I segreti del Basic

Tutti sanno utilizzare il Basic. Pochi in realtà conoscono esattamente come funziona. Questo articolo farà un po' di luce sul modo in cui il Commodore 64 gestisce i programmi Basic.

In questo articolo vedremo in dettaglio come il Commodore 64 e il Commodore 128 immagazzinano i programmi Basic, e come utilizzare questa tecnica per ottenere utilissimi vantaggi.Vedremo come ottenere le REM del listato in reverse, come rendere il listato Basic più leggibile, come proteggere o sproteggere (senza approfondire eccessivamente) un programma interamente Basic, come nascondere alcune parti di programma agli occhi della routine di List, come ottenere un AutoRun permanente, come ottenere un Merge di due programmi Basic senza utilizzare il linguaggio macchina e altro ancora.

Locazioni e puntatori

I programmi Basic vengono automaticamente memorizzati a partire da una predeterminata locazione di memoria, evitando così il problema di decidere dove locare ciò che digitate (problema con cui si ha a che fare se si programma in linguaggio macchina), risparmiando così tempo e decisioni. Il programma Basic inizia dalla locazione numero 2049 sul C64 e dalla 7169 o 16385 sul C128. Questo significa che a partire da questa locazione verrà memorizzato tutto ciò che scriverete preceduto da un numero di linea (le istruzioni date in modo diretto non vengono ricordate).

Il motivo per cui il C128 dispone di due valori è molto semplice: se si sta usando la pagina grafica (data con il comando Graphic n, con n > 0>) l'inizio Basic sarà in 16385, mentre se non la si sta utilizzando, l'inizio Basic sarà in 7169. Ricordate a tale proposito che per abbassare l'inizio Basic in 7169 (sul C128) dovete utilizzare l'istruzione Graphic CLR, che farà perdere il contenuto della pagina grafica.

Vi sono due locazioni che puntano all'inizio del Basic. Queste due locazioni sono la 43 e la 44 sul C64 e la 45 e 46 sul C128. Ciò significa che se modificate il contenuto di queste locazioni, modificherete l'inizio dell'area Basic, trasferendola in un'altra zona.

Il Basic inizia alla locazione numero:

```
PEEK<43> + PEEK<44>*256
```

sul Commodore 64, e alla locazione numero:

```
PEEK>45> + PEEK<46>*256
```

sul Commodore 128.

Sul C64 infatti, la locazione 43 contiene il valore 1, mentre la 44 contiene il valore 8: ciò significa che l'inizio Basic è in <1+8*256> ossia 2049 come già detto.

D'ora in avanti ci riferiremo solo al C64: chi possiede un C128 deve ricordarsi di utilizzare i puntatori del suo computer e non quelli del C64 forniti negli esempi. La forma con cui un programma Basic viene posta in memoria è la seguente:

```
$ LINK (lb)
* LINK (hb)
* NUMERO DI LINEA (lb)
* NUMERO DI LINEA (hb)
* istruzioni Basic introdotte
* 0
* LINK successivo (lb)
* LINK successivo (hb), ecc.
```

Vedremo fra poco un esempio che chiarirà ogni dubbio in merito.

Allo stesso modo in cui, nel C64, le locazioni 43/44 puntano all'inizio del Basic, le locazioni 45/46 puntano alla fine. Ciò significa che un programma Basic sul C64 occupa un numero di locazioni così ottenibile:

```
PRINT <PEEK<<45> PEEK<46>
*256>
<PEEK<43>+PEEK<44>*256>
```

E' intuibile quindi che i puntatori 45/46 vengono modificati ogni volta che introducete, modificate o cancellate una linea di programma in memoria. Sul Commodore 128 non esistono i puntatori di fine Basic perché il programma Basic risiede in un suo banco di memoria dedicato (il banco 0): quindi sul C128 per conoscere la locazione finale del Basic occorre scrivere: PRINT 65279 - FRE <0>.

