCH.EXE
```

e dovrete vedere una schermata come quella della figura 26-1. Notate che state esaminando il *file sorgente*. Questo perché Turbo Debugger è conosciuto come collaudatore a livello di sorgente.

Ora che siete in Turbo Debugger, potete analizzare la procedura DISPATCHER senza sapere dove si trova. Premete Alt-V per aprire il menu View; premete quindi V per visualizzare la finestra delle variabili (Figura 26-5). A questo punto utilizzate i tasti cursore per andare su dispatcher, e premete Invio per visualizzare il codice di DISPATCHER.

Potete utilizzare i tasti per lo spostamento del cursore per andare all'istruzione CALL WORD PTR [BX].

Quando avete il cursore sulla linea dell'istruzione CALL WORD PTR [BX], premete F4 e poi Shift-F2. Vedrete che Dskpatch disegnerà il suo schermo e, una volta premuto Shift-F2, tornerete a Turbo Debugger.

```

File View Run Breakpoints Data Window Options READY
Module: dskpatch File: dskpatch.asm 51
EXTRN CLEAR_SCREEN:PROC, READ_SECTOR:PROC
EXTRN INIT_SEC_DISP:PROC, WRITE_HEADER:PROC
EXTRN WRITE_PROMPT_LINE:PROC, DISPATCHER:PROC
DISK_PATCH
PROC
▶ MOV AX,DGROUP ;Mette il segmento dati in AX
MOV DS,AX ;Imposta DS per puntare ai dati
CALL CLEAR_SCREEN
CALL WRITE_HEADER
CALL READ_SECTOR
CALL INIT_SEC_DISP
LEA DX,EDITOR_PROMPT
CALL WRITE_PROMPT_LINE
JMP DISPATCHER

MOV AH,4Ch ;Torna al DOS
INT 21h
DISK_PATCH ENDP
Watches

```

Figura 26-3. La vista iniziale di Dskpatch.exe in Turbo Debugger.

Variables			
disp_line	06F4F:0134	disk_patch	06F4F:0000
dispatcher	06F4F:02DC	sector_offset	0 (0h)
edit_byte	06F4F:0595	current_sector_no	0 (0h)
editor_prompt	"Premere un tasto fu	disk_drive_no	0 (00h)
erase_phantom	06F4F:0555	lines_before_sector	'0' 2 (02h)
goto_xy	06F4F:0285	header_line_no	' ' 0 (00h)
header_line_no	' ' 0 (00h)	header_part_1	"Disco"
header_part_1	"Disco"	header_part_2	"Settore"

Figura 26-4. La finestra delle variabili in Turbo Debugger permette di saltare direttamente ad una procedura.

Questa volta non vedrete lo schermo di Dskpatch ma quello di Turbo Debugger. Per tornare a Dskpatch, premete Alt-F5; con un tasto qualsiasi tornerete al Turbo Debugger.

A questo punto bisogna vedere il valore di [BX] in modo da sapere che procedura deve chiamare Dskpatch. Per questo è necessario aggiungere un *watch*, che permette di vedere un valore. Premete Ctrl-W per aprire un box di dialogo e digitate [BX]. Come potrete vedere nella finestra 'Watches', 8600 è esattamente il valore trovato con Debug. Ma in questo modo tale valore è stato trovato più velocemente.

```

File View Run Breakpoints Data Window Options READY
Module: dispatch File: dispatch.asm 86 1
    CMP     AL,68             ;F10--uscita?
    JE      END_DISPATCH    ;Si, esci

    LEA     BX,DISPATCH_TABLE

SPECIAL_LOOP:
    CMP     BYTE PTR [BX],0   ;Fine tabella?
    JE      NOT_IN_TABLE     ;Si, tasto non presente in tabella
    CMP     AL,[BX]          ;Corrisponde a questo elemento di tab
    JE      DISPATCH        ;Si, allora smista
    ADD     BX,3              ;No, prova prossimo elemento
    JMP     SPECIAL_LOOP     ;Controlla successivo elemento di tab

DISPATCH:
    INC     BX                ;Punta a indirizzo di procedura
    CALL   WORD PTR [BX]     ;Richiama procedura
    JMP    DISPATCH_LOOP    ;Attende altro tasto
NOT_IN_TABLE:
    ;Non produce nulla, legge solo caratt

```

```

Watches 2
[bx] word 38288 (7680h)

```

F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu

Figura 26-5. Turbo Debugger dopo l'esecuzione di Dskpatch fino all'istruzione CALL

Digitate Alt-X per uscire da Turbo Debugger. Non dimenticate di cambiare DW in DB nel file DISPATCH.ASM.

Potreste cambiare anche il file linkinfo. Avevate aggiunto il parametro /v in modo che TLINK aggiungesse le informazioni di debug al file eseguibile. Ma queste informazioni di debug rendono il file un po' più grande. In qualsiasi caso, potreste voler rimuovere il parametro /v prima di dare il programma ad altre persone.

SOMMARIO

Con questo capitolo sono terminate le discussioni sulle tecniche di collaudo. Nel prossimo capitolo, aggiungerete le procedure per far scorrere lo schermo nel mezzo settore successivo. Quindi, nella parte finale di questo libro, si discuteranno degli argomenti avanzati.

In ogni caso non dimenticate di mettere a posto l'errore messo in DISPATCH_TABLE.

L'ALTRO MEZZO SETTORE

Idealmente Dskpatch dovrebbe agire come un word processor nel momento in cui provate a spostare il cursore oltre il limite inferiore della finestra in cui è visualizzato il settore. La versione di Dskpatch disponibile sul disco che accompagna questo libro fa esattamente questo, ma questo grado di sofisticazione non lo raggiungerete qui. In questo capitolo sarà aggiunto lo scheletro di due nuove procedure, SCROLL_UP e SCROLL_DOWN, che fanno scorrere lo schermo. Nella versione di Dskpatch su disco, SCROLL_UP e SCROLL_DOWN sono in grado di far scorrere un numero di linee qualsiasi da una a sedici (servono sedici linee per visualizzare mezzo settore). Le versioni di SCROLL_UP e SCROLL_DOWN che sono illustrate in questo capitolo scorrono solo di mezzo settore.

SCORRERE DI MEZZO SETTORE

La vecchia versione di PHANTOM_UP e PHANTOM_DOWN riportava il cursore all'inizio del mezzo settore visualizzato quando si provava a spostare il cursore oltre l'inizio o la fine della finestra. Cambierete PHANTOM_UP e PHANTOM_DOWN in modo da poter chiamare SCROLL_UP o SCROLL_DOWN quando il cursore si sposta oltre l'inizio o la fine della finestra di visualizzazione. Queste due procedure spostano il cursore e lo portano nella nuova posizione.

Ecco le versioni modificate di PHANTOM_UP e PHANTOM_DOWN (in PHANTOM.ASM):

Listato 27-1. Cambiamenti in PHANTOM.ASM

```

PHANTOM_UP      PROC
CALL    ERASE_PHANTOM          ;Cancella alla posizione corrente
DEC     PHANTOM_CURSOR_Y      ;Sposta cursore verso l'alto di una riga
JNS    WASNT_AT_TOP          ;Non era al limite superiore, scrive cursore
MOV    PHANTOM_CURSOR_Y,0    ;Era al limite, quindi riportalo qui
CALL  SCROLL_DOWN          ;Era al limite superiore, far scorrere
WASNT_AT_TOP:
CALL    WRITE_PHANTOM         ;Scrive cursore fantasma a nuova posizione
RET
PHANTOM_UP      ENDP

```

```

PHANTOM_DOWN PROC
    CALL ERASE_PHANTOM ;Cancella alla posizione corrente
    INC PHANTOM_CURSOR_Y ;Sposta cursore verso il basso di una riga
    CMP PHANTOM_CURSOR_Y,16 ;Limite inferiore?
    JB WASNT_AT_BOTTOM ;No, scrive quindi il cursore fantasma
    MOV PHANTOM_CURSOR_Y,15 ;Era al limite, quindi riportalo qui
    CALL SCROLL_UP ;Si, far scorrere
WASNT_AT_BOTTOM:
    CALL WRITE_PHANTOM ;Scrivo il cursore fantasma
    RET
PHANTOM_DOWN ENDP

```

Non dimenticate di cambiare i commenti nell'intestazione per PHANTOM_UP e PHANTOM_DOWN, per dire che queste procedure ora utilizzando SCROLL_UP e SCROLL_DOWN_

Listato 27-2. Cambiamenti a PHANTOM.ASM

```

;-----;
; Queste quattro procedure spostano i cursori fantasma. ;
; ;
; Usa: ERASE_PHANTOM, WRITE_PHANTOM ;
; SCROLL_DOWN, SCROLL_UP ;
; Legge: PHANTOM_CURSOR_X, PHANTOM_CURSOR_Y ;
; Scrive: PHANTOM_CURSOR_X, PHANTOM_CURSOR_Y ;
;-----;

```

SCROLL_UP e SCROLL_DOWN sono entrambe semplici procedure, dal momento che visualizzano solo l'altra metà del settore. Per esempio, se state guardando la prima metà del settore, e PHANTOM_DOWN chiama SCROLL_UP, vedrete l'altra metà del settore. SCROLL_UP cambia SECTOR_OFFSET in 256, l'inizio dell'altra metà del settore, sposta il cursore all'inizio della visualizzazione, visualizza la metà del settore, e scrive il cursore fantasma all'inizio della finestra.

Potete vedere tutti i dettagli sia di SCROLL_UP che di SCROLL_DOWN nel listato seguente. Aggiungete queste due procedure a PHANTOM.ASM

Listato 27-3. Aggiungete queste procedure a PHANTOM.ASM

```

    EXTRN DISP_HALF_SECTOR:PROC, GOTO_XY:PROC
.DATA
    EXTRN SECTOR_OFFSET:WORD
    EXTRN LINES_BEFORE_SECTOR:BYTE
.CODE
;-----;
; Queste due procedure permettono lo spostamento tra le due ;
; visualizzazioni di mezzo settore. ;

```



```

;
; Usa:          WRITE_PHANTOM, DISP_HALF_SECTOR, ERASE_PHANTOM, GOTO_XY      ;
;              SAVE_REAL_CURSOR, RESTORE_REAL_CURSOR                        ;
; Legge:       LINES_BEFORE_SECTOR                                         ;
; Scrive:      SECTOR_OFFSET, PHANTOM_CURSOR_Y                             ;
;-----;
SCROLL_UP PROC
    PUSH    DX
    CALL    ERASE_PHANTOM           ;Cancella il cursore fantasma
    CALL    SAVE_REAL_CURSOR       ;Salva la posizione del cursore reale
    XOR     DL,DL                   ;Imposta cursore per visualizzare mezzo settore
    MOV     DH,LINES_BEFORE_SECTOR
    ADD     DH,2
    CALL    GOTO_XY
    MOV     DX,256                  ;Visualizza secondo mezzo settore
    MOV     SECTOR_OFFSET,DX
    CALL    DISP_HALF_SECTOR
    CALL    RESTORE_REAL_CURSOR     ;Ripristina posizione del cursore reale
    MOV     PHANTOM_CURSOR_Y,0     ;Cursore all'inizio secondo mezzo settore
    CALL    WRITE_PHANTOM          ;Ripristina il cursore fantasma
    POP     DX
    RET
SCROLL_UP ENDP

SCROLL_DOWN PROC
    PUSH    DX
    CALL    ERASE_PHANTOM           ;Cancella il cursore fantasma
    CALL    SAVE_REAL_CURSOR       ;Salva la posizione del cursore reale
    XOR     DL,DL                   ;Imposta cursore per visualizzazione mezzo settore
    MOV     DH,LINES_BEFORE_SECTOR
    ADD     DH,2
    CALL    GOTO_XY
    XOR     DX,DX                   ;Visualizza primo mezzo settore
    MOV     SECTOR_OFFSET,DX
    CALL    DISP_HALF_SECTOR
    CALL    RESTORE_REAL_CURSOR     ;Ripristina posizione del cursore reale
    MOV     PHANTOM_CURSOR_Y,15    ;Cursore in fondo al primo mezzo settore
    CALL    WRITE_PHANTOM          ;Ripristina cursore fantasma
    POP     DX
    RET
SCROLL_DOWN ENDP

```

SCROLL_UP e SCROLL_DOWN funzionano abbastanza bene, anche se c'è un piccolo problema. Avviate Dskpatch e lasciate il cursore all'inizio dello schermo. Premete il tasto cursore verso l'alto, e vedrete che Dskpatch rivisualizza la metà del settore. Perché? Non avete controllato questa condizione. Dskpatch visualizza nuovamente il settore quando provate ad andare oltre l'inizio o la fine del settore stesso. Eccovi una sfida: modificate Dskpatch in modo che controlli queste condizioni. Se il

cursore fantasma è all'inizio del settore e premete il cursore verso l'alto, il programma non deve fare nulla. Lo stesso se il cursore si trova in fondo al settore.

SOMMARIO

Con questo capitolo è finito il lavoro su Dskpatch (fatta eccezione per il Capitolo 30, in cui modificherete Dskpatch per una veloce scrittura sullo schermo). L'intento era quello di utilizzare Dskpatch come esempio dell'evoluzione di un programma in assembly, e nello stesso tempo fornirvi un programma utile e funzionante. Ma Dskpatch sviluppato qui non è così finito come potreste pensare. Troverete molte altre caratteristiche nella versione su disco.

Questo libro continuerà ora con una serie di caratteristiche avanzate: rilocazione, scrivere dei programmi .COM, scrivere direttamente sullo schermo, scrivere procedure C in linguaggio assembly, e programmi TSR o residenti in RAM.

PARTE IV

CARATTERISTICHE AVANZATE

RILOCAZIONE

La maggior parte dei programmi presentati nella Parte II e III di questo libro sono di tipo EXE con due segmenti, uno per il codice ed uno per i dati. C'è comunque un punto che non è mai stato toccato: la rilocazione. In questo capitolo, sarà dato un rapido sguardo al processo di rilocazione e ai passaggi che esegue il DOS quando carica un programma .EXE in memoria.

Per mostrare il processo di rilocazione, si costruirà un programma .COM che esegua la sua rilocazione (dal momento che il DOS non fornisce il supporto per la rilocazione dei programmi .COM). Siccome non è stato ancora spiegato come costruire programmi con estensione .COM, inizierete a familiarizzare con alcune direttive che servono per scrivere programmi .COM.

PROGRAMMI .COM

Anche se in questo libro avete imparato a scrivere programmi .EXE, che saranno quelli che utilizzerete la maggior parte delle volte, alcuni programmi devono essere .COM (come i programmi residenti in RAM, che saranno analizzati nel Capitolo 32, di cui potete vederne un esempio in questo capitolo). Per questo tipo di programmi, non è possibile utilizzare le definizioni semplificate dei segmenti (come .CODE), dal momento che queste direttive supportano solo i programmi .EXE. Bisogna invece utilizzare altre definizioni.

Le direttive per la definizione dei segmenti assomigliano molto alla definizione delle procedure, come potete vedere in quest'esempio:

```

    _TEXT      SEGMENT
                .
                .
                .
    _TEXT      ENDS

```

Invece che iniziare un segmento con .CODE, bisogna racchiudere il codice tra le direttive SEGMENT e ENDS. Bisogna anche fornire il nome del segmento (_TEXT in quest'esempio).

Oltre alla definizione di segmento, bisogna utilizzare un'altra direttiva chiamata ASSUME. Quando utilizzate le direttive semplificate, l'assemblatore capisce, dalla

direttiva `.MODEL`, quali segmenti sono puntati dal registro `SEGMENT`. Con le direttive complete, invece, bisogna fornire questo tipo di informazioni all'assemblatore (dal momento che non è possibile utilizzare la direttiva `.MODEL`). Per questo si utilizza una nuova direttiva, `ASSUME`, come nell'esempio che segue:

```
ASSUME  CS:_TEXT, DS:_DATA, SS:STACK
```

Questo indica all'assemblatore che il registro `CS` punta al codice, il registro `DS` punta al segmento dati, e `SS` punta al segmento stack. La direttiva `.MODEL` fornisce automaticamente queste informazioni all'assemblatore.

Alla fine, un programma `.COM`, contenuto interamente in un singolo segmento, inizia con 256-byte di PSP. Per riservare dello spazio per il PSP, i programmi `.COM` devono iniziare con `ORG 100h`. `ORG` indica all'assemblatore di iniziare il codice del programma a 100h (o 256) byte nel segmento. Vedrete tutti questi dettagli nella sezione seguente e nel Capitolo 32.

RILOCAZIONE

Ogni programma `.EXE` inizia con il codice sottoriportato che imposta il registro `DS` in modo che punti al segmento dati (che attualmente consiste in un gruppo di segmenti chiamati `DGROUP`):

```
MOV  AX,DGROUP
MOV  DS,AX
```

Il problema è quello di capire la provenienza del valore per `DGROUP`. Se ci pensate, i programmi possono essere caricati in qualsiasi parte della memoria, il che significa che il valore di `DGROUP` non sarà conosciuto fino a quando non si conosce la posizione in cui verrà caricato il programma in memoria. Il DOS esegue un'operazione conosciuta come *rilocazione* quando carica un programma `.EXE` in memoria. Questo processo di rilocazione, cambia i numeri in modo che `DGROUP` rifletta la locazione attuale del programma in memoria.

Per capire questo processo, scriverete un programma `.COM` che esegue una sua rilocazione. Il nocciolo del problema sta nell'impostare il registro `DS` all'inizio del segmento `_DATA`, e il registro `SS` all'inizio del segmento `STACK`. Come prima cosa bisogna assicurarsi che i tre segmenti siano caricati in memoria nell'ordine corretto:

```
Segmento Codice (_TEXT)
Segmento Dati (_DATA)
Segmento di Stack (_STACK)
```

Fortunatamente questo viene già fatto automaticamente. Quando utilizzate le direttive complete, i segmenti vengono caricati nell'ordine in cui appaiono nel file sorgente.

Se utilizzerete le tecniche seguenti per impostare i registri, dovrete assicurarvi di conoscere l'ordine con cui LINK carica i vostri segmenti (potete utilizzare il file .MAP per controllare l'ordine dei segmenti).

Come si può calcolare il valore di DS? Iniziate a guardare le tre etichette inserite nei vari segmenti del listato seguente. Queste etichette sono END_OF_CODE_SEG, END_OF_DATA_SEG, e END_OF_STACK_SEG. Non sono esattamente quello che vi aspettavate. Perché no? Quando si definiscono i settori in questo modo:

```
_TEXT      SEGMENT
```

(bisogna utilizzare le definizioni complete per i programmi .COM), nessuno ha indicato al linker il modo in cui combinare i vari segmenti. Bisogna quindi iniziare ogni nuovo segmento su un limite come, per esempio, 32C40h. Dato che il linker salta al limite successivo per iniziare ogni segmento, molto spesso è presente un'area vuota tra i segmenti. Con l'etichetta END_OF_CODE_SEG all'inizio di _DATA, includete quest'area vuota. Se avete messo END_OF_CODE_SEG alla fine di _TEXT, l'area vuota non sarà inclusa.

Per il valore del registro DS, _DATA inizia a 39AF:0130, o 39C2:0000. L'istruzione OFFSET _TEXT:END_OF_CODE_SEG fornirà 130h, che è il numero di byte utilizzato da _TEXT. Dividete questo numero per 16 per trovare il numero da aggiungere a DS in modo che punti a _DATA. Si utilizza la stessa tecnica per impostare SS.

Ecco il listato del programma, incluse le istruzioni per la rilocazione, necessarie per un file .COM:

```
        ASSUME CS:_TEXT, DS:_DATA, SS:STACK

_TEXT   SEGMENT
        ORG    100h                ;Riserva spazio per l'area dati del programma .COM
WRITE_STRING  PROC FAR
        MOV    AX,OFFSET _TEXT:END_OF_CODE_SEG
        MOV    CL,4                ;Calcola il numero di paragrafi
        SHR    AX,CL              ; utilizzati dal codice segmento

        MOV    BX,CS
        ADD    AX,BX
        MOV    DS,AX              ;Imposta il registro DS a _DATA

        MOV    BX,OFFSET _DATA:END_OF_DATA_SEG
        SHR    BX,CL              ;Calcola i paragrafi usati dal segmento dati
        ADD    AX,BX              ;Aggiunge il valore utilizzato per il segmento dati
        MOV    SS,AX              ;Imposta il registro SS per lo STACK
        MOV    AX,OFFSET STACK:END_OF_STACK_SEG
        MOV    SP,AX

        MOV    AH,9
        LEA    DX,STRING          ;Carica l'indirizzo della stringa
        INT    21H                ;Scrive la stringa

        MOV    AH,4Ch             ;Chiede di tornare al DOS
```

```

                INT     21h                ;Ritorna al DOS

WRITE_STRING   ENDP

_TEXT         ENDS

_DATA        SEGMENT
END_OF_CODE_SEG LABEL      BYTE
STRING       DB     "Ciao, come stai?$"
_DATA        ENDS

STACK        SEGMENT
END_OF_DATA_SEG LABEL      BYTE
                DB     10 DUP ('STACK  ') ;'STACK' seguito da tre spazi
END_OF_STACK_SEG LABEL      BYTE
STACK        ENDS

                END     WRITE_STRING

```

Assemblete e processate con LINK questo programma, come se fosse un programma .EXE, quindi digitate:

```
EXE2BIN WRITESTR WRITESTR.COM
```

per convertire writestr.exe in un programma .COM, EXE2BIN converte un file EXE in un file BINario (.COM); in altre parole EXE to BINARY.

Potete vedere il risultato del lavoro nella seguente sessione di debug:

```

a>debug writestr.com
-U
3E05:0100    B83001        MOV  AX,0130
3E05:0103    B104          MOV  CL,04
3E05:0105    D3E8          SHR  AX,CL
3E05:0107    8CCB          MOV  BX,CS
3E05:0109    03C3          ADD  AX,BX
3E05:010B    8ED8          MOV  DS,AX
3E05:010D    BB2000        MOV  BX,0020
3E05:0110    D3EB          SHR  BX,CL
3E05:0112    03C3          ADD  AX,BX
3E05:0114    8ED0          MOV  SS,AX
3E05:0116    B85000        MOV  AX,0050
3E05:0119    8BE0          MOV  SP,AX
3E05:011B    B409          MOV  AH,09
3E05:011D    8D160000     LEA  DX,[1000]

```


-U

```

3E05:0121    CD21          INT    21
3E05:0123    B44C          MOV    AH, 4C
3E05:0125    CD21          INT    21
3E05:0127    0000          ADD    [BX+SI], AL
3E05:0129    0000          ADD    [BX+SI], AL
3E05:012B    0000          ADD    [BX+SI], AL
3E05:012D    0000          ADD    [BX+SI], AL
3E05:012F    004865        ADD    [BX+SI+65], CL
3E05:0132    6C            DB     6C
3E05:0133    6C            DB     6C
3E05:0134    6F            DB     6F
3E05:0135    2C20          SUB    AL, 20
3E05:0137    44            INC    SP
3E05:0138    4F            DEC    DI
3E05:0139    53            PUSH   BX
3E05:013A    206865        AND    [BP+SI+65], CH
3E05:013D    7265          JB     01A4
3E05:013F    2E            CS:
3E05:0140    2400          AND    AL, 00

```

-G 121

```

AX=0950 BX=0002 CX=0104 DX=0000 SP=0050 BP=0000 SI=0000 DI=0000
DS=3E18 ES=3DF5 SS=3E1A CS=3E05 IP=0121 NV UP EI PL NZ NA PO NC
3E05:0121 CD21          INT    21

```

Raramente dovrete fare questo tipo di rilocazione dal momento che il DOS la fa automaticamente per i programmi .EXE. Ma questo vi aiuta a capire cosa succede all'interno del computer.

PROGRAMMI .COM E PROGRAMMI .EXE

Questo capitolo termina evidenziando le differenze tra file .COM e .EXE e sul diverso metodo di caricamento in memoria usato dal DOS.

Un programma .COM salvato sul disco è essenzialmente un'immagine della memoria. Per questo un programma .COM è ristretto ad un singolo segmento, anche se esegue una propria rilocazione, come avete fatto in questo capitolo.

Un programma .EXE, invece, lascia al DOS la rilocazione. Questo rende molto facile l'utilizzo di segmenti multipli. Per questa ragione, i grossi programmi sono .EXE e non .COM.

Per avere un'idea ben precisa della differenza tra .COM e .EXE, ecco come questi vengono caricati ed avviati dal DOS. Quando il DOS carica un programma .COM in memoria, segue questi passaggi:

- Come prima cosa crea il Program Segment Prefix (PSP), che è l'area da 256 byte vista nel Capitolo 22. Questo PSP contiene la linea di comandi digitata.
- L'intero file .COM viene copiato dal disco in memoria, immediatamente dopo i 256 byte PSP.
- Il DOS imposta i registri dei tre segmenti DS, ES, e SS all'inizio del PSP.
- Il DOS imposta il registro SP alla fine del segmento (generalmente FFFE, che è l'ultima parola del segmento).
- Finalmente il DOS salta all'inizio del programma, imposta il registro CS all'inizio del PSP e il registro IP a 100h (l'inizio del programma .COM).

Al contrario i passaggi per caricare un file .EXE sono molto più complessi, dal momento che il DOS deve eseguire la rilocazione. Ma dove trova le informazioni necessarie per la rilocazione?

Ogni file .EXE ha un'intestazione contenuta all'inizio del programma. Questa intestazione, o *tabella di rilocazione*, è sempre lunga almeno 512 byte, e contiene tutte le informazioni necessarie al DOS per la rilocazione. Con le versioni più recenti del Macro Assembler, la Microsoft ha incluso un programma chiamato EXEMOD che serve per vedere le informazioni contenute in quest'intestazione. Per esempio questa è l'intestazione della versione .EXE di WRITESTR:

```
A>exemod writestr.exe
```

```
Microsoft (R) EXE File Header Utility Version 4.02
Copyright (C) Microsoft Corp 1985-1987. All rights reserved.
```

writestr.exe	(hex)	(dec)
.EXE size (bytes)	290	656
Minimum load size (bytes)	90	144
Overlay number	0	0
Initial CS:IP	0000:0000	
Initial SS:SP	0004:0050	80
Minimum allocation (para)	0	0
Maximum allocation (para)	FFFF	65535
Header size (para)	20	32
Relocation table offset	1E	30
Relocation entries	1	1

```
A>
```

Alla fine di questa tabella, potete vedere una singola rilocazione per l'istruzione MOV AX,DGROUP. Ogni volta che si costruisce un riferimento ad un indirizzo di segmento, come con 'MOV AX,DGROUP, LINK' si aggiunge una rilocazione nella tabella. L'indirizzo del segmento non è conosciuto fino a quando il DOS carica il programma in memoria.

Ci sono anche altre informazioni interessanti nella tabella; per esempio i valori iniziali CS:IP e SS:SP. La tabella indica al DOS quanta memoria serve al programma prima di

poter essere eseguito. Ecco i passaggi che segue il DOS nel caricare un programma .EXE:

- Il DOS crea il PSP, come per i programmi .COM.
- Controlla l'intestazione del file .EXE per vedere dove finisce l'intestazione stessa e dove inizia il programma. Carica quindi il resto del programma in memoria dopo il PSP.
- Quindi, utilizzando le informazioni dell'intestazione, il DOS trova e imposta correttamente tutti i riferimenti nel programma che devono essere rilocati, come i riferimenti agli indirizzi dei segmenti.
- Il DOS imposta quindi i registri ES e DS, in modo che puntino all'inizio del PSP. Se il vostro programma ha il proprio segmento dati, il programma deve cambiare DS e/o ES in modo che puntino al segmento dati.
- Il DOS imposta SS:SP in accordo con le informazioni trovate nell'intestazione. Nel caso illustrato, SS:SP sarà posto a 0004:0050. Questo significa che il DOS imposterà SP a 0050, e SS in modo che sia quattro paragrafi più alto rispetto alla fine del PSP.
- Alla fine il DOS salta all'inizio del programma utilizzando l'indirizzo fornito nell'intestazione. Questo imposta il registro CS all'inizio del segmento di codice, e IP allo scarto dato dall'intestazione del programma .EXE.

DETTAGLI SUI SEGMENTI E SU ASSUME

In questo capitolo imparerete una caratteristica chiamata *sovrapposizione del segmento*, che sarà utilizzata nel prossimo capitolo quando si scriverà direttamente sullo schermo. Durante questa spiegazione daremo un'occhiata ad ASSUME e alle definizioni dei segmenti completi.

SOVRAPPOSIZIONE DEL SEGMENTO

Fino ad ora avete sempre letto e scritto i dati localizzati nel segmento dati. In questo libro è stato trattato un singolo segmento dati, quindi non c'è stata ragione di leggere o scrivere i dati in altri segmenti.

Ma, in alcuni casi, serve più di un segmento dati. Un esempio classico è la scrittura diretta sullo schermo: molti programmi scrivono sullo schermo portando i dati direttamente nella memoria video e scavalcando completamente il BIOS, per guadagnare in velocità. La memoria video, nel PC, è localizzata al segmento B800h per l'adattatore colore/grafici e al segmento B000h per l'adattatore monocromatico. Scrivere direttamente nella memoria video, significa scrivere in segmenti differenti. In questa sezione, scriverete un piccolo programma che mostra come sia possibile scrivere in due segmenti differenti, utilizzando i registri DS e ES per puntare a tali segmenti. Infatti, la maggior parte dei programmi che scrivono direttamente nella memoria video, utilizzano il registro ES per puntare a tale area, come sarà illustrato nel capitolo successivo.

Nell'esempio seguente, utilizzerete la definizione completa dei segmenti per avere un controllo più ampio rispetto alla definizione semplificata. La maggior parte delle volte utilizzerete la definizione semplificata (per scrivere direttamente sullo schermo, per esempio), ma in questo capitolo sarà utilizzata la definizione completa per fornire maggiori esempi sul suo utilizzo, e per capire la dichiarazione ASSUME che serve con la definizione completa dei segmenti.

Ecco il programma. E' molto corto, contiene due segmenti dati, e una variabile per ogni segmento dati:

```
DOSSEG

_DATA    SEGMENT
DS_VAR  DW          1
_DATA    ENDS
```

```

EXTRA_SEG      SEGMENT PUBLIC
ES_VAR         DW      2
EXTRA_SEG      ENDS

STACK          SEGMENT          STACK
DB      10 DUP ('STACK  ')      ;'STACK' seguito da tre spazi
STACK          ENDS

_TEXT          SEGMENT
ASSUME CS:_TEXT, DS:_DATA, ES:EXTRA_SEG, SS:STACK

TEST_SEG      PROC
MOV  AX,_DATA      ;Indirizzo del segmento per _DATA
MOV  DS,AX         ;Imposta il registro ds per _DATA
MOV  AX,EXTRA_SEG  ;Indirizzo segmento per EXTRA_SEG

MOV  AX,DS_VAR     ;Legge la variabile dal segmento dati
MOV  BX,ES:ES_VAR  ;Legge la variabile dal segmento extra

MOV  AH,4Ch        ;Chiede di tornare al DOS
INT  21h           ;Ritorna al DOS
TEST_SEG      ENDP

_TEXT          ENDS

END    TEST_SEG

```

Utilizzerete questo programma per studiare la sovrapposizione dei segmenti e la direttiva ASSUME.

Notate che il segmento dati e il segmento di stack è stato messo *prima* del segmento codice, e che la direttiva ASSUME è stata messa dopo tutte le dichiarazioni di segmento. Come è possibile vedere in questa sezione, questo è il risultato dell'utilizzo dei due segmenti dati.

Date un'occhiata alle due istruzioni MOV del programma:

```

MOV  AX,DS_VAR
MOV  BX,ES:ES_VAR

```

ES: all'inizio della seconda istruzione indica all'8088 di utilizzare il registro ES, invece di DS, per questa operazione (per leggere i dati dal segmento extra). Ogni istruzione ha un registro di segmento standard che utilizza quando fa riferimento ai dati.

Ecco come funziona: l'8088 ha quattro istruzioni speciali, una per ognuno dei quattro registri di segmento. Queste istruzioni sono istruzioni di *sovrapposizione del segmento*, e indicano all'8088 di utilizzare un registro di segmento specifico, piuttosto che quello standard, quando le istruzioni che seguono la sovrapposizione del segmento cercano di leggere o scrivere nella memoria.

Per esempio, l'istruzione MOV AX,ES:ES_VAR è codificata in due istruzioni. Se disassemblate il programma vedrete queste due istruzioni:

```
2CF4:000D          ES:
2CF4:000E    8B1E0000    MOV  BX, [0000]
```

Questo mostra che l'assemblatore trasforma le istruzioni in istruzioni di sovrapposizione del segmento, seguite dall'istruzione MOV. Ora l'istruzione MOV leggerà i suoi dati dal segmento ES piuttosto che dal segmento DS.

Se tracciate il programma, vedrete che la prima istruzione MOV imposta AX uguale a 1 (DS_VAR) e la seconda MOV imposta BX uguale a 2 (ES_VAR). In altre parole si leggono i dati da due segmenti differenti.

UN ALTRO SGUARDO AD ASSUME

Ecco cosa succede quando si rimuove ES: dal programma. Cambiate la riga:

```
MOV  BX, ES:ES_VAR
```

in modo che risulti:

```
MOV  BX, ES_VAR
```

In questo modo non indicate più all'assemblatore che volete utilizzare il registro ES per leggere dalla memoria, ma che volete utilizzare il segmento di default (DS): giusto? Sbagliato. Utilizzate debug per vedere il risultato della modifica. Vedrete che c'è sempre la sovrapposizione del segmento ES: prima dell'istruzione MOV. Come ha fatto l'assemblatore a sapere che la variabile è un segmento extra invece di un segmento dati? Utilizzando le informazioni della direttiva ASSUME.

La direttiva ASSUME indica all'assemblatore che il registro DS punta al segmento DATA_SEG, mentre ES punta a EXTRA_SEG. Ogni volta che si scrive un'istruzione che utilizza una variabile di memoria, l'assemblatore cerca la dichiarazione della variabile per vedere in che segmento è dichiarata. Quindi cerca attraverso la lista ASSUME per vedere quale segmento di registro punta a quel segmento. L'assemblatore utilizza questo registro di segmento quando genera un'istruzione.

Nel caso dell'istruzione MOV BX,ES_VAR, l'assemblatore nota che ES_VAR si trova nel segmento chiamato EXTRA_SEG e il registro ES punta a quel segmento; per questo motivo genera un'istruzione di sovrapposizione di segmento ES: su se stesso. Se si fosse voluto spostare ES_VAR in STACK_SEG, l'assemblatore avrebbe generato un'istruzione di sovrapposizione di segmento SS:. L'assemblatore genera automaticamente qualsiasi istruzione di sovrapposizione segmento necessaria, in modo che, naturalmente, la direttiva ASSUME rifletta il contenuto attuale dei registri di segmento.

SOMMARIO

In questo capitolo avete imparato parecchie cose sui segmenti e su come l'assemblatore lavora con loro. Come prima cosa avete imparato la sovrapposizione dei segmenti, che permette di leggere e scrivere dati in altri segmenti. Utilizzerete questa sovrapposizione nel prossimo capitolo, quando scriverete i caratteri direttamente sullo schermo senza utilizzare il BIOS.

Il prossimo capitolo parla della scrittura diretta sullo schermo. Questo serve per incrementare in modo drastico la velocità di scrittura dei caratteri sullo schermo.

UNA WRITE_CHAR MOLTO VELOCE

All'inizio di questo libro è stato detto che molte persone che programmano in assembly lo fanno per ragioni di velocità. I programmi in linguaggio assembly sono quasi sempre più veloci di quelli scritti in altri linguaggi. Ma forse avrete notato che Dskpatch non scrive sullo schermo così velocemente come molti altri programmi in commercio. Perché è così lento?

Fino ad ora avete utilizzato le routine del BIOS per visualizzare i caratteri sullo schermo. E come vedrete in questo capitolo le routine del BIOS sono molto lente. La maggior parte dei programmi scavalcano la ROM BIOS e scrivono i caratteri direttamente nella memoria video.

In questo capitolo modificherete Dskpatch in modo che scriva i caratteri molto velocemente sullo schermo. Sfortunatamente, dovrete fare un certo numero di modifiche a Dskpatch per ottenere una visualizzazione veloce: non è possibile scrivere una nuova procedura WRITE_CHAR per ragioni che scoprirete presto.

IL SEGMENTO DI SCHERMO

Prima di poter scrivere i caratteri direttamente nella memoria video, servono alcune informazioni come, per esempio, la posizione della memoria video e la modalità in cui i caratteri vengono salvati in questa memoria.

La prima domanda ha una semplice risposta. La memoria video ha un suo segmento, che si può trovare in B800h o B000h. Perché ci sono due segmenti differenti? Ci sono due classi di adattatori, monocromatici (MDA) e a colori (CGA, EGA, e VGA), ed è possibile disporre contemporaneamente di entrambi gli adattatori nel computer. Per questo motivo questi hanno due indirizzi differenti.

Gli adattatori monocromatici fanno riferimento all'adattatore monocromatico IBM, alla scheda grafica Hercules, e alle schede EGA e VGA collegate a video monocromatici. Le schede monocromatiche visualizzano i caratteri sullo schermo in verde, bianco o ambra (dipende dal video), ed hanno solo un limitato set di 'colori': normale, evidenziato, inverso, e sottolineato. Le schede monocromatiche hanno il loro segmento in B000h.

Gli adattatori colore, invece, possono visualizzare 16 colori differenti alla volta, e possono anche utilizzare la modalità grafica (che non sarà trattata in questo libro). Gli adattatori più comunemente usati sono EGA e VGA, anche se è ancora diffusa anche la CGA. Gli adattatori colore hanno la memoria video in B800h.

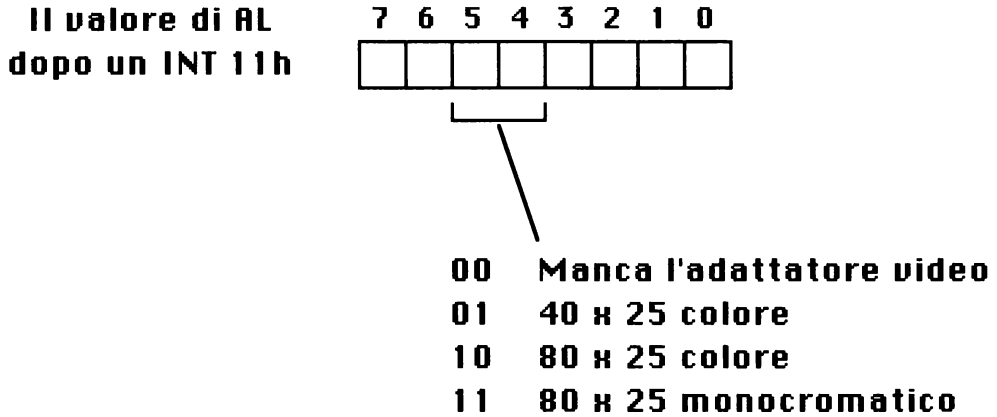


Figura 30-1. I Flag di INT 11h

Molti utenti non sanno che tipo di adattatore hanno e non si interessano di saperlo. Deve essere il programma a determinare quale adattatore è attivo.

Per questo è possibile utilizzare INT 11h, che fornisce una lista dell'equipaggiamento installato. Come potete vedere dalla figura 30-1, i bit 4 e 5 forniscono il tipo di video (monocromatico o a colori). In altre parole, il segmento di schermo sarà in B000h (monocromatico) se entrambi i bit sono a 1, e in B800h (colore) in caso contrario (non è contemplato il caso in cui non sia installato un adattatore).

Dal momento che non si conosce quale segmento utilizzare fino a quando non si avvia il programma, bisogna chiamare una procedura, `INIT_WRITE_CHAR`, che determina il segmento da utilizzare prima di effettuare qualsiasi chiamata a `WRITE_CHAR`. Questa chiamata sarà inserita all'inizio di `Disk_patch` in modo da essere sicuri di chiamarla prima di scrivere qualsiasi carattere sullo schermo. Ecco le modifiche da effettuare a `DSKPATCH.ASM` per aggiungere questa chiamata:

Listato 30-1. Cambiamenti a `DSKPATCH.ASM`

```

EXTRN WRITE_PROMPT_LINE:PROC, DISPATCHER:PROC
EXTRN INIT_WRITE_CHAR      :PROC
DISK_PATCH PROC
MOV  AX,DGROUP          ;Inserisce il segmento dati in AX
MOV  DS,AX              ;Imposta DS per puntare ai dati

CALL INIT_WRITE_CHAR
CALL CLEAR_SCREEN
CALL WRITE_HEADER
.
.
.

```

Quindi aggiungete INIT_WRITE_CHAR a VIDEO_IO.ASM

Listato 30-2. Aggiungete questa procedura a VIDEO_IO.ASM

```

PUBLIC INIT_WRITE_CHAR

;-----;
; Dovete chiamare questa procedura prima di chiamare WRITE_CHAR dal      ;
; momento che WRITE_CHAR utilizza delle informazioni impostate da      ;
; questa procedura                                                       ;
;                                                                           ;
; Scrive:          SCREEN_SEG                                             ;
;-----;
INIT_WRITE_CHAR PROC
    PUSH  AX
    PUSH  BX
    MOV   BX,0B800h                ;Impostazione per l'adattatore colore
    INT   11h                      ;Richiede le informazioni sull'equipaggiamento
    AND   AL,30h                   ;Conserva solo il tipo di video
    CMP   AL,30h                   ;E' un adattatore monocromatico?
    JNE   SET_BASE                 ;No è a colori, quindi usa B800
    MOV   BX,0B000h
SET_BASE
    MOVSCREEN_SEG,BX              ;Salva il segmento video
    POP   BX
    POP   AX
    RET   INIT_WRITE_CHAR        ENDP

```

Notate che il segmento video è salvato in SCREEN_SEG (che aggiungerete dopo). WRITE_CHAR utilizzerà questa variabile quando la utilizzerà per scrivere direttamente sullo schermo.

Ora che sapete come localizzare la memoria video, bisogna sapere come sono salvati i caratteri e i loro attributi.

ORGANIZZAZIONE DELLA MEMORIA VIDEO

Se utilizzate debug per vedere la memoria video quando sulla prima riga dello schermo compare:

```
DSKPATCH ASM
```

vedrete i seguenti dati (per una scheda colore):

```

-D800:0
      800:0000  44 07 53 07 4B 07 50 07-41 07 54 07 43 07 48 07   D.S.K.P.A.T.C.H.
      800:0010  20 07 41 07 53 07 4D 07-20 07 20 07 20 07 20 07   .A.S.M. . . . .
      .
      .
      .

```

In altre parole c'è uno 07 tra ogni carattere sullo schermo. Come forse ricorderete dal Capitolo 18, 7 è l'attributo per il testo normale (e 70h è quello per l'attributo in inverso). In poche parole ogni carattere sullo schermo utilizza una parola di memoria video, con il codice del carattere nel byte basso e l'attributo nel byte alto. Scrivete una nuova versione di WRITE_CHAR che scriva i caratteri direttamente sullo schermo. Apportate questi cambiamenti a Video_io.asm:

Listato 30-3. Cambiamenti a VIDEO_IO.ASM

```

PUBLIC WRITE_CHAR
EXTRN CURSOR_RIGHT:PROC
;-----
; Questa procedura invia un carattere allo schermo scrivendo direttamente
; nella memoria video; in questo modo un carattere come il backspace è
; trattato come un qualsiasi altro carattere e visualizzato. Questa procedura
; deve effettuare parecchie operazioni per aggiornare la posizione del
; cursore.
;
;      DL      Byte da stampare sullo schermo
;
; Usa:      CURSOR_RIGHT
; Legge:    SCREEN_SEG
;-----
WRITE_CHAR PROC
      PUSH AX
      PUSH BX
      PUSH CX
      PUSH DX
      PUSH ES

      MOV AX,SCREEN_SEG      ;Prende il segmento per la memoria video
      MOV ES,AX             ;Punta ES alla memoria video

      PUSH DX               ;Salva il carattere da scrivere
      MOV AH,3              ;Chiede la posizione del cursore
      XOR BH,BH             ;Sulla pagina 0
      INT 10h              ;Prende riga,colonna
      MOV AL,DX             ;Mette la riga in AL
      MOV BL,80             ;Ci sono 80 caratteri per linea
      MUL BL                ;AX = riga * 80
      ADD AL,DL             ;Aggiunge la colonna
      ADC AH,0

```

```

    SHL    AX,1           ;Converte nel byte offset
    MOV    BX,AX         ;Mette il byte offset del cursore in BX
    POP    DX            ;Ripristina il carattere

    MOV    DH,7         ;Utilizza l'attributo normale
    MOV    ES:[BX],DX   ;Scrive il carattere/attributo sullo schermo
    CALL   CURSOR_RIGHT ;Ora si sposta alla posizione successiva del cursore

    POP    ES
    POP    DX
    POP    CX
    POP    BX
    POP    AX
    RET

WRITE_CHAR    ENDP

```

Bisogna ora aggiungere una variabile di memoria a VIDEO_IO.ASM:

Listato 30-4. *Aggiungete DATA_SEG all'inizio di VIDEO_IO.ASM*

```

.MODEL SMALL

.DATA
SCREEN_SEG    DW    0B800h    ;Segmento del buffer video

.CODE

```

Dopo aver fatto questi cambiamenti ricostruite Dskpatch (dovete riassemblare DSKPATCH e VIDEO_IO) e provate la nuova versione. Quello che troverete è che Dskpatch non scriverà sullo schermo più velocemente di prima; per una ragione molto semplice. Il cursore deve essere mosso dopo aver scritto ogni carattere, e questo è un processo molto lento.

ALTA VELOCITÀ

La soluzione sta nel riscrivere la routine in VIDEO_IO e CURSOR per tener traccia della posizione del cursore invece che spostarlo; il cursore sarà spostato solo quando sarà necessario. Per questo si devono introdurre due nuove variabili di memoria: SCREEN_X e SCREEN_Y. Non è molto difficile da fare, ma bisogna cambiare un certo numero di procedure e riscriverne di nuove.

C'è anche un'altra ottimizzazione che è possibile fare. Fino ad ora WRITE_CHAR calcola l'offset del cursore nel buffer video ogni volta che lo si chiama. Ma dato che si tiene traccia della posizione del cursore, si può tenere traccia anche del suo offset nella variabile SCREEN_PTR:

Listato 30-5. Cambiamenti a WRITE_CHAR in VIDEO_IO.ASM

```

; Usa:          CURSOR_RIGHT          ;
; Legge:       SCREEN_SEG, SCREEN_PTR ;
;-----;
WRITE_CHAR     PROC
    PUSH  AX
    PUSH  BX
    PUSH  CX
    PUSH  DX
    PUSH  ES

    MOV   AX,SCREEN_SEG      ;Prende il segmento per la memoria video
    MOV   ES,AX             ;Punta ES alla memoria video
    MOV   BX, SCREEN_PTR    ;Punta al carattere nella memoria video

    PUSH  DX                ;Salva il carattere da scrivere
    MOV   AH,3              ;Chiede la posizione del cursore
    XOR   BH,BH             ;Sulla pagina 0
    INT  10h                ;Prende riga,colonna
    MOV   AL,DH             ;Mette la riga in AL
    MOV   BL,00             ;Ci sono 80 caratteri per linea
    MUL  BL                 ;AX = riga * 80
    ADD  AL,DL              ;Aggiunge la colonna
    ADC  AH,0
    SHL  AX,1               ;Converte nel byte offset
    MOV  BX,AX              ;Mette il byte offset del cursore in BX
    POP  DX                 ;Ripristina il carattere

    MOV  DH,7               ;Utilizza l'attributo normale
    MOV  ES:[BX],DX         ;Scrive il carattere/attributo sullo schermo
    CALL CURSOR_RIGHT       ;Si sposta alla posizione successiva

    POP  ES
    POP  DX
    POP  CX
    POP  BX
    POP  AX
    RET
WRITE_CHAR     ENDP

```

Come potete vedere WRITE_CHAR è diventata abbastanza semplice. Dovete anche aggiungere tre nuove variabili di memoria a DATA_SEG in VIDEO_IO.ASM:

Listato 30-6. Cambiamenti a .DATA in VIDEO_IO.ASM

```

.DATA
    PUBLIC SCREEN_PTR
    PUBLIC SCREEN_X, SCREEN_Y

```

```

SCREEN_SEG    DW    0D800h    ;Segmento del buffer video
SCREEN_PTR    DW    0          ;Offset del cursore nella memoria video
SCREEN_X     DB    0          ;Posizione del cursore
SCREEN_Y     DB    0

```

```
.CODE
```

E finalmente, ecco i cambiamenti a WRITE_ATTRIBUTE_N_TIMES in modo che scriva direttamente sullo schermo:

Listato 30-7. Cambiamenti a WRITE_ATTRIBUTE_N_TIMES in VIDEO_IO.ASM

```

; Usa:          CURSOR_RIGHT          ;
; Legge:        SCREEN_SEG, SCREEN_PTR ;
;-----;
WRITE_ATTRIBUTE_N_TIMES  PROC
    PUSH  AX
    PUSH  BX
    PUSH  CX
    PUSH  DX
    PUSH  DI
    PUSH  ES

    MOV   AX,SCREEN_SEG    ;Prende il segmento per la memoria video
    MOV   ES,AX           ;Punta ES alla memoria video
    MOV   DI,SCREEN_PTR    ;Punta al carattere nella memoria video
    INC   DI               ;Punta all'attributo sotto al cursore
    MOV   AL,DL           ;Mette l'attributo in AL

ATTR_LOOP
    STOSB                ;Salva un attributo
    INC   DI              ;Si sposta all'attributo successivo
    INC   SCREEN_X        ;Si sposta alla colonna successiva
    LOOP ATTR_LOOP        ;Scrive N attributi

    DEC   DI              ;Punta all'inizio del carattere successivo
    MOV   SCREEN_PTR,DI

    POP   ES
    POP   DI
    POP   DX
    POP   CX
    POP   BX
    POP   AX
    RET

WRITE_ATTRIBUTE_N_TIMES  ENDP

```

La maggior parte di queste procedure dovrebbero essere chiare, a parte la nuova istruzione: STOSB (*STOre String Byte*). STOSB è l'opposto dell'istruzione LODSB che carica un byte da DS:SI e incrementa il registro SI. STOSB, in altre parole, salva il byte

contenuto in AL nell'indirizzo ES:DI, e incrementa DI.

Tutti gli altri cambiamenti da fare (a parte un'eccezione di un piccolo cambiamento in KBD_IO) sono le procedure in CURSOR.ASM. Come prima cosa bisogna cambiare GOTO_XY in modo che imposti SCREEN_X e SCREEN_Y e calcoli il valore di SCREEN_PTR:

Listato 30-8. Cambiamenti a GOTO_XY in CURSOR.ASM

```

PUBLIC GOTO_XY

.DATA
    EXTRN SCREEN_PTR:WORD
    EXTRN SCREEN_X:BYTE, SCREEN_Y:BYTE

.CODE
;-----;
; Questa procedura sposta il cursore ;
; ;
; DH Riga (Y) ;
; DL Colonna (X) ;
;-----;
GOTO_XY PROC
    PUSH AX
    PUSH BX
    MOV BH,0 ;Visualizza pagina 0
    MOV AH,2 ;Richiama SET CURSOR POSITION
    INT 10h ;Lascia agire la ROM BIOS

    MOV AL,DH ;Prende il numero di riga
    MOV BL,80 ;Moltiplica per 80 caratteri per linea
    MUL BL ;AX = riga * 80
    ADD AL,DL ;Aggiunge la colonna
    ADC AH,0 ;AX = riga * 80 + colonna
    SHL AX,1

    MOV SCREEN_PTR,AX ;Salva l'offset del cursore
    MOV SCREEN_X,DL ;Salva la posizione del cursore
    MOV SCREEN_Y,DH

    POP BX
    POP AX
    RET
GOTO_XY ENDP

```

Come potete vedere è stato spostato il calcolo dell'offset del carattere sotto al cursore da WRITE_CHAR, da dove era prima a qui.

Dovete anche modificare CURSOR_RIGHT in modo da aggiornare le variabili di memoria:

Listato 30-9. Cambiamenti a *CURSOR_RIGHT* in *CURSOR.ASM*

```

PUBLIC  CURSOR_RIGHT

.DATA
    EXTRN  SCREEN_PTR:WORD      ;Punta al carattere sotto al cursore
    EXTRN  SCREEN_X:BYTE, SCREEN_Y:BYTE

.CODE
;-----;
; Questa procedura sposta il cursore a destra di una posizione o alla      ;
; riga successiva se il cursore si trova a fine riga.                        ;
;                                                                           ;
; Usa:          SEND_CRLF                                             ;
; Scrive:       SCREEN_PTR, SCREEN_X, SCREEN_Y                         ;
;-----;

CURSOR_RIGHT  PROC
    INC  SCREEN_PTR          ;Si sposta al carattere successivo
    INC  SCREEN_PTR
    INC  SCREEN_X           ;Si sposta alla colonna successiva
    CMP  SCREEN_X, 79       ;Si assicura che la colonna sia <= 79
    JBE  OK
    CALL SEND_CRLF         ;Va alla linea successiva

OK:
    RET
CURSOR_RIGHT  ENDP

```

Bisogna cambiare anche *CLEAR_TO_END_OF_LINE* in modo che utilizzi *SCREEN_X* e *SCREEN_Y* invece che la locazione del cursore reale:

Listato 30-10. Cambiamenti a *CLEAR_TO_END_OF_LINE* in *CURSOR.ASM*

```

PUSH  CX
PUSH  DX
MOV   AH,3           ;Legge la posizione corrente del cursore
XOR   BH,BH         ;a pagina 0
INT   10h           ;Si hanno ora (X,Y) in DL, DH
MOV   DL, SCREEN_X
MOV   DH, SCREEN_Y
MOV   AH,6           ;Imposta per cancellare fino alla fine della riga
XOR   AL,AL         ;Cancella finestra

```

I prossimi passaggi hanno bisogno di alcune spiegazioni. Dal momento che non viene più aggiornata la posizione del cursore vero, il cursore vero e quello virtuale saranno spesso fuori sincronia. La maggior parte delle volte non è un problema, ma ci sono alcuni casi in cui i due devono essere sincronizzati. Per esempio, prima di chiedere l'input all'utente, bisogna spostare il cursore nella posizione in cui si pensa che dovrebbe essere. Questo è fattibile con la procedura *UPDATE_REAL_CURSOR*, che sposta il cursore reale.

Dall'altra parte, *SEND_CRLF* sposta il cursore vero, quindi bisogna chiamare *UPDATE_VIRTUAL_CURSOR* per spostare il cursore virtuale nella posizione in cui si

trova il cursore vero dopo SEND_CRLF.
Ecco le due procedure che servono a CURSOR.ASM:

Listato 30-11. Aggiungete queste procedure a CURSOR.ASM

```

PUBLIC  UPDATE_REAL_CURSOR
;-----;
; Questa procedura sposta il cursore vero nella posizione corrente del ;
; cursore virtuale. Deve essere chiamato appena prima della richiesta ;
; di input all'utente. ;
;-----;
UPDATE_REAL_CURSOR      PROC
    PUSH  DX
    MOV   DL,SCREEN_X      ;La posizione del cursore virtuale
    MOV   DH,SCREEN_Y
    CALL  GOTO_XY          ;Sposta il cursore reale in questa posizione
    POP   DX
    RET
UPDATE_REAL_CURSOR      ENDP

PUBLIC  UPDATE_VIRTUAL_CURSOR
;-----;
; Questa procedura aggiorna la posizione del cursore virtuale in ;
; accordo con la posizione del cursore reale ;
;-----;
UPDATE_VIRTUAL_CURSOR   PROC
    PUSH  AX
    PUSH  BX
    PUSH  CX
    PUSH  DX
    MOV   AH,3             ;Chiede la posizione del cursore
    XOR   BH,BH           ;in pagina 0
    INT  10h              ;Salva la posizione del cursore in DH, DL
    CALL  GOTO_XY          ;Sposta il cursore virtuale in questa posizione
    POP   DX
    POP   CX
    POP   BX
    POP   AX
    RET
UPDATE_VIRTUAL_CURSOR   ENDP

```

Notate che si sta usando GOTO_XY per aggiornare le tre variabili SCREEN_X, SCREEN_Y, e SCREEN_PTR.

Bisogna quindi aggiornare alcune procedure per poter utilizzare le due precedenti. Ecco i cambiamenti a SEND_CRLF:

Listato 30-12: *Cambiamenti a SEND_CRLF in CURSOR.ASM*

```

;
; Usa          UPDATE_VIRTUAL_CURSOR
;-----;
SEND_CRLF PROC
    PUSH    AX
    PUSH    DX
    MOV     AH,2                ;Richiede funzione output carattere
    MOV     DL,CR              ;Invia un carattere di ritorno carrello
    INT     21h
    MOV     DL,LF              ;Invia un carattere di avanzamento riga
    INT     21h
    CALL    UPDATE_VIRTUAL_CURSOR ;Aggiorna posizione cursore virtuale
    POP     DX
    POP     AX
    RET
SEND_CRLF ENDP

```

Questo assicura che si possa conoscere la posizione del cursore dopo aver spostato il cursore reale sulla riga successiva.

Ecco infine le modifiche da effettuare a READ_STRING, che tiene la posizione del cursore reale e di quello virtuale in sincronismo durante l'inserimento da tastiera:

Listato 30-13. *Cambiamenti a READ_STRING in KBD_IO.ASM*

```

EXTRN UPDATE_REAL_CURSOR:PROC
;-----;
;
;
; Usa:         BACK_SPACE, WRITE_CHAR, UPDATE_REAL_CURSOR
;-----;
READ_STRING PROC
    PUSH    AX
    PUSH    BX
    PUSH    SI
    MOV     SI,DX                ;Usa SI per registro indice e
START_OVER:
    CALL    UPDATE_REAL_CURSOR
    MOV     BX,2                ;BX per offset da inizio buffer
    .
    .
    .
READ_NEXT_CHAR:
    CALL    UPDATE_REAL_CURSOR  ;Sposta cursore reale su quello virtuale
    MOV     AH,7
    INT     21h

```

Assemblete nuovamente i tre file che sono stati cambiati, e processateli quindi con LINK per creare Dskpatch. Noterete che la visualizzazione è molto più veloce che in precedenza.

SOMMARIO

Velocizzare WRITE_CHAR è stato un po' duro dal momento che si sono dovute modificare un certo numero di procedure. I programmi che scrivono velocemente sullo schermo sono decisamente migliori di quelli che impiegano un certo periodo di tempo per visualizzare delle informazioni. Il prossimo capitolo tratta un altro argomento interessante che probabilmente interesserà parecchi di voi: scrivere procedure e funzioni per il linguaggio C in linguaggio assembly. Per quelli che utilizzano un altro linguaggio, il prossimo capitolo può essere comunque un punto d'inizio.

PROCEDURE C IN ASSEMBLY

In questo capitolo vi sarà mostrato come scrivere in linguaggio assembly procedure che possono essere utilizzate in programmi C. Il C è uno dei linguaggi di programmazione ad alto livello più popolari. (Se volete scrivere delle procedure per altri linguaggi, come il Pascal o il BASIC, probabilmente le procedure di questo capitolo funzioneranno senza bisogno di modifiche; sarà sufficiente cambiare solamente la direttiva `.MODEL`).

Originariamente scritto da Dennis Ritchie ai Bell Laboratories, il C è diventato abbastanza popolare dal momento che è un linguaggio moderno e ad alto livello. Ma dato che si tratta di un linguaggio di programmazione di utilizzo generale, ci possono essere delle volte in cui si ha la necessità di scrivere delle parti in assembly, sia per guadagnare in velocità, che per poter accedere a routine di basso livello della macchina.

UNA PROCEDURA PER CANCELLARE LO SCHERMO PER IL C

Inizierete riscrivendo un procedura relativamente semplice, `CLEAR_SCREEN`, in modo da poterla chiamare direttamente dal C. Come potrete vedere, scrivere programmi in linguaggio assembly da utilizzare in programmi C è abbastanza semplice.

Nota: Per assemblare i programmi in questo capitolo, vi serve il Microsoft MASM 5.1 o superiore, il Turbo Assembler, o l'ultima versione di OPTASM che supporta la miscelazione di linguaggi di programmazione del MASM 5.1. Sarà anche utilizzato il compilatore Microsoft C per gli esempi in questo capitolo.

La direttiva `.MODEL`, utilizzata finora, permette di definire il modello di memoria del programma creato. Con la versione 5.1 del MASM, Microsoft ha aggiunto un'estensione alla direttiva `.MODEL` che permette di scrivere dei programmi da associare a diversi linguaggi di programmazione (inclusi C e Pascal). Per indicare al MASM che si sta

scrivendo una procedura in C, bisogna aggiungere una "C" alla fine:

```
.MODEL    SMALL,C
```

Iniziate a riscrivere CLEAR_SCREEN dando un'occhiata alla versione assembly scritta nella Parte II di questo libro:

```

PUBLIC  CLEAR_SCREEN
;-----;
; Questa procedura cancella l'intero schermo.
;-----;
CLEAR_SCREEN  PROC
    PUSH  AX
    PUSH  BX
    PUSH  CX
    PUSH  DX
    XOR   AL,AL           ;Cancella l'intera finestra
    XOR   CX,CX           ;L'angolo superiore sinistro è a (0,0)
    MOV   DH,24           ;La riga inferiore dello schermo è la 24
    MOV   DL,79           ;Il limite destro è la colonna 79
    MOV   BH,7            ;Utilizza l'attributo normale per gli spazi
    MOV   AH,6            ;Richiama la funzione SCROLL-UP
    INT   10h             ;Cancella la finestra
    POP   DX
    POP   CX
    POP   BX
    POP   AX
    RET
CLEAR_SCREEN  ENDP

```

Questa è una procedura in linguaggio assembly abbastanza semplice. Tutto quello che bisogna fare per convertirla in una procedura C, come potete vedere dal listato seguente, è di rimuovere un certo numero di istruzioni. Ecco il nuovo file CLIB.ASM, che sarà utilizzato per contenere tutte le procedure C scritte in linguaggio assembly:

Listato 31-1. Il nuovo file CLIB.ASM

```

.MODEL    SMALL,C

.CODE

;-----;
; Questa procedura cancella l'intero schermo.
;-----;
CLEAR_SCREEN  PROC
    XOR   AL,AL           ;Cancella l'intera finestra
    XOR   CX,CX           ;L'angolo superiore sinistro è a (0,0)
    MOV   DH,24           ;La riga inferiore dello schermo è la 24
    MOV   DL,79           ;Il limite destro è la colonna 79

```

```
MOV    BH,7                ;Utilizza l'attributo normale per gli spazi
MOV    AH,6                ;Richiama la funzione SCROLL-UP
INT    10h                 ;Cancella la finestra
RET
CLEAR_SCREEN    ENDP

END
```

(Se state utilizzando il Turbo Assembler, dovete aggiungere due linee dopo .MODEL, con MASM51 nella prima riga, e QUIRKS sulla seconda). Notate che sono state rimosse tutte le istruzioni PUSH e POP che servivano per salvare e richiamare i registri. Sono state utilizzate queste istruzioni nella programmazione in assembly in modo da non dover tener traccia di quali registri vengono cambiati dalle procedure chiamate. Questo rende la programmazione in linguaggio assembly molto più semplice. Le procedure C, d'altro canto, non devono salvare i registri AX, BX, CX o DX, come vedrete più tardi. E' possibile quindi utilizzare le procedure senza salvare e richiamare i quattro registri.

Nota: Non dovete salvare e cancellare i registri AX, BX, CX, o DX in procedure C scritte in linguaggio assembly. Dovete, invece, salvare e cancellare i registri di segmento SI, DI, BP, se vengono cambiati in una procedura.

Si Possono Cambiare: AX, BX, CX, DX, ES
Si Devono Conservare: SI, DI, BP, SP, CS, DS, SS

Ecco un programma in C che utilizza clear_screen().

Listato 31-2. // file test.c

```
main()
{
    clear_screen();
}
```

Utilizzate i seguenti passaggi per assemblare CLIB.ASM, compilare test.c, e linkare entrambi i file in test.exe:

```
MASM CLIB;
CL -C TEST.C
LINK TEST+CLIB,TEST,TEST/MAP;
```

(Il comando CL -C compila un file senza processarlo con LINK). L'ultima linea è un po' più complicata del solito, perché è stato indicato a Link di creare un file map in

modo da sapere dove trovare `clear_screen()` nel Debug. Anche se `test.exe` è un programma molto semplice, la mappa della memoria (`test.map`) potrebbe essere lunga a causa di alcuni appesantimenti presenti nei programmi in C. Ecco una versione abbreviata di questa mappa che mostra le informazioni che possono interessare:

```

      .
      .
      .
Address          Publics by Name
0054:00EC        STKHQQ
0000:001A        _clear_screen
0054:01D8        _edata
0054:01E0        _end
0054:00DA        _environ
0054:00B3        _errno
0000:01A2        _exit
0000:0010        _main
      .
      .
      .

```

Program entry point at 0000:002A

Come potete vedere, la procedura è chiamata `_clear_screen` invece che `clear_screen`. La maggior parte dei compilatori C mettono un underscore (`_`) prima dei nomi delle procedure per ragioni storiche che sono state ormai dimenticate (i compilatori C mettono underscore anche davanti ai nomi di variabili).

Avrete anche notato che non è stato incluso un `PUBLIC CLEAR_SCREEN` per rendere `CLEAR_SCREEN` disponibile ad altri file. Questo è un'altro cambiamento che l'opzione `"/C"` ha fatto per voi. L'opzione `"/C"` aggiunta a `.MODEL` cambia la direttiva `PROC` in modo che definisca automaticamente ogni procedura come `PUBLIC`. In altre parole, se state scrivendo procedure C in linguaggio assembly (utilizzando `.MODEL SMALL,C`), tutte le procedure saranno dichiarate `PUBLIC` automaticamente.

Caricate `test.exe` in Debug per vedere se MASM ha fatto altri cambiamenti. Utilizzando l'indirizzo trovato nella mappa precedente (1A), ecco il codice per `_clear_screen`:

A>DEBUG TEST.EXE

-U 1A

```

4A8A:001A 32C0          XOR  AL,AL
4A8A:001C 33C9          XOR  CX,CX
4A8A:001E B618          MOV  DH,18
4A8A:0020 B24F          MOV  DL,4F
4A8A:0022 B707          MOV  BH,07

```



```

4A8A:0024 B406          MOV  AH,06
4A8A:0026 CD10          INT  10
4A8A:0028 C3 RET
.
.
.

```

Questo è esattamente quello che avete scritto in CLIB.ASM. In altre parole, il parametro “,C” alla fine della direttiva .MODEL ha solo cambiato il nome della procedura da clear_screen a _clear_screen e l’ha dichiarata pubblica.

PASSARE I PARAMETRI

Fino ad ora avete utilizzato i registri per passare i parametri alle procedure, che funzionano egregiamente fino a quando si hanno meno di sei parametri (che richiedono i sei registri, AX, BX, CX, DX, SI, e DI). I programmi C, invece, utilizzano lo stack per passare i parametri alle procedure; e qui entra in gioco l’estensione .MODEL del MASM 5.1. Il MASM genera automaticamente la maggior parte del codice che serve per lavorare con i parametri passati nello stack.

Per vedere come funziona, convertite alcune procedure in procedure C. Inizierete con una procedura per scrivere una serie di caratteri sullo schermo. Si potrebbe convertire WRITE_STRING ma, dato che WRITE_STRING utilizza un certo numero di altre procedure, scriverete una nuova WRITE_STRING che utilizza il BIOS per scrivere i caratteri sullo schermo. Questa nuova procedura utilizza la funzione 14 di INT 10h per scrivere ogni carattere sullo schermo. Questo non sarà sicuramente veloce, ma è abbastanza semplice e permette di non perdersi in un lungo codice sorgente.

Ecco la versione C di WRITE_STRING che bisogna aggiungere a CLIB.ASM:

Figura 31-3. Aggiungete questa procedura a CLIB.ASM

```

;-----;
; Questa procedura scrive una stringa di caratteri sullo schermo. ;
; La stringa deve terminare con DB 0 ;
; ;
; write_string(string); ;
; char *string; ;
;-----;
WRITE_STRING PROC USES SI, STRING:PTR BYTE
    PUSHF ;Salva il flag di direzione
    CLD
    MOV SI,STRING ;Mette gli indirizzi per LODSB in SI

STRING_LOOP:
    LODSB ;Porta un carattere nel registro al

```

```

OR     AL,AL           ;E' già stato trovato lo 0?
JZ     END_OF_STRING  ;Si, la stringa è finita
MOV    AH,14          ;Chiama la funzione per la scrittura del carattere
XOR    BH,BH          ;Scrive in pagina 0
INT    10h            ;Scrive un carattere sullo schermo
JMP    STRING_LOOP

END_OF_STRING:
        POPF
        RET
WRITE_STRING ENDP

```

La maggior parte del codice dovrebbe esservi familiare dal momento che è stato utilizzato nella versione veloce di `WRITE_STRING`. Una linea, tuttavia, è differente. Notate che sono state aggiunte due informazioni alla fine della direttiva `PROC`.

La prima, `USES SI`, indica al MASM che si sta utilizzando il registro `SI` nelle procedure. Come detto in precedenza, le procedure `C` devono salvare e ripristinare i registri `SI` e `DI` se li modificano. Come vedrete presto, la direttiva `USES SI` indica al MASM di generare del codice per salvare e ripristinare il registro `SI` automaticamente!

La seconda istruzione è utilizzata per passare un parametro al programma. `STRING:PTR BYTE` indica semplicemente che si desidera chiamare il parametro `STRING` che punta (`PTR`) ad un carattere (`BYTE`), che è il primo carattere della stringa. Assegnando un nome a questo parametro, è possibile utilizzare il valore del parametro semplicemente utilizzando il nome, come in `MOV SI,STRING`.

Tutto questo diventerà chiaro quando vedrete il codice generato da MASM. Assemblate la nuova `CLIB.ASM`, e apportate i seguenti cambiamenti a `test.c`:

```

main()
{
    clear_screen();
    write_string("Questa è una stringa!");
}

```

Ricompilate `test.c` (con `cl -c test.c`) e processatelo con il comando `LINK TEST+CLIB,TEST,TEST/MAP,;`

Date un'occhiata al nuovo `map` file e vedrete che `_write_string` è in `33h` (potreste vedere un numero diverso, dal momento che il numero varia a seconda del compilatore utilizzato):

```

0000:0024    _clear_screen
0056:01EA    _edata
0056:01F0    _end
0056:00DA    _environ
0056:00B3    _errno
0000:01C6    _exit
0000:0010    _main
0000:0033    _write_string

```

Ecco il codice generato da MASM per `write_string` (le istruzioni generate da MASM sono in grassetto):

```

-U 33
4A8A:0033    55           PUSH BP
4A8A:0034    8BEC        MOV BP, SP
4A8A:0036    56           PUSH SI
4A8A:0037    9C           PUSHF
4A8A:0038    FC           CLD
4A8A:0039    8B7604      MOV SI, [BP+04]
4A8A:003C    AC           LODSB
4A8A:003D    0AC0        OR AL, AL
4A8A:003F    7408        JZ 0049
4A8A:0041    B40E        MOV AH, 0E
4A8A:0043    32FF        XOR BH, BH
4A8A:0045    CD10        INT 10
4A8A:0047    EBF3        JMP 003C
4A8A:0049    9D           POPF
4A8A:004A    SE         POP SI
4A8A:004B    SD         POP BP
4A8A:004C    C3           RET

```

Come potete vedere, MASM aggiunge un certo numero di istruzioni a quelle che avete scritto. Le istruzioni `PUSH SI` e `POP SI` dovrebbero essere chiare dal momento che è stato detto che MASM salva e ripristina il registro `SI` quando incontra un'istruzione `USES SI`. Per le altre istruzioni servono alcune spiegazioni.

Il registro `BP` è un registro ad uso generale di cui non è stato detto molto. Se date uno sguardo alla tabella degli indirizzamenti nell'Appendice D, noterete che `BP` è leggermente differente dagli altri registri; il segmento di default per `[BP]` è il registro `SS` invece che `DS`. Questo è interessante perché, come detto in precedenza, i programmi C passano i parametri allo stack invece che ai registri. Quindi l'istruzione:

```
MOV SI, [BP+04]
```

leggerà sempre dallo stack, anche se `SS` non è lo stesso di `DS` o `ES`. Siccome è molto conveniente utilizzare il registro `BP` per lavorare con lo stack, le procedure C utilizzano il registro `BP` per accedere ai parametri passati a loro stesse nello stack. Per utilizzare il registro `BP`, bisogna impostarlo al valore corrente di `SP`, e l'istruzione `MOV BP, SP` serve proprio a questo scopo. Ma siccome le procedure C chiamate utilizzano il registro `BP` per accedere ai parametri, bisogna salvare e ripristinare lo stesso registro `BP`. In questo modo l'assemblatore genera automaticamente queste istruzioni (senza i commenti, naturalmente) che permettono di utilizzare il registro `BP` per leggere i parametri dallo stack:

```

PUSH  BP                ;Salva il registro BP corrente
MOV   BP,SP            ;Imposta BP per puntare ai parametri.
.
.
.
POP   BP                ;Ripristina la vecchia versione di BP

```

La Figura 31-1 mostra come sarebbe lo stack per una procedura con due parametri che utilizza il registro SI. La chiamata C, `c_call(param1, param2)`, spinge i parametri nello stack, da destra a sinistra. Spingendo per primo il parametro più a destra, e il più a sinistra come ultimo, il primo parametro sarà sempre vicino all'inizio dello stack, in altre parole, il più vicino a SP.

L'istruzione CALL creata da `c_call(param1, param2)` spinge gli indirizzi di ritorno nello stack, nel punto in cui la procedura prende il controllo. Notate a questo punto che l'istruzione PUSH SI appare *dopo* l'istruzione MOV BP,SP. Dopo aver impostato il valore di BP, siete liberi di cambiare lo stack con le istruzioni PUSH, POP, e

`c_call(parametro1, parametro2)`

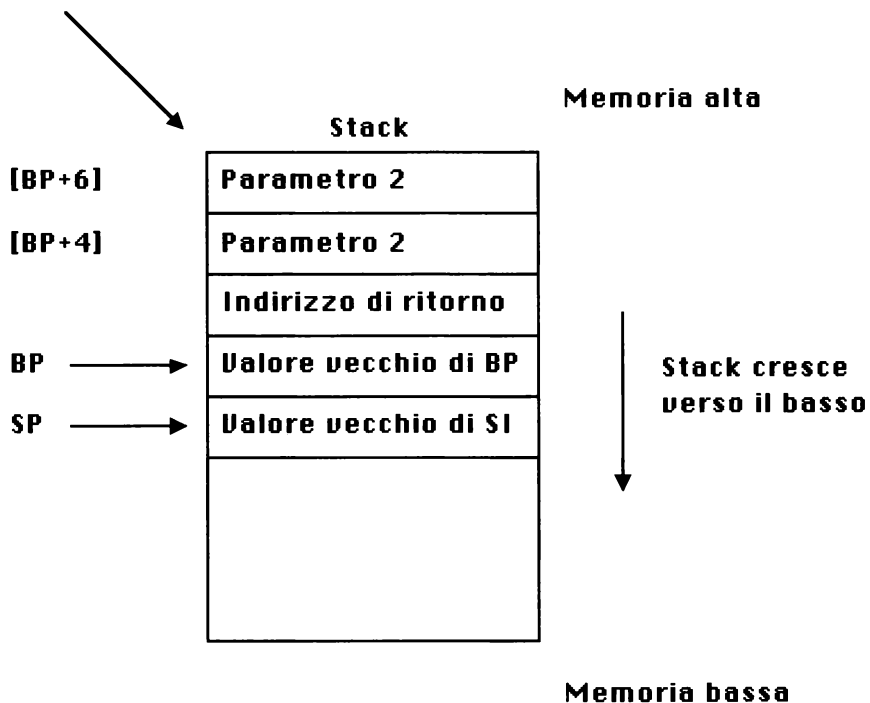


Figura 31-1. Come il C passa i parametri nello stack

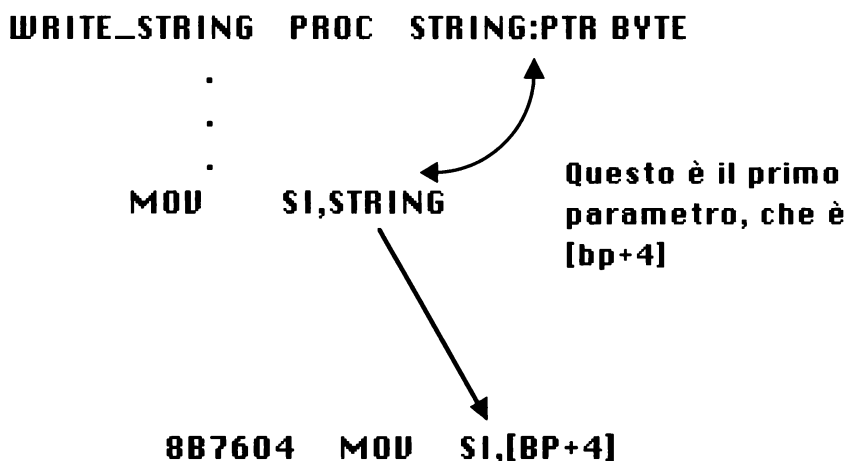


Figura 31-2. L'assembler sa dove trovare i parametri.

chiamando altre procedure. Siccome MASM genera tutte le istruzioni che servono, non dovete preoccuparvi di inserirle nel codice sorgente.

Il primo parametro dovrebbe sempre essere alla stessa distanza da BP, che è 4 per il modello di memoria SMALL. Guardando al listato disassemblato della procedura, noterete che l'assemblatore ha trasformato l'istruzione MOV SI,STRING in MOV SI,[BP+4]. Se avete utilizzato un modello di memoria con procedure FAR, questo sarebbe stato trasformato in MOV SI,[BP+6].

Solo per la cronaca, il C passa i parametri nello stack in ordine opposto rispetto agli altri linguaggi di alto livello. Pascal, BASIC, e FORTRAN, per esempio, spingono per primo il primo parametro nello stack, e l'ultimo per ultimo; questo significa che l'ultimo parametro sarà il più vicino all'inizio dello stack (SP).

Se pensate per un momento, capirete che la distanza da BP al primo parametro dipende dal numero di parametri passati allo stack. Questo non è un problema in Pascal, BASIC, o FORTRAN dove le chiamate alle procedure *devono* avere lo stesso numero di parametri come definito nelle procedure.

Nelle procedure C, in ogni caso, potete passare più parametri rispetto a quelli definiti. La funzione C printf() è un buon esempio. Il numero di parametri che si passa a printf() dipende da quanti argomenti % si hanno nella stringa. Per permettere alle procedure in C di avere un numero variabile di parametri, bisogna spingere i parametri in ordine inverso in modo che l'ultimo sia il più vicino a SP e non dipenda dal numero di parametri inseriti nello stack.

UN ESEMPIO A DUE PARAMETRI

Prima di continuare, ecco un'altra procedura che troverete utile nei vostri programmi in C:

Listato 31-4. Aggiungete questa procedura a *CLIB.ASM*.

```

-----;
; Questa procedura sposta il cursore ;
; ;
; goto_xy(x, y); ;
; int x, y; ;
-----;
GOTO_XY PROC X:WORD, Y:WORD
MOV AH,2 ;Chiamata per SET CURSOR POSITION
MOV BH,0 ;Visualizza la pagina 0
MOV DH,BYTE PTR (Y) ;Numero linea (0..n)
MOV DL,BYTE PTR (X) ;Numero colonna (0..79)
INT 10h ;Sposta il cursore
RET
GOTO_XY ENDP

```

E questi sono i cambiamenti da effettuare a *test.c* per utilizzare `goto_xy()`:

```

main()
{
    clear_screen();
    goto_xy(35, 10);
    write_string("Questa è una stringa!");
}

```

Ci sono due cose interessanti in `goto_xy()`. Come prima cosa notate che la dichiarazione dei due parametri (X e Y) è nello stesso ordine in cui sono stati scritti nella procedura call: `goto_xy(x,y)`. Non sarebbe possibile scrivere questi parametri nello stesso ordine per un linguaggio, come il Pascal, che inserisce i parametri nello stack in ordine differente.

Notate che X e Y sono stati definiti come parole, invece che come byte. E' stato fatto così perché il C (e altri linguaggi di altro livello) non inserisce mai un byte (BYTE) nello stack, ma inserisce solo delle parole (WORD). E per questo c'è una buona ragione: l'istruzione PUSH inserisce solo parole nello stack e non byte. In `goto_xy`, questo non è un problema a meno che non si voglia spostare un byte nei registri DH e DL. L'istruzione:

```
MOV DL,X
```

non funzionerebbe poiché l'assemblatore riporterebbe un errore. Bisogna invece utilizzare `BYTE PTR X` per accedere a `X` come byte. Ma anche questo non funzionerebbe per il modo in cui le estensioni MASM 5.1 sono scritte nell'assemblatore. Quindi le definizioni `X:WORD` e `Y:WORD` nella direttiva `PROC` sono implementate nell'assemblatore come macro. Le macro, che non saranno spiegate in questo libro, sono una modalità per aggiungere delle *caratteristiche* all'assemblatore. I parametri `X` e `Y` sono delle macro, quindi quando scrivete `MOV DL,X,X` questo viene espanso in un testo definito dal MASM:

```
X      ->      WORD PTR [BP+4]
```

Se poi inserite `BYTE PTR`, l'assemblatore non saprà come comportarsi:

```
BYTE PTR X      ->      BYTE PTR WORD PTR [BP+4]
```

E' possibile fissare questo problema aggiungendo delle parentesi a `X` e `Y`, in modo da indicare all'assemblatore che `[BP+4]` fa riferimento ad una parola, ma lo si vuole trattare come byte:

```
BYTE PTR (X)    ->      BYTE PTR (WORD PTR [BP+4])
```

Le parentesi indicano semplicemente all'assemblatore di processare per primo tutto quello che si trova tra parentesi.

FORNIRE I VALORI DELLE FUNZIONI

Oltre a scrivere delle procedure C in assembly, probabilmente vorrete anche scrivere delle funzioni C in assembly; nulla di difficoltoso. Le funzioni C ritornano dei valori nei seguenti registri: i byte in `AL`, le parole in `AX`, e le parole lunghe (due byte) in `DX:AX`, con la parola bassa in `AX`. (Se volete che fornisca dei tipi di tre byte o più lunghi di quattro byte, dovete consultare la *Microsoft Mixed-Language Programming Guide* o *Turbo Assembler User's Guide* per maggiori dettagli).

Nota: Ecco i registri utilizzati per ritornare i valori ai programmi C:

Byte	<code>AL</code>
Parola	<code>AX</code>
Parola Lunga	<code>DX:AX</code>

La procedura seguente, che dovrete aggiungere a `CLIB.ASM`, è una nuova versione di `READ_KEY` che ritorna i codici estesi dei tasti ai programmi C:

Listato 31-5. *Aggiungete questa procedura a CLIB.ASM.*

```

-----;
; Questa procedura legge un tasto dalla tastiera. ;
; ;
; key 0 read_key(); ;
-----;
READ_KEY PROC
    XOR    AH,AH
    INT    16h            ;Legge i caratteri/scan code dalla tastiera
    OR     AL,AL         ;E' un codice esteso?
    JZ     EXTENDED_CODE ;Sì
NOT_EXTENDED:
    XOR    AH,AH         ;Ritorna solo il codice ASCII
    JMP    DONE_READING

EXTENDED_CODE:
    MOV    AL,AH         ;Mette lo scan code in AL
    MOV    AH,1
DONE_READING:
    RET
READ_KEY ENDP

```

Ecco la versione di test.c che cancella lo schermo, visualizza una stringa in prossimità del centro dello schermo, e aspetta la pressione della barra spaziatrice prima di ritornare al DOS:

```

main()
{
    clear_screen();
    goto_xy(35,10);
    write_string("Questa è una stringa!");
    while (read_key() != ' ')
        ;
}

```

SOMMARIO

Se volete scrivere delle procedure per altri linguaggi, dovete consultare la documentazione per il linguaggio utilizzato. Non tutti i compilatori dello stesso linguaggio (come il Pascal) usano le stesse convenzioni. Quindi, anche se MASM e Turbo Assembler supportano le stesse convenzioni, ci possono essere delle differenze se non utilizzate compilatore e assembler della stessa casa produttrice.

Il prossimo capitolo, l'ultimo capitolo tecnico, discute la parte più avanzata di questo libro: scrivere dei programmi residenti in RAM.

DISKLITE, UN PROGRAMMA RESIDENTE IN RAM

In questo capitolo, l'ultimo per quanto riguarda la programmazione, si discuterà un argomento avanzato: scrivere dei programmi residenti in RAM. Nel fare questo utilizzerete molte cose che sono state spiegate in questo libro, e scriverete anche un programma utile.

I PROGRAMMI RESIDENTI IN RAM

I programmi residenti in RAM sono scritti per la maggior parte in linguaggio assembly per poter accedere al BIOS, alla memoria, e per renderli compatti. Il programma Disklite che costruirete in questa sede, per esempio, è lungo solo 247 byte. Siccome questo tipo di programmi rimangono nella memoria del computer fino a quando il computer stesso non viene riavviato, è molto importante che siano dei programmi compatti per poter lasciare il posto ad altre applicazioni.

I programmi residenti in RAM, generalmente, devono lavorare a stretto contatto con il BIOS e con l'hardware della macchina. Disklite, per esempio, controlla le routine del BIOS che leggono e scrivono sui dischi in modo da poter visualizzare sullo schermo la luce che indica l'operatività del drive.

Molti programmatori vogliono vedere la luce del drive durante la compilazione, per poterne seguire il processo. Quando una compilazione dura 30 secondi o un minuto, non c'è molto altro da fare. E' anche utile vedere la luce del drive durante una scrittura o lettura sul disco per capire se c'è un accesso al disco o meno. Ma cosa succede se mettete il computer lontano dal vostro tavolo (o se utilizzate un IBM PS/2 Modello 80)? O se avete un disco fisso su scheda che non ha la luce? In entrambi i casi, Disklite fornisce una 'luce' d'attività sullo schermo che si accende quando c'è un accesso al disco, e indica anche a quale disco si sta accedendo.

INTERCETTARE GLI INTERRUPT

Come detto in precedenza, Disklite visualizza la luce del drive controllando le routine del BIOS che leggono e scrivono sui dischi.

Tutte le operazioni di lettura/scrittura su disco sono eseguite attraverso le routine di INT 13h del BIOS. Gli interrupt, come già illustrato nel Capitolo 11, si servono di una

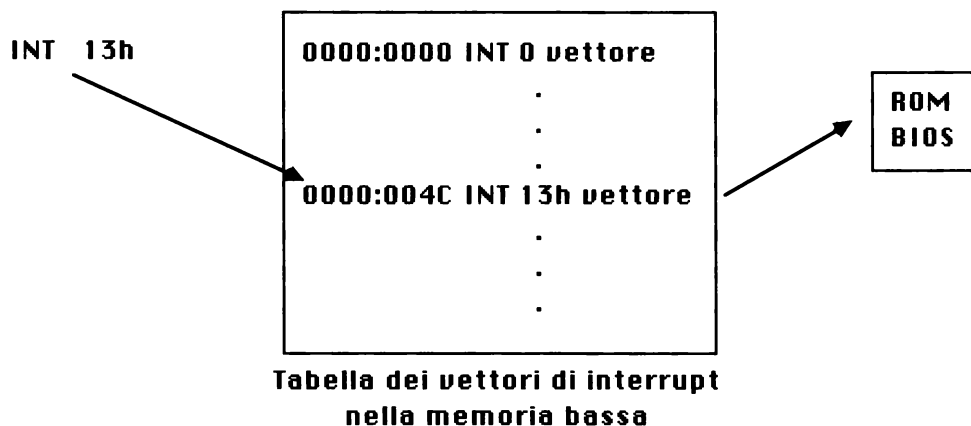


Figura 32-1. *INT 13h utilizza il vettore d'interrupt a 4Ch per determinare l'indirizzo della routine da chiamare.*

tabella di vettori all'inizio della memoria per determinare quale routine chiamare. Ogni vettore di interrupt in questa tabella è lungo due parole, dato che contiene l'indirizzo FAR della routine che elabora l'interrupt.

Quindi l'istruzione INT 13h utilizzerà l'indirizzo 0:4Ch (13h volte 4) come indirizzo della routine che elabora la funzione INT 13h. In altre parole, è possibile cambiare questo indirizzo per puntare alla routine desiderata invece che alla routine del BIOS. Questo è precisamente quello che vogliamo fare.

La Figura 32-1 mostra come INT 13h chiama la routine del BIOS. Ora immaginate di cambiare il vettore di interrupt per puntare alla nuova procedura; in questo modo avete il controllo sull'istruzione INT 13h. Questo è, parzialmente, quello che serviva. Se prendete il controllo totale di INT 13h, dovrete scrivere una routine che svolga l'intero lavoro di INT 13h, e le nuove funzioni che si vogliono aggiungere.

Invece che rimpiazzare ciecamente il vettore di INT 13h, prima lo si deve salvare nel programma. Quindi, si utilizzano le routine del BIOS di INT 13h simulando una chiamata INT alle routine. Simulare una chiamata INT è come fare un'istruzione CALL; bisogna tuttavia salvare i flag nello stack in modo che possano essere ripristinati da un'istruzione IRET (Interrupt RETurn). Tutto quello che bisogna fare, quindi, è salvare l'indirizzo delle routine di INT 13h nella variabile ROM_DISKETTE_INT, in modo da poter passare il controllo alle routine del BIOS INT 13h con questo tipo di istruzioni:

```
PUSHF
CALL ROM_DISKETTE_INT
```

Quando la ROM conclude l'accesso al disco, il controllo tornerà all'utente. Questo significa che è possibile eseguire delle parti di codice sia prima che dopo una chiamata alla funzione del disco nella ROM, il che è esattamente quello che serve per poter visualizzare, e rimuovere, la lettera del drive.

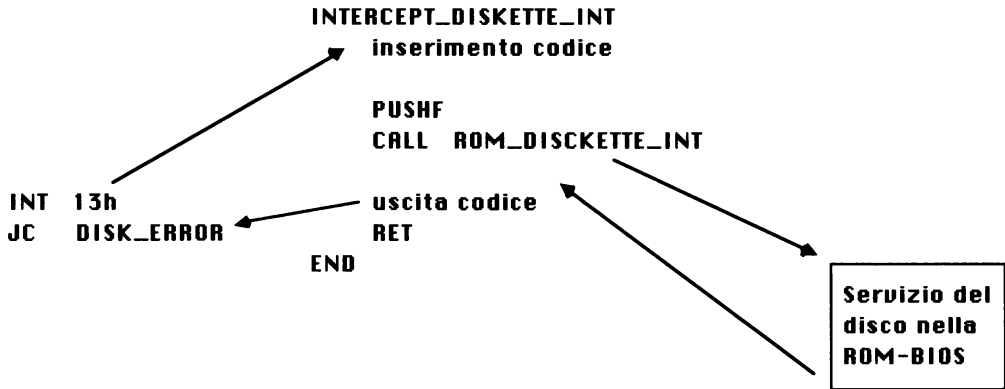


Figura 32-2. Intercettare l'INT 13h

La Figura 32-2 mostra questi passaggi in modo dettagliato.

Nota: La tecnica presentata in questa sezione funziona con la maggior parte delle routine del BIOS. Dato che il DOS non è un sistema operativo multitasking, non potete fare delle chiamate a funzioni DOS con un interrupt a meno che non siate assolutamente sicuri che il DOS non sia a metà di un'operazione. Ci sono alcuni modi per assicurarsi di questo, ma sono abbastanza difficili, e non saranno spiegati in questo libro.

DISKLITE

La maggior parte delle istruzioni di Disklite dovrebbero esservi familiari. Ci sono comunque alcuni dettagli che sono un po' fuori dall'ordinario.

Come prima cosa notate che non vengono salvati o ripristinati i registri nelle procedure di Disklite, mentre vengono invece marcati i registri che vengono alterati. Vengono quindi salvati tutti i registri che possono essere alterati all'inizio di INTERCEPT_DISKETTE_INT; questi si devono salvare solo una volta in modo da mantenere al minimo l'utilizzo dello stack.

Le routine d'interrupt generalmente devono essere scritte in modo che non utilizzino una grande parte dello stack dal momento che possono interferire con qualche altra procedura che utilizza lo stack. Non bisognerà occuparsi dello spazio dello stack in questo programma dal momento che è stato dato uno stack sufficientemente largo. Non è possibile garantire che sia sempre fornito uno stack largo quando si richiede un INT 13h. Per questa ragione, la maggior parte dei programmi residenti in memoria imposta un proprio stack.

Le due procedure GET_DISPLAY_BASE, SAVE_SCREEN e WRITE_TO_SCREEN dovrebbero essere abbastanza chiare alla luce di quanto spiegato nell'ultimo capitolo; SAVE_SCREEN salva i due caratteri nell'angolo in alto a destra, e WRITE_TO_SCREEN è utilizzata sia per visualizzare la lettera del drive utilizzato che per ripristinare i due caratteri che erano sullo schermo prima di visualizzare la lettera del drive in uso.

DISPLAY_DRIVE_LETTER è anch'essa semplice. INT 13h preleva il numero del drive dal registro DL. Per i drive dei floppy disk, DL conterrà 0 per il drive A, 1 per il drive B e così via. Per i dischi fissi, DL parte a 80h. Quindi, per prelevare la lettera del disco fisso correntemente in uso, bisogna sottrarre 80h e quindi aggiungere il numero dei floppy disk drive (questo perché il primo disco fisso appare sempre dopo l'ultimo floppy disk).

Veniamo ora a INIT_VECTORS e GET_NUM_FLOPPIES. INIT_VECTORS mostra i dettagli sull'installazione di una procedura per intercettare un vettore d'interrupt e per mantenere questo programma in memoria dopo essere tornati al DOS. Come prima cosa sarà visualizzato un messaggio dell'autore. Quindi sarà chiamato GET_NUM_FLOPPIES per impostare NUM_FLOPPIES al numero di floppy disk collegati al computer. Quindi sarà letto e impostato il vettore INT 13h con le funzioni 35h e 25h di INT 21h per leggere e impostare i vettori di interrupt.

Notate che le routine di inizializzazione sono state messe alla fine di Disklite. Come è facile capire, queste routine sono utilizzate solo una volta, quando si avvia Disklite. La chiamata alla funzione DOS INT 27h, chiamata *Terminate but Stay Resident*, torna al DOS ma mantiene il programma in memoria. Questa chiamata mantiene un offset in DX per il primo byte che non si desidera mantenere in memoria. Quindi, impostando DX in modo che punti a INIT_VECTORS, si indicherà al DOS di mantenere tutto Disklite in memoria *eccetto* INIT_VECTORS e GET_NUM_FLOPPIES. Potete mettere quanti codici di inizializzazione desiderate in questo punto, in quanto non consumate memoria dopo l'installazione di Disklite; questa è una caratteristica veramente interessante.

Inserite il programma seguente in DISKLITE.ASM. Assemblatelo, processatelo con LINK e convertitelo in un programma .COM (digitando EXE2BIN DISKLITE DISKLITE.COM). Dopo aver eseguito questo programma, una X in inverso (dove X rappresenta la lettera del drive in uso) apparirà nell'angolo in alto a destra quando accederete a un drive. Per verificare il programma, eseguite CHKDSK su tutti i drive.

Listato 32-1. Il Programma DISKLITE.ASM

```

;-----
; Disklite riproduce sullo schermo le luci dei drive nel momento in cui viene ;
; effettuato un accesso al disco. ;
; La differenza sta nel fatto che sullo schermo la luce del drive in uso ;
; appare solo durante delle operazioni di lettura e/o scrittura sull'unità, e ;
; non resta accesa quando il disco gira senza alcuna attività. ;
; ;
; Questo programma intercetta il vettore INT 13h, che è il punto ;
; d'ingresso per le routine BIOS dei drive. Disklite visualizza la ;
; lettera del drive in uso nell'angolo in alto a destra, e ripristina ;
; questa parte di schermo prima di uscire. ;
;-----

```

```

;-----;
; Ecco il punto d'ingresso di DISKLITE. Salta alle routine di          ;
; inizializzazione che sono alla fine del programma e possono essere  ;
; rilasciate dalla memoria del computer dopo essere state utilizzate.  ;
;-----;
CODE_SEG SEGMENT
    ASSUME CS:CODE_SEG, DS:CODE_SEG
        ORG    100h                ;Riservato per il DOS Program Segment Prefix
BEGIN:   JMP    INIT_VECTORS
AUTHOR_STRING DB    "Disklite Installato, by John Socha"
        DB    0Dh, 0Ah, '$'

ROM_DISKETTE_INT DD    ?

