di Sergio Polini

Quando si può attivare un TSR

La volta scorsa abbiamo visto cosa è un programma residente e abbiamo accennato alla sua attivazione mediante un interrupt. Ci siamo però anche soffermati sugli aspetti che rendono pericolosa un'attivazione poco cauta: dal momento che le funzioni del BIOS e del DOS non sono «rientranti», si possono verificare situazioni di blocco del sistema; si può arrivare a non poter proseguire in altro modo che resettando la macchina. Occorre quindi essere «cauti». Ora vedremo in concreto cosa questo voglia dire

Avrete modo di constatare che la scrittura di programmi residenti non è un'impresa banale. La unit TSR ha anzi proprio lo scopo di racchiudere in un unico file tutte le complicazioni, lasciando a disposizione dell'utente niente altro che una semplicissima **interface**: quattro variabili, una procedura di installazione, una funzione da usare in luogo della familiare *IOResult*.

Ad essa si accompagna però una **implementation** decisamente più complessa, tanto che saremo costretti ad esaminarne i dettagli in più riprese. Per questo motivo, la prima cosa che vi propongo è lo «scheletro» del file TSR.PAS (figura 1); vi invito a conservarlo con cura, in modo da poter poi ricostruire il file completo con i vari pezzi che vedremo in questa e nelle prossime puntate. Ricordo comunque che, come al solito, i sorgenti completi (TSR.PAS, TSRINT.ASM e TSRDEMO.PAS) sono già disponibili su MC-Link nel file TSRTP100.ZIP.

La interface della unit TSR

Ho cercato di rendere quanto più agile possibile l'uso della unit (figura 2): per le applicazioni più semplici può essere sufficiente la sola procedura *Installa*. Questa ha cinque parametri: *Nome* non è altro che una stringa con il nome del programma, che viene usata solo nei messaggi che confermano o meno l'avvenuta installazione; *Prog* è un parametro di tipo **procedure** con il quale viene passata la procedura che dovrà essere eseguita dal programma residente; *ID* è il byte di identificazione del programma, ad uso dell'INT 2Fh; in *Scan* e *Shift* si passa la combinazione di tasti con la quale si vuole che il programma venga attivato. Ad esempio, con

```
Installa('DUMP',Dump,$F1,$3B,8);
```

si installa il programma in modo che verrà eseguita la procedura *Dump* ogni volta che si premerà Alt-F1. Se si vuole attivare il programma residente con i soli tasti di shift (Shift destro e sinistro, Ctrl e Alt, i cui codici - 1, 2, 4 e 8 - possono anche essere sommati tra loro), *Scan* deve essere zero. Ad esempio:

```
Installa('DUMP',Dump,$F1,0,6);
```

per i tasti Ctrl (4) e Shift sinistro (2).

Rimandiamo per il momento una discussione esauriente degli ultimi tre parametri; per ora ci limiteremo ad osservare che la procedura passata in *Prog* sarà, per così dire, il corpo principale del programma residente: sarà equivalente al blocco **begin end** con il quale si chiude un programma normale, dal quale un programma normale inizia l'esecuzione. C'e una sola regola da osservare:

```
interface
  uses
  type
  var
  procedure Installa
  function ErroreIO
implementation
  (*$L TSRINT*)
  type
  var
  procedure CLI
  procedure STI
  procedure PUSHF
  procedure FSAVE
  procedure FRSTOR
  procedure GetIntVec10
  procedure GetIntVec13
  function GetIntVec2F
  procedure SetIntVec10
  procedure SetIntVec13
  procedure SetIntVec2F
  procedure SetInt2FVuoto
  procedure NuovoStack
  procedure PrevStack
  function InDOS
  function In8259
  function InBIOS
  function TSRAttivabile
  procedure Beep
  procedure NuovoInt5
  procedure NuovoInt8
  procedure NuovoInt9
  procedure NuovoInt1B
  procedure NuovoInt23
  procedure NuovoInt24
  procedure NuovoInt28
  procedure GetInfoEstesaErrori
  procedure SetInfoEstesaErrori
  procedure GetPSP
  procedure SetPSP
  procedure GetDTA
  procedure SetDTA
  procedure GetInfoVideo
  procedure GetStatoVideo
  procedure SetStatoVideo
  procedure EseguiTSR
  function CercaFlag
  procedure Installa
  function ErroreIO
```

Figura 1 - Lo «scheletro» del file TSR.PAS. Vi potrà essere utile per ricomporre il file completo con i «pezzi» che vedremo in questo e nei prossimi numeri.

i parametri di tipo procedure devono essere **procedure** chiamate con una *far call*, e quindi, per restare al nostro esempio, *Dump* dovrà essere preceduta da una direttiva $F+ se dichiarata nello stesso file in cui viene chiamata *Installa*.