Provate ora a scrivere due linee Basic e a vedere come esse vengono poste in memoria; il programma dimostrativo che analizzeremo sarà il seguente:

```
10 PRINT"UNO"
20 PRINT"DUE"
```

Fate attenzione a non aggiungere uno spazio <CHR$>32>> dopo la PRINT. E' inutile dire che il programma è puramente dimostrativo. La **tavola 1** mostra il contenuto delle locazioni di memoria in cui sono memorizzate le due linee Basic.

La linea Basic 10 è posta in memoria a partire dalla locazione numero 2049. Ecco i significati delle varie locazioni: 2049 e 2050 contengono rispettivamente 12 e 8; abbiamo detto che le prime due locazioni rappresentano il link (ossia l'indirizzo da cui partirà la prossima linea Basic). Infatti 12 + 8*256 dà come risultato proprio 2060, ossia la locazione della linea Basic successiva. Il motivo per cui si moltiplica il secondo numero per 256

e lo si somma al primo è che un byte può contenere al massimo il valore 255: numeri maggiori vanno divisi in due parti (il byte basso LB e il byte alto HB), e questa è la formula per riottenere il numero originario. Le locazioni 2951 e 2052 contengono rispettivamente 10 e 0: questo è il numero di linea Basic diviso nella parte alta e bassa (10 +0*256 + 10). La 2053 contiene il valore 153 (vedere in **tavola 2** la tabella dei Token e dei codici Ascii): 153 è infatti il Token della istruzione Basic Print.

Salvare l'istruzione Print sotto forma Ascii sarebbe uno spreco di memoria troppo elevato, per cui ogni parola riservata Basic ha un suo Token, ossia un valore superiore a 128 con cui viene salvata in memoria, occupando così solo un byte e non cinque.

Dalla locazione 2054 compresa alla 2058 compresa troviamo ciò che si era scritto dopo la Print, ossia il contenuto "UNO" memorizzato sotto forma di codici Ascii. Infatti il codice Ascii del carattere " è 34, del carattere U è 85 e così via. A questo punto la linea 10 è terminata, e infatti la locazione 2059 contiene uno 0 (ossia 'fine linea'). Dopo questa cella inizia la seconda linea introdotta (in questo caso la 20) con le locazioni 2060 e 2061 che contengono il link puntante in 2071 >23 + 8*256 + 2071> e vedremo poi cosa indica lo zero extra puntato da queste due locazioni.

2062 e 2063 contengono il valore 20 (seconda linea Basic). 2064 contiene ancora il valore 153: è la seconda istruzione Print presente nel programma. Da 2965 a 2069 troviamo gli altri codici Ascii del secondo messaggio che stampiamo con la linea 20, ossia "DUE". In 2070 troviamo ancora uno 0 che indica la fine della linea 20. In 2071 e in 2072 vi sono altri due zeri: si tratta dei due zeri consecutivi che ci indicano la fine del programma Basic in memoria. Le locazioni 45/46 puntano infatti all'ultimo di questi due zeri ossia alla fine del programma Basic in memoria.

A questo punto si dà per scontato che sia chiaro come un programma Basic viene posto in memoria. Se così non fosse, il metodo migliore è quello di eseguire un paio di prove, con linee Basic molto corte, e poi di andare a verificare cosa è realmente presente in memoria digitando:

```
FORA+PEEK<43>+PEEK<44>
*256TOPEEK<<45>+PEEK
<46>*256:PRINTA,PEEK <A>
:NEXT (return)
```

ottenendo così l'elenco delle locazioni in cui è memorizzato il programma affiancate dal loro contenuto.

Ricordate infine che la locazione precedente l'inzio del Basic deve contenere uno 0 (in questo caso la locazione è la 2048): se così non fosse, comparirebbero dei messaggi di SYNTAX ERROR e il RUN non verrebbe preso in considerazione. E' quindi importantissimo quando si va a spostare l'inizio del Basic che la locazione precedente l'inizio venga posta a 0: la formula generica che lo permette è:

```
POKE<PEEKL<43>+PEEK<44>
*256>-1,0 (return) (45/46 sul C128)
```

Ecco quindi come ingannare il tipo più banale di AutoRun: basta digitare in modo diretto: POKE 2048,1 (return), e caricare il programma: l'autorun dato servirà solo a fornire un ?SINTAX ERROR e a ridarci il controllo. Basta ora rimettere uno 0 nella locazione 2048 e tutto è a posto.

