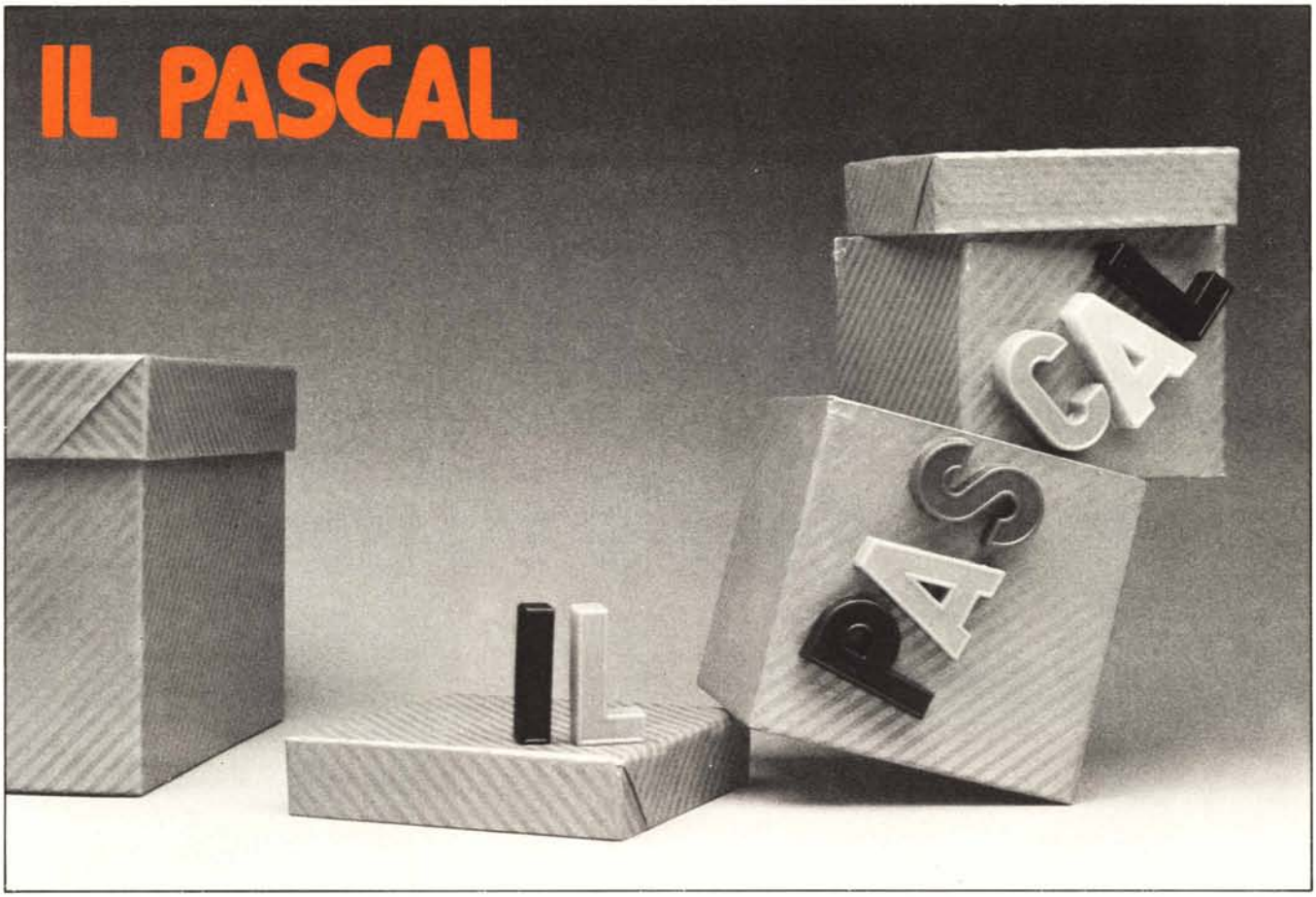
IL PASCAL



Quinta parte

I sottoprogrammi e le funzioni di ingresso/uscita

Siamo arrivati alla puntata conclusiva di questa breve rassegna del linguaggio PASCAL: dopo avere esaminato a fondo le istruzioni per la creazione di architetture di dati anche molto complesse e quelle "operative" che consentono un controllo molto modulare delle operazioni che il programma deve eseguire, affrontiamo oggi due argomenti di peso specifico molto diverso.