La funzione *ErroreIO* viene in aiuto in quei programmi che vogliono poter eseguire senza rischi operazioni di lettura da disco o di scrittura su disco e stampante o altre periferiche. È evidente che, se abbiamo già sottolineato in passato l'importanza della gestione degli errori critici in programmi normali, un programma residente è anche più esigente al riguardo: non si può certo tollerare che il programma termini in modo irregolare in conseguenza di un drive aperto o di una stampante senza carta. Il Turbo Pascal, fin dalla versione 4.0, consente una qualche gestione degli errori critici, in modo non completissimo ma certo sufficiente: la funzione *IOResult* ritorna tra gli altri alcuni codici di errore (quelli tra 150 e 162) riservati appunto agli errori critici. *ErroreIO* fa esattamente la stessa cosa, al punto che dovrebbe essere usata proprio come *IOResult* e in sostituzione di questa: ritorna zero se non si sono verificati errori, oppure un numero, la cui decodifica è la stessa che trovate nelle appendici del manuale del Turbo Pascal 5.x. Vedremo in un prossimo appuntamento come ciò è stato realizzato, e come interpretare certe «stranezze» che capitano con le versioni 2.x del DOS (lasciatemi precisare subito che si tratta di comportamenti da attribuire più alle imperfezioni di quelle versioni, superate nelle successive, che ad anomalie della unit).

Un programma residente, così come non può morire a causa di un errore critico, non può neppure terminare per un Ctrl-C o un Ctrl-Break; vengono quindi intercettati gli interrupt 23h e 1Bh, in modo che non abbiano altro effetto che quello di rendere TRUE, rispettivamente, le variabili *CtrlC* e *CtrlBreak*. Anche esse sono nella **interface**, al fine di consentire al programma che «usa» la unit TSR di riconoscere queste situazioni (ne vedremo un esempio quando esamineremo il programma TSRDEMO PAS).