Prime applicazioni

Iniziamo a vedere le prime applicazioni ottenibili dopo le premesse date.

1. Modifica dell'inizio Basic.

Le applicazioni da cui si trarrà vantaggio imparando a utilizzare questa tecnica sono talmente numerose che non è possibile neppure un sommario elenco. Proviamo ora come applicazione a modificare l'inizio del Basic alle precedenti due linee che andranno copiate senza alcuna modifica. Si è già dimostrato come la seconda linea (la 20), inizi dalla locazione numero 2060. Digitate ora in modo diretto

```
POKE43,PEEK<2049>:POKE44,PE-
EK<2050> <R>
```

dove <R> indica la pressione del tasto Return. Come noterete, chiedendo il listato, la linea 10 non esiste più (provate a dare un RUN per accertarvene). Il motivo è semplicissimo: con la precedente istruzione siamo andati a leggere il primo link (quello della linea 10) il quale punta alla seconda linea e a tali valori abbiamo posto l'inizio del Basic: in tal modo questo link contiene già i due valori (LB e HB) in cui dobbiamo trasferire l'inizio del Basic, risparmiandoci così tutti i calcoli sulla divisione in LB e HB. Se non salvassimo il programma a questo punto, salveremmo la sola linea 20, in quanto la 10, pur essendo in memoria a tutti gli effetti, non viene considerata dal computer come linea Basic. Digitate ora in modo diretto:POKE43,1:POKE44,8 e la linea 10 ricompare nuovamente (eseguibile, listabile e

Tavola 1. Struttura di un programma Basic

Locazione	Contenuto	Carattere Ascii
2049	12 (link LB)	-
2050	8 (link HB)	-
2051	10 (linea LB)	-
2052	0 (linea HB)	-
2053	153 (print)	-
2054	34	"
2055	85	u
2056	78	n
2057	79	o
2058	34	"
2059	0 (fine linea)	-
2060	23 (link LB)	-
2061	8 (link HB)	-
2062	20 (linea LB)	-
2063	0 (linea HB)	-
2064	153 (print)	-
2065	34	"
2066	68	d
2067	85	u
2068	69	e
2069	34	"
2070	0 (fine linea)	-
2071	0 (fine prg)	-
2072	0 (fine prg)	-

salvabile). Alcune prove vi permetteranno di comprendere bene questa tecnica. La prima applicazione pratica è la seguente: se dotate un programma di autorun (si vedrà più avanti come fare) potete modificare l'inizio Basic dall'interno del programma che conterrà delle linee essenziali, in modo da escludere dall'area Basic il numero di linee desiderato. Un salvataggio del programma dato a questo punto salverebbe solo una fetta di programma e non tutto il listato, rendendo inutilizzabile il salvataggio eseguito.

2. Modifica dei link.

A differenza della modifica dell'inizio del Basic, la modifica dei link ha effetto solo sul listato del programma e non sull'esecuzione dello stesso. Provate a resettare il computer e a ridigitare sempre le due linee Basic precedenti e a scrivere in modo diretto:

```
POKE2049,PEEK<2060>:POKE
2050,PEEK<2061> (return)
```

Chiedete ora il listato: noterete che la linea 20 è scomparsa. Provate poi a dare il RUN: vedrete come la linea 20, anche se non letta dalla routine di LIST, viene eseguita regolarmente. Cosa abbiamo modificato? Con la poke precedente, abbiamo fatto puntare il primo link <2049/2050> al valore contenuto nel secondo link <2060 /2061>; in tal modo il primo link (della linea 10), punta direttamente alla fine del programma Basic, ma lo zero contenuto in 2059 informa sempre che la linea 10 termina a quel punto, e la routine di list crede che il programma sia terminato (il link successivo, infatti, lo porta a leggere i due zeri indicanti la fine del programma). Le applicazioni di questa tecnica sono decisamente interessanti: se fate puntare un link di linea anche all'interno di un programma a un altro link situato più avanti, tutte le linee intermedie non verranno più lette dalla routine di list, e anche chi conosce questi trucchi desisterà ben presto dal rimettere le cose a posto, dato l'eccessivo numero di controlli da eseguire se il programma è almeno di media lunghezza. Questo tipo di protezione parziale è una protezione statica: se salvate un programma con i link modificati, il programma verrà salvato ugualmente (o caricato) ma le linee nascoste continueranno a essere tali.

