L'istruzione POKE

A partire da questa puntata e a meno di "interruzioni" dovute a contributi da parte dei lettori, occuperemo lo spazio dedicato all'MBASIC con qualcosa di, speriamo, interessante, ma al tempo stesso alquanto complesso e delicato: ci proponiamo di andare ad analizzare "dall'interno" l'MBASIC stesso, studiando volta per volta le sue istruzioni e le sue subroutine più interessanti. La difficoltà insita in tale lavoro consiste nel fatto che, come più volte accennato nelle scorse puntate, non facciamo riferimento ad alcun testo in particolare, ma soltanto all'MBASIC. COM (inteso come un file da analizzare), sull'"analizzatore" ZSID (una versione per Z80 del DDT) e sull'esperienza maturata in anni di studio ed applicazioni dell'Assembler.

Con questo ancora una volta ci rendiamo conto che talora potremo incorrere in qualche inesattezza (segnalazioni in merito da parte dei lettori saranno le benvenute), ma tutto sommato confidiamo in pieno nei progettisti della Microsoft, i quali hanno fatto veramente un ottimo lavoro...

Innanzitutto dobbiamo dire che l'MBASIC è stato scritto in Assembler dell'8080 e non dello Z80: questo lo diciamo in quanto nel corso delle puntate faremo sempre riferimento al più potente microprocessore, al cui set di istruzioni è compatibile quello dell'8080. Il fatto che l'MBASIC è scritto "in 8080" comporta (ed è manifestato dal fatto) che si incontrano particolari "trucchi" oramai superati nello Z80; inoltre non si trova traccia, ovviamente, dei registri indice IX ed IY, nonché di istruzioni utilissime quali la LDIR.

Un'altra considerazione riguarda invece l'utilizzazione pratica delle informazioni riportate in questi articoli: già nei numeri 34, 35 e 36 di MC, nell'ambito della rubrica "I trucchi del CP/M", abbiamo trattato una parte di questo argomento, proponendo tra l'altro alcune istruzioni, che si possono introdurre semplicemente nel proprio Basic.

In un certo senso con questi articoli effettuiamo una sorta di passo indietro, per andare a conoscere più da vicino il programma in esame: perciò in parecchie occasioni faremo implicito o esplicito riferimento alle notizie riportate in tali numeri di MC. Un'ultima annotazione e poi iniziamo l'analisi: nello studiare le varie parti dell'MBASIC, sfrutteremo il metodo ingegneristico detto "Top-Down" applicato alla programmazione, metodo secondo il quale si analizza un programma già esistente spezzandolo via via in parti sempre più piccole da analizzare a loro volta singolarmente.

Tutte le volte che si utilizza tale approccio e perciò in particolar modo nell'analisi di un programma scritto da altri, è assolutamente necessario conoscere alla perfezione che cosa fa quel tale programma o una certa subroutine: non importa a questo punto se alcune parti, in cui abbiamo frazionato il problema riescono più difficili di altre, dal momento che è in prima analisi più importante avere una visione globale, da raffinare solo a seconda di ciò che ci si prefigge.

Evidentemente noi ci fermeremo ad un certo livello di analisi, per non appesantire troppo il discorso e soprattutto per lasciare ai lettori interessati il gusto di procedere oltre.

Consigliamo dunque fin d'ora di tenere sottomano i tre numeri di MC già citati, nonché il numero 38, dove a pag. 163 e seguenti troviamo la "jump table" dei comandi e delle funzioni dell'MBASIC.

L'istruzione POKE

Iniziamo dunque l'analisi con questa istruzione ben nota ai lettori: l'abbiamo scelta per prima dal momento che ci sembra molto istruttiva per quanto ci proponiamo di fornire ai lettori. Dalla tabella di cui sopra apprendiamo che il comando in esame, come subroutine Assembler, inizia all'indirizzo esadecimale *22B1H*: vediamo subito il significato di questa affermazione ed in particolare in quale contesto ha senso parlare di POKE e di subroutine corrispondente.

La distinzione tra i termini "funzione" e "comando", innanzitutto comporta una differenza notevole fra le strutture delle relative subroutine: tale distinzione si ha in quanto differente è quella che si chiama in gergo la "semantica" di un comando e di una funzione.

In termini pratici per semantica si intende propriamente "ciò che l'istruzione esegue", in contrapposizione alla "sintassi", che invece si occupa delle regole secondo le quali deve essere impostato un comando o scritta una funzione.

Semanticamene, perciò, le funzioni in generale comportano un certo calcolo logico, algebrico o trascendentale che sia, del quale forniscono un risultato sotto forma di un bit, di un byte, di un valore intero, reale o in doppia precisione; inoltre consentono di conoscere "lo stato" di un certo elemento del computer stesso o di un programma (la quantità di memoria, la posizione del cursore, il valore di un flag, ecc).

I comandi invece, come d'altro canto è implicito nel loro nome, forzano l'interprete oppure il Sistema Operativo stesso a compiere una determinata azione o serie di azioni (stampa di un risultato, accensione di una periferica, partenza di un programma, ecc.).