Se l'uso dei sottoprogrammi è utile nei linguaggi comuni, in un linguaggio strutturato come il PASCAL essi sono addirittura indispensabili per poter sfruttare appieno la potenzialità di questo strumento software: programmare in PASCAL senza adoperare sottoprogrammi sarebbe come suonare il pianoforte con un dito solo.

Ben diverso è invece il discorso riguardante le istruzioni di input/output, per due motivi fondamentali.

Il primo motivo è comune a tutti i linguaggi di programmazione, e riguarda la *compatibilità* delle istruzioni con la macchina su cui il linguaggio viene implementato. Ogni calcolatore ha infatti un suo proprio modo di accedere alle unità periferiche: alcuni usano istruzioni speciali (INA, OTA), altri scrivono e leggono sui registri di interfaccia con le normali istruzioni di trasferimento. Per di più un compilatore si appoggia di norma su un ben determinato *sistema operativo*, e come purtroppo sappiamo, ogni sistema ha il proprio modo di gestire, ad esempio, i files sulle memorie di massa, o le interfacce con i terminali. Tutto ciò fa sì che, nella definizione *teorica* di un linguaggio, la parte riguardante i colloqui con le periferiche debba per forza restare un po' nel vago: toccherà poi alle singole implementazioni definire cosa si può fare, e così il PASCAL del PDP-11 prevederà questo e quest'altro, mentre il PASCAL dell'IBM consentirà, quello e quell'altro.

Il secondo motivo è invece specifico del PASCAL: nonostante gli onesti sforzi di Wirth e dei suoi collaboratori, il linguaggio è risultato di livello un tantino troppo elevato per permettere un interfacciamento stretto con la struttura della macchina, e così ci dobbiamo accontentare di un insieme di istruzioni di I/O più limitato di quello del BASIC e dello stesso FORTRAN; tuttavia molte volte nella vita la semplicità è d'aiuto, e l'apparente "povertà" delle istruzioni READ e WRITE nasconde in realtà una gran chiarezza e una sorprendente facilità d'uso.

I sottoprogrammi

Il concetto di *sottoprogramma* è ben noto a chiunque abbia avuto a che fare con del software a livello appena più alto di quello elementare, qualunque sia il linguaggio usato, dagli assemblatori ai più sofisticati linguaggi specializzati.

La necessità di effettuare chiamate a sottoprogrammi ha una motivazione di tipo *logico* e una di tipo *pratico*, e si presenta ogni volta che il programma richiede l'esecuzione di gruppi di istruzioni uguali fra loro.

In questi casi la scrittura pura e semplice di queste istruzioni ogni volta che ce n'è bisogno porta ad un'inutile *ripetizione* del codice in parti diverse del programma. Se la parte comune è abbastanza lunga e le chiamate sono numerose, ecco che si genera un grande ingombro di memoria per ripetere delle istruzioni che potrebbero stare in un posto solo: non per nulla esistono le istruzioni di salto! La soluzione è appunto la creazione di un *sottoprogramma*, ossia di un'unità software *a sé stante*, che può essere raggiunta dal programma principale, tramite un'istruzione di *salto*, da vari punti: il *ritorno* al punto di chiamata è gestito dal sottoprogramma stesso con un'apposita istruzione (solitamente un salto indiretto). Così la parte di istruzioni compresa nel sottoprogramma viene eseguita molte volte, come il software richiede, ma compare fisicamente in memoria una volta sola: questa è la motivazione di tipo *pratico* per l'uso dei sottoprogrammi, ed il confronto fra le due metodologie è mostrato in figura 1.