3. Merge di due programmi.

L'operazione di Merge consiste nell'unione grezza di due programmi, uno residente in memoria e l'altro su disco o cassetta. Date le premesse fatte, siamo ora in grado di compiere il modo velocissimo questa operazione in modo manuale e senza avere più bisogno di alcun programma in linguaggio macchina. Abbiamo detto che un programma Basic termina con due zeri indicanti appunto la fine del programma, e che l'ultimo di questi zeri è puntato da 45/46. Abbiamo visto inoltre come l'ultimo link dell'ultima linea Basic ci fornisca direttamente la locazione da cui dovremmo caricare il secondo programma (in coda a uno già presente in memoria). Resta l'ultimo problema: quanto dista il link finale (quello da cui dovremo prelevare i valori) dalla fine del Basic? Un minimo di ragionamento ci mostra come tale valore vari col variare dell'ultima linea Basic in memoria. Ecco l'idea: se aggiungiamo in coda al programma in memoria sempre la stessa linea Basic, il link in questione disterà sempre lo stesso numero di locazioni dalla fine del Basic e potremo così scrivere una regola generale applicabile sempre. Ecco come procedere per punti:

- aggiungete in coda al programma Basic in memoria una linea nuova contenente una sola REM (es.40000 REM).
- scrivete in modo diretto:PRINT PEEK<45> + PEEK <46>*256 (return) e prendete nota del numero che verrà fornito (lo chiameremo X).
- scrivete sempre in modo diretto: POKE43,PEEK<X-8> (return) e POKE44,PEEK<X-7> (return), dove X è il valore fornito al punto precedente.
- caricate ora il secondo programma Basic con un normale LOAD.
- digitate in modo corretto: POKE43,1 e POKE44,8.

I due programmi Basic sono ora uniti in memoria e consecutivamente, e possono essere eseguiti, listati e salvati. Come tutti i programmi di Merge è naturalmente richiesto che il primo numero di linea del secondo programma sia superiore all'ultimo numero di linea del primo programma in memoria, e che i due programmi possano essere entrambi contenuti in memoria senza generare degli out of memory.

Il funzionamento dovrebbe essere chiaro: introducendo la REM del punto uno abbiamo modo di sapere dove si trova l'ultimo link che ci fornisce la locazione da cui caricare il secondo programma per fare in modo che sia linkato correttamente. Il secondo punto preleva proprio questi due valori e li pone direttamente in 43/44, ossia alza l'inizio del Basic al punto da cui si deve caricare il programma B. Ora il LOAD carica il secondo programma proprio da questa locazione (correttamente linkata) e le ultime due poke rimettono l'inizio Basic ai suoi abitudinari valori (ossia 2049).

Inutile soffermarsi sull'utilità di questa routine: una volta imparata a memoria consente in un attimo di unire due programmi Basic senza perdere tempo.

Un consiglio:se create una enciclopedia di routine Basic per i vostri programmi, la salvate su periferica e la numerate con un numero di linee altissimo, potete usare il precedente Merge per caricare la routine desiderata ogni volta che lo volete. Occorrerà poi avere sempre in memoria un programma in L/M che risieda fuori dal Basic per renumerare tutto il programma ogni volta che avete caricato una routine da periferica. Potete usarlo anche per vedere la Directory senza perdere il programma: basterà in tal caso dare un New, prima di riabbassare l'inizio Basic in 2049.

Ricordate che ci sono alcune forme di caricamento della Direcotory che non tutti conoscono:

- LOAD"$*=PRO", che carica la Directory dei soli file programma,
- LOAD"$*=SEQ", che carica solo i file sequenziali ecc.
- Valide anche USR e REL, e la sostituzione dell'asterisco con una chiave di ricerca.