DISPLAY_BASE DW    ?
OLD_DISPLAY_CHARS DB    4 DUP (?)
DISPLAY_CHARS DB    'A', 70h, ':', 70h
NUM_FLOPPIES DB    ?                ;Numero di floppy drive

UPPER_LEFT EQU    (80 - 2) * 2

;-----;
; Questa procedura intercetta le chiamate al vettore di I/O dei drive  ;
; e svolge le seguenti operazioni:                                     ;
; ;                                                                     ;
; 1. Controlla se lo schermo è in modalità testo a 80 colonne.      ;
;    Disklite non scriverà se lo schermo non è in questa modalità.  ;
; 2. Visualizza la lettera del drive in uso, "A:" per esempio,      ;
;    nell'angolo in alto a destra.                                   ;
; 3. Chiama la vecchia routine BIOS per eseguire il lavoro.        ;
; 4. Ripristina i due caratteri nell'angolo in alto a destra dello  ;
;    dello schermo.                                                 ;
;-----;
INTERCEPT_DISKETTE_INT PROC FAR
    Assume CS:CODE_SEG, DS:Nothing
    PUSHF                ;Salva i vecchi flag
    PUSH AX
    PUSH SI
    PUSH DI
    PUSH DS
    PUSH ES
    CALL GET_DISPLAY_BASE ;Calcola la base dello schermo
    CALL SAVE_SCREEN     ;Salva due caratteri in alto a destra
    CALL DISPLAY_DRIVE_LETTER ;Visualizza la lettera del drive
    POP ES
    POP DS
    POP DI
    POP SI
    POP AX
    POPF                ;Ripristina i vecchi flag

    PUSHF                ;Simula una chiamata INT

```

```

CALL ROM_DISKETTE_INT          ; alla vecchia routine BIOS

PUSHF
PUSH AX
PUSH SI
PUSH DI
PUSH DS
PUSH ES
LEA SI,OLD_DISPLAY_CHARS      ;Punta alla vecchia immagine dello schermo
CALL WRITE_TO_SCREEN          ;Ripristina i due caratteri sullo schermo
POP ES
POP DS
POP DI
POP SI
POP AX
POPF                            ;Ripristina i vecchi flag
RET 2                            ;Lascia il flag di stato intatto
INTERCEPT_DISKETTE_INT      ENDP

;-----;
; Questa procedura calcola l'indirizzo di segmento per l'adattatore che ;
; si sta utilizzando. ;
; ;
; Distrugge:      AX ;
;-----;
GET_DISPLAY_BASE PROC          NEAR
    Assume CS:CODE_SEG, DS:Nothing
    INT 11h                    ;L'equipaggiamento corrente
    AND AX,30h                 ;Isola il display flag
    CMP AX,30h                 ;E' un display monocromatico?
    MOV AX,0B800h              ;Si imposta per un display colore/grafico
    JNE DONE_GET_BASE
    MOV AX,0B000h              ;Si imposta per un display monocromatico
DONE_GET_BASE:
    MOV DISPLAY_BASE,AX        ;Salva il display base
    RET
GET_DISPLAY_BASE ENDP

;-----;
; Questa procedura salva i due caratteri nell'angolo in alto a destra ;
; dello schermo, in modo da poterli utilizzare in seguito. ;
; ;
; Distrugge:      AX, SI, DI, DS, ES ;
;-----;
SAVE_SCREEN PROC              NEAR
    Assume CS:CODE_SEG, DS:Nothing
    MOV SI,UPPER_LEFT          ;Legge i caratteri dallo schermo
    LEA DI,OLD_DISPLAY_CHARS    ;Scrive i caratteri nella memoria locale
    MOV AX,DISPLAY_BASE
    MOV DS,AX
    MOV AX,CS                  ;Punta ai dati locali

```

```

        MOV     ES,AX
        CLD
        MOVSW                      ;Muove i due caratteri
        MOVSW

RET
SAVE_SCREEN      ENDP

;-----;
; Questa procedura visualizza la lettera del drive nell'angolo in alto ;
; a destra dello schermo ;
; ;
; Distrugge:      AX, SI ;
;-----;
DISPLAY_DRIVE_LETTER      PROC      NEAR
        Assume CS:CODE_SEG, DS:Nothing
        MOV     AL,DL                ;Il numero del drive
        CMP     AL,80h               ;E' un disco fisso?
        JB     DISPLAY_LETTER       ;No, quindi continua
        SUB     AL,80h               ;Converte nel numero di disco fisso
        ADD     AL,NUM_FLOPPIES      ;Converte nel numero di disco corretto
DISPLAY_LETTER:
        ADD     AL,'A'               ;Lo converte in una lettera di drive
        LEA    SI,DISPLAY_CHARS
        MOV     CS:[SI],AL          ;Salva questo carattere
        CALL   WRITE_TO_SCREEN
        RET
DISPLAY_DRIVE_LETTER      ENDP

;-----;
; Questa procedura scrive due caratteri nell'angolo in alto a destra. ;
; ;
; Inserimento:    CS:SI Immagine dei due caratteri ;
; Distrugge:      AX, SI, DI, DS, ES ;
;-----;
WRITE_TO_SCREEN PROC      NEAR
        Assume CS:CODE_SEG, DS:Nothing
        MOV     DI,UPPER_LEFT        ;Scrive i caratteri sullo schermo
        MOV     AX,DISPLAY_BASE      ;L'indirizzo del segmento dello schermo
        MOV     ES,AX
        MOV     AX,CS                ;Punta ai dati locali
        MOV     DS,AX
        CLD
        MOVSW                      ;Muove due caratteri
        MOVSW
        RET
WRITE_TO_SCREEN ENDP

;-----;
; Questa procedura incatena Disklite sul vettore di I/O in modo da ;
; poter monitorare l'attività del disco. ;
;-----;
INIT_VECTORS      PROC      NEAR

```

```

Assume CS:CODE_SEG, DS:Nothing
LEA  DX,AUTHOR_STRING      ;La stringa dell'autore
MOV  AH,9                  ;Visualizza questa stringa
INT  21h

CALL  GET_NUM_FLOPPIES     ;Controlla quanti floppy sono installati

MOV  AH,35h                ;Chiede un vettore di interrupt
MOV  AL,13h                ;Prende il vettore per INT 13h
INT  21h                   ;Mette il vettore in ES:BX
MOV  Word Ptr ROM_DISKETTE_INT,BX
MOV  Word Ptr ROM_DISKETTE_INT[2],ES

MOV  AH,15h                ;Chiede di impostare un vettore di interrupt
MOV  AL,13h                ;Imposta il vettore INT 13h a DS:DX
MOV  DX,Offset INTERCEPT_DISKETTE_INT
INT  21h                   ;Imposta INT 13h per puntare alla procedura

MOV  DX,Offset INIT_VECTORS ;Fine della parte residente
INT  27h                   ;Termina ma resta residente
INIT_VECTORS  ENDP

;-----;
; Questa procedura determina quanti drive logici sono presenti nel ;
; sistema. La lettera del drive successivo sarà utilizzata per ;
; il disco fisso. ;
;-----;
GET_NUM_FLOPPIES PROC      NEAR
    Assume CS:CODE_SEG, DS:Nothing
    INT  11h
    MOV  CL,6
    SHR  AX,CL
    AND  AL,3
    INC  AL                  ;Fornisce 0 per un floppy
    CMP  AL,1                ;E' un sistema ad un floppy?
    JA  DONE_GET_FLOPPIES   ;No, è il numero corretto

    MOV  AL,2                ;Sì, ci sono 2 drive logici
DONE_GET_FLOPPIES:
    MOV  NUM_FLOPPIES,AL    ;Salva questo numero
    RET
GET_NUM_FLOPPIES ENDP

CODE_SEG  ENDS

END      BEGIN

```


GUIDA AL DISCO

Il disco che accompagna questo libro contiene la maggior parte degli esempi che sono stati mostrati nei capitoli precedenti, oltre ad una versione avanzata del programma. I file sono divisi in due gruppi: gli esempi dei capitoli e la versione avanzata del programma Dskpatch. Questa appendice spiega cosa si trova sul disco.

GLI ESEMPI DEI CAPITOLI

Tutti gli esempi sono tratti dai Capitoli compresi tra 9 e 27, dal Capitolo 30 e dal 32. Gli esempi nei capitoli precedenti sono sufficientemente corti per poterli digitare da soli. A partire dal Capitolo 9 inizia la costruzione di Dskpatch, che è composto da nove file.

Dal momento che questi file sono stati cambiati e perfezionati di capitolo in capitolo, sul disco non c'è stato spazio a sufficienza per salvare tutti gli esempi. Quindi sul disco troverete gli esempi nel modo in cui appaiono alla fine del capitolo.

La tabella seguente mostra quando (in che capitolo) sono state apportate delle modifiche ai file. Se volete essere sicuri che il file che state utilizzando sia il più aggiornato, o se non volete digitare il file stesso, fate riferimento a questa tabella per trovare i nomi dei nuovi file.

Questa è la lista completa dei file contenuti nel disco che accompagna il libro (non include la versione avanzata di Dskpatch):

VIDEO_9.ASM	VIDEO_16.ASM	DISP_S19.ASM	KBD_IO24.ASM
VIDEO_10.ASM	DISK_I16.ASM	KBD_IO19.ASM	DISPAT25.ASM
VIDEO_13.ASM	DSKPAT17.ASM	DISK_I19.ASM	DISPAT26.ASM
TEST13.ASM	DISP_S17.ASM	DISP_S21.ASM	DISK_I26.ASM
DISP_S14.ASM	CURSOR17.ASM	PHANTO21.ASM	PHANTO27.ASM
CURSOR14.ASM	VIDEO_17.ASM	VIDEO_21.ASM	DSKPAT30.ASM
VIDEO_14.ASM	DISK_I17.ASM	DISPAT22.ASM	KBD_IO30.ASM
DISP_S15.ASM	DISP_S18.ASM	EDITOR22.ASM	CURSOR30.ASM
DISK_I15.ASM	CURSOR18.ASM	PHANTO22.ASM	VIDEO_30.ASM
DISP_S16.ASM	VIDEO_18.ASM	KBD_IO23.ASM	CLIB.ASM
DSKPAT19.ASM	DISPAT19.ASM	TEST23.ASM	DISKLITE.ASM

UNA VERSIONE AVANZATA DI DSKPATCH

Come è già stato detto, il disco contiene qualcosa di più degli esempi riportati in questo libro. Dskpatch non è stato ultimato con la fine del Capitolo 27, e ci sono parecchie cose che bisognerebbe aggiungere. Il disco contiene una versione quasi finita. Ecco una rapida panoramica su cosa troverete.

Come già spiegato, Dskpatch può leggere il settore precedente o successivo. Se volete leggere il settore 576, dovete premere il tasto F4 per 575 volte. Un po' troppo! E se desiderate vedere un settore che appartiene ad un determinato file? La versione su disco di Dskpatch è in grado di leggere sia settori assoluti che settori appartenenti ad un determinato file.

La versione avanzata di Dskpatch ha subito troppi cambiamenti per descriverli in dettaglio in questa sede; questa nuova versione è sempre costituita da nove file, che troverete sul disco:

DSKPATCH.ASM	DISPATCH.ASM	DISP_SEC.ASM	KBD_IO.ASM
CURSOR.ASM	EDITOR.ASM	PHANTOM.ASM	VIDEO_IO.ASM
DISK_IO.ASM	DSKPATCH.COM		

Troverete anche una versione .COM assemblata e linkata pronta per essere eseguita. Quando avvierete Dskpatch, vedrete che sono stati fatti parecchi cambiamenti solo guardando lo schermo. La versione avanzata utilizza otto tasti funzione e mostra una riga in fondo allo schermo con la descrizione dei tasti funzione.

Ecco una descrizione dei tasti funzione:

- F2 E' già stato discusso in questo libro. Premete Shift-F2 per scrivere il settore sul disco.
- F3, F4 F3 legge il settore precedente, mentre F4 legge il settore successivo.
- F5 Cambia il drive selezionato. Premete F5 e digitate una lettera (senza i due punti - :), o inserite il numero del drive, come 0. Quando premete il tasto Invio, Dskpatch si sposterà sulla nuova unità e leggerà il settore. E' possibile cambiare Dskpatch in modo che non legga il nuovo settore quando cambia drive.
- F6 Cambia il numero del settore. Premete F6 e digitate il nuovo numero in decimale. Dskpatch leggerà il settore.
- F7 Porta Dskpatch in modalità file. Digitate il nome del file e Dskpatch leggerà un settore di quel file. A questo punto con F3 e F4 potete far scorrere i settori. F5 vi riporta in modalità normale.
- F8 Richiede l'offset all'interno del file. E' come F4 eccetto il fatto che legge i settori del file specificato. Se digitate un offset di 3, Dskpatch leggerà il quarto settore.
- F10 Esce da Dskpatch. Se premete questo tasto accidentalmente, vi troverete al DOS, e perderete tutti i cambiamenti fatti all'ultimo settore. Potete cambiare il programma in modo che chieda conferma prima di uscire.

Altri cambiamenti non sono così evidenti come i precedenti. Per esempio, la versione finale di Dskpatch permette di scorrere una riga alla volta. Quindi, se spostate il cursore sulla riga in fondo allo schermo e premete il tasto per lo spostamento del cursore verso il basso, Dskpatch scorrerà di una sola riga.

Sono inoltre stati aggiunti nuovi tasti:

- Home sposta il cursore all'inizio del mezzo settore e fa scorrere la visualizzazione in modo da poter vedere il mezzo settore.
- End sposta il cursore fantasma in basso a destra del mezzo settore visualizzato, e fa scorrere la visualizzazione in modo da poter vedere la seconda metà del settore.
- PgUp fa scorrere la visualizzazione di quattro righe. Questa è una caratteristica interessante quando si desidera spostarsi all'interno del settore. Se premete PgUp quattro volte, vedrete l'ultima metà del settore.
- PgDn fa scorrere la visualizzazione di quattro righe in direzione opposta a PgUp.

Se desiderate, potete cambiare Dskpatch secondo le vostre necessità. Ecco perché il disco contiene tutti i file sorgente della versione avanzata di Dskpatch. Per esempio potreste perfezionare la capacità di elaborare gli errori. Come sapete, se premete F4 potreste arrivare alla fine del disco o del file senza che Dskpatch riporti il contatore all'ultimo settore del file o del disco. Se lo desiderate potete risolvere questo piccolo problema.

Oppure potreste voler velocizzare la scrittura a video. Per fare questo dovrete riscrivere alcune procedure come `WRITE_CHAR` e `WRITE_ATTRIBUTE_N_TIMES`, in modo che scrivano direttamente nella memoria video. Attualmente utilizzano le routine del BIOS che, come sapete, sono molto lente. Se siete veramente ambiziosi, provate a scrivere le routine per la scrittura nella memoria video.

Buona fortuna.

Chapter Number	DSKPATCH	DISPATCH	DISP_SEC	KBD_IO	CURSOR	EDITOR	PHANTOM	VIDEO_IO	DISK_IO	TEST
9								VIDEO_9.ASM		
10								VIDEO_10.ASM		
13								VIDEO_13.ASM		TEST13.ASM
14			DISP_S14.ASM		CURSOR14.ASM			VIDEO_14.ASM		
15			DISP_S15.ASM						DISK_I15.ASM	
16			DISP_S16.ASM					VIDEO_16.ASM	DISK_I16.ASM	
17	DSKPAT17.ASM		DISP_S17.ASM		CURSOR17.ASM			VIDEO_17.ASM	DISK_I17.ASM	
18			DISP_18.ASM		CURSOR18.ASM			VIDEO_18.ASM		
19	DSKPAT19.ASM	DISPAT19.ASM	DISP_19.ASM	KBD_IO19.ASM					DISK_I19.ASM	
21			DISP_S21.ASM				PHANT021.ASM	VIDEO_21.ASM		
22		DISPAT22.ASM				EDITOR22.ASM	PHANT022.ASM			
23				KBD_IO23.ASM						TEST23.ASM
24				KBD_IO24.ASM						
25		DISPAT25.ASM								
26		DISPAT26.ASM							DISK_I26.ASM	
27							PHANT027.ASM			
30	DSKPAT30.ASM			KBD_IO30.ASM	CURSOR30.ASM			VIDEO_30.ASM		

Disco A Settore 0

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	0123456789ABCDEF
00	EB	34	90	49	42	4D	20	20	33	2E	33	00	02	04	01	00	04ÉIBM 3.3
10	02	00	02	EF	A9	F8	2B	00	11	00	00	00	11	00	00	00	n-°+ < . <
20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
30	00	00	00	00	01	00	FA	33	C0	0E	D0	BC	00	7C	16	07	-3LÄL !
40	BB	78	00	36	C5	37	1E	56	16	53	BF	2B	7C	B9	0B	00	x 6+7AU S1+i
50	FC	AC	26	00	3D	00	74	03	26	8A	05	AA	8A	C4	E2	F1	%&C= t &è -è-Γ±
60	06	1F	09	47	02	C7	07	2B	7C	FB	CD	13	72	67	A0	10	vèG +iJ= rgá>
70	7C	98	F7	26	16	7C	03	06	1C	7C	03	06	0E	7C	A3	3F	iù7!i =&<iYA !
80	7C	A3	37	7C	B8	20	00	F7	26	11	7C	0B	1E	0B	7C	03	!ú7!i =&<iYA !
90	C3	48	F7	F3	01	06	37	7C	BB	00	05	A1	3F	7C	E8	9F	H=ç 7i i?iøf
A0	00	B8	01	02	E8	B3	00	72	19	0B	FB	B9	0B	00	BE	D9	7 rviJ dJ
B0	7D	F3	A6	75	0D	0D	7F	20	BE	E4	7D	B9	0B	00	F3	A6	}ç= u ià dJ}- ç=
C0	74	18	BE	77	7D	E8	6A	00	32	E4	CD	16	5E	1F	8F	04	t^Jw}øj 2Σ= ^vA
D0	8F	44	02	CD	19	BE	C4	7D	EB	EB	A1	1C	05	33	D2	F7	âD =v-}øøiL 3T=
E0	36	0B	7C	FE	C0	A2	3C	7C	A1	37	7C	A3	3D	7C	BB	00	6 !uL6<i?iú=i
F0	07	A1	37	7C	E8	49	00	A1	18	7C	2A	06	3B	7C	40	38	i?iøi i^!* :iø8

Premere un tasto funzione o introdurre carattere o byte esadecimale: _

1 Salva 2Prec. 3Succ. 4Drive 5Sett. 6File 7Offset 8 9 Uscita

Figura A-1. La versione finale di Dskpatch

LISTATI DI DSKPATCH

DESCRIZIONE DELLE PROCEDURE

Questa appendice contiene la versione finale di Dskpatch. Se scriverete dei programmi per un uso personale, in questa appendice troverete delle procedure che vi potranno essere molto utili. E' stata inclusa una breve descrizione di ogni procedura per aiutarvi ad individuarle.

CURSOR.ASM

CLEAR_SCREEN Come il comando BASIC CLS; cancella lo schermo

CLEAR_TO_END_OF_LINE Cancella tutti i caratteri dalla posizione del cursore alla fine della linea corrente

CURSOR_RIGHT Sposta il cursore di una posizione verso destra, senza sovrascrivere il vecchio carattere.

GOTO_XY Molto simile al comando BASIC LOCATE; sposta il cursore sullo schermo.

SEND_CRLF Invia un paio di carriage-return/line-feed sullo schermo. Questa procedura sposta semplicemente il cursore all'inizio della riga successiva.

UPDATE_READ_CURSOR Sposta il cursore reale alla posizione del cursore virtuale.

UPDATE_VIRTUAL_CURSOR Sposta il cursore virtuale alla posizione del cursore reale.

DISK_IO.ASM

NEXT_SECTOR Aggiunge uno al numero di settore corrente, quindi legge quel settore in memoria e riscrive lo schermo di Dskpatch.

PREVIOUS_SECTOR Legge il settore precedente. Questa procedura sottrae uno dal vecchio numero di settore (CURRENT_SECTOR_NO) e legge il nuovo settore nella variabile di memoria SECTOR. Riscrive anche il video.

READ_SECTOR Legge un settore (512 byte) dal disco nel buffer di memoria, SECTOR.

WRITE_SECTOR Scrive un settore (512 byte) dal buffer di memoria, SECTOR, al disco.

DISPATCH.ASM

DISPATCHER La parte centrale del programma, legge i caratteri dalla tastiera e chiama le altre procedure per svolgere i vari lavori. Aggiungete qualsiasi nuovo comando a DISPATCH_TABLE in questo file.

DISP_SEC.ASM

DISP_HALF_SECTOR Visualizza tutti i caratteri in esadecimale e ASCII per la prima metà del settore, chiamando DISP_LINE per 16 volte.

DISP_LINE Visualizza una linea. DISP_HALF_SECTOR chiama questa procedura 16 volte per visualizzare una metà del settore.

INIT_SEC_DISP Inizializza la visualizzazione del settore in Dskpatch. Questa procedura riscrive lo schermo, con i bordi, i numeri esadecimali, ma non riscrive l'intestazione o il prompt.

WRITE_HEADER Scrive l'intestazione sulla prima linea dello schermo. Questa procedura visualizza il numero del drive e il numero del settore visualizzato.

WRITE_PROMPT_LINE Scrive una stringa sulla riga prompt, quindi cancella il resto della riga per rimuovere dei vecchi caratteri.

WRITE_TOP_HEX_NUMBERS Scrive la riga di numeri esadecimali nella parte alta della schermata del mezzo settore. Non è utile per altri compiti.

DSKPATCH.ASM

DISK_PATCH Il programma principale di Dskpatch. DISK_PATCH chiama semplicemente un certo numero di altre procedure, che svolgono tutto il lavoro.

Include inoltre la maggior parte delle definizioni per le variabili che sono utilizzate dal programma.

EDITOR.ASM

EDIT_BYTE *modifica* un byte nella parte di settore visualizzata, cambiando il byte sia in memoria (SECTOR) che sullo schermo. Dskpatch utilizza questa procedura per cambiare i byte nei settori.

WRITE_TO_MEMORY Chiamata da EDIT_BYTE per cambiare un singolo byte in SECTOR. Questa procedura cambia il byte puntato dal cursore fantasma.

KBD_IO.ASM

BACK_SPACE Utilizzato da READ_STRING per cancellare un carattere, sia dallo schermo che dal buffer di tastiera, quando si preme il tasto Backspace.

CONVERT_HEX_DIGIT Converte un singolo carattere ASCII nell'equivalente esadecimale. Per esempio, la procedura converte la lettera A nel numero esadecimale 0AH. **Nota:** CONVERT_HEX_DIGIT funziona solo con le lettere maiuscole.

HEX_TO_BYTE Converte una stringa di due caratteri dal corrispondente valore esadecimale, come A5, in un singolo byte con quel valore. HEX_TO_BYTE si aspetta che i due caratteri siano in lettere maiuscole.

READ_BYTE Utilizza READ_STRING per leggere una stringa di caratteri. Questa procedura riporta i tasti speciali, un singolo carattere, o un byte esadecimale se avete digitato un numero esadecimale a due cifre.

READ_DECIMAL Legge un numero decimale senza segno dalla tastiera, utilizzando READ_STRING per leggere i caratteri. READ_DECIMAL può leggere i numeri tra 0 e 65535.

READ_KEY Legge un singolo tasto dalla tastiera e ritorna un numero tra 0 e 255 per i caratteri ordinari, e oltre 100h per i tasti speciali.

READ_STRING Legge una stringa di caratteri dalla tastiera. Questa procedura legge anche i tasti speciali, mentre la funzione DOS READ_STRING non è in grado di farlo.

STRING_TO_UPPER Una procedura di uso generale, per convertire le stringhe in lettere maiuscole.

PHANTOM.ASM

ERASE_PHANTOM Rimuove i due cursori fantasma dallo schermo riportando l'attributo del carattere a normale (7) per tutti i caratteri sotto ai cursori fantasma.

MOV_TO_ASCII_POSITION Sposta il cursore reale all'inizio del cursore fantasma nella finestra ASCII.

MOV_TO_HEX_POSITION Sposta il cursore reale all'inizio del cursore fantasma nella finestra esadecimale.

PHANTOM_DOWN Sposta il cursore fantasma verso il basso e fa scorrere i dati se provate ad andare oltre la sedicesima riga.

PHANTOM_LEFT Sposta il cursore a sinistra di un dato, ma non oltre il bordo sinistro della finestra.

PHANTOM_RIGHT Sposta il cursore a destra di un dato, ma non oltre il bordo destro della finestra.

PHANTOM_UP Sposta il cursore vero l'alto, e fa scorrere i dati se provate ad andare oltre il limite superiore.

RESTORE_REAL_CURSOR Riporta il cursore nella posizione registrata da **SAVE_REAL_CURSOR**.

SAVE_REAL_CURSOR Salva la posizione del cursore reale in due variabili. Chiamate questa procedura prima di spostare il cursore reale se desiderate ripristinare la posizione del cursore reale dopo i cambiamenti alla visualizzazione.

SCROLL_DOWN Invece che scorrere la visualizzazione del mezzo schermo, visualizza la prima metà del settore. Troverete una versione avanzata di **SCROLL_DOWN** sul disco che accompagna il libro. La versione avanzata fa scorrere la visualizzazione solo di una riga.

SCROLL_UP Chiamata da **PHANTOM_DOWN** quando provate a spostare il cursore fantasma oltre la fine della finestra. La versione contenuta in questo libro non fa scorrere lo schermo: scrive la seconda metà del settore. Sul disco troverete una versione più avanzata di **SCROLL_UP** e **SCROLL_DOWN** per far scorrere una riga alla volta.

WRITE_PHANTOM Disegna il cursore fantasma sul video: uno nella finestra esadecimale, l'altro nella finestra ASCII. Questa procedura cambia semplicemente l'attributo del carattere in 70H per utilizzare i caratteri neri su sfondo bianco.

VIDEO_IO.ASM

Contiene la maggior parte delle procedure di uso generale che potete utilizzare nei vostri programmi.

INIT_WRITE_CHAR Chiamate questa procedura prima di chiamare qualsiasi altra procedura di questo file. Inizializza i dati utilizzati dalle routine che scrivono direttamente nella memoria video.

WRITE_ATTRIBUTE_N_TIMES Una procedura che serve per cambiare gli attributi ad un gruppo di N caratteri. **WRITE_PHANTOM** utilizza questa procedura per cancellare i cursori fantasma, e **ERASE_PHANTOM** la utilizza per rimuovere i cursori fantasma.

WRITE_CHAR Scrive un carattere sullo schermo. Siccome utilizza le routine del BIOS, questa procedura non aggiunge dei significati speciali ai caratteri stessi. Quindi, un carriage-return apparirà sullo schermo come una nota musicale (il carattere 0DH). Chiamate **SEND_CRLF** se volete spostare il cursore all'inizio della riga successiva.

WRITE_CHAR_N_TIMES Scrive N copie di un carattere sullo schermo. Questa procedura è utile per disegnare delle linee sullo schermo.

WRITE_DECIMAL Scrive una parola sullo schermo come un numero decimale senza segno compreso tra 0 e 65535.

WRITE_HEX Prende un numero di un byte, e lo scrive sullo schermo come un numero esadecimale a due cifre.

WRITE_HEX_DIGIT Scrive un numero esadecimale di una cifra sullo schermo. Questa procedura converte un semi-byte (nibble - 4 bit) in un carattere ASCII e lo scrive sullo schermo.

WRITE_PATTERN Disegna una cornice intorno al settore visualizzato. Potete utilizzare **WRITE_PATTERN** per disegnare dei pattern arbitrari sullo schermo.

WRITE_STRING Una procedura veramente utile con la quale potete scrivere le stringhe sullo schermo. L'ultimo carattere della stringa deve essere un byte zero.

LISTATI DEL PROGRAMMA PER LE PROCEDURE DI DSKPATCH

DSKPATCH MAKE FILE

Ecco il Makefile che potete utilizzare con l'utility Make della Microsoft per creare automaticamente Dskpatch.