Un programma residente può limitarsi ad operare con somma discrezione, senza alterare il video (pensate ad un programma che si limiti a creare un file contenente una copia dello schermo), oppure può produrre proprie schermate; in quest'ultimo caso è evidente che, quando ne usciremo, dovranno essere

```
implementation
(*$L TSRINT*)

type
  BytePtr = ^byte;
  RecInfoEstesaErrori = record AX,BX,CX,DX,SI,DI,DS,ES,W1,W2,W3: word end;

var
  MultiplexID       : byte;          (* byte di identificazione per INT 2Fh *)
  InInt5            : word;          (* > 0 se attivo INT 05h *)
  InInt8            : word;          (* > 0 se attivo INT 08h *)
  InInt9            : word;          (* > 0 se attivo INT 09h *)
  InInt10           : word;          (* > 0 se attivo INT 10h *)
  InInt13           : word;          (* > 0 se attivo INT 13h *)
  InInt28           : word;          (* > 0 se attivo INT 28h *)
  PrevInt5          : procedure;     (* INT 5 originario *)
  PrevInt8          : procedure;     (* INT 8 originario *)
  PrevInt9          : procedure;     (* INT 9 originario *)
  PrevInt1B         : procedure;     (* INT 1Bh originario *)
  PrevInt23         : procedure;     (* INT 23h originario *)
  PrevInt24         : procedure;     (* INT 24h originario *)
  PrevInt28         : procedure;     (* INT 28h originario *)
  ErroreCritico     : word;          (* codice di errore critico *)
  Tasto             : byte;          (* codice di scansione e stato di *)
  StatoShift        : byte;          (*   shift che attivano il TSR *)
  InTSRKey          : boolean;       (* TRUE se Tasto e StatoShift *)
  InTSRProg         : boolean;       (* TRUE se TSR in esecuzione *)
  Stato80x87        : array[1..94] of byte; (* stato del coprocessore *)
  PrevSS, PrevSP    : word;          (* stack originario *)
  NuovoSS, NuovoSP  : word;          (* stack del TSR *)
  PrevBreak         : boolean;       (* stato originario del Break flag *)
  VersioneDOS       : word;          (* ad es.: DOS 3.30 -> $031E *)
  InfoEstesaErrori  : RecInfoEstesaErrori; (* per funzione DOS $59 *)
  AddrFlagErrCrit   : BytePtr;       (* indirizzo flag errori critici *)
  AddrFlagInDOS     : BytePtr;       (* indirizzo flag InDOS *)
  PrevPSP, NuovoPSP : word;          (* PSP originario e del TSR *)
  SegPrevDTA        : word;          (* segmento DTA originaria *)
  OfsPrevDTA        : word;          (* offset DTA originario *)
  Modo              : byte;          (* "modo" del video *)
  Righe             : byte;          (* numero di righe del video *)
  Riga              : byte;          (* riga e colonna del cursore prima *)
  Colonna           : byte;          (*   dell'attivazione del TSR *)
  TSRProg           : procedure;     (* procedura eseguita dal TSR *)

procedure CLI; inline($FA);          (* disabilita gli interrupt *)

procedure STI; inline($FB);          (* abilita gli interrupt *)

procedure PUSHF; inline($9C);        (* mette i flag nello stack *)

procedure FSAVE;                     (* salva stato del coprocessore *)
  inline($9B/                        (* WAIT                         *)
         $DD/$36/Stato80x87);        (* FSAVE Stato80x87             *)

procedure FRSTOR;                    (* ripristina stato del coproc. *)
  inline($9B/                        (* WAIT                         *)
         $DD/$26/Stato80x87);        (* FRSTOR Stato80x87            *)

procedure EseguiTSR; forward;
```

```
interface

uses Crt, Dos;

type
  Proc          = procedure;
  TipoScheda    = (MDA, CGA, EGA, VGA);
  BuffVideo     = array[1..2000] of word;

var
  CtrlBreak     : boolean;
  CtrlC         : boolean;
  SchedaVideo   : TipoScheda;
  MemoriaVideo  : BuffVideo;

procedure Installa(Nome:string; Prog:Proc; ID:byte; Scan:byte; Shift:byte);

function ErroreIO: word; (* da usare in luogo di IOResult *)
```

Figura 2 - La interface della unit TSR.

```
procedure GetIntVec10; external;    (* cfr. TSRINT.ASM *)

procedure GetIntVec13; external;    (* cfr. TSRINT.ASM *)

function GetIntVec2F: boolean; external;  (* cfr. TSRINT.ASM *)

procedure SetIntVec10; external;    (* cfr. TSRINT.ASM *)

procedure SetIntVec13; external;    (* cfr. TSRINT.ASM *)

procedure SetIntVec2F; external;    (* cfr. TSRINT.ASM *)

procedure SetInt2FVuoto; external; (* cfr. TSRINT.ASM *)

procedure NuovoStack;               (* imposta lo stack del TSR *)
inline($FA/                         (* CLI                      *)
       $8C/$16/PrevSS/              (* MOV  PrevSS,SS           *)
       $89/$26/PrevSP/              (* MOV  PrevSP,SP           *)
       $8E/$16/NuovoSS/             (* MOV  SS,NuovoSS          *)
       $8B/$26/NuovoSP/             (* MOV  SP,NuovoSP          *)
       $FB);                        (* STI                      *)

procedure PrevStack;                (* ripristina stack origin. *)
inline($FA/                         (* CLI                      *)
       $8E/$16/PrevSS/              (* MOV  SS,PrevSS           *)
       $8B/$26/PrevSP/              (* MOV  SP,PrevSP           *)
       $FB);                        (* STI                      *)

function InDOS: boolean;            (* TRUE se in corso attivita' DOS *)
begin                               (*   che non si puo' interrompere *)
  InDOS := ((AddrFlagInDOS^ <> 0) and (InInt28 = 0)) or (AddrFlagErrCrit^ <> 0
end;

function In8259: boolean;           (* TRUE se in corso un hardware *)
var                                 (*   interrupt                  *)
  Stato: byte;
begin
  Port[$20] := $0B;
  Stato := 0;                       (* per perdere un po' di tempo  *)
  Stato := Port[$20];
  In8259 := Stato <> 0
end;

function InBIOS: boolean;           (* TRUE se in corso interrupt   *)
begin                               (*   del BIOS                   *)
  InBIOS := (InInt5 > 0) or (InInt9 > 0) or (InInt10 > 0) or (InInt13 > 0)
end;

function TSRAttivabile: boolean;
begin
  TSRAttivabile := (not InDOS) and (not InBIOS) and (not In8259)
end;
```