4. Reverse e colorazione.

Osservando la **tavola 2** noterete come alcuni codici presenti non corrispondano né a una lettera né a un Token, ma diano ugualmente degli effetti interessanti. Digitate il programma esempio:

```
10 REM*
20 PRINT "CIRO"
```

e scrivete in modo diretto: POKE 2054,5 (return). La seconda linea del listato apparirà sempre di colore bianco e l'asterisco dopo la REM è sparito. Cos'è successo? Semplicemente è stato creato uno spazio extra in memoria (l'asterisco) e manualmente è stato sostituito con il valore 5 (ossia il colore bianco). Provate a premere STOP+RESTORE e a scrivere PRINT CHR$<5> per sincerarvene. Il valore 5 posto dopo una REM viene in realtà eseguito, e non mostrato come una parte del LIST. Il programma infatti ignora la presenza del 5 in memoria, perché la REM lo fa passare all'esecuzione della linea successiva, mentre la routine di LIST lo deve leggere per forza, e leggendolo lo esegue. I codici che possiamo utilizzare per esperi-

menti di questo tipo sono i seguenti:

```
valore  effetto
--------------------
5. colora di bianco
8. disabilità SHIFT/COMMODOR-
...riabilita SHIFT/COMMODORE
13. cursore a capo
14. set minuscolo/maiuscolo
17. cursore giù
18. reverse on
28. colora di rosso
30. colora di verde
31. colora di blu
204. stampa SINTAX ERROR
```

Volendo quindi per esempio ottenere tutte le Rem del programma scritte in Reverse, basta sostituire il primo carattere seguente la Rem con un 18: l'effetto termina con la fine della linea su cui si pone il reverse (a differenza dei colori che restano attivi). Una routine che esegue l'operazione manualmente (empirica) è la seguente:

```
60000 I=PEEK<43>+PEEK<44>
*256
60010 F+PEEK<45>+PEEK<46>
*256
60020 FORA=1TOF
60030IFPEEK<A>=143 AND PEEK
<A+1>+32THENPOKEA+2,18
60040 NEXT
```

Abbiamo detto che è un modo empirico di procedere per il seguente motivo: il programmino (da aggiungere in coda al programma che si desidera e poi da cancellare) testa la presenza di un 143 (token della REM) e se è seguito da uno spazio (32) allora esegue la modifica. Ma un 143 seguito da un 32 può trovarsi nel programma anche per altri motivi (parti di un link o altro) e quindi la routine non è infallibile. Per accertarsene occorre vedere il listato: se tutto è OK, allora siamo a posto, se compaiono caratteri strani è necessario eseguire l'operazione manualmente (funziona in un buon 90% dei casi). Le modifiche date in questo modo restano permanentemente anche dopo il salvataggio e il successivo ricaricamento. Un valore 204 provoca invece, chiedendo il LIST, la stampa di un messaggio di errore e l'interruzione dell'esecuzione.

Se possedete un C128 (in modo 128) astenetevi da questi esperimenti: il C128 possiede oltre 140 comandi Basic e molti Token sono memorizzati nella parte bassa seguita dalla parte alta, per cui sorgono problemi inaspettati da tutte le parti; potete invece utilizzare il metodo della sostituzione manuale ogni volta che lo volete. La tecnica della sostituzione di caratteri dopo una Rem è utilizzata anche da chi vuole inserire cortissime routine in LM inglobandole all'interno del programma Basic.

Tavola 2.

Codici token del Basic

Token	Codice Ascii	Token	Codice Ascii
abs	182	or	176
and	175	peek	194
asc	198	poke	151
atn	193	pos	185
chr$	199	print	153
close	160	print#	152
clr	156	read	135
cmd	157	rem	143
cont	154	restore	140
cos	190	return	142
data	131	right$	201
def	150	rnd	187
dim	134	run	138
end	128	save	148
exp	189	sgn	180
fn	165	sin	191
for	129	spc	166
fre	184	sqr	186
get	161	status	---
get#	---	step	169
gosub	141	stop	144
goto	137	str$	196
if	139	sys	158
input	133	tab	163
input#	132	tan	192
int	181	then	167
left$	200	time	---
len	195	time$	---
let	136	to	164
list	155	usr	183
load	147	val	197
log	188	verify	149
mid$	202	wait	146
new	162	+	170
next	130	-	171
not	168	*	172
on	145	:	173
open	159	=	178

5.Protezioni e sprotezioni.