```
DSKPATCH.OBJ:  DSKPATCH.ASM
               MASM DSKPATCH;

DISK_IO.OBJ:   DISK_IO.ASM
               MASM DISK_IO;

DISP_SEC.OBJ:  DISP_SEC.ASM
               MASM DISP_SEC

VIDEO_IO.OBJ:  VIDEO_IO.ASM
               MASM VIDEO_IO;

CURSOR.OBJ:    CURSOR.ASM
               MASM CURSOR;

DISPATCH.OBJ: DISPATCH.ASM
               MASM DISPATCH;

KBD_IO.OBJ:    KBD_IO.ASM
               MASM KBD_IO;

PHANTOM.OBJ:   PHANTOM.ASM
               MASM PHANTOM;

EDITOR.OBJ:    EDITOR.ASM
               MASM EDITOR;

DSKPATCH.EXE:  DSKPATCH.OBJ DISK_IO.OBJ DISP_SEC.OBJ VIDEO_IO.OBJ CURSOR.OBJ
               DISPATCH.OBJ KBD_IO.OBJ PHANTOM.OBJ EDITOR.OBJ
               LINK @LINKINFO
```

DSKPATCH LINKINFO FILE

E questo è il file linkinfo:

```
dskpatch disk_io disp_sec video_io cursor +
dskpatch kbd_io phantom editor
dskpatch
dskpatch /MAP;
```

CURSOR.ASM

```

CR      EQU    13                ;Carriage Return
LF      EQU    01                ;Line feed

.MODEL  SMALL
.CODE

        PUBLIC CLEAR_SCREEN
;-----;
; Questa procedura cancella l'intero schermo.
;-----;
CLEAR_SCREEN  PROC
        PUSH  AX
        PUSH  BX
        PUSH  CX
        PUSH  DX
        XOR   AL,AL                ;Cancella intera finestra
        XOR   CX,CX                ;L'angolo superiore sinistro è a (0,0)
        MOV  DH,24                ;La riga inferiore dello schermo è la 24
        MOV  DL,79                ;Il limite destro è la colonna 79
        MOV  BH,7                 ;Utilizza l'attributo normale per gli spazi
        MOV  AH,6                 ;Richiama la funzione SCROLL-UP
        INT  10h                 ;Cancella la finestra
        POP  DX
        POP  CX
        POP  BX
        POP  AX
        RET
CLEAR_SCREEN  ENDP

        PUBLIC GOTO_XY
.DATA
        EXTRN SCREEN_PTR:WORD
        EXTRN SCREEN_X:BYTE, SCREEN_Y:BYTE
.CODE
;-----;
; Questa procedura sposta il cursore
;-----;
;      DH      Riga (Y)
;      DL      Colonna (X)
;-----;
GOTO_XY  PROC
        PUSH  AX
        PUSH  BX
        MOV  BH,0                 ;Visualizza pagina 0
        MOV  AH,2                 ;Richiama SET CURSOR POSITION
        INT  10h                 ;Lascia agire la ROM BIOS

        MOV  AL,DH                ;Prende il numero di riga
        MOV  BL,80                ;Moltiplica per 80 caratteri per linea

```

```

        MUL    BL                ;AX = riga * 80
        ADD    AL,DL             ;Aggiunge la colonna
        ADC    AH,0              ;AX = riga * 80 + colonna
        SHL    AX,1
        MOV    SCREEN_PTR,AX     ;Salva l'offset del cursore
        MOV    SCREEN_X,DL       ;Salva la posizione del cursore
        MOV    SCREEN_Y,DH

        POP    BX
        POP    AX
        RET

GOTO_XY      ENDP

        PUBLIC  CURSOR_RIGHT

.DATA
        EXTRN  SCREEN_PTR:WORD   ;Punta al carattere sotto al cursore
        EXTRN  SCREEN_X:BYTE, SCREEN_Y:BYTE

.CODE
;-----;
; Questa procedura sposta il cursore a destra di una posizione o alla ;
; riga successiva se il cursore si trova a fine riga. ;
; ;
; Usa:          SEND_CRLF ;
; Scrive:       SCREEN_PTR, SCREEN_X, SCREEN_Y ;
;-----;

CURSOR_RIGHT  PROC
        INC    SCREEN_PTR        ;Si sposta al carattere successivo
        INC    SCREEN_PTR
        INC    SCREEN_X          ;Si sposta alla colonna successiva
        CMP    SCREEN_X,79       ;Si assicura che la colonna sia <= 79
        JBE    OK
        CALL   SEND_CRLF         ;Va alla linea successiva

OK:
        RET

CURSOR_RIGHT  ENDP

        PUBLIC  UPDATE_REAL_CURSOR

;-----;
; Questa procedura sposta il cursore vero nella posizione corrente del ;
; cursore virtuale. Deve essere chiamato appena prima della richiesta ;
; di input all'utente. ;
;-----;

UPDATE_REAL_CURSOR  PROC
        PUSH   DX
        MOV    DL,SCREEN_X       ;La posizione del cursore virtuale
        MOV    DH,SCREEN_Y
        CALL   GOTO_XY           ;Sposta il cursore reale in questa posizione
        POP    DX
        RET

UPDATE_REAL_CURSOR  ENDP

```

```

PUBLIC UPDATE_VIRTUAL_CURSOR
;-----;
; Questa procedura aggiorna la posizione del cursore virtuale in      ;
; accordo con la posizione del cursore reale                          ;
;-----;
UPDATE_VIRTUAL_CURSOR      PROC
    PUSH    AX
    PUSH    BX
    PUSH    CX
    PUSH    DX
    MOV     AH,3              ;Chiede la posizione del cursore
    XOR     BH,BH            ;in pagina 0
    INT     10h              ;Salva la posizione del cursore in DH, DL
    CALL    GOTO_XY          ;Sposta il cursore virtuale in questa posizione
    POP     DX
    POP     CX
    POP     BX
    POP     AX
    RET
UPDATE_VIRTUAL_CURSOR      ENDP

PUBLIC CLEAR_TO_END_OF_LINE
;-----;
; Questa procedura cancella la riga dalla posizione corrente del cursore ;
; alla fine della riga stessa.                                          ;
;-----;
CLEAR_TO_END_OF_LINE      PROC
    PUSH    AX
    PUSH    BX
    PUSH    CX
    PUSH    DX
    MOV     DL,SCREEN_X
    MOV     DH,SCREEN_Y
    MOV     AH,6              ;Imposta per cancellare fino alla fine della
riga
    XOR     AL,AL            ;Cancella finestra
    MOV     CH,DH            ;Tutto sulla stessa riga
    MOV     CL,DL            ;Inizia dalla posizione del cursore
    MOV     DL,79            ;E termina alla fine della riga
    MOV     BH,7              ;Usa gli attributi normali
    INT     10h
    POP     DX
    POP     CX
    POP     BX
    POP     AX
    RET
CLEAR_TO_END_OF_LINE      ENDP

```

```
                PUBLIC SEND_CRLF
;-----;
; Utilizza      UPDATE_VIRTUAL_CURSOR      ;
;-----;
SEND_CRLF PROC
    PUSH  AX
    PUSH  DX
    MOV   AH,2                ;Richiede funzione output carattere
    MOV   DL,CR              ;Invia un carattere di ritorno carrello
    INT   21h
    MOV   DL,LF              ;Invia un carattere di avanzamento riga
    INT   21h
    CALL  UPDATE_VIRTUAL_CURSOR    ;Aggiorna la posizione del cursore
virtuale
    POP   DX
    POP   AX
    RET
SEND_CRLF ENDP

                END
```


DISK_IO.ASM

```

.MODE      SMALL

.DATA

        EXTRN  SECTOR:BYTE
        EXTRN  DISK_DRIVE_NO:BYTE
        EXTRN  CURRENT_SECTOR_NO:WORD

.CODE

        PUBLIC PREVIOUS_SECTOR
        EXTRN  INIT_SEC_DISP:PROC, WRITE_HEADER:PROC
        EXTRN  WRITE_PROMPT_LINE:PROC

.DATA

        EXTRN  CURRENT_SECTOR_NO:WORD, EDITOR_PROMPT:BYTE

.CODE

;-----;
; Questa procedura legge il settore precedente, se possibile. ;
; ; ;
; Usa:          WRITE_HEADER, READ_SECTOR, INIT_SEC_DISP ;
;              WRITE_PROMPT_LINE ;
; Legge:       CURRENT_SECTOR_NO, EDITOR_PROMPT ;
; Scrive:     CURRENT_SECTOR_NO ;
;-----;
PREVIOUS_SECTOR          PROC
        PUSH  AX
        PUSH  DX
        MOV  AX,CURRENT_SECTOR_NO      ;Rileva il numero del settore corrente
        OR   AX,AX                      ;Non decrementa se già 0
        JZ   DONT_DECREMENT_SECTOR
        DEC  AX
        MOV  CURRENT_SECTOR_NO,AX      ;Salva nuovo numero di settore
        CALL WRITE_HEADER
        CALL READ_SECTOR
        CALL INIT_SEC_DISP              ;Visualizza nuovo settore
        LEA  DX,EDITOR_PROMPT
        CALL WRITE_PROMPT_LINE
DONT_DECREMENT_SECTOR:
        POP  DX
        POP  AX
        RET
PREVIOUS_SECTOR          ENDP

        PUBLIC NEXT_SECTOR
        EXTRN  INIT_SEC_DISP:PROC, WRITE_HEADER:PROC
        EXTRN  WRITE_PROMPT_LINE:PROC

.DATA

        EXTRN  CURRENT_SECTOR_NO:WORD, EDITOR_PROMPT:BYTE

```

```

.CODE
;-----;
; Legge il settore successivo. ;
; ;
; Usa: WRITE_HEADER, READ_SECTOR, INIT_SEC_DISP ;
; WRITE_PROMPT_LINE ;
; Legge: CURRENT_SECTOR_NO, EDITOR_PROMPT ;
; Scrive: CURRENT_SECTOR_NO ;
;-----;
NEXT_SECTOR PROC
    PUSH AX
    PUSH DX
    MOV AX,CURRENT_SECTOR_NO
    INC AX ;Passa al settore successivo
    MOV CURRENT_SECTOR_NO,AX
    CALL WRITE_HEADER
    CALL READ_SECTOR
    CALL INIT_SEC_DISP ;Visualizza il nuovo settore
    LEA DX,EDITOR_PROMPT
    CALL WRITE_PROMPT_LINE
    POP DX
    POP AX
    RET
NEXT_SECTOR ENDP

PUBLIC READ_SECTOR
;-----;
; Questa procedura legge un settore (512 byte) in SECTOR. ;
; ;
; Legge: CURRENT_SECTOR_NO, DISK_DRIVE_NO ;
; Scrive: SECTOR ;
;-----;
READ_SECTOR PROC
    PUSH AX
    PUSH BX
    PUSH CX
    PUSH DX
    MOV AL,DISK_DRIVE_NO ;Numero drive
    MOV CX,1 ;Legge solo 1 settore
    MOV DX,CURRENT_SECTOR_NO ;Numero settore logico
    LEA BX,SECTOR ;Dove memorizzare questo settore
    INT 25h ;Legge il settore
    POPF ;Elimina flag impostati su stack dal DOS
    POP DX
    POP CX
    POP BX
    POP AX
    RET
READ_SECTOR ENDP

```

```
                PUBLIC WRITE_SECTOR
;-----;
; Questa procedura riscrive sul disco il settore.           ;
;                                                           ;
; Legge:          DISK_DRIVE_NO, CURRENT_SECTOR_NO, SECTOR ;
;-----;
WRITE_SECTOR    PROC
                PUSH    AX
                PUSH    BX
                PUSH    CX
                PUSH    DX
                MOV     AL,DISK_DRIVE_NO                    ;Numero drive
                MOV     CX,1                               ;Scrive 1 settore
                MOV     DX,CURRENT_SECTOR_NO              ;Settore logico
                LEA     BX,SECTOR
                INT     26h                                ;Scrive il settore su disco
                POPF                                ;Elimina il flag delle informazioni
                POP     DX
                POP     CX
                POP     BX
                POP     AX
                RET
WRITE_SECTOR    ENDP

                END
```

DISPATCH.ASM

```

.MODEL SMALL

.CODE

    EXTRN NEXT_SECTOR:PROC                ;In DISK_IO.ASM
    EXTRN PREVIOUS_SECTOR:PROC           ;In DISK_IO.ASM
    EXTRN PHANTOM_UP:PROC, PHANTOM_DOWN:PROC ;In PHANTOM.ASM
    EXTRN PHANTOM_LEFT:PROC, PHANTOM_RIGHT:PROC
    EXTRN WRITE_SECTOR:PROC              ;In DISK_IO.ASM

.DATA
;-----;
; Questa tabella contiene i tasti estesi ASCII ammessi e gli indirizzi delle ;
; procedure che devono essere richiamati alla pressione di ogni tasto. ;
; Il formato della tabella è ;
; DB 72 ;Codice esteso per cursore verso l'alto ;
; DW OFFSET PHANTOM_UP ;
;-----;
DISPATCH_TABLE LABEL BYTE
    DB 61 ;F3
    DW OFFSET _TEXT:PREVIOUS_SECTOR
    DB 62 ;F4
    DW OFFSET _TEXT:NEXT_SECTOR
    DB 72 ;Cursore verso l'alto
    DW OFFSET _TEXT:PHANTOM_UP
    DB 80 ;Cursore verso il basso
    DW OFFSET _TEXT:PHANTOM_DOWN
    DB 75 ;Cursore a sinistra
    DW OFFSET _TEXT:PHANTOM_LEFT
    DB 77 ;Cursore a destra
    DW OFFSET _TEXT:PHANTOM_RIGHT
    DB 85 ;SHIFT-F2
    DW OFFSET _TEXT:WRITE_SECTOR
    DB 0 ;Fine della tabella

.CODE

    PUBLIC DISPATCHER
    EXTRN READ_BYTE:PROC, EDIT_BYTE:PROC
    EXTRN WRITE_PROMPT_LINE:PROC

.DATA

    EXTRN EDITOR_PROMPT:BYTE

.CODE
;-----;
; Questa è la routine di smistamento principale. Durante le normali ;
; operazioni di editing e di visualizzazione questa procedura legge i ;
; caratteri dalla tastiera e, se il carattere è un tasto di comando (come ;
; ad esempio un tasto cursore), DISPATCHER richiama le procedure che ;
; effettuano il lavoro effettivo. Lo smistamento è effettuato per tutti ;
; i tasti elencati nella tabella DISPATCH_TABLE, dove gli indirizzi delle ;
; procedure sono memorizzati subito dopo i nomi dei tasti. ;
; Se il carattere non è un tasto speciale, dovrà essere introdotto ;
; direttamente nel buffer di settore (modalità di editing). ;

```

```

;
; Usa:          READ_BYTE, EDIT_BYTE, WRITE_PROMPT_LINE
; Legge:       EDITOR_PROMPT
;-----;
DISPATCHER    PROC
                PUSH    AX
                PUSH    BX
                PUSH    DX
DISPATCH_LOOP:
                CALL    READ_BYTE      ;Legge carattere in AX
                OR      AH,AH          ;AX = -1 se nessun carattere letto, 1
                                        ; per un codice esteso.
                JS      NO_CHARS_READ  ;Nessun carattere letto, riprova
                JNZ     SPECIAL_KEY    ;Letto un codice esteso
                MOV     DL,AL
                CALL    EDIT_BYTE      ;Era un carattere normale, modifica byte
                JMP     DISPATCH_LOOP   ;Legge un altro carattere

SPECIAL_KEY:
                CMP     AL,68           ;F10-uscita?
                JE      END_DISPATCH   ;Si, esci
                                        ;Usa BX per consultare la tabella
                LEA     BX,DISPATCH_TABLE
SPECIAL_LOOP:
                CMP     BYTE PTR [BX],0 ;Fine tabella?
                JE      NOT_IN_TABLE   ;Si, tasto non presente in tabella
                CMP     AL,[BX]        ;Corrisponde a questo elemento di tabella?
                JE      DISPATCH       ;Si, allora smista
                ADD     BX,3            ;No, prova il prossimo elemento
                JMP     SPECIAL_LOOP    ;Controlla il successivo elemento nella tabella

DISPATCH:
                INC     BX              ;Punta a indirizzo di procedura
                CALL    WORD PTR [BX]  ;Richiama procedura
                JMP     DISPATCH_LOOP   ;Attende altro tasto

NOT_IN_TABLE:
                                        ;Non produce nulla, legge solo il carattere
                                        ;successivo
                JMP     DISPATCH_LOOP

NO_CHARS_READ:
                LEA     DX,EDITOR_PROMPT
                CALL    WRITE_PROMPT_LINE ;Cancella i caratteri non validi
                JMP     DISPATCH_LOOP   ;Riprova

END_DISPATCH:
                POP     DX
                POP     BX
                POP     AX
                RET
DISPATCHER    ENDP

                END

```

DISP_SEC.ASM

```

.MODEL    SMALL

;-----;
; Caratteri grafici per cornice del settore.      ;
;-----;

VERTICAL_BAR EQU    0BAh
HORIZONTAL_BAR EQU   0CDh
UPPER_LEFT   EQU    0C9h
UPPER_RIGHT  EQU    0BBh
LOWER_LEFT   EQU    0C8h
LOWER_RIGHT  EQU    0BCh
TOP_T_BAR EQU   0CBh
BOTTOM_T_BAR EQU   0CAh
TOP_TICK EQU   0D1h
BOTTOM_TICK  EQU   0CFh

.DATA

TOP_LINE_PATTERN LABEL    BYTE
    DB    ' ',7
    DB    UPPER_LEFT,1
    DB    HORIZONTAL_BAR,12
    DB    TOP_TICK,1
    DB    HORIZONTAL_BAR,11
    DB    TOP_TICK,1
    DB    HORIZONTAL_BAR,11
    DB    TOP_TICK,1
    DB    HORIZONTAL_BAR,12
    DB    TOP_T_BAR,1
    DB    HORIZONTAL_BAR,18
    DB    UPPER_RIGHT,1
    DB    0

BOTTOM_LINE_PATTERN LABEL    LABEL    BYTE
    DB    ' ',7
    DB    LOWER_LEFT,1
    DB    HORIZONTAL_BAR,12
    DB    BOTTOM_TICK,1
    DB    HORIZONTAL_BAR,11
    DB    BOTTOM_TICK,1
    DB    HORIZONTAL_BAR,11
    DB    BOTTOM_TICK,1
    DB    HORIZONTAL_BAR,12
    DB    BOTTOM_T_BAR,1
    DB    HORIZONTAL_BAR,18
    DB    LOWER_RIGHT,1
    DB    0

.DATA?

```

```

        EXTRN  SECTOR:BYTE

.CODE

        PUBLIC  INIT_SEC_DISP
        EXTRN  WRITE_PATTERN:PROC, SEND_CRLF:PROC
        EXTRN  GOTO_XY:PROC, WRITE_PHANTOM:PROC

.DATA

        EXTRN  LINES_BEFORE_SECTOR:BYTE
        EXTRN  SECTOR_OFFSET:WORD

.CODE

;-----;
; Questa procedura inizializza la visualizzazione di mezzo settore.      ;
;                                                                           ;
; Usa:          WRITE_PATTERN, SEND_CRLF, DISP_HALF_SECTOR                ;
;               WRITE_TOP_HEX_NUMBERS, GOTO_XY, WRITE_PHANTOM             ;
; Legge:       TOP_LINE_PATTERN, BOTTOM_LINE_PATTERN                       ;
;               LINES_BEFORE_SECTOR                                       ;
; Scrive:      SECTOR_OFFSET                                              ;
;-----;
INIT_SEC_DISP  PROC
        PUSH  DX
        XOR   DL,DL                ;Sposta il cursore all'inizio
        MOV   DH,LINES_BEFORE_SECTOR
        CALL  GOTO_XY
        CALL  WRITE_TOP_HEX_NUMBERS
        LEA  DX,TOP_LINE_PATTERN
        CALL  WRITE_PATTERN
        CALL  SEND_CRLF
        XOR  DX,DX                ;Comincia all'inizio del settore
        MOV  SECTOR_OFFSET,DX     ;Imposta l'offset del settore a 0
        CALL DISP_HALF_SECTOR
        LEA  DX,BOTTOM_LINE_PATTERN
        CALL WRITE_PATTERN
        CALL WRITE_PHANTOM       ;Scrive il cursore fantasma
        POP  DX
        RET
INIT_SEC_DISP  ENDP

        PUBLIC  WRITE_HEADER

.DATA

        EXTRN  HEADER_LINE_NO:BYTE
        EXTRN  HEADER_PART_1:BYTE
        EXTRN  HEADER_PART_2:BYTE
        EXTRN  DISK_DRIVE_NO:BYTE
        EXTRN  CURRENT_SECTOR_NO:WORD

.CODE

        EXTRN  WRITE_STRING:PROC, WRITE_DECIMAL:PROC
        EXTRN  GOTO_XY:PROC

```

```

-----;
; Questa procedura scrive l'header con la lettera del drive e il      ;
; numero del settore                                               ;
;                                                                     ;
; Usa:   GOTO_XY, WRITE_STRING, WRITE_CHAR, WRITE_DECIMAL          ;
; Legge: HEADER_LINE_NO, HEADER_PART_1, HEADER_PART_2, DISK_DRIVE_NO ;
;        CURRENT_SECTOR_NO                                         ;
-----;
WRITE_HEADER    PROC
    PUSH    DX
    XOR     DL,DL                ;Sposta il cursore sulla linea dell'header
    MOV     DH,HEADER_LINE_NO
    CALL    GOTO_XY
    LEA     DX,HEADER_PART_1
    CALL    WRITE_STRING
    MOV     DL,DISK_DRIVE_NO
    ADD     DL,'A'                ;Scrive disco A, B, ...
    CALL    WRITE_CHAR
    LEA     DX,HEADER_PART_2
    CALL    WRITE_STRING
    MOV     DX,CURRENT_SECTOR_NO
    CALL    WRITE_DECIMAL
    POP     DX
    RET
WRITE_HEADER    ENDP

    EXTRN  WRITE_CHAR_N_TIMES:PROC, WRITE_HEX:PROC, WRITE_CHAR:PROC
    EXTRN  WRITE_HEX_DIGIT:PROC, SEND_CRLF:PROC

-----;
; Questa procedura scrive i numeri da 0 a F nella riga superiore della ;
; visualizzazione di mezzo settore.                                   ;
;                                                                     ;
; Usa:   WRITE_CHAR_N_TIMES, WRITE_HEX, WRITE_CHAR                 ;
;        WRITE_HEX_DIGIT, SEND_CRLF                                ;
-----;
WRITE_TOP_HEX_NUMBERS    PROC
    PUSH    CX
    PUSH    DX
    MOV     DL,' '                ;Scrive 9 spazi per il lato sinistro
    MOV     CX,9
    CALL    WRITE_CHAR_N_TIMES
    XOR     DH,DH                ;Inizia da 0
HEX_NUMBER_LOOP:
    MOV     DL,DH
    CALL    WRITE_HEX
    MOV     DL,' '
    CALL    WRITE_CHAR
    INC     DH
    CMP     DH,10h                ;Finito?
    JB     HEX_NUMBER_LOOP

```



```

        MOV     DL, ' '           ;Scrive i numeri esadecimali nella finestra
ASCII
        MOV     CX,2
        CALL    WRITE_CHAR_N_TIMES
        XOR     DL,DL
HEX_DIGIT_LOOP:
        CALL    WRITE_HEX_DIGIT
        INC     DL
        CMP     DL,10h
        JB     HEX_DIGIT_LOOP
        CALL    SEND_CRLF
        POP     DX
        POP     CX
        RET
WRITE_TOP_HEX_NUMBERS      ENDP

        PUBLIC  DISP_HALF_SECTOR
        EXTRN   SEND_CRLF:PROC
;-----;
; Questa procedura visualizza mezzo settore (256 byte)
;
; Inserimento: DS:DX Distanza in SECTOR, in byte - deve essere
;                un multiplo di 16.
;
; Usa:           DISP_LINE, SEND_CRLF
;-----;
DISP_HALF_SECTOR          PROC
        PUH     CX
        PUSH    DX
        MOV     CX,16           ;Visualizza 16 righe
HALF_SECTOR:
        CALL    DISP_LINE
        CALL    SEND_CRLF
        ADD     DX,16
        LOOP   HALF_SECTOR
        POP     DX
        POP     CX
        RET
DISP_HALF_SECTOR          ENDP

        PUBLIC  DISP_LINE
        EXTRN   WRITE_HEX:PROC
        EXTRN   WRITE_CHAR:PROC
        EXTRN   WRITE_CHAR_N_TIMES:PROC
;-----;
; Questa procedura visualizza una riga di dati, o 16 byte, prima in
; esadecimale e poi in ASCII.
;
; Inserimento: DS:DX Distanza in SECTOR, in byte.
;
; Usa:           WRITE_CHAR, WRITE_HEX, WRITE_CHAR_N_TIMES
; Legge:         SECTOR
;-----;

```

```

DISP_LINE      PROC
                PUSH  BX
                PUSH  CX
                PUSH  DX
                MOV   BX,DX                ;La distanza è più utile in BX
                MOV   DL,' '
                MOV   CX,3                ;Scrive 3 spazi prima della linea
                CALL  WRITE_CHAR_N_TIMES
                                ;Scrive la distanza in esadecimale
                CMP   BX,100h            ;La prima cifra è 1?
                JB   WRITE_ONE           ;No, uno spazio è già in DL
                MOV   DL,'1'            ;Sì, allora inserisci 1 in DL per l'output
WRITE_ONE:
                CALL  WRITE_CHAR
                MOV   DL,BL                ;Copia il byte basso in DL per l'output hex
                CALL  WRITE_HEX
                                ;Scrive separatore
                MOV   DL,' '
                CALL  WRITE_CHAR
                MOV   DL,VERTICAL_BAR    ;Traccia il lato sinistro del riquadro
                CALL  WRITE_CHAR
                MOV   CX,16
                PUSH  BX
HEX_LOOP:
                MOV   DL,SECTOR[BX]      ;Preleva 1 byte
                CALL  WRITE_HEX          ;Visualizza il byte in esadecimale
                MOV   DL,' '            ;Scrive uno spazio tra i numeri
                CALL  WRITE_CHAR
                INC   BX
                LOOP  HEX_LOOP

                MOV   DL,' '            ;Scrive separatore
                CALL  WRITE_CHAR
                MOV   DL,' '            ;Aggiunge un altro spazio prima dei caratteri
                CALL  WRITE_CHAR

                MOV   CX,16
                POP   BX                ;Riporta la distanza in SECTOR
ASCII_LOOP:
                MOV   DL,SECTOR[BX]
                CALL  WRITE_CHAR
                INC   BX
                LOOP  ASCII_LOOP

                MOV   DL,' '            ;Traccia il lato destro del riquadro
                CALL  WRITE_CHAR
                MOV   DL,VERTICAL_BAR
                CALL  WRITE_CHAR

                POP   DX
                POP   CX
                POP   BX
                RET

```

```
DISP_LINE      ENDP

                PUBLIC WRITE_PROMPT_LINE
                EXTRN CLEAR_TO_END_OF_LINE:PROC, WRITE_STRING:PROC
                EXTRN GOTO_XY:PROC

.DATA
                EXTRN PROMPT_LINE_NO:BYTE

.CODE
;-----;
; Questa procedura scrive la riga di messaggio sullo schermo e cancella la ;
; parte restante della riga. ;
; ;
; Inserimento: DS:DX Indirizzo della riga di messaggio ;
; ;
; Usa: WRITE_STRING, CLEAR_TO_END_OF_LINE, GOTO_XY ;
; Legge: PROMPT_LINE_NO ;
;-----;
WRITE_PROMPT_LINE PROC
                PUSH DX
                XOR DL,DL ;Scrive la riga di messaggio e
                MOV DH,PROMPT_LINE_NO ; sposta il cursore su quella riga
                CALL GOTO_XY
                POP DX
                CALL WRITE_STRING
                CALL CLEAR_TO_END_OF_LINE
                RET
WRITE_PROMPT_LINE ENDP

                END
```

DSKPATCH.ASM

```

DOSSEG
.MODEL    SMALL

.STACK

.DATA

        PUBLIC SECTOR_OFFSET
;-----;
; SECTOR_OFFSET è l'offset della visualizzazione ;
; di mezzo settore nel settore intero. Deve      ;
; essere un multiplo di 16 e non maggiore di 256 ;
;-----;
SECTOR_OFFSET          DW          0

        PUBLIC CURRENT_SECTOR_NO, DISK_DRIVE_NO
CURRENT_SECTOR_NO      DW          0      ;Inizialmente settore 0
DISK_DRIVE_NO          DB          0      ;Inizialmente Drive A:

        PUBLIC LINES_BEFORE_SECTOR, HEADER_LINE_NO
        PUBLIC HEADER_PART_1, HEADER_PART_2
;-----;
; LINES_BEFORE_SECTOR è il numero di righe vuote ;
; nella parte alta dello schermo prima della    ;
; visualizzazione di mezzo settore.             ;
;-----;
LINES_BEFORE_SECTOR    DB          2
HEADER_LINE_NO         DB          0
HEADER_PART_1          DB          'Disco ',0
HEADER_PART_2          DB          '      Settore ',0
        PUBLIC PROMPT_LINE_NO, EDITOR_PROMPT
PROMPT_LINE_NO         DB          21
EDITOR_PROMPT          DB          'Premere il tasto funzione, o digita'
                        DB          ' il byte esadecimale: ',0

.DATA?

        PUBLIC SECTOR
;-----;
; L'intero settore (fino a 8192 byte) è salvato ;
; in quest'area di memoria.                    ;
;-----;
SECTOR    DB      8192 DUP (?)

.CODE

        EXTRN CLEAR_SCREEN:PROC, READ_SECTOR:PROC
        EXTRN INIT_SEC_DISP:PROC
        EXTRN WRITE_PROMPT_LINE:PROC, DISPATCHER:PROC
        EXTRN INIT_WRITE_CHAR:PROC

```

```
DISK_PATCH          PROC
    MOV     AX,DGROUP      ;mette il segmento dati in AX
    MOV     DS,AX         ;imposta DS in modo da puntare ai dati

    CALL    INIT_WRITE_CHAR
    CALL    CLEAR_SCREEN
    CALL    WRITE_HEADER
    CALL    READ_SECTOR
    CALL    INIT_SEC_DISP
    LEA    DX,EDITOR_PROMPT
    CALL    WRITE_PROMPT_LINE
    CALL    DISPATCHER
    MOV     AH,4Ch         ;Torna al DOS

    INT     21h
DISK_PATCH          ENDP

END     DISK_PATCH
```

EDITOR.ASM

```

.MODEL    SMALL

.CODE

.DATA

        EXTRN  SECTOR:BYTE
        EXTRN  SECTOR_OFFSET:WORD
        EXTRN  PHANTOM_CURSOR_X:BYTE
        EXTRN  PHANTOM_CURSOR_Y:BYTE

.CODE
;-----;
; Questa procedura scrive un byte su SECTOR, alla locazione di memoria      ;
; puntata dal cursore fantasma.                                           ;
;                                                                           ;
;          DL      Byte da scrivere su SECTOR                             ;
;                                                                           ;
; L'offset è calcolato da                                                 ;
;   OFFSET = SECTOR_OFFSET + (16 * PHANTOM_CURSOR_Y) + PHANTOM_CURSOR_X  ;
;                                                                           ;
; Legge:          PHANTOM_CURSOR_X, PHANTOM_CURSOR_Y, SECTOR_OFFSET       ;
; Scrive:         SECTOR                                                  ;
;-----;
WRITE_TO_MEMORY PROC
        PUSH  AX
        PUSH  BX
        PUSH  CX
        MOV  BX,SECTOR_OFFSET
        MOV  AL,PHANTOM_CURSOR_Y
        XOR  AH,AH
        MOV  CL,4                      ;Moltiplica PHANTOM_CURSOR_Y per 16
        SHL  AX,CL
        ADD  BX,AX                      ;BX = SECTOR_OFFSET + (16 * Y)
        MOV  AL,PHANTOM_CURSOR_X
        XOR  AH,AH
        ADD  BX,AX                      ;Questo è l'indirizzo!
        MOV  SECTOR[BX],DL             ;Ora, memorizza il byte
        POP  CX
        POP  BX
        POP  AX
        RET
WRITE_TO_MEMORY ENDP

        PUBLIC EDIT_BYTE
        EXTRN  SAVE_REAL_CURSOR:PROC, RESTORE_REAL_CURSOR:PROC
        EXTRN  MOV_TO_HEX_POSITION:PROC, MOV_TO_ASCII_POSITION:PROC
        EXTRN  WRITE_PHANTOM:PROC, WRITE_PROMPT_LINE:PROC
        EXTRN  CURSOR_RIGHT:PROC, WRITE_HEX:PROC, WRITE_CHAR:PROC

.DATA

```

```

        EXTRN  EDITOR_PROMPT:BYTE

.CODE
;-----;
; Questa procedura modifica un byte in memoria e sullo schermo.           ;
;                                                                           ;
; Inserimento: DL Byte da scrivere su SECTOR, e modificare sullo schermo  ;
;                                                                           ;
; Usa:          SAVE_REAL_CURSOR, RESTORE_REAL_CURSOR                       ;
;               MOV_TO_HEX_POSITION, MOV_TO_ASCII_POSITION                 ;
;               WRITE_PHANTOM, WRITE_PROMPT_LINE, CURSOR_RIGHT            ;
;               WRITE_HEX, WRITE_CHAR, WRITE_TO_MEMORY                    ;
; Legge:        EDITOR_PROMPT                                             ;
;-----;
EDIT_BYTE  PROC
        PUSH  DX
        CALL  SAVE_REAL_CURSOR
        CALL  MOV_TO_HEX_POSITION      ;Porta sul numero esadecimale nella
        CALL  CURSOR_RIGHT             ; finestra esadecimale
        CALL  WRITE_HEX                ;Scrive il nuovo numero
        CALL  MOV_TO_ASCII_POSITION    ;Si porta sul carattere nella finestra ASCII
        CALL  WRITE_CHAR               ;Scrive il nuovo carattere
        CALL  RESTORE_REAL_CURSOR      ;Riporta il cursore alla posizione iniziale
        CALL  WRITE_PHANTOM           ;Riscribe il cursore fantasma
        CALL  WRITE_TO_MEMORY          ;Salva questo nuovo byte in SECTOR
        LEA  DX,EDITOR_PROMPT
        CALL  WRITE_PROMPT_LINE
        POP  DX
        RET
EDIT_BYTE  ENDP

        END

```

KBD_IO.ASM

```

.MODEL    SMALL

BS        EQU    8                ;Carattere di Backspace
CR        EQU    13               ;Carattere Carriage Return
ESCAPE    EQU    27               ;Carattere Escape

.DATA

KEYBOARD_INPUT LABEL    BYTE
CHAR_NUM_LIMIT DB        0        ;Lunghezza del buffer di input
NUM_CHARS_READ DB        0        ;Numero di caratteri letti
CHARS      DB        80 DUP (0)   ;Buffer per input da tastiera

.CODE

        PUBLIC  STRING_TO_UPPER
;-----;
; Questa procedura converte i caratteri della stringa, usando il formato ;
; del DOS per le stringhe, in tutte lettere maiuscole. ;
; ;
; DS:DX  Indirizzo del buffer di stringa ;
;-----;
STRING_TO_UPPER PROC
        PUSH    AX
        PUSH    BX
        PUSH    CX
        MOV     BX,DX
        INC     BX                ;Punta contatore di carattere
        MOV     CL,[BX]          ;Conteggio di caratteri nel secondo byte del buffer
        XOR     CH,CH            ;Azzera byte superiore del contatore
UPPER_LOOP:
        INC     BX                ;Punta al carattere successivo nel buffer
        MOV     AL,[BX]
        CMP     AL,'a'           ;Controlla se si tratta di lettera minuscola
        JB     NOT_LOWER        ;No
        CMP     AL,'z'
        JA     NOT_LOWER
        ADD     AL,'A'-'a'       ;Converte in lettera maiuscola
        MOV     [BX],AL
NOT_LOWER:
        LOOP   UPPER_LOOP
        POP     CX
        POP     BX
        POP     AX
        RET
STRING_TO_UPPER ENDP

;-----;
; Questa procedura converte un carattere da ASCII (esadecimale) a un ;

```



```

; nibble (4 bit). ;
; ;
; AL Carattere da convertire ;
; Riporta: AL nibble ;
; DF Impostato in caso di errore, altrimenti azzerato ;
;-----;
CONVERT_HEX_DIGIT PROC
    CMP AL,'0' ;E' una cifra ammessa?
    JB BAD_DIGIT ;No
    CMP AL,'9' ;Non è ancora sicuro
    JA TRY_HEX ;Potrebbe essere una cifra esadecimale
    SUB AL,'0' ;E' decimale, converti in nibble
    CLC ;Azzerà il riporto, nessun errore
    RET