Figura 3 - La prima parte della implementation della unit TSR.

ripristinate sia l'apparenza del video che la posizione del cursore. Per far ciò si deve accertare che tipo di video è installato e in che «modo» questo si trova (i programmi realizzati con la unit TSR possono essere attivati solo se si è in modo testo, con 25 righe e 80 colonne); si deve anche salvare e poi ripristinare il tutto. Alcune delle variabili usate a questo scopo sono dichiarate nella **interface** perché potrebbero tornare utili anche in altre occasioni: *SchedaVideo* ci dice che tipo di video è installato (MDA, CGA, EGA o VGA), *MemoriaVideo* contiene una copia del video come era subito prima dell'attivazione del programma residente (nel caso si voglia registrarne una copia su disco; è proprio questo quello che fa SNAP.ASM, il programma illustrato nella *MS-DOS Encyclopedia*).

Credo possiate riconoscere che non c'è nulla di complicato. Vedremo subito, invece, di quanti dettagli bisogna occuparsi per la *implementation*, cominciando dal tema di questo mese: quando si può attivare un TSR.

Fino a che punto il DOS è rientrante

Dire che le funzioni del DOS non sono rientranti non è del tutto esatto; anche se si tratta di un sistema monoutente, infatti, il DOS deve comunque gestire situazioni in cui una sua funzione viene interrotta da qualcosa in modo tale da poter poi essere portata regolarmente a termine. Basta pensare ai soliti errori critici: abbiamo già visto che, quando si scrive una routine da associare all'INT 24h, questa può anche contenere una fase interattiva di dialogo con l'utente, purché vengano usate solamente le funzioni da 01h a 0Ch; rispettando questa condizione, una routine che intercetti gli errori critici può fare quello che vuole, senza per questo pregiudicare la corretta prosecuzione del programma interrotto.

Tutto dipende da quelle aree di memoria a cui avevamo accennato il mese scorso: le funzioni da 01h a 0Ch usano un'area chiamata *IOStack*, quelle da 0Dh in poi usano in genere *DiskStack*. Se si verifica un errore critico, viene assegnato un valore diverso da zero ad un flag chiamato *ErrorMode*; quando poi la routine associata all'INT 24h esegue una delle funzioni da 01h a 0Ch, queste, trovando un valore diverso da zero in *ErrorMode*, usano un'area chiamata *AuxStack* invece dell'usuale *IOStack*: ciò vuol dire che, se pure l'errore si è verificato durante l'esecuzione di una di quelle funzioni, l'area *IOStack* non viene alterata dalle funzioni chiamate dalla routine associata all'INT 24h.

Non tutte le funzioni da 0Dh in poi usano *DiskStack*. Tra le eccezioni, sono per noi rilevanti soprattutto le funzioni che leggono e impostano l'indirizzo di un *Program Segment Prefix* (PSP), rispettivamente la 51h e la 50h, e quella che legge l'informazione «estesa» sugli errori (59h), disponibile a partire dal DOS 3.0. Le prime due si comportavano come le funzioni da 01h a 0Ch nelle prime versioni del DOS, ma usano lo stack del programma chiamante a partire dalla versione 3.0; la terza usava anch'essa in principio *IOStack* o *AuxStack* secondo il valore del flag *ErrorMode*, ma a partire dal DOS 3.1 usa sempre *AuxStack*.

Questa chiacchierata ci serve a vari scopi. È chiaro, ad esempio, che non potremo attivare un TSR quando il flag *ErrorMode* sarà diverso da zero. Vedremo anche che, dovendo un TSR preservare il «contesto» del programma interrotto, dovrà anche impostare un proprio PSP e dovrà farlo in modo da evitare l'uso di *IOStack*; potremo constatare, per altro verso, che la funzione 59h pone meno problemi.