Per protezione di un programma si intende la non accessibilità al listato. Chi è in grado di programmare in linguaggio macchina (ed è in grado di redigere un intero programma in LM senza il Basic) è già in grado di inventarsi da solo sistemi per proteggere il suo lavoro. I problemi sorgono volendo proteggere un listato Basic dagli occhi dei copiatori di programmi e dai ladri di routine. Un sistema valido è la modifica dei link all'interno del programma, già spiegata in questo articolo. Un altro sistema adottato (ma nessuno vieta di usarli anche tutti contemporaneamente) è quello di dotare il programma di AutoRun e disabilitare il tasti STOP e RESTORE. Sugli AutoRun si sono visti in giro programmi validi affiancati da altri a dir poco ridicoli. Daremo qui alcune idee su come fornire un AutoRun da disco un po' valido (gli AutoRun su cassetta sono laboriosi da eseguire a mano e quindi è meglio procurarsi un programma protettore e vedere come funziona per costruirsene un altro migliore). Per AutoRun si intende la partenza del programma appena caricato, in maniera automatica.

Si distinguono due tipi di AutoRun:
1. S AutoRun a riempimento di buffer.
2. * AutoRun immediati.

I secondi sono i più efficaci e vedremo come sia facile costruirsene uno senza bisogno del linguaggio macchina e in pochi secondi. Gli AutoRun a riempimento di buffer, come dice il nome, funzionano mandando in esecuzione un programmino in linguaggio macchina che riempie il buffer di tastiera con le lettere R-U-N-CHR$<13> e poi torna al Basic. Tornati al Basic viene scritto immediatamente RUN (ret.) e il programma parte. La routine in LM è la seguente:

```
CLI
LDA #4
STA 198
LDA #'R
STA 631
LDA #'U
STA 632
LDA #'N
STA 633
LDA #13
STA 634
SEI
RTS
```

I numeri sono espressi in decimale, con 'R si intende il codice ASCII della lettera R e così via. Andando a locare questa routine da 828 (buffer del registratore) e abbassando l'inizio del Basic in 770 (ove è contenuta la routine di READY) con POKE43,2 e POKE44,3, il primo dei due passi è compiuto. Ora dovete dividere il valore 828 nella parte alta e bassa secondo la formula:

```
H+INT<X/256>
L+X-H*256
```

dove H è la parte alta, L la parte bassa, e X il numero in questione (in questo caso 828). Ora dovete scrivere in modo diretto:POKE770,L:POKE771,H:SAVE"[clr]nome programma",8 (return). Terminato il salvataggio basta riportare il Basic ai soliti valori con POKE43,1 e POKE44,8. Nella formula appena data L rappresenta la parte bassa del numero, mentre H è la parte alta.

Perché funziona questo AutoRun? La soluzione è facile come sempre: la routine di READY (quella che viene eseguita a partire dalla locazione puntata da 770/771 ogni volta che vediamo il READY sullo schermo) può essere modificata a piacere e fatta puntare a ciò che desideriamo (purché in linguaggio macchina). Se la routine di READY, tramite modifica in 770/771 viene fatta puntare al programmino che riempie il buffer di tastiera con le lettere RUN (return) si ha la stampa del messaggio RUN (return) appena terminato il caricamento del programma. Infatti, quando caricate un programma, al termine vedete proprio la scritta READY. Il RUN impartito fa quindi partire il programma.

Abbiamo abbassato l'inizio del Basic in 770 e 771 perché così facendo salviamo tutta la memoria a partire dalla locazione numero 770 fino alla locazione numero PEEK<45>+PEEK<46>*256 e quindi salviamo alche il programmino di autorun in LM, la memoria video (1024/2023) e tutto il programma Basic da 2049 in poi. Quando carichiamo il programma salvato con un LOAD"nome",8,1 andiamo a modificare di nuovo le locazioni 770 e 771 e le facciamo riportare alla routine di AutoRun e quindi il programma parte. E' importante caricare il programma utilizzando l'indirizzo secondario (,8,1) perché in caso contrario il caricamento avverrebbe a partire dall'area Basic (2049).