Ecco che perciò intuitivamente un comando potrà essere impartito semplicemente ordinando al complesso insieme "interprete/ Sistema Operativo" di eseguire una certa determinata subroutine, finita la quale (e cioè dopo aver fisicamente eseguito il comando), il controllo deve passare per forza all'interprete o al Sistema Operativo.

Tornando perciò alla nostra POKE, noi sappiamo che si tratta di un comando che consente di scrivere in un certo punto della

memoria un certo valore: la sintassi della POKE è la seguente:

```
POKE <address>, <byte>
```

come dire che, come ben sanno gli "smanettomani", per porre il valore 20H nella locazione di indirizzo 1234H basterà impostare:

```
POKE 1234H,10H
```

In generale però tanto "<address>" quanto "<byte>" possono essere dei valori calcolati con un'espressione, si badi bene, intera: la differenza tra i due valori è che "<address>" potrà avere un valore compreso tra 0000H e FFFFH, mentre "<byte>" potrà variare tra 00H e FFH.

Vediamo dunque cosa dobbiamo aspettarci dalla subroutine corrispondente: per un istante pensiamo di essere noi l'interprete Basic ed in particolare quella parte di esso che prende il nome di "Analizzatore Lessicale-Sintattico". Questo strano essere, che i neofiti non conosceranno, è un oggetto che analizza passo passo il programma, istruzione per istruzione, alla ricerca di ciò che la sintassi richiede, prendendo eventualmente opportune decisioni se si verificano eventi contrari alle regole definite dalla sintassi stessa.

Routine Utily dell'MBASIC

ADDRESS 22C2H	Calcola un'espressione e pone il risultato in DE
GETCHAR 43C7H	Legge il testo e trova il byte posto subito dopo la chiamata.
BYTE 2066H	Calcola un'espressione e pone il risultato in A

Ecco che, scrivendo "POKE <address>, <byte>", dopo aver incontrato il comando POKE, l'analizzatore passa il controllo alla subroutine relativa all'istruzione stessa, subroutine che dovrà continuare lei ad analizzare il resto del comando per verificare che la sintassi sia stata seguita.

Perciò la subroutine relativa a POKE:
— dovrà trovare subito un'espressione da calcolare per ottenere un <address>, che memorizzerà poi da qualche parte
— dovrà trovare una virgola di separazione
— dovrà trovare un'altra espressione, che calcolerà, per ottenere a sua volta un <byte> da andare a scrivere proprio nell'indirizzo calcolato e memorizzato prima.

Qualsiasi altra cosa trovi al di fuori di queste, dovrà segnalare errore con un generico "SYNTAX ERROR": ecco che se scriviamo

```
POKE 0;5
```

dobbiamo aspettarci la segnalazione di errore in quanto al posto della virgola abbiamo messo un "punto e virgola".

Andiamo dunque a vedere cosa effettivamente troviamo a partire dall'indirizzo *22B1H*: avendo in mente già cosa dobbiamo aspettarci e con un pizzico di intuizione (che certo non guasta nel nostro compito) ecco ciò che leggiamo:

```
CALL 22C2H
PUSH DE
CALL 5D41H
CALL 43C7H
DEFB ','
CALL 2066H
POP DE
LD (DE),A
RET
```

Non spaventiamoci, ma vediamo di cosa si tratta: dopo una prima chiamata a *22C2H* salviamo la coppia DE nello stack. Guarda caso poco prima della "RET" troviamo il ripristino della coppia DE con il valore precedentemente salvato nello stack (POP DE) ed infine troviamo la memorizzazione del contenuto dell'accumulatore nella cella il cui indirizzo è posto in DE.

Non ci vuole molto per capire che è proprio quello che dobbiamo fare con la POKE!

Allora ci possiamo "sbilanciare" dicendo che la subroutine *22C2H* analizza il testo, calcola un'espressione, ponendone il risultato in DE.

Lasciando da parte la routine *5D41H*, che probabilmente effettua un controllo sul range di valori di <address>, troviamo la *43C7H*, della quale abbiamo già parlato nel n. 34: ricordiamo che, analizzandola, si scopre che il byte dopo la chiamata stessa rappresenta il carattere ASCII che dobbiamo incontrare nel testo. In particolare nel nostro caso troviamo il valore *2CH* (il codice ASCII relativo alla virgola) che altrimenti non avrebbe senso come istruzione Assemler (INC L): i lettori scettici troveranno moltissime conferme di questo fatto in altri casi in cui l'istruzione Assembler trarrebbe soltanto in inganno.

Ecco che perciò la *43C7H* significa: analizza il testo per trovare il byte posto subito dopo la chiamata stessa e nel caso in cui tale byte non venga trovato, uscire dalla routine per errore di sintassi.

Infine la subroutine *2066H* analizza il testo per trovare un'espressione, da calcolare ed il cui risultato porre nell'accumulatore. Detto ciò si ritorna alla POP DE e a quanto detto prima.

Per ricapitolare perciò quanto abbiamo ricavato, possiamo tradurre la routine in termini simbolici, che cercheremo di usare ancora in seguito nell'analisi di altre istruzioni:

```
POKE
CALL ADDRESS
PUSH DE
CALL SETADDR
CALL GETCHAR
DEFB ','
CALL BYTE
POP DE
LD (DE),A
RET
```

MC