Tutto ciò, inoltre, ci mostra come il comportamento del DOS non dipende solo dal numero della funzione chiamata e dai valori passati nei registri, ma anche da alcuni flag interni.

```
function CercaFlag(var AddrFlagInDOS, AddrFlagErrCrit: BytePtr): boolean;
var
  Reg: registers;
  i  : word;
begin
  Reg.AH := $34;
  MsDos(Reg);
  AddrFlagInDOS := Ptr(Reg.ES,Reg.BX);
  if (VersioneDOS >= $30A) and (VersioneDOS < $0A00) then
    AddrFlagErrCrit := Ptr(Reg.ES,Reg.BX-1)
  else begin
    AddrFlagErrCrit := nil;
    i := 0;
    while i < $FFFF do begin        (* cerca in tutto il segmento *)
      if Byte(Ptr(Reg.ES,i)^) = $CD then
        if Byte(Ptr(Reg.ES, i+1)^) = $28 then
          (* se si e' trovato INT 28h, ...          *)
          (* cerca TEST SS:[FlagErrCrit],0FFh ecc. *)
          if Byte(Ptr(Reg.ES,i-12)^) = $F6 then begin
            AddrFlagErrCrit := Ptr(Reg.ES,Word(Ptr(Reg.ES,i-10)^));
            i := $FFFE              (* forza l'uscita dal loop *)
          end
          (* .. oppure CMP SS:[FlagErrCrit],0 ecc. *)
          else if Byte(Ptr(Reg.ES,i-7)^) = $80 then begin
            AddrFlagErrCrit := Ptr(Reg.ES,Word(Ptr(Reg.ES,i-5)^));
            i := $FFFE              (* forza l'uscita dal loop *)
          end;
      Inc(i)
    end
  end;
  if AddrFlagErrCrit <> nil then CercaFlag := TRUE
  else CercaFlag := FALSE
end;
```

Figura 4 - La funzione CercaFlag, che cerca gli indirizzi dei flag InDOS e ErrorMode per assegnarli, rispettivamente, ai puntatori AddrFlagInDOS e AddrFlagErrCrit. La variabile VersioneDOS viene avvalorata nella procedura Installa, mediante uno Swap(DosVersion). DosVersion è una funzione predefinita che ritorna la versione del DOS «al contrario»: con un DOS 3.30 si ottiene 7683, in esadecimale $1E03, cioè prima 30 poi 3. Con Swap si invertono i due byte, per semplificare i confronti. Il numero di versione $0A00 è quello del «box» DOS sotto OS/2.

Il flag InDOS e l'INT 28h

Oltre a quello appena visto, un altro flag di vitale importanza è quello chiamato *InDOS*. Si tratta di un byte che viene incrementato ogni volta che viene chiamata una funzione DOS (mediante INT 21h) e decrementato quando questa termina: esaminando il valore di quel byte si può quindi sapere se è in corso un'attività del DOS; se *InDOS* è diverso da zero non è prudente attivare il TSR.

Naturalmente le cose non sono così semplici. In primo luogo, quel byte viene sempre azzerato quando si verifica un errore critico; ciò avviene perché la routine associata all'INT 24h potrebbe scegliere di non tornare alla funzione che era inciampata nell'errore, e se così fosse il flag non verrebbe decrementato. È questo un altro motivo per tenere d'occhio il flag *ErrorMode*, come del resto avevamo già deciso. Ci sono però situazioni in cui l'attesa di trovarsi in una situazione di «DOS inattivo» potrebbe risultare troppo lunga: tutte quelle situazioni in cui un programma si ferma aspettando eventi esterni, quali la pressione di un tasto da parte dell'utente. Se ciò avviene perché sono in azione le famose funzioni da 01h a 0Ch, c'è la possibilità di fare qualcosa: quelle funzioni infatti «aspettano» mediante un loop da cui viene ripetutamente chiamato l'INT 28h. A questo è normalmente associato un IRET, ma vi si può associare una routine di attivazione del TSR.

In definitiva, un TSR può essere attivato se il flag *ErrorMode* è zero, e se anche *InDOS* è zero o è diverso da zero ma si è nel mezzo di una routine associata all'INT 28h. Se anche una sola di queste condizioni non è vera, la funzione *InDOS* della unit TSR (vedi la figura 3) ritorna TRUE a significare che il DOS è attivo e non può essere interrotto.