L'ultima cosa da chiarire è il nome che abbiamo dato al programma: perché abbiamo utilizzato il [clr] all'interno del nome del programma? Si è detto che anche lo schermo video viene salvato assieme al programma Basic, quindi se non facessimo in questo modo, verrebbero salvate anche le istruzioni che stiamo scrivendo sullo schermo, dato che lo schermo (da1024) resta visibile durante il caricamento. Questo espediente cancella il video durante l'ultima fase del salvataggio e lascia solo il nome del programma scritto sullo schermo in alto a sinistra. Una volta salvato il programma su disco, è opportuno cambiargli nome per evitare di vedere sempre quello sgradevole simbolo in reverse all'interno del nome. Per compiere tale operazione si deve usare la routine di RENAME, previa apertura del canale 15 (vedere il manuale del disk in proposito). Se tutto il procedimento non fosse ben chiaro, si consideri la differenza fra un LOAD"nome",8 e un LOAD"nome",8,1. Nel primo caso il programma viene caricato a partire dalla locazione puntata da 43/44, mentre nel secondo caso viene caricato a partire dalla locazione da cui è stato salvato. Ciò significa che assieme al programma salvato su disco, il computer salva anche un paio di locazioni che indicano da quale zona deve essere caricato il programma. Un programmino per leggere queste due locazioni è il seguente:

```
10 SYS65511:OPEN8,8,12,"nome
programma"
20 GET#*,A*:GET#8,B$:CLOSE8
30 I=ASC<A$+Z$>+ASC<B$+Z$>
*256:END
```

al termine la variabile I conterrà la locazione da cui il programma è stato salvato, e da cui verrà caricato con LOAD"nome",8,1. Se I vale 2049, è un programma Basic, in caso contrario è molto probabilmente un programma in LM, oppure un programma Basic salvato da un C128 o un programma Basic salvato dopo una modifica di puntatori.

Di natura analoga al precedente è il secondo tipo di AutoRun, quello permanente, molto più efficace del precedente.

Immaginate il metodo appena visto: a rendere scomoda la preparazione di questo AutoRun è l'inserimento del linguaggio macchina e il fatto che il programma non parte direttamente, ma viene prima eseguita una routine in LM che dà l'AutoRun. Se il programma potesse partire con una SYS di attivazione tutta l'operazione diventerebbe talmente veloce e immediata da sembrare quasi un gioco. In altre parole, immaginiamo ancora un attimo di avere un programma che parta con una SYS (e non con un RUN): basterebbe solo abbassare il Basic in 770 come nel programma precedente, e far puntare la routine di READY (in 770) all'inizio del programma che parte con una SYS e il gioco sarebbe fatto. Naturalmente, un programma che parte con una SYS deve essere interamente in linguaggio macchina, a meno che...non venga compilato.

Prendete un programma Basic a caso e compilatelo con un qualsiasi compilatore (PETSPEED, BLITZ o ciò che possedete). Caricate il programma compilato in memoria e abbassate l'inizio del Basic in 770, con una POKE 43,2 e POKE44,3. Dividete la SYS di attivazione del compilatore (quella che viene indicata nell'unica linea Basic che il compilatore lascia, e che è 2073 per PETSPEED, o un altro valore per gli altri compilatori) nella parte alta e bassa seconda la formula prima descritta, quindi salvate il programma con POKE770,L:POKE771,H:SAVE"[clr]nome",B. Resettate ora il computer e verificate il corretto funzionamento del programma appena salvato caricandolo con LOAD"?nome",8,1. Premendo RUN/STOP+RESTORE, la fuoriuscita dal programma non fa altro che farlo ripartire da capo, facendovi perdere per sempre il controllo diretto della tastiera e facendovi ritrovare di continuo all'interno del programma in esecuzione.

Fausto Molinari