Oltre ad occupare meno memoria, i sottoprogrammi danno anche un grande aiuto in termini di *leggibilità* di un programma: una sola istruzione di chiamata, che normalmente include un nome simbolico indicante l'operazione da eseguire, sostituisce tutta una parte di software che altrimenti comparirebbe qua e là nel programma senza soluzione di continuità: un bell'aiuto per non perdere il filo del discorso in diecimila particolari! Ecco la motivazione di tipo *logico* per l'uso dei sottoprogrammi.

A queste due motivazioni il PASCAL ne aggiunge un'altra di tipo *strutturale*: non è infatti possibile programmare in modo strutturato, ossia, per quanto abbiamo detto nelle scorse puntate, programmare in PASCAL, senza un uso intensivo delle chiamate a sottoprogrammi.

Si prenda ad esempio lo schema a blocchi di figura 2 che rappresenta una tipologia software abbastanza comune: occorre eseguire il blocco C ogni volta che si riscontra una variabile (X o Y) diversa da zero.

Il programma PASCAL che descrive lo schema a blocchi di figura 2 potrebbe essere il seguente:

```
if X = 0 then
    begin
    . (A)
    :
if Y = 0 then
    begin
    . (B)
    :
     end
  else begin
    . (C)
    :
     end
  end
  else begin
    . (C)
    :
     end
```

Sebbene vi possa essere modo di evitarlo, in questa stesura "strutturata" il blocco di istruzioni C ha dovuto essere scritto due volte; rispetto all'analogo programma BASIC:

```
    IF X <>0 GOTO C
A   .....
    IF Y <>0 GOTO C
B   .....
    GOTO....
C   .....
```

si è guadagnato in chiarezza ma si è perso in occupazione di memoria: per una volta la famigerata "programmazione a spaghetti" si è rivelata più compatta di quella strutturata.

Se però si può chiamare il blocco C come un *sottoprogramma*, ecco che la chiarezza della programmazione strutturata non inficia più l'occupazione di memoria, e il programma stesso diventa, naturalmente, più leggibile: i sottoprogrammi, aprendo una *parentesi* nel software, sono essi stessi un elemento della programmazione strutturata, ed è questo il motivo base del loro impiego così diffuso nel PASCAL.

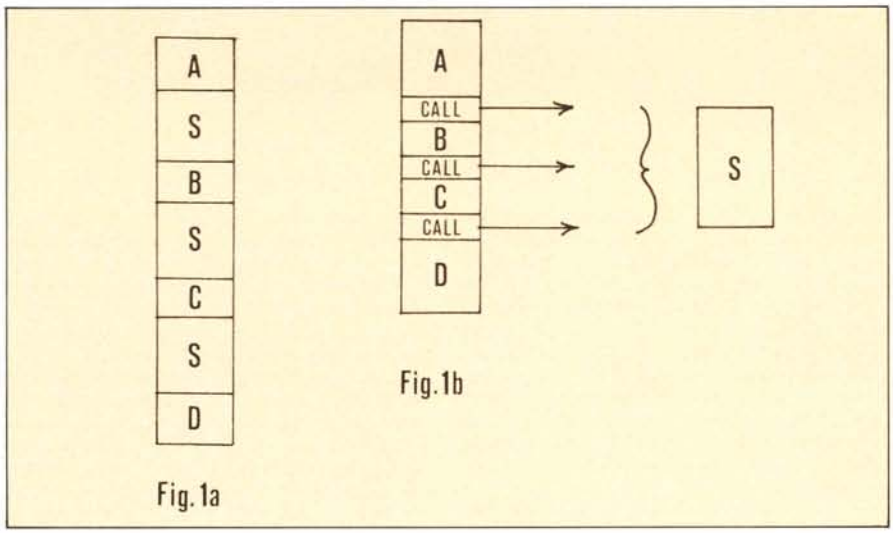
Un'ultima puntualizzazione sui sottoprogrammi riguarda la loro *parametricità*. Molto spesso accade che un sottoprogramma debba compiere le operazioni su variabili che cambiano ad ogni chiamata.