Un'ultima nota. Quanto appena detto comporta che si potrebbe attivare un TSR mentre è in esecuzione una delle funzioni da 01h a 0Ch, e quindi mentre è in uso *IOStack*. La *MS-DOS Encyclopedia* non lo dice, ma mi pare evidente che il programma residente dovrebbe evitare di usare funzioni di quel gruppo, per non alterare *IOStack*. Finché si programma in Pascal comunque non ci sono problemi: il Turbo Pascal 5.x, infatti, usa solo funzioni DOS da 0Eh e superiori.

Rimane un solo problema: dove andare a cercare i flag del DOS. Per *InDOS* è tutto facile: ce lo dice la funzione 34h, che ne ritorna l'indirizzo in ES:BX. Quanto all'indirizzo di *ErrorMode*, tutto dipende dalla versione del DOS; se siamo da 3.1 in poi, il flag si trova nel byte subito prima di *InDOS*, e quindi trovato l'uno si trova anche l'altro; altrimenti bisogna mettersi a scandire il codice in linguaggio macchina nel segmento ritornato in ES dalla funzione 34h, in cerca di particolari sequenze di istruzioni, anch'esse variabili secondo la versione del DOS. Il tutto è implementato nella funzione *CercaFlag* (figura 4), diligente conversione in Turbo Pascal delle routine in Assembler «raccomandate» dalla Microsoft.

BIOS e 8259A

Più semplice il caso del BIOS: non si parla di «rientranza» , punto e basta. Non rimane altro che sostituire agli interrupt del BIOS altre routine che riproducano il meccanismo del flag *InDOS*: si incrementa un flag, si esegue l'interrupt originario, si decrementa il flag. La unit TSR fa proprio questo, intercettando gli interrupt 05h, 10h e 13h (rispettivamente: *print screen*, I/O su video, I/O su disco); vedremo il mese prossimo i relativi dettagli.

Non basta tuttavia fare i conti con DOS e BIOS. Oltre agli interrupt software, infatti, ci sono anche interrupt hardware (08h per il timer, 09h per la tastiera, ecc.) che vengono gestiti mediante un 8259A della Intel: questo inibisce gli interrupt che abbiano priorità più bassa di quello eventualmente in esecuzione, il quale a sua volta, quando termina, deve notificarlo all'8259A. Se un TSR interrompesse un interrupt hardware, ne potrebbe risultare l'inibizione di altri, con possibile blocco del sistema. La funzione *In8259* controlla appunto che ciò non si verifichi, che cioè non siano in esecuzione interrupt hardware. La funzione *TSRAttivabile* racchiude in sé tutti i controlli che abbiamo visto.

Le prime tessere del mosaico

Nella figura 3 c'è qualcosa in più rispetto a quello che ho cercato di illustrarvi questo mese: c'è tutta la prima parte della **implementation**, che vi propongo così come è per cercare di ridurre al minimo quel lavoro di ricostruzione del file TSR.PAS che vi ho chiesto di fare montando i vari pezzi pubblicati ora e nei prossimi numeri sullo «scheletro» della figura 1. Ed anche per evitare il rischio di allungare troppo il discorso. Credo (spero) che i commenti inseriti nel listati siano più che sufficienti.

Vi do appuntamento quindi tra trenta giorni, per vedere più da vicino l'intercettazione degli interrupt del BIOS nonché (se lo spazio ce lo consentirà) i meccanismi di attivazione di un TSR.

MC

NELCOM NELCOM NELCOM

VI OFFRE PERIFERICHE **N E C *** e ...

P2200	+ 4 Nastri neri originali	=	**L. 665.000**
P6+	+ 4 Nastri neri originali	=	**L. 1.195.000**
P6+/KIT COLORE	+ 4 Nastri neri + 2 nastri col.	=	**L. 1.435.000**
P7+	+ 4 Nastri neri originali	=	**L. 1.500.000**
P7+/KIT COLORE	+ 4 Nastri neri + 2 nastri col.	=	**L. 1.740.000**
P9-960XL	+ 4 Nastri neri + 2 nastri col.	=	**L. 2.854.000**
Laser 866+ 2Mbs	+ Kit aggiuntivo	=	**L. 4.208.000**
Laser 890 PostScript* 3 Mbs	+ Kit aggiuntivo	=	**L. 6.298.000**