TRY_HEX:
    CMP AL,'A' ;Non è ancora sicuro
    JB BAD_DIGIT ;Non esadecimale
    CMP AL,'F' ;Non è ancora sicuro
    JA BAD_DIGIT ;Non esadecimale
    SUB AL,'A'-10 ;E' esadecimale, converti in nibble
    CLC ;Azzerare riporto, nessun errore
    RET

BAD_DIGIT:
    STC ;Impostare riporto, errore
    RET

CONVERT_HEX_DIGIT ENDP

PUBLIC HEX_TO_BYTE
;-----;
; Questa procedura converte i due caratteri a DS:DX da esadecimale a un byte. ;
; ;
; DS:DX Indirizzo dei due caratteri del numero esadecimale ;
; Riporta: ;
; AL Byte ;
; CF Impostato in caso di errore, altrimenti azzerato ;
;-----;
HEX_TO_BYTE PROC
    PUSH BX
    PUSH CX
    MOV BX,DX ;Invia indirizzo in BX per indirizzamento indiretto
    MOV AL,[BX] ;Preleva la prima cifra
    CALL CONVERT_HEX_DIGIT
    JC BAD_HEX ;Se il riporto è impostato, la cifra hex è errata
    MOV CX,4 ;Ora moltiplica per 16
    SHL AL,CL
    MOV AH,AL ;Ne tiene una copia
    INC BX ;Preleva seconda cifra
    MOV AL,[BX]
    CALL CONVERT_HEX_DIGIT
    JC BAD_HEX ;Se il riporto è impostato, la cifra hex è errata
    OR AL,AH ;Unisce due semibyte
    CLC ;Azzerà riporto per assenza errore

```

```

DONE_HEX:
    POP    CX
    POP    BX
    RET

BAD_HEX:
    STC                                ;Imposta riporto per presenza errore
    JMP    DONE_HEX
HEX_TO_BYTE    ENDP

    PUBLIC READ_STRING
    EXTRN  WRITE_CHAR:PROC
    EXTRN  UPDATE_REAL_CURSOR:PROC

;-----;
; Questa procedura svolge una funzione molto simile alla funzione 0Ah del DOS.;
; Questa funzione però riporterà un carattere speciale se viene premuto un ;
; tasto funzione o un tasto speciale (dopo questi tasti non premere RETURN). ;
; ESCAPE cancellerà l'input e permetterà di ricominciare. ;
; ;
; DS:DX Indirizzo del buffer di tastiera. Il primo byte deve ;
; ;
; ; contenere il numero massimo di caratteri da leggere (più ;
; ; uno per RETURN). Il secondo byte verrà utilizzato da questa ;
; ; procedura per riportare il numero di caratteri letti ;
; ; effettivamente. ;
; ; 0 Nessun carattere letto ;
; ; -1 Letto un carattere speciale altrimenti il ;
; ; numero letto effettivamente (escluso tasto RETURN) ;
; ;
; Usa: BACK_SPACE, WRITE_CHAR, READ_KEY ;
;-----;
READ_STRING    PROC
    PUSH    AX
    PUSH    BX
    PUSH    SI
    MOV     SI,DX                                ;Usa SI per registro indice e
START_OVER:
    CALL    UPDATE_REAL_CURSOR
    MOV     BX,2                                ;BX per l'offset dall'inizio del buffer
    CALL    READ_KEY                            ;Legge un carattere dalla tastiera
    OR     AH,AH                                ;E' un carattere ASCII esteso?
    JZ     EXTENDED                            ;Si, legge carattere esteso
STRING_NOT_EXTENDED:                          ;Il carattere esteso è errore se buffer è
pieno
    CMP     AL,CR                                ;Carattere di ritorno carrello (CR)?
    JE     END_INPUT                            ;Si, input finito
    CMP     AL,BS                                ;Carattere BACKSPACE?
    JNE    NOT_BS                                ;No
    CALL    BACK_SPACE                            ;Si, cancella carattere
    CMP     BL,2                                ;Buffer vuoto?
    JE     START_OVER                            ;Si, puo' leggere ASCII esteso ancora
    JMP     SHORT READ_NEXT_CHAR                ;No, continua lettura caratteri normali
NOT_BS:    CMP     AL,ESCAPE                    ;Carattere di cancellazione buffer, Esc?
    JE     PURGE_BUFFER                        ;Si, allora cancella buffer

```

```

    CMP     BL,[SI]                ;Controlla se buffer e' pieno
    JA     BUFFER_FULL            ;Buffer pieno
    MOV     [SI+BX],AL            ;Altrimenti salva carattere nel buffer
    INC     BX                    ;Punta prossimo carattere libero nel buffer
    PUSH   DX
    MOV     DL,AL                 ;Invia carattere allo schermo
    CALL   WRITE_CHAR
    POP     DX
READ_NEXT_CHAR:
    CALL   UPDATE_REAL_CURSOR
    CALL   READ_KEY
    OR     AH,AH                 ;Un carattere ASCII esteso non è valido
                                ;quando il buffer non è vuoto
    JZ     STRING_NOT_EXTENDED    ;Il carattere è valido

;-----;
; Segnala una condizione di errore inviando un ;
; acustico allo schermo: chr$(7). ;
;-----;
SIGNAL_ERROR:
    PUSH   DX
    MOV     DL,7                 ;Emette segnale acustico inviando chr$(7)
    MOV     AH,2
    INT    21h
    POP     DX
    JMP     SHORT READ_NEXT_CHAR    ;Ora legge carattere successivo

;-----;
; Svuota il buffer di stringa e cancella tutti i ;
; caratteri visualizzati sullo schermo. ;
;-----;
PURGE_BUFFER:
    PUSH   CX
    MOV     CL,[SI]              ;BACKSPACE supera numero massimo di
    XOR     CH,CH
PURGE_LOOP:
    CALL   BACK_SPACE            ;caratteri nel buffer. BACK_SPACE
                                ;impedirà al cursore di tornare troppo
    LOOP  PURGE_LOOP            ;indietro
    POP     CX
    JMP     START_OVER           ;Può leggere ora caratteri ASCII estesi
                                ;dal momento che il buffer è vuoto

;-----;
; Il buffer era pieno, quindi non è possibile ;
; leggere un altro carattere. Invia un segnale ;
; acustico per avvisare l'utente della condizione ;
; di buffer pieno. ;
;-----;
BUFFER_FULL:
    JMP     SHORT SIGNAL_ERROR    ;Se il buffer è pieno, invia un segnale acustico

```

```

;-----;
; Legge il codice ASCII esteso e lo introduce ;
; nel buffer come carattere individuale, quindi ;
; riporta -1 come numero di caratteri letti. ;
;-----;
EXTENDED: ;Legge un codice ASCII esteso
MOV [SI+2],AL ;Introduce questo carattere nel buffer
MOV BL,OFFh ;Numero caratteri speciali letti
JMP SHORT END_STRING

;-----;
; Salva il conteggio dei caratteri letti e ;
; rientra. ;
;-----;
END_INPUT: ;Input terminato
SUB BL,2 ;Conteggio dei caratteri letti
END_STRING:
MOV [SI+1],BL ;Riporta numero di caratteri letti
POP SI
POP BX
POP AX
RET
READ_STRING ENDP

PUBLIC BACK_SPACE
EXTRN WRITE_CHAR:PROC

;-----;
; Questa procedura cancella i caratteri, uno alla volta, dal buffer e ;
; dallo schermo quando il buffer non è vuoto. BACK_SPACE ritorna ;
; quando il buffer è vuoto. ;
; ;
; Inserimento: DS:SI+BX Il carattere più recente ancora nel buffer ;
; Ritorna: DS:SI+BX Punta al carattere successivo più recente ;
; ;
; Usa: WRITE_CHAR ;
;-----;
BACK_SPACE PROC ;Cancella un carattere
PUSH AX
PUSH DX
CMP BX,2 ;Il buffer è vuoto?
JE END_BS ;Si, legge il carattere successivo
DEC BX ;Cancella un carattere dal buffer
MOV AH,2 ;Cancella un carattere dallo schermo
MOV DL,BS
INT 21h
MOV DL,20h ;Scrivo uno spazio in quella posizione
CALL WRITE_CHAR
MOV DL,BS ;Ripetizione
INT 21h
END_BS: POP DX
POP AX
RET
BACK_SPACE ENDP

```

```

PUBLIC READ_BYTE
;-----;
; Questa procedura legge o un singolo carattere ASCII o un numero ;
; esadecimale a due cifre. Questa è solo una versione di prova di READ_BYTE. ;
; ;
; Ritorna          AL      Codice carattere (ad eccezione di AH=0) ;
;                 AH      0 se legge un carattere ASCII ;
;                 ;      1 se legge un tasto speciale ;
;                 ;      -1 se non viene letto nessun carattere ;
; ;
; Usa:             HEX_TO_BYTE, STRING_TO_UPPER, READ_STRING ;
; Legge:           KEYBOARD_INPUT, etc. ;
; Scrive:          KEYBOARD_INPUT, etc. ;
;-----;
READ_BYTE PROC
    PUSH    DX
    MOV     CHAR_NUM_LIMIT,3 ;Ammette solo due caratteri (più RETURN)
    LEA    DX,KEYBOARD_INPUT
    CALL   READ_STRING
    CMP    NUM_CHARS_READ,1 ;Vede quanti caratteri
    JE     ASCII_INPUT ;Solo uno, lo tratta come carattere ASCII
    JB     NO_CHARACTERS ;Premuto solo RETURN
    CALL   STRING_TO_UPPER ;No, converte stringa in maiuscole
    LEA    DX,CHARS ;Indirizzo della stringa da convertire
    CALL   HEX_TO_BYTE ;Converte stringa da esadecimale a byte
    JC     NO_CHARACTERS ;Errore, segnala 'nessun carattere letto'
    XOR    AH,AH ;Segnala lettura di un carattere

DONE_READ:
    POP    DX
    RET

NO_CHARACTERS:
    XOR    AH,AH ;Imposta su 'nessun carattere letto'
    NOT    AH ;Ritorna -1 in AH

ASCII_INPUT:
    MOV    AL,CHARS ;Carica il carattere letto
    MOV    AH,1 ;Segnala lettura di un carattere
    JMP    DONE_READ

READ_BYTE ENDP

PUBLIC READ_KEY
;-----;
; Questa procedura legge un carattere dalla tastiera. ;
; versione di prova di READ_BYTE. ;
; ;
; Ritorna il byte in AL      Codice carattere (ad eccezione di AH=1) ;
;                 AH      0 se legge carattere ASCII ;
;                 ;      1 se legge un carattere speciale ;
;-----;
READ_KEY PROC
    XOR    AH,AH
    INT    16h ;Legge il carattere/scan code dalla tastiera
    OR     AL,AL ;E' un codice esteso?

```

```

        JZ     EXTENDED_CODE      ;Si
NOT_EXTENDED:
        XOR     AH,AH              ;Ritorna solo il codice ASCII
DONE_READING:
        RET

EXTENDED_CODE:
        MOV     AL,AH              ;Mette lo scan code in AL
        MOV     AH,1              ;Segnala codice esteso
        JMP     DONE_READING

READ_KEY ENDP

        END

        PUBLIC READ_DECIMAL
;-----;
; Questa procedura preleva il buffer dell'output di READ_STRING e      ;
; converte la stringa di cifre decimali in una parola.                  ;
;                                                                           ;
;          AX      Parola convertita da decimale                        ;
;          CF      Impostato in caso di errore, altrimenti azzerato    ;
;                                                                           ;
; Usa:           READ_STRING                                           ;
; Legge:        KEYBOARD_INPUT, etc.                                   ;
; Scrive:       KEYBOARD_INPUT, etc.                                   ;
;-----;
READ_DECIMAL PROC
        PUSH   BX
        PUSH   CX
        PUSH   DX
        MOV    CHAR_NUM_LIMIT,6    ;Il numero massimo è di 5 cifre (65535)
        LEA   DX,KEYBOARD_INPUT
        CALL  READ_STRING
        MOV   CL,NUM_CHARS_READ    ;Rileva numero di caratteri letti
        XOR   CH,CH                ;Imposta byte superiore del contatore a 0
        CMP   CL,0                 ;Errore se non viene letto alcun carattere
        JLE   BAD_DECIMAL_DIGIT    ;Nessun carattere letto, segnala errore
        XOR   AX,AX                ;Inizia con numero impostato a 0
        XOR   BX,BX                ;Comincia dall'inizio della stringa
CONVERT_DIGIT:
        MOV   DX,10                ;Moltiplica il numero per 10
        MUL   DX                    ;Moltiplica AX per 10
        JC   BAD_DECIMAL_DIGIT    ;CF impostato se MUL supera una parola
        MOV   DL,CHARS[BX]         ;Rileva cifra successiva
        SUB   DL,'0'               ;E la converte in un semibyte (4 bit)
        JS   BAD_DECIMAL_DIGIT    ;Cifra errata se < 0
        CMP   DL,9                 ;E' una cifra errata?
        JA   BAD_DECIMAL_DIGIT    ;Si
        ADD   AX,DX                ;No, allora aggiungila al numero
        INC   BX                    ;Punta al carattere successivo
        LOOP CONVERT_DIGIT        ;Preleva cifra successiva
DONE_DECIMAL:

```

```
        POP    DX
        POP    CX
        POP    BX
        RET
BAD_DECIMAL_DIGIT:
        STC                    ;Imposta riporto per segnalare errore
        JMP    DONE_DECIMAL
READ_DECIMAL    ENDP

        END
```

PHANTOM.ASM

```

.MODEL    SMALL

.DATA

REAL_CURSOR_X          DB    0
REAL_CURSOR_Y          DB    0
        PUBLIC  PHANTOM_CURSOR_X, PHANTOM_CURSOR_Y
PHANTOM_CURSOR_0 DB    0
PHANTOM_CURSOR_Y DB    0

.CODE

;-----;
; Queste quattro procedure spostano i cursori fantasma. ;
; ;
; Usa:          ERASE_PHANTOM, WRITE_PHANTOM ;
;              SCROLL_DOWN, SCROLL_UP ;
; Legge:       PHANTOM_CURSOR_X, PHANTOM_CURSOR_Y ;
; Scrive:      PHANTOM_CURSOR_X, PHANTOM_CURSOR_Y ;
;-----;

        PUBLIC  PHANTOM_UP
PHANTOM_UP      PROC
        CALL    ERASE_PHANTOM      ;Cancella alla posizione corrente
        DEC     PHANTOM_CURSOR_Y   ;Sposta cursore verso l'alto di una riga
        JNS    WASNT_AT_TOP       ;Non era al limite superiore, scrivo cursore
        CALL    SCROLL_DOWN       ;Era al limite superiore, far scorrere
WASNT_AT_TOP:
        CALL    WRITE_PHANTOM     ;Scrivo cursore fantasma a nuova posizione
        RET
PHANTOM_UP      ENDP

        PUBLIC  PHANTOM_DOWN
PHANTOM_DOWN    PROC
        CALL    ERASE_PHANTOM     ;Cancella alla posizione corrente
        INC     PHANTOM_CURSOR_Y   ;Sposta cursore verso il basso di una riga
        CMP     PHANTOM_CURSOR_Y,16 ;Limite inferiore?
        JB     WASNT_AT_BOTTOM    ;No, scrivo quindi il cursore fantasma
        CALL    SCROLL_UP        ;Si, far scorrere
WASNT_AT_BOTTOM:
        CALL    WRITE_PHANTOM     ;Scrivo il cursore fantasma
        RET
PHANTOM_DOWN    ENDP

        PUBLIC  PHANTOM_LEFT
PHANTOM_LEFT    PROC
        CALL    ERASE_PHANTOM     ;Cancella alla posizione corrente
        DEC     PHANTOM_CURSOR_X   ;Sposta cursore a sinistra di 1 posizione
        JNS    WASNT_AT_LEFT     ;Non era al limite sinistro, scrivo cursore
        MOV     PHANTOM_CURSOR_X,0 ;Era a limite sinistro, riscriverlo lì
WASNT_AT_LEFT:
        CALL    WRITE_PHANTOM     ;Scrivo cursore fantasma

```



```

        RET
PHANTOM_LEFT      ENDP

        PUBLIC PHANTOM_RIGHT
PHANTOM_RIGHT     PROC
        CALL  ERASE_PHANTOM           ;Cancella alla posizione corrente
        INC   PHANTOM_CURSOR_X        ;Sposta il cursore a destra di 1 posizione
        CMP   PHANTOM_CURSOR_X,16    ;Era al limite destro?
        JB   WASNT_AT_RIGHT
        MOV   PHANTOM_CURSOR_X,15    ;Era al limite destro, riscriverlo lì
WASNT_AT_RIGHT:
        CALL  WRITE_PHANTOM           ;Scrive cursore fantasma
        RET
PHANTOM_RIGHT     ENDP

        PUBLIC MOV_TO_HEX_POSITION
        EXTRN GOTO_XY:PROC

.DATA
        EXTRN LINES_BEFORE_SECTOR:BYTE

.CODE
;-----;
; Questa procedura sposta il cursore reale nella posizione del cursore      ;
; fantasma nella finestra esadecimale.                                     ;
;                                                                           ;
; Usa:          GOTO_XY                                                       ;
; Legge:        LINES_BEFORE_SECTOR, PHANTOM_CURSOR_X, PHANTOM_CURSOR_Y    ;
;-----;
MOV_TO_HEX_POSITION      PROC
        PUSH  AX
        PUSH  CX
        PUSH  DX
        MOV   DH,LINES_BEFORE_SECTOR    ;Trova la riga del cursore fantasma (0,0)
        ADD   DH,2                       ;Più la riga di hex e barra orizzontale
DH,PHANTOM_CURSOR_Y     ;DH = riga del cursore fantasma ADD
        MOV   DL,8                       ;Rientro a sinistra
        MOV   CL,3                       ;Ogni colonna usa 3 caratteri, quindi
        MOV   AL,PHANTOM_CURSOR_X       ;dobbiamo moltiplicare CURSOR_X per 3
        MUL   CL
        ADD   DL,AL                      ;E aggiungerlo al rientro per avere la colonna
        CALL  GOTO_XY                   ;del cursore fantasma
        POP   DX
        POP   CX
        POP   AX
        RET
MOV_TO_HEX_POSITION      ENDP

        PUBLIC MOV_TO_ASCII_POSITION
        EXTRN GOTO_XY:PROC

.DATA
        EXTRN LINES_BEFORE_SECTOR:BYTE

.CODE

```

```

;-----;
; Questa procedura sposta il cursore reale all'inizio del cursore ;
; fantasma nella finestra ASCII. ;
; ;
; Usa: GOTO_XY ;
; Legge: LINES_BEFORE_SECTOR, PHANTOM_CURSOR_X, PHANTOM_CURSOR_Y ;
;-----;
MOV_TO_ASCII_POSITION PROC
    PUSH AX
    PUSH DX
    MOV DH,LINES_BEFORE_SECTOR ;Trova la riga del cursore fantasma (0,0)
    ADD DH,2 ;Più la riga di hex e barra orizzontale
    ADD DH,PHANTOM_CURSOR_Y ;DH = riga del cursore fantasma
    MOV DL,59 ;Rientro a sinistra
    ADD DL,PHANTOM_CURSOR_X ;Aggiunge CURSOR_X per ottenere posizione
    CALL GOTO_XY ; X per cursore fantasma
    POP DX
    POP AX
    RET
MOV_TO_ASCII_POSITION ENDP

PUBLIC SAVE_REAL_CURSOR
;-----;
; Questa procedura salva la posizione del cursore reale nelle due ;
; variabili REAL_CURSOR_X e REAL_CURSOR_Y. ;
; ;
; Scrive: REAL_CURSOR_X, REAL_CURSOR_Y ;
;-----;
SAVE_REAL_CURSOR PROC
    PUSH AX
    PUSH BX
    PUSH CX
    PUSH DX
    MOV AH,3 ;Legge la posizione del cursore
    XOR BH,BH ; a pagina 0
    INT 10h ;E la riporta in DL,DH
    MOV REAL_CURSOR_Y,DL ;Salva la posizione
    MOV REAL_CURSOR_X,DH
    POP DX
    POP CX
    POP BX
    POP AX
    RET
SAVE_REAL_CURSOR ENDP

PUBLIC RESTORE_REAL_CURSOR
EXTRN GOTO_XY:PROC
;-----;
; Questa procedura riporta il cursore reale nella vecchia posizione, ;
; salvata in REAL_CURSOR_X e REAL_CURSOR_Y. ;
; ;
; Usa: GOTO_XY ;

```

```

; Legge:          REAL_CURSOR_X, REAL_CURSOR_Y          ;
;-----;
RESTORE_REAL_CURSOR      PROC
    PUSH  DX
    MOV   DL,REAL_CURSOR_Y
    MOV   DH,REAL_CURSOR_X
    CALL  GOTO_XY
    POP   DX
    RET
RESTORE_REAL_CURSOR      ENDP

    PUBLIC WRITE_PHANTOM
    EXTRN WRITE_ATTRIBUTE_N_TIMES:PROC

;-----;
; Questa procedura usa CURSOR_X e CURSOR_Y, tramite MOV_TO..., come ;
; coordinate per il cursore fantasma. WRITE_PHANTOM scrive il cursore ;
; fantasma. ;
; ;
; Usa:          WRITE_ATTRIBUTE_N_TIMES, SAVE_REAL_CURSOR RESTORE_REAL_CURSOR ;
;              MOV_TO_HEX_POSITION, MOV_TO_ASCII_POSITION ;
;-----;
WRITE_PHANTOM      PROC
    PUSH  CX
    PUSH  DX
    CALL  SAVE_REAL_CURSOR
    CALL  MOV_TO_HEX_POSITION      ;Coordinate cursore nella finestra hex
    MOV   CX,4                    ;Rende cursore fantasma largo 4 caratteri
    MOV   DL,70h
    CALL  WRITE_ATTRIBUTE_N_TIMES
    CALL  MOV_TO_ASCII_POSITION    ;Coordinate cursore nella finestra ASCII
    MOV   CX,1                    ;Qui il cursore è largo un carattere
    CALL  WRITE_ATTRIBUTE_N_TIMES
    CALL  RESTORE_REAL_CURSOR
    POP   DX
    POP   CX
    RET
WRITE_PHANTOM      ENDP

    PUBLIC ERASE_PHANTOM
    EXTRN WRITE_ATTRIBUTE_N_TIMES:PROC

;-----;
; Questa procedura cancella il cursore fantasma; funziona in modo ;
; contrario a WRITE_PHANTOM ;
; ;
; Usa:          WRITE_ATTRIBUTE_N_TIMES, SAVE_REAL_CURSOR, RESTORE_REAL_CURSOR ;
;              MOV_TO_HEX_POSITION, MOV_TO_ASCII_POSITION ;
;-----;
ERASE_PHANTOM      PROC
    PUSH  CX
    PUSH  DX
    CALL  SAVE_REAL_CURSOR
    CALL  MOV_TO_HEX_POSITION      ;Coordinate del cursore nella finestra hex

```

```

MOV     CX,4                ;Riporta a bianco su nero
MOV     DL,7
CALL    WRITE_ATTRIBUTE_N_TIMES
CALL    MOV_TO_ASCII_POSITION
MOV     CX,1
CALL    WRITE_ATTRIBUTE_N_TIMES
CALL    RESTORE_REAL_CURSOR
POP     DX
POP     CX
RET

ERASE_PHANTOM  ENDP

        EXTRN  DISP_HALF_SECTOR:PROC, GOTO_XY:PROC

.DATA
        EXTRN  SECTOR_OFFSET:WORD
        EXTRN  LINES_BEFORE_SECTOR:BYTE

.CODE
;-----;
; Queste due procedure permettono lo spostamento tra le due ;
; visualizzazioni di mezzo settore. ;
; ;
; Usa:          WRITE_PHANTOM, DISP_HALF_SECTOR, ERASE_PHANTOM, GOTO_XY ;
;              SAVE_REAL_CURSOR, RESTORE_REAL_CURSOR ;
; Legge:       LINES_BEFORE_SECTOR ;
; Scrive:      SECTOR_OFFSET, PHANTOM_CURSOR_Y ;
;-----;

SCROLL_UP PROC
    PUSH  DX
    CALL  ERASE_PHANTOM      ;Cancella cursore fantasma
    CALL  SAVE_REAL_CURSOR   ;Salva la posizione del cursore reale
    XOR   DL,DL              ;Imposta cursore per visualizzazione mezzo settore
    MOV   DH,LINES_BEFORE_SECTOR
    ADD   DH,2
    CALL  GOTO_XY
    MOV   DX,256              ;Visualizza secondo mezzo settore
    MOV   SECTOR_OFFSET,DX
    CALL  DISP_HALF_SECTOR
    CALL  RESTORE_REAL_CURSOR ;Ripristina posizione del cursore reale
    MOV   PHANTOM_CURSOR_Y,0 ;Cursore all'inizio secondo mezzo settore
    CALL  WRITE_PHANTOM      ;Ripristina il cursore fantasma
    POP   DX
    RET

SCROLL_UP ENDP

SCROLL_DOWN PROC
    PUSH  DX
    CALL  ERASE_PHANTOM      ;Cancella il cursore fantasma
    CALL  SAVE_REAL_CURSOR   ;Salva la posizione del cursore reale
    XOR   DL,DL              ;Imposta cursore per visualizzazione mezzo settore
    MOV   DH,LINES_BEFORE_SECTOR
    ADD   DH,2
    CALL  GOTO_XY

```

```
XOR    DX,DX                ;Visualizza primo mezzo settore
MOV    SECTOR_OFFSET,DX
CALL  DISP_HALF_SECTOR
CALL  RESTORE_REAL_CURSOR   ;Ripristina posizione del cursore reale
MOV    PHANTOM_CURSOR_Y,15 ;Cursore in fondo al primo mezzo settore
CALL  WRITE_PHANTOM        ;Ripristina cursore fantasma
POP    DX
RET
SCROLL_DOWN    ENDP

END
```

VIDEO_IO.ASM

```

.MODEL    SMALL

.DATA
    PUBLIC SCREEN_PTR
    PUBLIC SCREEN_X, SCREEN_Y
SCREEN_SEG    DW    0B800h    ;Segmento del buffer di schermo
SCREEN_PTR    DW    0        ;Offset del cursore nella memoria video
SCREEN_X      DB    0
SCREEN_Y      DB    0

.CODE

    PUBLIC WRITE_STRING
;-----;
; Questa procedura scrive una stringa di caratteri sullo schermo. La ;
; stringa deve terminare con DB 0 ;
; ;
; Inserimento: DS:DX Indirizzo della stringa ;
; ;
; Usa: WRITE_CHAR ;
;-----;
WRITE_STRING    PROC
    PUSH    AX
    PUSH    DX
    PUSH    SI
    PUSHF                    ;Salva flag direzione
    CLD                      ;Imposta direzione per incremento (in avanti)
    MOV     SI,DX            ;Invia indirizzo a SI per LODSB
STRING_LOOP:
    LODSB                    ;Carica un carattere nel registro AL
    OR     AL,AL            ;Già arrivato a 0?
    JZ     END_OF_STRING    ;Sì, abbiamo finito con la stringa
    MOV    DL,AL            ;No, scrive carattere
    CALL   WRITE_CHAR
    JMP    STRING_LOOP
END_OF_STRING:
    POPF                    ;Ripristina flag di direzione
    POP    SI
    POP    DX
    POP    AX
    RET
WRITE_STRING    ENDP

    PUBLIC WRITE_HEX
;-----;
; Questa procedura converte il byte nel registro DL in esadecimale e ;
; scrive le due cifre esadecimali alla posizione corrente del cursore. ;
; ;
; Inserimento: DL Byte da convertire in esadecimale. ;
; ;

```

```

; Usa:          WRITE_HEX_DIGIT          ;
;-----;
WRITE_HEX PROC          ;Punto di inserimento
    PUSH  CX          ;Salva registri usati in questa procedura
    PUSH  DX
    MOV   DH,DL       ;Copia il byte
    MOV   CX,4        ;Preleva il nibble alto in DL
    SHR  DL,CL
    CALL  WRITE_HEX_DIGIT ;Visualizza prima cifra esadecimale
    MOV   DL,DH       ;Preleva il nibble basso in DL
    AND  DL,0Fh      ;Cancella il nibble alto
    CALL  WRITE_HEX_DIGIT ;Visualizza la seconda cifra esadecimale
    POP   DX
    POP   CX
    RET
WRITE_HEX ENDP

        PUBLIC  WRITE_HEX_DIGIT
;-----;
; Questa procedura converte i 4 bit bassi di DL in una cifra esadecimale
; e la scrive sullo schermo.
;
; Inserimento: DL      I 4 bit inferiori contengono numero da
;                  visualizzare in esadecimale
;
; Usa:          WRITE_CHAR
;-----;
WRITE_HEX_DIGIT PROC
    PUSH  DX          ;Salva i registri utilizzati
    CMP  DL,10        ;Questo nibble è <10?
    JAE  HEX_LETTER   ;No, converte in lettera
    ADD  DL,"0"       ;Si, converte in una cifra
    JMP  Short WRITE_DIGIT ;Ora scrive questo carattere
HEX_LETTER:
    ADD  DL,"A"-10    ;Converte in lettera esadecimale
WRITE_DIGIT:
    CALL  WRITE_CHAR   ;Visualizza la lettera sullo schermo
    POP   DX          ;Ripristina vecchio valore di AX
    RET
WRITE_HEX_DIGIT ENDP

        PUBLIC  INIT_WRITE_CHAR
;-----;
; Dovete chiamare questa procedura prima di chiamare WRITE_CHAR dal momento
; che WRITE_CAHN utilizza delle informazioni impostate da questa procedura
;
; Scrive:      SCREEN_SEG
;-----;
INIT_WRITE_CHAR PROC
    PUSH  AX
    PUSH  BX

```

```

MOV    BX,0B800h           ;Imposta per l'adattatore colore
INT    11h                 ;Richiede le informazioni sull'equipaggiamento
AND    AL,30h              ;Prende solo il tipo di video
CMP    AL,30h              ;E' un adattatore monocromatico?
JNE    SET_BASE            ;No è a colori, quindi usa B000
MOV    BX,0B000h
SET_BASE
MOVSCREEN_SEG,BX           ;Salva il segmento video
POP    BX
POP    AX
RET    INIT_WRITE_CHAR    ENDP

```

```

PUBLIC WRITE_CHAR
EXTRN  CURSOR_RIGHT:PROC

```

```

;-----;
; Questa procedura invia un carattere allo schermo scrivendo direttamente ;
; nella memoria video; in questo modo un carattere come il backspace è ;
; trattato come un qualsiasi altro carattere e visualizzato. Questa procedura ;
; deve effettuare parecchie operazioni per aggiornare la posizione del ;
; cursore. ;
; ;
; DL Byte da stampare sullo schermo ;
; ;
; Usa: CURSOR_RIGHT ;
; Legge: SCREEN_SEG ;
;-----;
WRITE_CHAR PROC
    PUSH AX
    PUSH BX
    PUSH DX
    PUSH ES

    MOV AX,SCREEN_SEG ;Prende il segmento per la memoria video
    MOV ES,AX ;Punta ES alla memoria video
    MOV BX,SCREEN_PTR ;Punta al carattere nella memoria schermo

    MOV DH,7 ;Utilizza l'attributo normale
    MOV ES:[BX],DX ;Scrive il carattere/attributo sullo schermo
    CALL CURSOR_RIGHT ;Si sposta alla posizione successiva del cursore

    POP ES
    POP DX
    POP BX
    POP AX
    RET
WRITE_CHAR ENDP

```

```

PUBLIC WRITE_DECIMAL

```

```

;-----;
; Questa procedura serve per scrivere un numero a 16 bit senza segno ;
; in notazione decimale. ;

```



```

; Inserimento:   DX      N : numero senza segno a 16-bit.      ;
;                                                       ;
; Usa:           WRITE_HEX_DIGIT                               ;
;-----;
WRITE_DECIMAL   PROC      NEAR
                PUSH  AX      ;Salva i registri utilizzati
                PUSH  CX
                PUSH  DX
                PUSH  SI
                MOV   AX,DX
                MOV   SI,10    ;Dividerà per 10 usando SI
                XOR   CX,CX    ;Conta le cifre inserite nello stack
NON_ZERO:
                XOR   DX,DX    ;Imposta la parola superiore di N a 0
                DIV  SI      ;Calcola N/10 e (N mod 10)
                PUSH  DX      ;Inserisce una cifra nello stack
                INC   CX      ;Aggiunge un'altra cifra
                OR    AX,AX    ;Ancora N = 0?
                JNE  NON_ZERO ;No, continua
WRITE_DIGIT_LOOP:
                POP   DX      ;Preleva le cifre in ordine inverso
                CALL  WRITE_HEX_DIGIT
                LOOP  WRITE_DIGIT_LOOP
END_DECIMAL:
                POP   SI
                POP   DX
                POP   CX
                POP   AX
                RET
WRITE_DECIMAL   ENDP