Non sempre è così: basti pensare ad un sottoprogramma che ad esempio compatta o i files sull'unità a disco, o azzera una certa zona fissa di memoria; ma nella maggior parte dei casi un sottoprogramma deve eseguire le medesime operazioni su dati che cambiano di volta in volta.

La soluzione a questo problema non indifferente (se io faccio un sottoprogramma che esegue una funzione fra A e B e scrive il risultato in C, mi occorrerebbe un'altra routine uguale se volessi fare la stessa cosa con altre variabili D, E ed F) è di fare eseguire al sottoprogramma le operazioni su delle variabili apposite dette *parametri*, e gestirsi in qualche modo il trasporto di questi parametri.

Supponiamo ad esempio di scrivere un sottoprogramma SUB che calcoli una certa funzione di due variabili X e Y e scriva il risultato in Z: per fare eseguire la funzione sulle variabili A e B ed ottenere il risultato in C si può agire in questo modo:

```
X=A
Y=B
CALL SUB
C=Z
```



(N.B.) La scrittura è BASIC.

In un altro punto del programma, per eseguire la funzione sulle variabili D ed E ed ottenere il risultato in F, si scrive la medesima sequenza di istruzioni sostituendo queste ultime variabili alle A, B e C del caso precedente.

In quasi tutti i linguaggi ad alto livello questa operazione può essere eseguita automaticamente, associando al sottoprogramma una *lista di parametri* interessati all'operazione. Nel nostro esempio; al sottoprogramma SUB devono essere associati i tre parametri X, Y e Z. In BASIC la scrittura di un sottoprogramma parametrico avviene in questo modo:

```
SUBR SUB (X, Y, Z)
(corpo del sottoprogramma)
SUBEND (istruzione di ritorno)
```

Una volta scritto il sottoprogramma in questo modo, occorre elencare ad ogni chiamata i parametri *attuali* che sostituiscono quelli *simbolici* nel corso del sottoprogramma: avremo così la chiamata:

```
CALL SUB (A, B, C)
```

che esegue la funzione su A e B e restituisce il risultato in C, e la chiamata:

```
CALL SUB (D, E, F)
```

che esegue la funzione su D ed E e restituisce il risultato in F. In entrambi i casi i parametri listati nella chiamata sostituiscono quelli listati nella definizione del sottoprogramma.

Due ultime osservazioni: innanzitutto la chiamata *parametrica* appena vista non è equivalente alla scrittura precedente con le istruzioni di assegnamento X=A etc., perché, in un sottoprogramma parametrico i parametri simbolici non esistono, cioè ad essi non è associata *alcuna cella di memoria*, non sono insomma delle variabili vere e proprie: il sottoprogramma esegue la funzione *direttamente* sui parametri passati nella lista di chiamata. Questo fatto permette ad un sottoprogramma di essere *rientrante*, cioè di essere interrotto, magari da un interrupt a livello più elevato, rieseguito a priorità più alta con altri parametri e successivamente ripreso dal punto dove stava, con i parametri precedenti e senza inficiare minimamente il risultato. Se invece il sottoprogramma, ad esempio il nostro SUB, calcolasse la funzione su effettive variabili X, Y e Z, una interruzione e una riesecuzione con parametri diversi modificherebbero il contenuto di queste variabili (che sono sempre le stesse), e al momento di riprendere da dove si era fermato, il sottoprogramma si ritroverebbe dati diversi da prima.

La rientranza è poi condizione indispensabile per la *ricorsività* di un sottoprogramma, e questa ultima caratteristica è fondamentale per la programmazione in PASCAL.