NOVITÀ

P2200PLUS			=	**L. 738.000**
MULTISYNC NEC* 2A	800x600	0,31	=	**L. 920.000**
MULTISYNC NEC* 3D	1024x768	0,28	=	**L. 1.228.000**
MULTISYNC NEC* 4D	1024x768	0,28	=	**L. 2.320.000**
MULTISYNC NEC* 5D	1280x1024	0,31	=	**L. 4.050.000**

... PERSONAL COMPUTERS A PREZZI ECCEZIONALI

I COMPONENTI ESSENZIALI dei ns. Personal Computers sono di ALTA QUALITÀ e COMPLETA AFFIDABILITÀ

- *Scheda Madre: NEAT con SHADOW RAM; SHADOW ROM; Page Interlave; LIM EMS 4.0 e Set Up esteso*
- *Controller FD-HD = ADAPTEC* Interl. 1:1 o RLL 1:1*
- *Floppy disk = TEAC**
- *Hard disk = NEC* / MICROSCIENCE* / RONDIME* / QUANTUM**
- *Schede video a colori = TRIDENT* / ATI "WONDER" 1024x768 - 512 K*
- *DOS 4.1 Microsoft**

B 286/21.5 L.M.
Versione tavolo
1 Mbs RAM 75 ms
Fd. 5"¼ 1.2 Mbs
Fd. 3"½ 1.44 Mbs
Hd. 44 Mbs / 28 ms
1 P.P. + 2 RS232
Hercules
Monitor Mono 14"
Cavo stampante

L. 2.990.000

C 286/21.5 L.M.
Versione tavolo
1 Mbs RAM 75 ns
Fd. 5"¼ 1.2 Mbs
Fd. 3"½ 1.44 Mbs
Hd. RLL 1:1 65 Mbs/28 ms
1 P.P. + 2 RS232
Super VGA 1024x768 - 512 K
Multisync NEC* 2A 14"
Cavo stampante

L. 4.200.000

D 286/21.5 L.M.
Versione tavolo
2 Mbs RAM 70 ns
Fd. 5"¼ 1.2 Mbs
Fd. 3"½ 1.44 Mbs
Hd. - 105 Mbs - 64 K Cache Memory 18 ms
1 P.P. + 2 RS232
Coprocessore Matematico 16 MHz
Super VGA 1024x768 - 512 K
Multisync NEC* 3D 14"
Cavo stampante
Winchester removibile 44 Mbs/20 ms

L. 8.350.000

B 386/34.5 L.M.
Mini Tower
2 Mbs RAM 75 ms
Fd. 5"¼ 1.2 Mbs
Fd. 3"½ 1.44 Mbs
Hd. 44 Mbs / 28 ms
1 P.P. + 2 RS232
Hercules
Monitor Mono 14"
Cavo stampante

L. 4.200.000

C 386/34.5 L.M.
Mini Tower
2 Mbs RAM 75 ns
Fd. 5"¼ 1.2 Mbs
Fd. 3"½ 1.44 Mbs
Hd. RLL 1:1 65 Mbs/28 ms
1 P.P. + 2 RS232
Super VGA 1024x768 - 512 K
Multisync NEC* 2A 14"
Cavo stampante

L. 6.400.000

D 386/34.5 L.M.
Tower
4 Mbs RAM 70 ms
Fd. 5"¼ 1.2 Mbs
Fd. 3"½ 1.44 Mbs
Hd. - 105 Mbs - 64 K Cache Memory 18 ms
Coprocessore Matematico 25 MHz
Super VGA 1024x768 - 512 K
Multisync NEC* 4D 16"
Cavo stampante
Winchester removibile 44 Mbs/20 ms

L. 11.300.000

Drivers Software NEC · Parti di Ricambio Originali · Accessori ed Optionals*

ESPERIENZA DI 12 ANNI

Garanzia 12 mesi franco Torino · Spedizioni gratuite in tutta Italia
Vendita per corrispondenza · Prezzi IVA esclusa · * = Marchi registrati

Corso Casale, 120 - 10132 TORINO - Telefoni (011) 88.58.22 / 83.73.30 - Telefax (011) 8123813

APERTI ANCHE AL SABATO