                PUBLIC  WRITE_CHAR_N_TIMES
;-----;
; Questa procedura scrive più di una copia di un carattere      ;
;                                                       ;
; Inserimento:   DL      Codice carattere                       ;
;               CX      Numero di copie del carattere          ;
;                                                       ;
; Usa:           WRITE_CHAR                                     ;
;-----;
WRITE_CHAR_N_TIMES   PROC
                PUSH  CX
N_TIMES:
                CALL  WRITE_CHAR
                LOOP  N_TIMES
                POP   CX
                RET
WRITE_CHAR_N_TIMES   ENDP

                PUBLIC  WRITE_ATTRIBUTE_N_TIMES
                EXTRN  CURSOR_RIGHT:PROC

```

```

;-----;
; Questa procedura imposta l'attributo per N caratteri, iniziando dalla ;
; posizione corrente del cursore. ;
; ;
; CX Numero di caratteri da impostare l'attributo ;
; DL Nuovo attributo per il carattere ;
; ;
; Usa: CURSOR_RIGHT ;
; Legge: SCREEN_SEG, SCREEN_PTR ;
;-----;
WRITE_ATTRIBUTE_N_TIMES PROC
    PUSH AX
    PUSH CX
    PUSH DI
    PUSH ES

    MOV AX,SCREEN_SEG ;Prende il segmento per la memoria video
    MOV ES,AX ;Punta es alla memoria video
    MOV DI,SCREEN_PTR ;Punta al carattere nella memoria video
    INC DI ;Punta all'attributo sotto al cursore
    MOV AL,DL ;Mette l'attributo in AL
ATTR_LOOP:
    STOSB ;Salva un attributo
    INC DI ;Si sposta all'attributo successivo
    INC SCREEN_X ;Si sposta alla colonna successiva
    LOOP ATTR_LOOP ;Scrive N attributi

    DEC DI ;Punta all'inizio del carattere successivo
    MOV SCREEN_PTR,DI

    POP ES
    POP DI
    POP CX
    POP AX
    RET
WRITE_ATTRIBUTE_N_TIMES ENDP

PUBLIC WRITE_PATTERN
;-----;
; Questa procedura traccia una linea sullo schermo sulla base dei dati ;
; seguenti ;
; ;
; DB {carattere, numero di copie del carattere}, 0 ;
; Dove {x} significa che x può essere ripetuto un qualsiasi numero di volte. ;
; ;
; Inserimento: DS:DX Indirizzo del modello da tracciare ;
; ;
; Usa: WRITE_CHAR_N_TIMES ;
;-----;
WRITE_PATTERN PROC
    PUSH AX
    PUSH CX

```

```
        PUSH  DX
        PUSH  SI
        PUSHF                    ;Salva il flag di direzione
        CLD                      ;Imposta il flag direzione per l'incremento
        MOV   SI,DX              ;Sposta la distanza nel registro SI per LODSB
PATTERN_LOOP:
        LODSB                    ;Carica il carattere in AL
        OR   AL,AL              ;I dati sono finiti (0h)?
        JZ   END_PATTERN        ;Si, ritorna
        MOV  DL,AL              ;No, imposta la scrittura del carattere N volte
        LODSB                    ;Carica il contatore in AL
        MOV  CL,AL              ;E lo invia in CX per WRITE_CHAR_N_TIMES
        XOR  CH,CH              ;Imposta a zero il byte alto di CX
        CALL WRITE_CHAR_N_TIMES
        JMP  PATTERN_LOOP
END_PATTERN:
        POPF                    ;Ripristina il flag direzione
        POP  SI
        POP  DX
        POP  CX
        POP  AX
        RET
WRITE_PATTERN  ENDP