Infine vorrei spendere due parole sul parametro (Z) che contiene — nel nostro esempio — il valore di ritorno della funzione calcolata dal sottoprogramma SUB. In molti linguaggi, e fra essi il PASCAL, un sottoprogramma con un solo parametro di ritorno prende il nome di *funzione* "tout court", e può essere richiamato in un modo tutto particolare, associando al proprio nome il valore del parametro di ritorno. La funzione può quindi comparire in un'istruzione come una variabile qualsiasi: ad esempio le istruzioni

```
C=SUB (A, B)
F=SUB (D, E)
```

sostituiscono le CALL dell'esempio precedente, se SUB è stato definito come *funzione*. Sono funzioni le normali SIN, LOG, EXP... di tutti i linguaggi: le possibilità di definire sottoprogrammi come funzioni consente di richiamarli allo stesso modo.

E veniamo infine alla scrittura PASCAL dei sottoprogrammi, sia per quanto riguarda la loro *definizione*, sia per quanto riguarda la loro *chiamata*.

Si ricordi innanzitutto che un blocco *compound* (ad esempio quello che costituisce il programma) contiene un ben preciso spazio dedicato alla definizione dei sottoprogrammi, fra le dichiarazioni di tipo e le istruzioni operative. In un programma PASCAL i sottoprogrammi stanno quindi in un posto ben definito.

Un sottoprogramma PASCAL è composto di un normalissimo blocco *compound* preceduto da un'intestazione contenente il nome della routine e la eventuale lista dei parametri simbolici, in questo modo:

```
procedure nome (par:tipo, par:tipo...);
begin
end;
```

Vi sono alcune annotazioni interessanti da fare: innanzitutto i parametri simbolici, a cui è unito il tipo devono comparire nella parte *var* del blocco. Ciò può sembrare a prima vista scomodo, ma in questo modo diventa semplicissimo trasformare una parte di codice in un sottoprogramma: basta aggiungere l'istruzione di intestazione, e automaticamente le variabili "parametrizzate" perdono ogni realtà e diventano simboliche, garantendo la rientranza.

In secondo luogo il blocco *compound* è completo, e può quindi contenere al suo interno definizioni di tipo, di variabile e anche di altri sottoprogrammi. La cosa può sembrare buffa, ma questi sottoprogrammi a livello inferiore possono essere chiamati soltanto all'interno del blocco *compound* che li definisce, e ciò aiuta ad organizzarsi nella strutturazione dell'intero programma.

Infine si fa notare che l'istruzione di *chiamata* di un sottoprogramma PASCAL consiste *nel solo nome* del sottoprogramma stesso, senza parole chiave come CALL o GOSUB. Nel corso di un programma PASCAL si possono dunque incontrare istruzioni formate da uno strano nome, eventualmente dotato di una lista di parametri: questa è un'istruzione di chiamata ad un sottoprogramma.

Come già detto, il PASCAL prevede la definizione di funzioni, la cui chiamata funge anche da parametro di ritorno: la loro definizione differisce da quella dei sottoprogrammi soltanto nell'istruzione di intestazione, che contiene la parola chiave *function* anziché *procedure* e un'indicazione del tipo della funzione; quest'ultima parte è necessaria perché al momento della chiamata la funzione viene trattata come una variabile, e deve esservi associato un tipo.

All'interno della funzione, poi, esisterà almeno un'istruzione di assegnamento alla funzione stessa, che viene così trattata, per l'appunto come una variabile, e in particolare come un parametro di ritorno.

Un esempio chiarirà quanto esposto.

Si considerino le seguenti funzioni:

```
function FACT (N:integer)=integer;
begin     if N=0 then FACT=1
          else FACT=N*FACT (N-1)
end;
    functionc NEWTON (N, K:integer) =
integer;
    begin
    NEWTON=FACT (N) / (FACT (K)*
FACT (N-K))
    end;
```

La prima di queste funzioni calcola il fattoriale di un numero N intero secondo la nota definizione ricorsiva: il risultato (FACT) è a sua volta un numero intero, e ciò è specificato nell'ultima parte dell'intestazione, ove si pone la funzione uguale ad una definizione di tipo.

La funzione FACT viene poi richiamata nella seconda funzione NEWTON, che calcola il coefficiente di Newton($\binom{n}{k}$) di due numeri interi.

Se ora vogliamo divertirci a scrivere un sottoprogramma, possiamo costruirci il triangolo di Tartaglia:

```
var COEF : array [0...10] of integer;
procedure SVIL (N . integer);
begin
var K : integer;
for N=0 to K do
COEF (K)=NEWTON (K, N -K)
end;
```

Il sottoprogramma SVIL scrive nel vettore COEF i coefficienti di sviluppo di un polinomio $(a+b)^n$, ad esempio 1, 4, 6, 4, 1 per N = 4: provare per vedere.

Il programma seguente a queste definizioni (occorrono infatti anche le due funzioni FACT e NEWTON) potrebbe stampare il triangolo di Tartaglia in questo modo:

```
for I=1 to 10 do
begin SVIL (I);
writeln (COEF [J] for J=0 to I)
end;
```

Incontriamo per la prima volta in quest'esempio l'istruzione *writeln*, una delle istruzioni di ingresso/ uscita: abbastanza semplice come si vede, merita soltanto un breve approfondimento.

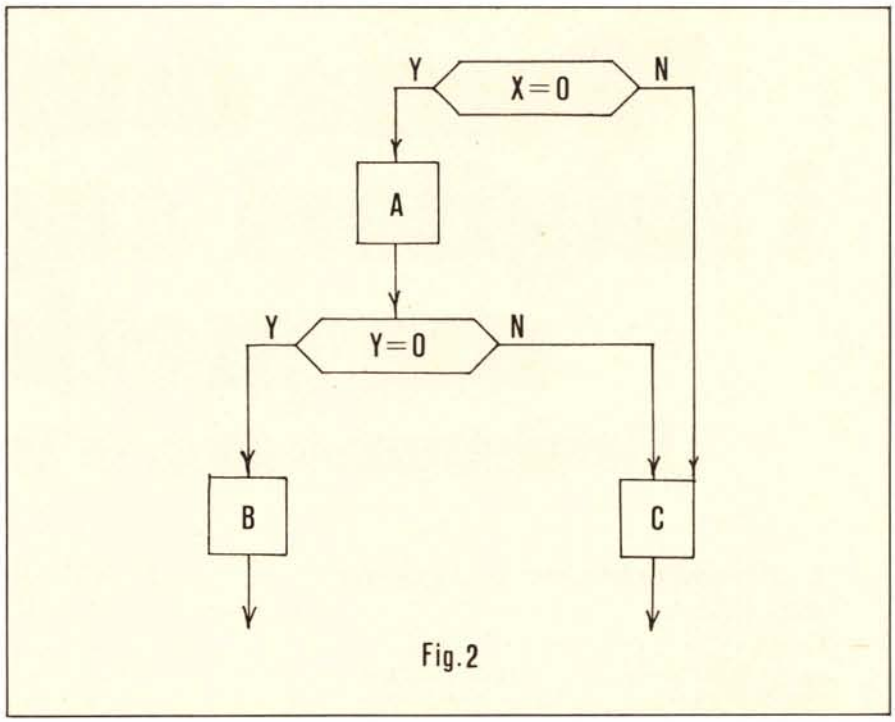


Fig.2

Le istruzioni di ingresso e uscita

Come già detto all'inizio di questo articolo, le istruzioni di I/O sono state lasciate da Wirth un po' nel vago: sta alle singole implementazioni stabilire con rigore il formato e la potenza reale di queste istruzioni.