        END
```


MESSAGGI DI ERRORE COMUNI

Questa appendice elenca alcuni messaggi di errore che potreste incontrare utilizzando MASM, LINK e EXE2BIN. Se non trovate l'errore in questa appendice, fate riferimento al manuale del macro assembler o al manuale DOS.

I messaggi d'errore sono divisi in tre gruppi: uno per il MASM, uno per LINK e uno per EXE2BIN. In ogni sezione i messaggi sono elencati in ordine alfabetico.

MASM

Block nesting error: Vedrete questo messaggio in concomitanza con *Open procedures* o con *Open segments*. Fate riferimento alla descrizione di questi due messaggi d'errore.

End of file, no END directive: Significa che avete dimenticato la direttiva END alla fine del file, oppure che dovete inserire una riga vuota dopo la direttiva END già esistente. La versione Microsoft del macro assembler, si aspetta di trovare una riga vuota alla fine del file. Se non c'è questa riga, il MASM non leggerà la direttiva END.

Open procedures: Questo messaggio significa che avete dimenticato una direttiva PROC o ENDP, o che i nomi non sono gli stessi in PROC/ENDP. Assicuratevi che ad ogni PROC corrisponda un ENDP e controllate i nomi sia in PROC che ENDP per assicurarsi che corrispondano.

Open segments: Questo messaggio d'errore dovrebbe apparire solo quando state utilizzando la definizione completa dei segmenti. Significa che avete dimenticato una direttiva SEGMENT o ENDS, o che i nomi in SEGMENT/ENDS non corrispondono. Assicuratevi che ad ogni SEGMENT corrisponda un ENDS e controllate i nomi sia in SEGMENT che ENDS per assicurarsi che corrispondano.

Symbol not defined: Ci sono tre cose che dovete guardare se incappate in questo messaggio d'errore:

1. Avete scritto male un nome. Controllate la linea in cui compare l'errore per assicurarsi di averla scritta correttamente.
2. Potreste aver sbagliato a scrivere il nome quando avete dichiarato una

PROC o una variabile. Controllate il nome sbagliato sulla linea d'errore con i nomi nelle dichiarazioni di PROC e delle variabili.

3. Potreste aver dimenticato una dichiarazione EXTRN, o il nome in EXTRN potrebbe essere stato scritto male.

LINK

Fixup offset exceeds field width: In genere questo è l'errore più difficile da scovare. Questo messaggio generalmente significa che avete dichiarato alcune procedure come FAR, ma successivamente le avete dichiarate come NEAR in una dichiarazione EXTRN.

Può anche significare che il vostro programma è più lungo dei 64K consentiti per i programmi small. Potete controllare questo errore guardando il campo dimensione nel map file.

Questo messaggio dovrebbe apparire solo se state utilizzando la definizione completa dei segmenti. Può anche apparire quando un segmento viene frammentato. In certi casi, i due segmenti potrebbero essere più di 64K, il che significa che la CALL, per funzionare, deve essere una FAR CALL.

Se non sembra essere questo il problema, dovete cercare più a fondo. Potreste trovare qualche aiuto nel map file. Per esempio, controllate l'ordine dei segmenti. Potrebbero non essere in ordine.

Symbol defined more than once: Questo messaggio significa che, probabilmente, avete definito la stessa procedura o variabile in due file differenti. Assicuratevi di aver definito ogni nome in un solo file, quindi utilizzate EXTRN dove dovete utilizzare la stessa procedura o variabile.

Unresolved externals: Quando vedete questo messaggio, o manca la direttiva PUBLIC nel file in cui avete dichiarato la variabile o la procedura, oppure avete scritto male il nome in una dichiarazione EXTRN o CALL in un altro file.

Questo errore può essere anche causato se ci si dimentica di processare un file con LINK. Dovete quindi aggiungere il nuovo file al Makefile o al file batch che utilizzate.

Warning: no stack segment: Questo non è un vero messaggio d'errore, è semplicemente un avvertimento. Vedrete sempre questo messaggio quando create i file .COM. Ignoratelo in questi casi.

EXE2BIN

Probabilmente non utilizzerete EXE2BIN molto spesso in quanto vi serve solo se volete creare i file .COM. Ma se lo utilizzate, probabilmente, vedrete un solo messaggio d'errore:

File cannot be converted: Questo messaggio non è di grande aiuto. La maggior parte delle volte può significare una delle tre cose seguenti:

1. I segmenti sono nell'ordine sbagliato, quindi avete un segmento in memoria prima di CODESEG. Controllate la mappa per vedere se è questo il problema.
2. Il programma principale non è il primo che avete elencato nella lista. Dato che lo deve essere, provate a riprocessare ancora il tutto con LINK per vedere se questo era il problema. Ancora una volta potete controllare la mappa per vedere se questo era il problema.
3. Il programma principale non ha un ORG 100h come prima definizione

TABELLE VARIE

Tabella D-1. Codici dei Caratteri ASCII

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0		43	2B	+	86	56	U	129	81	ù
1	1		44	2C	,	87	57	W	130	82	ú
2	2	@	45	2D	-	88	58	X	131	83	û
3	3	A	46	2E	.	89	59	Y	132	84	ü
4	4	B	47	2F	/	90	5A	Z	133	85	ÿ
5	5	C	48	30	0	91	5B	[134	86	ÿ
6	6	D	49	31	1	92	5C	\	135	87	ÿ
7	7	E	50	32	2	93	5D]	136	88	ÿ
8	8	F	51	33	3	94	5E	^	137	89	ÿ
9	9	G	52	34	4	95	5F		138	8A	ÿ
10	A	H	53	35	5	96	60	`	139	8B	ÿ
11	B	I	54	36	6	97	61	a	140	8C	ÿ
12	C	J	55	37	7	98	62	b	141	8D	ÿ
13	D	K	56	38	8	99	63	c	142	8E	ÿ
14	E	L	57	39	9	100	64	d	143	8F	ÿ
15	F	M	58	3A	:	101	65	e	144	90	ÿ
16	0	N	59	3B	;	102	66	f	145	91	ÿ
17	11	O	60	3C	<	103	67	g	146	92	ÿ
18	12	P	61	3D	=	104	68	h	147	93	ÿ
19	13	Q	62	3E	>	105	69	i	148	94	ÿ
20	14	R	63	3F	?	106	6A	j	149	95	ÿ
21	15	S	64	40	@	107	6B	k	150	96	ÿ
22	16	T	65	41	A	108	6C	l	151	97	ÿ
23	17	U	66	42	B	109	6D	m	152	98	ÿ
24	18	V	67	43	C	110	6E	n	153	99	ÿ
25	19	W	68	44	D	111	6F	o	154	9A	ÿ
26	1A	X	69	45	E	112	70	p	155	9B	ÿ
27	1B	Y	70	46	F	113	71	q	156	9C	ÿ
28	1C	Z	71	47	G	114	72	r	157	9D	ÿ
29	1D	[72	48	H	115	73	s	158	9E	ÿ
30	1E]	73	49	I	116	74	t	159	9F	ÿ
31	1F	^	74	4A	J	117	75	u	160	A0	ÿ
32	20	_	75	4B	K	118	76	v	161	A1	ÿ
33	21	`	76	4C	L	119	77	w	162	A2	ÿ
34	22		77	4D	M	120	78	x	163	A3	ÿ
35	23		78	4E	N	121	79	y	164	A4	ÿ
36	24		79	4F	O	122	7A	z	165	A5	ÿ
37	25		80	50	P	123	7B	{	166	A6	ÿ
38	26		81	51	Q	124	7C		167	A7	ÿ
39	27		82	52	R	125	7D	}	168	A8	ÿ
40	28		83	53	S	126	7E	~	169	A9	ÿ
41	29		84	54	T	127	7F		170	AA	ÿ
42	2A		85	55	U	128	80		171	AB	ÿ

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
172	AC	¼	193	C1	␣	214	D6	␣	235	EB	␣
173	AD	½	194	C2	␣	215	D7	␣	236	EC	␣
174	AE	¾	195	C3	␣	216	D8	␣	237	ED	␣
175	AF	␣	196	C4	␣	217	D9	␣	238	EE	␣
176	B0	␣	197	C5	␣	218	DA	␣	239	EF	␣
177	B1	␣	198	C6	␣	219	DB	␣	240	F0	␣
178	B2	␣	199	C7	␣	220	DC	␣	241	F1	␣
179	B3	␣	200	C8	␣	221	DD	␣	242	F2	␣
180	B4	␣	201	C9	␣	222	DE	␣	243	F3	␣
181	B5	␣	202	CA	␣	223	DF	␣	244	F4	␣
182	B6	␣	203	CB	␣	224	E0	␣	245	F5	␣
183	B7	␣	204	CC	␣	225	E1	␣	246	F6	␣
184	B8	␣	205	CD	␣	226	E2	␣	247	F7	␣
185	B9	␣	206	CE	␣	227	E3	␣	248	F8	␣
186	BA	␣	207	CF	␣	228	E4	␣	249	F9	␣
187	BB	␣	208	D0	␣	229	E5	␣	250	FA	␣
188	BC	␣	209	D1	␣	230	E6	␣	251	FB	␣
189	BD	␣	210	D2	␣	231	E7	␣	252	FC	␣
190	BE	␣	211	D3	␣	232	E8	␣	253	FD	␣
191	BF	␣	212	D4	␣	233	E9	␣	254	FE	␣
192	C0	␣	213	D5	␣	234	EA	␣	255	FF	␣

Tabella D-2. Codici dei Colori

0	Nero
1	Blu
2	Verde
3	Ciano
4	Rosso
5	Violetto
6	Marrone
7	Bianco

Attributo = colore di sfondo * 16 + colore in primo piano

Aggiungete 8 al colore in primo piano per la versione evidenziata, o aggiungete 0 al colore di sfondo per renderlo lampeggiante

Molti tasti della tastiera (come i tasti funzione) ritornano un codice a due caratteri quando leggete i tasti attraverso il DOS: un decimale 0 seguito da uno scan code (codice di scansione). La tabella seguente mostra gli scan code per tutti i tasti che non hanno un equivalente ASCII.

Tabella D-3. Codici Estesi di Tastiera

15	Shift Tab
16-25	Alt + Q, W, E, R, T, Y, U, I, O, P
30-38	Alt + A, S, D, F, G, H, J, K, L
44-50	Alt + Z, X, C, V, B, N, M
59-68	Da F1 a F10
71	Home
72	Freccia in Alto
73	PgUp
75	Freccia a Sinistra
77	Freccia a Destra
79	End
80	Freccia in Basso
81	PgDn
82	Ins
83	Del
84-93	Da Shift F1 a Shift F10
94-103	Da Control F1 a Control F10
104-113	Da Alt F1 a Alt F10
114	Control PrtSc
115	Control Freccia a Sinistra
116	Control Freccia a Destra
117	Control End
118	Control PgDn
119	Control Home
120-131	Control Alt + 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, -, =
132	Control PgUp

Tabella delle Modalità di Indirizzamento

Modalità di Indirizzamento	Formato dell'Indirizzo	Registro Segmento Utilizzato
Registro	registro (come AX)	Nessuno
Immediata	dati (come 12345)	Nessuno
<i>Modalità di Indirizzamento della Memoria</i>		
Indiretta di Registro	[BX]	DS
	[BP]	SS
	[DI]	DS
	[SI]	DS
Base Relativa*	etichetta[BX]	DS
	etichetta[BP]	SS
Diretta Indicizzata*	etichetta[DI]	DS
	etichetta[SI]	DS
Base Indicizzata*	etichetta[BX+SI]	DS
	etichetta[BX+DI]	DS
	etichetta[BP+SI]	SS
	etichetta[BP+DI]	SS
Comandi Stringa: (MOVSW, LODSB, e così via)		Leggono da DS:SI Scrivono su ES:DI

* Etichetta[...] può essere sostituita con [spost + ...], dove *spost* è lo spostamento. Quindi si potrebbe scrivere [10 + BX] e l'indirizzo sarebbe 10 + BX.

Tabella D-5. Le Funzioni di INT 10h

(AH)=0 **Imposta la modalità di visualizzazione.** Il registro AL contiene il numero della modalità.

MODALITÀ TESTO

(AL)=0	40 per 25, bianco e nero
(AL)=1	40 per 25, colore
(AL)=2	80 per 28, bianco e nero
(AL)=3	80 per 25, colore
(AL)=7	80 per 25, adattatore monocromatico

MODALITÀ GRAFICHE

(AL)=4	320 per 200, colore
(AL)=5	320 per 200, bianco e nero
(AL)=6	640 per 200, bianco e nero

(AH) = 1 **Imposta la dimensione del cursore.**

(CH) Linea di scan iniziale del cursore. La prima linea è 0 su entrambi i modi di visualizzazione monocromatico e grafico, mentre l'ultima linea è 7 per gli adattatori grafici a colori e 13 per gli adattatori monocromatici. Il range valido è tra 0 e 31.

(CL) Ultima linea di scan del cursore.

L'impostazione all'accensione per l'adattatore grafico a colori è CH=6 e CL=7. Per l'adattatore monocromatico CH=11 e CL=12

(AH) = 2 **Imposta la posizione del cursore.**

(DH,DL) Riga, colonna della nuova posizione del cursore; l'angolo in alto a destra è (0,0).

(BH) Numero di pagina. Questo è il numero della pagina di visualizzazione. L'adattatore grafico a colori ha spazio per parecchie pagine di visualizzazione, ma la maggior parte dei programmi utilizzano la pagina 0.

- (AH) = 3 **Legge la posizione del cursore.**
- (BH) Numero di pagina.
In uscita (DH, DL) Riga, colonna del cursore
 (CH, CL) Dimensione del cursore
- (AH) = 4 **Legge la posizione della penna luminosa** (vedere il Tech. Ref. Man.).
- (AH) = 5 **Seleziona la pagina di visualizzazione attiva.**
- (AL) Nuovo numero pagina (da 0 a 7 per i modi 0 e 1; da 0 a 3 per i modi 2 e 3)
- (AH) = 6 **Scorrimento verso l'alto.**
- (AL) Numero di linee da cancellare nella parte bassa della finestra. Normalmente viene cancellata una sola linea. Impostate a zero per cancellare l'intera finestra.
(CH, CL) Riga, colonna dell'angolo in alto a sinistra della finestra
(DH, DL) Riga, colonna dell'angolo in basso a destra della finestra
(BH) Visualizza gli attributi da utilizzare per cancellare le righe
- (AH) = 7 **Scorrimento verso il basso.**
- Come lo scorrimento verso l'alto (funzione 6), ma si fa riferimento alle righe nella parte alta della finestra.
- (AH) = 8 **Legge l'attributo e il carattere sotto al cursore.**
- (BH) Visualizza la pagina (solo modalità testo)
(AL) Carattere da scrivere
(AH) Attributo del carattere letto (solo modalità testo)
- (AH) = 9 (BX) Visualizza la pagina (solo modalità testo)
(CX) Numero di volte che bisogna scrivere il carattere e l'attributo sullo schermo

(AL) Carattere da scrivere
(BL) Attributo da scrivere

(AH) = 10 **Scrivi il carattere sotto al cursore** (con attributo normale).

(BH) Visualizza la pagina
(CX) Numero di volte da scrivere il carattere
(AL) Carattere da scrivere

(AH)= da 11 a 13 **Varie funzioni grafiche.** (Vedere il Tech. Ref. Man. per maggiori dettagli)

(AH) = 14 **Scrittura carattere** Scrive un carattere sullo schermo e sposta il cursore alla prossima posizione.

(AL) Carattere da scrivere
(BL) Colore del carattere (solo modalità grafica)
(BH) Pagina da visualizzare (modalità testo)

(AH) = 15 **Riporta lo stato corrente del video**

(AL) Visualizza la modalità correntemente impostata
(AH) Numero di caratteri per linea
(BH) Attiva le pagine da visualizzare

Questa tabella contiene le funzioni relative all'istruzione INT 16h utilizzate in questo libro per leggere i caratteri dalla tastiera.

Tabella D-6. Le Funzioni di INT 16h

(AH)=0 **Legge dalla Tastiera.** Questa funzione aspetta che l'utente digiti un carattere sulla tastiera. Riporta il codice ASCII in AL e lo scan code in AH. Per i tasti estesi, AL sarà impostato a 0. Fate riferimento alla Tabella D-2 per una lista degli scan code.

- (AL) Codice ASCII del tasto premuto (0 per i tasti speciali).
- (AH) Scan Code del tasto premuto

(AH)=1 **Stato della Tastiera.** Questa funzione controlla se c'è qualche tasto da leggere.

- ZF 0, se c'è un carattere in attesa
 1, se non ci sono caratteri in attesa
- (AL) Codice ASCII del carattere da leggere
- (AH) Scan Code del carattere da leggere

(AH)=2 **Stato dello Shift.** Questa funzione ritorna un byte con lo stato dei vari tasti shift:

(AL)	Stato dei tasti shift:							
	7	6	5	4	3	2	1	
	1	Insert attivo
	.	1	Caps Lock attivo
	.	.	1	Num Lock attivo
	.	.	.	1	.	.	.	Scroll Lock attivo
	1	.	.	Tasto Alt premuto
	1	.	Tasto Shift di Sinistra premuto
	1	Tasto Shift di Destra premuto

Questa tabella contiene le funzioni dell'istruzione INT 21h utilizzate in questo libro. Per una lista più completa, dovrete comprare il manuale di riferimento tecnico del DOS.

Tabella D-7. Le Funzioni di INT 21h

- (AH)=1 **Inserimento da tastiera.** Questa funzione aspetta che digitiate un carattere sulla tastiera, lo visualizza sullo schermo, e riporta il codice ASCII nel registro AL. Per i codici estesi, questa funzione ritorna due caratteri: un ASCII 0 seguito dallo scan code (Vedere la Tabella D-2).
- (AL) Carattere letto dalla tastiera
- (AH)=2 **Visualizzazione sullo schermo.** Visualizza un carattere sullo schermo. Parecchi caratteri hanno un significato speciale in questa funzione:
- 7 Suono: manda un tono di un secondo all'altoparlante.
8 Backspace: sposta il cursore a sinistra di una posizione.
9 Tab: si sposta al tabulatore successivo. I tabulatori sono impostati ogni 8 caratteri.
0Ah Avanzamento Riga: si sposta alla riga successiva.
0Dh Ritorno a Capo: si sposta all'inizio della riga corrente.
(DL) Carattere da visualizzare sullo schermo.
- (AH)=8 **Input da tastiera senza eco.** Legge un carattere dalla tastiera, ma non lo visualizza sullo schermo.
- (AL) Carattere letto dalla tastiera
- (AH)=9 **Visualizza una stringa.** Visualizza la stringa puntata dai registri DS:DX. Dovete contrassegnare la fine della riga con un segno di dollaro (\$)
- DS:DX Punta alla stringa da visualizzare
- (AH)=0Ah **Legge la stringa.** Legge la stringa dalla tastiera. Vedere il Capitolo 23 per maggiori dettagli.
- (AH)=25h **Imposta il vettore di interrupt.** Imposta un vettore di interrupt per puntare ad una nuova routine.

(AL) Numero di interrupt
DS:DX Indirizzo del nuovo interrupt

(AH)=35h **Vettore di interrupt.** Ottiene l'indirizzo della routine di interrupt attraverso il numero di interrupt dato in AL.

(AL) Numero di interrupt
ES:BX Indirizzo dell'interrupt

(AH)=4Ch **Uscita al DOS.** Ritorna al DOS, come INT 20h, ma funziona sia per i programmi .COM e .EXE. La funzione INT 20h funziona solo con i programmi .COM.

(AL) Ritorna il codice. Normalmente importato a 0, ma è possibile impostarlo con un qualsiasi altro numero e utilizzare i comando DOS IF e ERRORLEVEL per trovare gli errori.

I seguenti due interrupt sono chiamate DOS per leggere e scrivere i settori dei dischi.

Tabella D-8. Funzioni per Leggere/Scrivere i Settori

INT 25h - Legge i Settori da Disco

In ingresso:

(AL)	Numero drive (0=A, 1=B, e così via)
(CX)	Numero di settori da leggere in una volta
(DX)	Numero del primo settore da leggere (il primo settore è 0)
DS:BX	Indirizzo di trasferimento: dove scrivere i settori letti.

INT 26h - Scrive i Settori su Disco

In ingresso:

(AL)	Numero drive (0=A, 1=B, e così via)
(CX)	Numero di settori da scrivere in una volta
(DX)	Numero del primo settore da scrivere (il primo settore è 0)
DS:BX	Indirizzo di trasferimento: inizio dei dati che si desiderano scrivere sul disco.

Informazioni ritornate da INT 25h, INT 26h

Sia INT 25h che INT 26h ritornano le seguenti informazioni nel registro AX. Lasciano anche un flag nello stack, in modo da poter utilizzare un'istruzione POP o POPF per rimuovere questa parola dallo stack (vedere il Capitolo 15 per un esempio).

Ritorni:

Flag di riporto	Impostato se c'è stato un errore, nel qual caso l'errore sarà in AX.
(AL)	codice di errore DOS
(AH)	Contiene una delle informazioni seguenti:
80h	Il drive non risponde
40h	L'operazione Seek è fallita
08h	CRC sbagliato durante la lettura
04h	Impossibile trovare il settore richiesto
03h	Tentativo di scrittura su disco protetto in scrittura
02h	Altri errori

Distrugge

AX, BX, CX, DX, SI, DI, BP

INDICE ANALITICO

.DATA, direttiva, 128
.DATA?, direttiva, 149
.MODEL direttiva, 106, 117
 linguaggi di alto livello e, 294
 PUBLIC e, 294

80286/80386, microprocessori, 3

A

A, comando di Debug, 35
Adattatore grafico a colori, memoria, 279
ADC, 43
ADD, 21
Addizione con riporto, 43
Addizione, aritmetica esadecimale, 5
Aggiungere, testo in grassetto, 135
AL, registro
 LODSB, istruzione, 162
 STOSB, istruzione, 285
AND, istruzione, 56
Area all'inizio del programma, PSP, 102
Aritmetica esadecimale, 5
ASCII, caratteri, 32
ASCII, codice, 367
ASCII, esteso, 370
Asm, file sorgente ".asm", 73
Assegnazione, direttiva EQU, 135
Assegnazione, istruzione MOV, 36
Assembla, comando, 35
Assemblatore
 automatico, 141
 commenti, 78

direttive, 75, 78
 ASSUME, 267, 277
 BYTE, 196
 .CODE, 104, 118, 267
 .DATA, 128
 .DATA?, 149
 DB, 130, 150
 DOSSEG 104
 DUP(?), 150
 END, 75
 ENDP e PROC 84, 295
 ENDS e SEGMENT, 267
 EQU, 135
 EXTRN, 117
 FAR e NEAR, 150
 .MODEL, 291
 NEAR e FAR, 105
 OFFSET, 196
 ORG, 268
 PUBLIC, 88, 294
 PROC e ENDP, 84, 295
 PTR, 196
 .STACK, 101
 SEGMENT, 267
 USES, 295
 WORD, 197
etichette, 79
output, file oggetto, 76
segmento, definizione completa, 267
ASSUME, direttiva, 267, 277
Attributi dei caratteri in memoria, 282
Attributi, carattere
 tabella dei colori, 185
 in memoria, 281

inverso e normale, 185
 WRITE_ATTRIBUTES_N_TIMES, 212, 285
 scrittura, 284
 scrittura di caratteri e, 186
 Attributo normale, 186
 AX, registro di uso generale, 20

B

B, numero binario, 14
 BACKSPACE, 235
 Barrato, testo, 135
 Base 16, esadecimale, 6
 Base 2, binaria, 14
 Basic input output system, BIOS, 169
 BASIC, comando CLS, 174
 BIOS, Basic input output system in ROM, 169
 INT 10h, funzioni VIDEO_IO, 169-170, 372
 funzione 2, imposta il cursore, 174
 funzione 3, legge la posizione del
 cursore, 188
 funzione 6, scorre di una pagina in alto,
 172, 372
 funzione 9, scrive car./attributo, 186
 INT 11h, flag di equipaggiamento, 282
 INT 13h, servizi del disco, 303
 INT 16h, servizi di tastiera, 198, 375
 Bit, 17
 gruppi di quattro, Nibble, 54
 impostare con OR, 95
 Borland Turbo Debugger, 256
 BP, registro, 297
 BS, costante, 237
 Bug, trovare in programmi estesi, 249
 Bug, trovare, 252
 BX, registro di uso generale, 19
 Byte e parole, 196
 BYTE PTR, 197
 Byte, 17
 BYTE, direttiva, 196

C

C e linguaggio assembly, 291
 C, nomi di procedure, 294
 C, nomi di variabili, 294
 C, parametri, 295, 300
 C, procedure
 CLEAR_SCREEN, 292
 GOTO_XY, 300
 READ_KEY, 301

 WRITE_STRING, 295
 C, uso dei registri, 293
 C, valori di ritorno, 301
 CALL, istruzione, 63
 NEAR e FAR, 105, 118
 segmenti, 105
 stack, 76
 CALL, lunga, 105
 Cambiare i registri in Debug, R, 20
 Cambiare la memoria in Debug, 21, 25
 Cancellare i caratteri, BACK_SPACE, 235
 Cancellare i registri con XOR, 93
 Cancellare le finestre, 173
 Cancellare lo schermo, 172
 CLS, 174
 dal C, 292
 Cancellazione caratteri, BACK_SPACE, 236
 Caratteri grafici, 367
 Caratteri, attributi,
 WRITE_ATTRIBUTE_N_TIMES, 208, 285
 Caricare un byte con LODSB, 62
 Caricare un settore, comando L, 111
 CGA, memoria video, 279
 CL, registro, 41, 60
 CLD, flag, 162
 CLD, istruzione, 162
 CLEAR_SCREEN per il C, 292
 CLEAR_SCREEN, 173
 CLIB.ASM, 292
 CLS, comando BASIC, 174
 CMP, istruzione di confronto, 51
 confrontare con 0 con OR, 94
 CodeView, 254
 cambio schermo, 254
 Codice di scansione, 59
 INT 16h, 198, 375
 Codice macchina, 20
 Codici dei caratteri, 367
 estesi, 370
 leggere una stringa, 237
 leggere con INT 16h, 198, 375
 leggere con READ_BYTE, 198, 229, 239
 codici speciali, 59
 scrivere gli attributi e, 186
 scrivere stringhe di, 180
 Codici dei colori, 185, 369
 Codici di tastiera estesi, 59, 370
 Collaudo, 4
 PUBLIC, 88, 294
 livello sorgente, 253
 simbolico, 254, 258
 tecniche di, 249, 252

Colori, tabella, 185
 EXE2BIN, 270
 e ASSUME, 267, 277
 e ORG, 269
 e segmenti, 268
 Combinare parole e byte, 196
 Combinare tipi di dati diversi, 196
 Commenti e progettazione modulare, 88
 Commenti, il ";", 78
 COMPAQ DOS 3.31, 147
 Condizioni di limite, 53, 61
 Condizioni di limite, stampare un numero in
 esadecimale, 53
 Confronto con OR, 94
 Controllare READ_BYTE con TEST, 224
 Controllare READ_DECIMAL, 231
 Conversione da binario a decimale, 93
 Conversione da decimale a esadecimale, 11
 Conversione da esadecimale a decimale, 7
 Conversione di numeri negativi in
 complemento a due, 17
 CONVERT_HEX_DIGIT, 225
 Costanti, CR, BS ed ESCAPE, 237
 Costanti, direttiva EQU, 135
 CR, costante, 237
 CR, ritorno a capo, 135
 CRLF, ritorno a capo, avanzamento riga, 134
 CS, segmento codice, 23, 97, 268
 CURRENT_SECTOR_NO, 176
 Cursor.asm, 135, 173, 187
 Corsore
 movimento, INT 10h, 2, 170, 174, 372
 posizione, leggere, 189
 spostare dal C, 299
 spostare il, 174
 spostare a destra, 187
 virtuale, 287
 Corsore virtuale, 287
 CURSOR_RIGHT, 187, 287
 CX, contatore, 121
 CX, registro di uso generale, 19
 CY, flag di riporto, 41

D

D, comando di Debug, 39

Dati

ASSUME, direttiva, 267, 277
 DISPATCH_TABLE, 196, 249
 modo di indirizzamento immediato, 125
 segmento, 276

DB, definisce byte, 130, 150

Debug, 4

e MS-DOS, 113
 comando G e punti di interruzione, 46
 tracciare, 107
 comando L, 111
 avviare e uscire, 5
 comando T, 23

Debugger

CodeView, 254
 Debug, 5
 Turbo, 275

Debugging simbolico, 253, 254

Decimale, conversione esadecimale in, 7

Decimale, conversione in esadecimale, 11

Definisce byte, direttiva DB, 130, 150

Definisce con la direttiva EQU, 135

Definizione completa di segmenti, 267

DI, registro, 93

Dimensione del disco, tabella, 111

Dimensione, disco, 111

Directory del disco, 111

Directory, avvio sul disco, 111

Directory, dischetto, 111

Direttive, 75

comandi dell'assemblatore, 75

ASSUME, 267, 277

BYTE, 213, 196

.DATA, 128

.DATA?, 149

DOSSEG, 104

DUP(?), 150

END, 75

ENDS, 267

SEGMENT, 267

.MODEL, 291

OFFSET, 196

ORG, 268

PROC e ENDP, 84, 295

PTR, 132, 196

PUBLIC, 88, 294

SMALL, 106

.STACK, 101

USES, 295

WORD, 197

Direzione, flag, 162

DISPATCH_TABLE, 193, 247

Disassemblare, 34

Dischetto, INT 13h, 303

Dischi fissi di grossa capacità, 147

Dischi fissi, leggere i settori, 147, 368

Dischi, numero di floppy, 306

Disco, settori, 111
 leggere i settori con INT 25h, 146, 378
 leggere con READ_SECTOR, 162
 scrivere, 247, 378
 scrivere settori modificati con F2, 247

Disklite, 305

DISK_DRIVE_IO, 176

Disk_io.asm, 144, 162, 174, 199, 248

DISK_PATCH, 176, 181-182, 194, 280

Dispatch.asm, 181-182, 215, 243, 247

Dispatcher, 192, 193, 243,

DISP_HALF_SECTOR, 137, 144

DISP_LINE, 126, 132, 138, 154

Disp_sec.asm, 126, 132, 144, 153, 175, 207

DIV, 29

Dividere la memoria in segmenti, 20

Divisione, 29
 resto, 11

Documentazione, 121

DOS 4.0, 147

DOS, funzione 25h, leggere i settori, 146, 378

DOS, uscire al, 99

DOSSEG e gruppi, 129

DS, segmento dati, 99

Dskpatch.asm, 176, 181-182, 194, 280

Due schermi
 CodeView, 254
 Turbo Debugger, 256

DUP(?), direttiva, 150

DX, registro di uso generale, 19

E

E, comando di Debug, 21, 25

Editor.asm, 220

EDITOR_PTOMPT, 193

EDIT_BYTE, 221

EGA, 279

END, direttiva, 75

END, uso in file sorgenti separati, 119

ENDP, direttiva, 84, 295

Enter, comando di Debug, 21, 25

EQU, direttiva, 135

Equipaggiamento, flag, 280

ERASE_PHANTOM, 211

Errori, collaudo, 4

Errori, flag di riporto, 122

ES, segmento extra, 99

Esadecimale, 6
 conversione decimale in, 11
 conversione in decimale, 7

numeri nell'assemblatore, 74
 origini, 6
 stampare in, 57
 leggere una singola cifra, 60

ESCAPE, costante, 237

esclusivo, OR, 94

Esecuzione passo-passo, 23
 breakpoint, 46

Esecuzione, passo a passo, 23

Eseguire, 23

Esterno, direttiva EXTRN, 117

Etichette, 79
 indirizzi, 21
 CodeView e, 254
 Turbo Debugger e, 256

EXE in COM, EXE2BIN, 271

EXE, file "exe" e "com", 103

EXE, file "exe", 76

EXE, programmi, rilocalazione, 268

EXE, programmi, stack, 102

Exe2bin, 76, 271

EXTRN, direttiva, 117
 collegare i file, 118

F

F1-F10, tasti funzione speciali, 59

F2, tasto per scrivere i settori modificati, 247

F3, legge settore precedente, 201

F4, legge settore successivo, 201

F10, esce da dskpatch, 201

FAR RET, 105

FAR, direttiva, 105

File binari, EXE2BIN, 270

File di comando, LINK, 249

File non convertibile, 264

File oggetto, output assemblatore, 76

File, directory dei, 111

File, formato make, 142

File, nomi in Debug, 37

File, scrivere in Debug, 37

Fine file, 363

Fine linea, cancellare fino, 189

Finestre, cancellarle, 173

Flag di overflow, 50

Flag di riporto, 41
 errori riportati con, 121

Flag di stato, 49, 122
 CMP, istruzione, 51
 JA, 69
 JB, 69

JL, 53
 JLE, 60
 JNZ, 51
 JZ, 50
 OR istruzioni, 94
 overflow, 50
 salvare e ripristinare, 162
 Flag, registri, 106
 di riporto, 41
 di direzione, 162
 istruzione INT e, 106
 IRET, 107
 di overflow, 50
 istruzione POPF e, 148
 del segno, 50
 salvare e caricare, 67, 162
 zero, 49
 Floppy disk
 directory, 111
 numero di, 305
 settori, 111
 leggere con INT 25h, 147, 378
 leggere con READ_SECTOR, 161
 scrivere, 247, 378
 FOR-NEXT, istruzioni, 44
 Funzioni di tastiera, 375
 Funzioni DOS, 376-8

G

G, comando di Debug, 31, 33
 punti di interruzione, 46
 comando P, 47
 GET_NUM_FLOPPIES, 305
 Go, comando, 31
 GOSUB
 istruzione CALL, 63
 procedure, 63
 INT, 31
 GOTO_XY, 174, 286
 per il C, 300
 Grassetto, testo, 4, 135
 Gruppi, 130

H

H, aritmetica esadecimale, 5
 numeri esadecimali, 7
 H, per i numeri esadecimali nell'assemblatore,
 74
 Hardware installato, 280

HEADER_LINE_NO, 177
 HEADER_PART_1, 177
 HEADER_PART_2, 177
 Hercules, 279
 HEX_TO_BYTE, 227

I

IF-THEN, salti condizionali, 50
 istruzione CMP, 51
 flag di stato, 51
 Impostare i bit con OR, 95
 Impostare i vettori INT, 305
 INC, istruzione, 64
 Incrementare, INC, 64
 Indice di destinazione, registro, 93
 Indice, registri SI e DI, 93
 Indirizzamento di base relativa, 127, 130
 Indirizzamento di memoria indiretto, 127
 Indirizzamento, CS:IP, 97
 Indirizzi
 CALL e segmenti, 105
 CS:IP, 97
 effettivi e LEA, 146
 vettori di interrupt, 108, 305
 etichette, 79
 locazioni di memoria, 22
 mappa dei file, 250
 memoria, 21
 modi, 125, 130, 371
 indicizzato, 131
 relativo, 127, 130
 diretto, 130
 diretto indicizzato, 131
 immediato, 131
 indiretto, 127, 131
 registro, 130
 registro indiretto, 130
 tabella, 130, 371
 OFFSET, direttiva, 196
 PUBLIC, direttiva, 88, 294
 RET e segmenti, 105
 segmenti, 97, 99, 269
 Indirizzo effettivo, LEA, 146
 INIT_SEC_DISP, 161, 173, 207
 INIT_WRITE_CHAR, 280
 Inizio dello stack, 101
 Inserimento da tastiera, 68
 Inserimento da tastiera, funzione INT 21h, 1, 59
 Inserimento programmi, 35
 Installazione di programmi residenti in RAM,

305
 INT, istruzione, 31, 106
 INT 1, interrupt passo a passo, 107
 INT 10h, funzioni, 169-170, 372
 funzione 2, imposta la posizione del cursore, 71
 funzione 3, legge la posizione del cursore, 188
 funzione 6, scorre di una pagina verso l'alto, 172, 372
 funzione 9, scrive carattere/attributo, 186
 INT 13h, funzioni del disco, 303
 INT 16h, servizi di tastiera, 198, 375
 INT 20h, 33
 INT 21h, 31, 376
 funzione 1, legge carattere, 59
 funzione 8, leggere dei caratteri senza visualizzarli, 68
 funzione 9, scrittura stringa, 38
 funzione 25h, lettura vettori INT, 305
 funzione 35h, impostazione vettori INT, 305
 funzione 4Ch, uscita al DOS, 99
 INT 25h, lettura di un settore del disco, 146, 378
 INT 26h, scrittura di un settore del disco, 274, 278
 INT 27h, finisce ma resta residente, 305
 simulazione, 304
 comando P, 47
 Intel, 97
 Intercettare vettori di interrupt, 303
 Interrupt
 clock, 106
 istruzione INT, 106
 ritorno da, 107
 stack dopo un, 107
 dimensione stack, 305
 vettori, 21
 intercettare, 303
 leggere e impostare, 305
 Interrupt del clock, 107
 Intestazioni, 122, 179
 Intrasegment CALL, 105
 Intrasegment RET, 105
 IP, puntatore di istruzioni, 97
 IP, registro, 97
 IRET, ritorno da interrupt, 107
 Istruzioni logiche, AND, 56
 Istruzioni stringa
 LODSB, 162
 STOSB, 285

Istruzioni, linguaggio macchina, 20
 LEA, 145
 LODSB, 162
 segmento, 275
 STOSB, 285

J

JA, salta se sopra, 69
 JB, salta se sotto, 69
 JL, salta se minore di, 53
 JLE, salta se minore o uguale di, 60
 JNZ, salta se non è 0, 51
 JZ, salta se è 0, 50
 Kbd_io.asm, 197, 226, 236
 Kludge, 112, 97
 Kluge, 97

L

L, comando di Debug, 111
 LEA, istruzione, 145
 Legge i caratteri
 INT 21h funzione 1, 59
 READ_BYTE, 198, 229, 239
 stringhe di caratteri, 237
 senza echo, 68
 Legge il settore precedente, F3, 201
 Legge il settore successivo, F4, 201
 Legge la posizione del cursore, 187
 Legge un settore, 378
 Legge una stringa di caratteri, 237
 Leggere cifre esadecimali, 60
 Leggere dalla tastiera, 197
 Leggere i settori
 Debug L, 111
 funzione DOS 25h, 146, 378
 PREVIOUS_SECTOR e NEXT_SECTOR, 199
 READ_SECTOR, 162
 Leggere i settori del disco, funzione DOS INT 25h, 146, 378
 Leggere la memoria, LODSB, 162
 Leggere vettori INT, 305
 Leggi, progettazione modulare, 120
 LET, istruzione, 36
 LF, avanzamento riga, 134
 LIFO, 65
 stack, 65
 Line di comando, scrittura, 198
 LINES_BEFORE_SECTOR, 176
 Linguaggi ad alto livello, 294

.MODEL, 291
 Linguaggio macchina, 20, 23
 LINK, 76
 risposta automatica, 250
 mappa dei file e, 250
 /map, 250
 direttiva PUBLIC, 88, 294
 ordine di caricamento, 127
 file separati, 119, 127
 file insieme, 127
 Listare un programma, comando U, 34
 LOCATE, movimento del cursore, 174
 Localizzazione di memoria, indirizzi, 21
 LODSB, istruzione, 62
 LOOP, 44
 LOOP, istruzione, 44
 Luce, del disco, 305

M

Macro, 301
 Make, 141
 formato file Make, 142
 Makefile, nuova versione, 200
 Mappa dei file, creare, 250
 Mappa, 250
 MASM
 direttiva ASSUME, 267, 277
 messaggi di errore, 363
 ordine di caricamento dei segmenti, 127
 segmenti, 275
 Memoria video, 279
 Memoria, 20
 indirizzamento con CS:IP, 97
 modi di indirizzamento, 125, 371
 direttiva ASSUME, 267, 277
 indirizzamento relativo, 127, 130
 CodeView e, 254
 segmento dati, 99, 128
 direttiva DB, 130, 150
 indirizzamento diretto, 130
 dividere in segmenti, 20
 modificare con EDIT_BYTE, 221
 gruppi, 129
 come sono memorizzate le parole, 95
 indirizzamento indiretto, 127
 locazione, 21
 etichette per, 21
 mappa, 250
 modelli, 117
 .MODEL, 106, 291

 scarto, 22, 195
 ordine dei segmenti, 127
 ROM chip, 169
 video e attributi, 279-282
 registri di segmento, 99
 segmentazione, 97
 lo stack nella, 101
 Turbo Debugger e, 256
 scrivere in con WRITE_TO_MEMORY, 220
 Messaggi di errore
 EXE2BIN, 365
 LINK, 364
 MASM, 363
 cause possibili, 363
 Messaggi di errore comuni, 363
 MG, flag di segno, 50
 Microsoft CodeView, 254
 Microsoft e Debug, 113
 Mnemonico, 36
 Modificare la memoria, EDIT_BYTE, 221
 Modo di indirizzamento diretto, 130
 Modo immediato, 131
 Moltiplicare due parole, 40
 Moltiplicazione, 28
 per spostamento, SHL, 60
 MOV, 36
 LODSB, 162
 STOSB, 285
 MOVE_TO_ASCII_POSITION, 210
 MOVE_TO_HEX_POSITION, 209
 MS-DOS e debug, 113
 MUL, 28

N

N, comando di Debug, 38
 Near CALL, 105
 NEAR e FAR, procedure, 118
 Near RET, 105
 NEAR, direttiva, 105
 NEAR, etichette, 79
 NEXT_SECTOR, 199
 Nibble, un gruppo di quattro bit, 54
 Nidificazione blocco, errore, 363
 Nomi e Code View, 254
 Nomi e Turbo Debugger, 256
 Nomi in C, 288
 Nomi in Debug, 38
 Numeri binari, 14
 conversione in decimale, 91
 gruppo di quattro bit, nibble, 54

Numeri decimali, conversione, 91
 Numeri negativi, 16, 25
 flag di segno, 50
 bit di segno, 16
 Numeri positivi, flag di overflow, 50
 Numeri positivi, flag di segno, 50
 Numeri senza segno, 14
 JA e JB, 69
 flag di overflow, 50
 Numeri, conversione da binario a decimale, 91
 Numeri, flag di segno, 59
 Numero delle funzione per VIDEO_IO, 170, 372
 funzione 2, imposta la posizione del cursore,
 174
 funzione 3, legge la posizione del cursore,
 188
 funzione 6, scorre di una pagina verso l'alto,
 172, 372
 funzione 9, scrive carattere/attributo, 186
 Numero, flag di overflow, 59
 Nuovi programmi, punto d'inizio, 88
 NV, flag di overflow, 59
 NZ, flag zero, 58

O

OBJ, file ".obj", 90
 Offset nel segmento, 22, 196
 OFFSET, direttiva, 196
 Operazioni logiche, XOR, 94
 OR, istruzione, 94
 CMP un numero con 0, 94
 Ordine dei segmenti, 128
 Ordine di caricamento, Link, 127
 ORG, direttiva, 268
 OV, flag di overflow, 50

P

P, il comando per procedere, 47
 Parametri e BP, 298
 Parametri e C, 295, 300
 Parola, 16
 Parola, moltiplicazione, 30
 Parole, come sono salvate in memoria, 95
 Passare i parametri e BP, 298
 Passare le informazioni standard, 121
 PC-DOS e Debug, 113
 Phantom.asm, 208, 216, 261
 PHANTOM_CURSOR_Y, 208
 PHANTOM_DOWN, 216, 261

PHANTOM_LEFT, 216
 PHANTOM_RIGHT, 217
 PHANTOM_UP, 215, 261
 PL, flag di segno, 50
 POP e PUSH per salvare e richiamare i registri,
 121
 POPF, istruzione, 147, 155
 POS, leggere la posizione del cursore, 188
 Posizione, leggere il cursore, 188
 Preservare i registri in C, 293
 PREVIOUS_SECTOR, 199
 PRINT, funzione INT 21h, 8, 39
 Printaj.asm, 84
 PRINT_A_J, 83
 PROC e i parametri, 295, 299
 PROC, direttiva, 84, 117, 295
 Procedure in linguaggio assembly per il C, 291
 Procedure, 63
 indirizzi, direttiva OFFSET, 196
 Code View e, 254
 esterne, 117
 trovarle in memoria, 252
 variabili locali, 68
 renderle corte, 121
 .MODEL, 106, 291
 nomi in C, 294
 NEAR e FAR, 118
 parametri, 295, 300
 PROC e ENDP, 84, 295
 valori ritornati, 301
 salvare e ripristinare i registri, 67, 121
 Turbo Debugger e, 256
 Procedure, sorgenti
 BACK_SPACE, 235
 CLEAR_SCREEN, 173
 CLEAR_SCREEN in C, 291
 CLEAR_TO_END_OF_LINE, 189, 190
 CONVERT_HEX_DIGIT, 225
 CURSOR_RIGHT, 187, 285
 DISK_PATCH, 193, 243
 DISPATCHER, 193, 243
 EDIT_BYTE, 221
 ERASE_PHANTOM, 211
 GET_NUM_FLOPPIES, 306
 GOTO_XY, 286
 GOTO_XY in C, 300
 HEX_TO_BYTE, 227
 INIT_SEC_DISP, 161, 207
 INIT_WRITE_CHAR, 280
 MOV_TO_ASCII_POSITION, 210
 MOV_HEX_POSITION, 209
 NEXT_SECTOR, 199

PHANTOM_DOWN, 216, 262
 PHANTOM_LEFT, 216
 PHANTOM_RIGHT, 217
 PHANTOM_UP, 216, 262
 PREVIOUS_SECTOR, 199
 READ_DECIMAL, 231
 READ_KEY, 229
 READ_KEY in C, 301
 READ_STRING, 228, 238, 289
 RESTORE_REAL_CURSOR, 209
 SAVE_REAL_CURSOR, 209
 SCROLL_DOWN, 263
 SCROLL_UP, 262
 SEND_CRLF, 226
 STRING_TO_UPER, 227
 TEST, 224, 232
 UPDATE_REAL_CURSOR, 287
 UPDATE_VIRTUAL_CURSOR, 287
 WRITE_ATTRIBUTE_N_TIMES, 208
 WRITE_CHAR, 208, 283
 WRITE_HEADER, 179, 190
 WRITE_PHANTOM, 210
 WRITE_PROMPT_LINE, 198
 WRITE_SECTOR, 249
 WRITE_STRING, 180
 WRITE-STRING in C, 295
 WRITE_TO_MEMORY, 220
 Progettatori dell'8088, Intel, 97
 Progettazione modulare, 120
 blocchi di commento, 88
 Program Segment Prefix, 102
 Programmi estesi, 117, 127
 collaudo, 249
 Programmi residenti, 303
 Programmi, Disklite, 305
 Programmi, file sorgente, 76
 Programmi, residenti in RAM, 303
 Programmi, scheletro, 89
 Programmi, tracciamento con il comando P, 47
 PROMPT_LINE_NO, 193
 Prossima istruzione, 24
 Prossimo settore, F4, 201
 PSP all'inizio dei programmi, 102
 PSP, Program Segment Prefix, 101
 PTR, direttiva, 148, 196
 PUBLIC, direttiva, 88
 C e, 294
 map file e, 250
 Puntatore di Istruzioni, 24
 registro IP, 97
 Punti di interruzione per Debug, 46
 PUSH e POP, salvare e ripristinare i registri, 121

PUSHF, istruzione, 155

R

R, comando Debug, 19
 cambiare i registri un byte, 26
 cambiare i registri, 19
 RAM, programmi residenti, 303
 RCL, 42
 Read Only Memory, ROM, 169
 READ_BYTE, 198, 229, 239
 READ_DECIMAL, 231
 controllare, 232
 READ_KEY in C, 301
 READ_KEY, 229
 READ_SECTOR, 145, 162, 177
 READ_STRING, 226, 237, 289
 REAL_CURSOR_X, 208
 REAL_CURSOR_Y, 208
 Registri di byte, cambiare in Debug, 26
 Registri di visualizzazione, 19
 Registri, 19
 ASSUME, segmento, 267, 277
 BP, 297
 cambiare i byte con il comando R di Debug, 26
 cambiarli nel Debug, 20
 CS, 97
 visualizzarli con il comando R di Debug, 19
 flag, 107
 IP, 97
 modalità
 indicizzata di base, 131
 relativa di base, 127, 130
 modalità diretta, 130
 indicizzata diretta, 131
 immediata, 131
 memoria indiretta, 131
 registri, 125
 registri indiretti, 130
 salvarli e ripristinarli, 67, 121
 segmento, 13
 ASSUME, 267, 277
 registri SI e DI, 93
 utilizzo, 122
 in C, 293
 Registro di stato, POPF, 151
 vedere anche flag di stato
 RESTORE_REAL_CURSOR, 209
 RET, istruzione, 63
 NEAR e FAR, 118

segmenti, 105
 lo stack, 65
 RET, lungo, 105
 RETURN, RET, istruzioni, 63
 Riga di comando, 103
 Riga, prompt, 198
 Righe, cancellare fino alla fine, 189
 Rilocazione, 268
 Rimuovere gli errori, collaudo, 4
 Ripristinare i flag, POPF, 155
 Ripristinare i registri dallo stack, 67
 Ripristinare i registri in C, 293
 Risposta automatica, LINK, 250
 Ritornare i valori in C, 301
 Ritorno da Interrupt, IRET, 107
 ROM BIOS, funzioni
 INT 10h funzione VIDEO_IO, 170, 372
 INT 13h funzioni del disco, 303
 INT 16h servizi per la tastiera, 198, 375
 ROM, Read Only Memory, 169
 Rotazioni attraverso il riporto, 42
 Rotazioni, istruzioni, 42
 Rotazioni, SHL, 54
 Routine, nella ROM BIOS, 169

S

Salto condizionale, istruzioni, 50
 JA, salta se sopra, 69
 JB, salta se sotto, 69
 JL, salta se minore di, 53
 JLE, salta se minore o uguale, 60
 JNZ, salta se non è zero, 51
 JZ, salta se zero, 50
 Salvare e ripristinare i registri, 67, 88, 121
 Salvare i flag con l'istruzione INT, 106
 Salvare i flag, PUSHF, 155
 Salvare i registri in C, 293
 Salvare i registri nello stack, 67
 Salvare un file sul disco da Debug, 37
 Salvataggio temporaneo, lo stack, 65, 101
 SAVE_REAL_CURSOR, 210
 Scan code, vedi Codice di scansione
 Schermo, cancellare lo, 172
 Schermo, cancellarlo in C, 292
 Schermo, funzioni, 372
 vedere anche ROM BIOS
 Schermo, memoria, 279
 organizzazione, 281
 Schermo, scrittura veloce con GOTO_XY, 286
 Schermo, scrittura veloce con WRITE_CHAR,

283
 Schermo, utilizzare la ROM BIOS con, 172
 Scorrere la visualizzazione del settore, 262
 SCREEN_PTR, 284
 SCREEN_SEG, 282
 SCREEN_X e SCREEN_Y, 284
 Scrittura rapida sullo schermo
 GOTO_XY, 286
 INIT_WRITE_CHAR, 280
 Scrittura settore, 378
 Scrivere gli attributi,
 WRITE_ATTRIBUTE_N_TIMES, 208
 Scrivere i caratteri e gli attributi, 186
 Scrivere i settori del disco, 247, 378
 Scrivere i settori modificati, tasto F2, 247
 Scrivere in memoria, STOSB, 285
 Scrivere in memoria, WRITE_TO_MEMORY, 220
 Scrivere le stringhe di caratteri, 180
 Scrivere un file in Debug, 37
 Scrivere una stringa, 38
 SCROLL_UP e SCROLL_DOWN, 263
 SECTOR, 126, 135, 177
 SECTOR_OFFSET, 176
 SEG, sovrascrittura segmento, 275
 SEGMENT, direttiva, 267
 Segmenti frammentati, 364
 Segmenti multipli, 275
 Segmenti, 20, 97
 ASSUME, 267, 277
 CALL e RET, 105
 .CODE, 104, 118, 267
 .DATA, 128
 .DATA?, 149
 frammentati, 364
 definizione, 267
 gruppi, 130
 multipli, 275
 NEAR, 79
 NEAR e FAR, 105
 .STACK, 101
 .TEXT, 101
 Segmenti, direttive
 .CODE, 104, 118, 267
 .DATA, 128
 .STACK, 101
 Segmento Codice, 23, 99, 267
 .TEXT, 116, 101
 registro, CS, 97
 Segmento dati, 99, 267
 e gruppi, 129
 per le variabili di memoria, 128
 multipli, 275

Segmento extra, 99
 Segmento, registri, 99
 CS, 97
 DS, 99
 Segmento, sovrascrittura
 ASSUME, 277
 istruzioni, 275
 SEND_CRLF, 134, 289
 Senza fine, vedi ciclo
 Separazione file, 117
 Link, 76
 linking, 119
 design modulare, 120
 Settore precedente, F3, 201
 Settori modificati, riscriverli con F2, 247
 Settori per disco, 111
 Settori, disco, 111
 modificarli con EDIT_BYTE, 221
 tasto F2, scriverle modifiche, 247
 precedente e successivo con F3 e F4, 201
 leggerli, 379
 funzione DOS INT 25h, 146, 378
 PREVIOUS_SECTOR e
 NEXT_SECTOR, 199
 READ_SECTOR, 162
 scrivere il disco, 247, 378
 SHL, spostare l'istruzione, 55
 Short CALL, 105
 Short RET, 105
 SHR, spostamento, 55
 SI, registro, 93
 Simboli definiti più di una volta, 364
 Simbolo non definito, 363
 Simulare l'INT, 304
 SMALL, modello di memoria, 106
 .MODEL, direttiva, 117
 Software interrupt, istruzione INT, 106
 Sorgente, file, 73
 Cursor.asm, 135, 173, 187
 Disk_io.asm, 144, 162, 174, 199, 248
 Dispatch.asm, 181-182, 215, 243, 247
 Disp_sec.asm, 132, 144, 153, 175, 207
 Dskpatch.asm, 176, 193, 280
 Editor.asm, 220
 Kbd_io.asm, 197, 226, 236
 Phantom.asm, 208, 216, 261
 Test.asm, 117, 224, 232
 Test_seg.asm, 99
 Video_io.asm, 119, 134, 157, 181, 186, 213, 281
 Sottrazione,
 l'istruzione CMP, 60

SP, puntatore stack, 65, 101
 Spostamento aritmetico, SHL, 55, 60
 Spostamento, l'istruzione MOV, 36
 Spostare il cursore
 CURSOR_RIGHT, 187, 287
 GOTO_XY, 174, 286
 INT 10h funzione 2, 167-170, 372
 SS, segmento stack, 65, 99
 SS:SP, inizio dello stack, 101
 Stack, 65
 dopo l'istruzione INT, 107
 BP e, 297
 programmi .EXE e, 102
 gruppi e, 130
 LIFO, 65
 puntatore, SP, 65, 101
 salvare e ripristinare i registri, 67
 salvare i flag nello, 106
 segmento, 65, 99
 inizio dello stack, 101
 .STACK, direttiva, 101
 Stampare in esadecimale, 57
 Standard, 120
 STore String Byte, STOSB, istruzione, 285
 STOSB, istruzione, 285
 Stringa, leggere la, 237
 Stringa, scriverla con WRITE_STRING, 295
 Stringa, scriverla in C, 295
 STRING_TO_UPPER, 227
 SUB, 25
 Subroutine o procedure, 63
 Switch, LINK e /map, 250

T

Tabella
 caratteri, 387
 funzioni ROM BIOS per VIDEO_IO, 170, 372
 modi di indirizzamento, 371
 codici dei colori, 185, 369
 dimensione del disco, 111
 codici estesi della tastiera, 370
 funzioni INT 10h, 169-170, 372
 funzioni INT 16h, 198, 375
 funzioni INT 21h, 376
Tasti funzione
 codici dei caratteri, 59
 F2, tasto per scrivere i settori modificati, 247
 F3, legge settore precedente, 201

F4, legge settore successivo, 201
 F10, esce da dskpatch, 201
 Tasti funzione speciali
 input da tastiera, 59
 leggere i settori con READ_BYTE, 198,
 229, 239
 tabella, 370
 Tasti, 59
 Tastiera, codici estesi, 370
 Tastiera, leggere dal C, 301
 Tastiera, leggere, 197
 TEST, 224, 232
 Test.asm, 224, 232
 Testo da aggiungere, 135
 Testo da cancellare, 135
 Test_seg.asm, 99
 TEST_WRITE_DECIMAL, 93, 117
 TEST_WRITE_HEX, 85
 Tipi di dati, 196
 Trattino, prompt di Debug, 5
 Tre leggi del design modulare, 120
 Trovare le procedure in memoria, 252
 TSR, 306
 TSR, programmi, 303
 Turbo Assembler, 257, 292
 Turbo Debugger, 256

U

U, comando Debug, 34
 UPDATE_REAL_CURSOR, 287
 UPDATE_VIRTUAL_CURSOR, 287
 Uscire al DOS, INT 21h, funzione 4Ch, 99
 Uscire da Dskpatch - F10, 201
 Uscire da Dskpatch, 201
 Uscire, INT 20h, 33
 Uso generale, registri, 19

V

Valori di ritorno, 301
 Variabili
 modalità di indirizzamento, 125
 segmento dati, 128

DB, DW direttive, 130, 150
 etichette, 79
 memoria, 17
 registri come, 19
 non inizializzate con .DATA?, 149
 Variabili di memoria, 176
 .DATA?, 149
 CURRENT_SECTOR_NO, 176
 DISK_DRIVE_NO, 176
 DISPATCH_TABLE, 193, 247
 EDITOR_PROMPT, 193
 PHANTOM_CURSOR_X, 208
 PHANTOM_CURSOR_Y, 208
 PROMPT_LINE_NO, 284
 REAL_CURSOR_X, 208
 REAL_CURSOR_Y, 208
 SCREEN_PTR, 284
 SCREEN_SEG, 282
 SCREEN_X e SCREEN_Y, 284
 SECTOR, 135
 Variabili e CodeView, 254
 Variabili e Turbo Debugger, 256
 Variabili locali, 68, 89
 Variabili non inizializzate, .DATA?, 149
 Variabili, Dskpatch
 BOTTOM_LINE_PATTERN, 160
 CURRENT_SECTOR_NO, 176
 DISK_DRIVE_NO, 176
 DISKPATCH_TABLE, 196, 249
 EDITOR_PROMPT, 193
 HEADER_LINE_NO, 177
 HEADER_PART_1, 177
 HEADER_PART_2, 177
 LINES_BEFORE_SECTOR, 176
 PHANTOM_CURSOR_X, 208
 PHANTOM_CURSOR_Y, 208
 PROMPT_LINE_NO, 193
 REAL_CURSOR_X, 208
 REAL_CURSOR_Y, 208
 SECTOR, 126, 135, 177
 SECTOR_OFFSET, 176
 SCREEN_PTR, 284
 SCREEN_SEG, 282
 SCREEN_X e SCREEN_Y, 284

Finito di stampare nel mese di Settembre 1990
presso la "Grafica '85 print"
Rodano Millepini - Milano



**GRUPPO
EDITORIALE
JACKSON**

Cod. R935

PETER NORTON

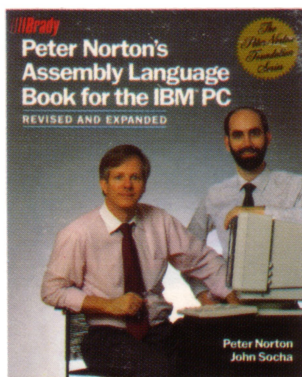
Linguaggio Assembly per PC IBM

Peter Norton
John Socha

La programmazione in Assembler, conosciuta come ostica ed impegnativa ma spesso risolutiva quando si richiedono procedure veloci e compatte, viene trattata in questo libro in modo sorprendentemente amichevole. La semplicità del linguaggio utilizzato non è però a scapito della completezza e del rigore, come altrimenti non potrebbe essere quando l'autore si chiama Peter Norton. Il libro è corredato di molti esempi e di routine pronte all'uso, che il lettore potrà analizzare, e modificare a piacere, perché presenti sul dischetto allegato.

Sommario

- Il linguaggio macchina: aritmetica dell'8088, visualizzazione e lettura di numeri e caratteri, le procedure.
- L'Assembler: procedure, progettazione modulare, visualizzazione dei settori del disco e della RAM.
- Le ROM BIOS dell' IBM PC
- Procedure C in Assembler
- Tecniche avanzate



GRUPPO EDITORIALE JACKSON

L. 75.000

Cod. R935

ISBN 88-256-0151-4



9 788825 601510

5 PETER NORSTON Lingua Asseribly per PC IBM

SECONDA
EDIZIONE