Per il momento dobbiamo vedercela soltanto con le più banali *read* e *write*, a cui si può associare una *lista* di parametri da leggere o scrivere. Non è prevista alcuna possibilità di specificare *su che periferica* va eseguita l'operazione. Sta ancora una volta ai singoli compilatori fornire il modo di accedere, ad esempio, ad una unità a disco: noi leggiamo e scriviamo unicamente su di un terminale.

Per l'istruzione *read* non vi sono problemi di sorta; la lista di parametri specifica le variabili da riempire con l'operazione di ingresso, e in questo è analoga alla INPUT del BASIC: con le istruzioni

```
var A, B, C : integer
read (A, B, C);
```

si leggono da terminale tre variabili intere.

Analogamente, la *write* corrisponde alla PRINT del BASIC, con una sottile differenza: il suffisso *ln* indica che dopo la scrittura il carrello si posiziona all'inizio di una nuova riga.

Così le istruzioni:

```
write ('non va')
write ('a capo')
```

generano questa stampa:

```
non va a capo
```

mentre le istruzioni

```
writeln ('questa invece')
writeln ('ci va')
```

generano questa stampa:

```
questa invece
ci va
```

Per di più, nel primo caso il carrello è rimasto posizionato dopo la parola "capo" sulla stessa riga, mentre nel secondo caso sta sotto la "C", all'inizio di una nuova riga.

Si possono stampare, come in BASIC, ogni combinazione e sequenza di stringhe, costanti e variabili, e loro espressioni.

Conclusione

Con questi due argomenti, trattati — specie il primo — anche in sede teorica, si conclude questa breve rassegna sul PASCAL. Non si pretende che abbiate imparato il linguaggio col poco che s'è detto, ma sarebbe già un ottimo risultato se fosse diventata chiara la filosofia della programmazione in questo affascinante linguaggio. PASCAL infatti, oltre e più ancora che un linguaggio di programmazione, è una proposta di standardizzazione del software in termini di programmazione strutturata: per questo motivo non c'è bisogno assoluto di un compilatore per potersi esercitare. Basta iniziare a "pensare" in PASCAL, e magari scrivere una prima stesura di un programma secondo le regole e le definizioni date in questa serie di articoli; e poi "tradurre" nel linguaggio disponibile (assemblatore, BASIC...) quello che si è scritto. Io ci ho provato, e questo metodo non l'ho abbandonato più.

Pietro Hasenmajer

"Senza dubbio i migliori supporti magnetici"

finalmente anche in Italia

L'affidabilità Maxell: una garanzia assoluta.

Già da molti anni Maxell è tra i migliori specialisti mondiali di memorie magnetiche. Una reputazione ormai così solida che sia i costruttori che gli utenti hanno una totale fiducia in noi. Questa immagine la dobbiamo alla qualità della nostra produzione, alla severità dei controlli cui la sottoponiamo e soprattutto alla nostra tecnica di rivestimento: una tecnica esclusiva.

Maxell: supporti magnetici dalle caratteristiche uniche.

- Un procedimento di rivestimento esclusivo, grazie al quale si ottengono proprietà magnetiche eccezionali e una grande affidabilità di lettura/registrazione.
- Un'elevata qualità della superficie, per un contatto ottimale delle testine magnetiche.
- Una totale compatibilità con tutti i sistemi standard di lettura/registrazione.
- Una prolungata resistenza dei prodotti, per una sicurezza massima degli archivi.

Se volete informazioni più dettagliate, telefonateci o scriveteci

maxell®
supporti magnetici
l'affidabilità

Distributori di zona
Lombardia: TELCOM - Milano (02) 4047648
Tre Venezie: HARDPOINT - Padova (049) 773962
Emilia Romagna: CTC - Bologna (051) 552430/80
Toscana Umbria: CSM - Firenze (055) 576589
Campania: EDL - Napoli (081) 611988-632335

scegli telcom

TELCOM s.r.l. 20148 Milano - Via M. Civitali, 75
Tel. (02) 4047648 (3 linee ric. aut.)
Telex 335654 TELCOM I