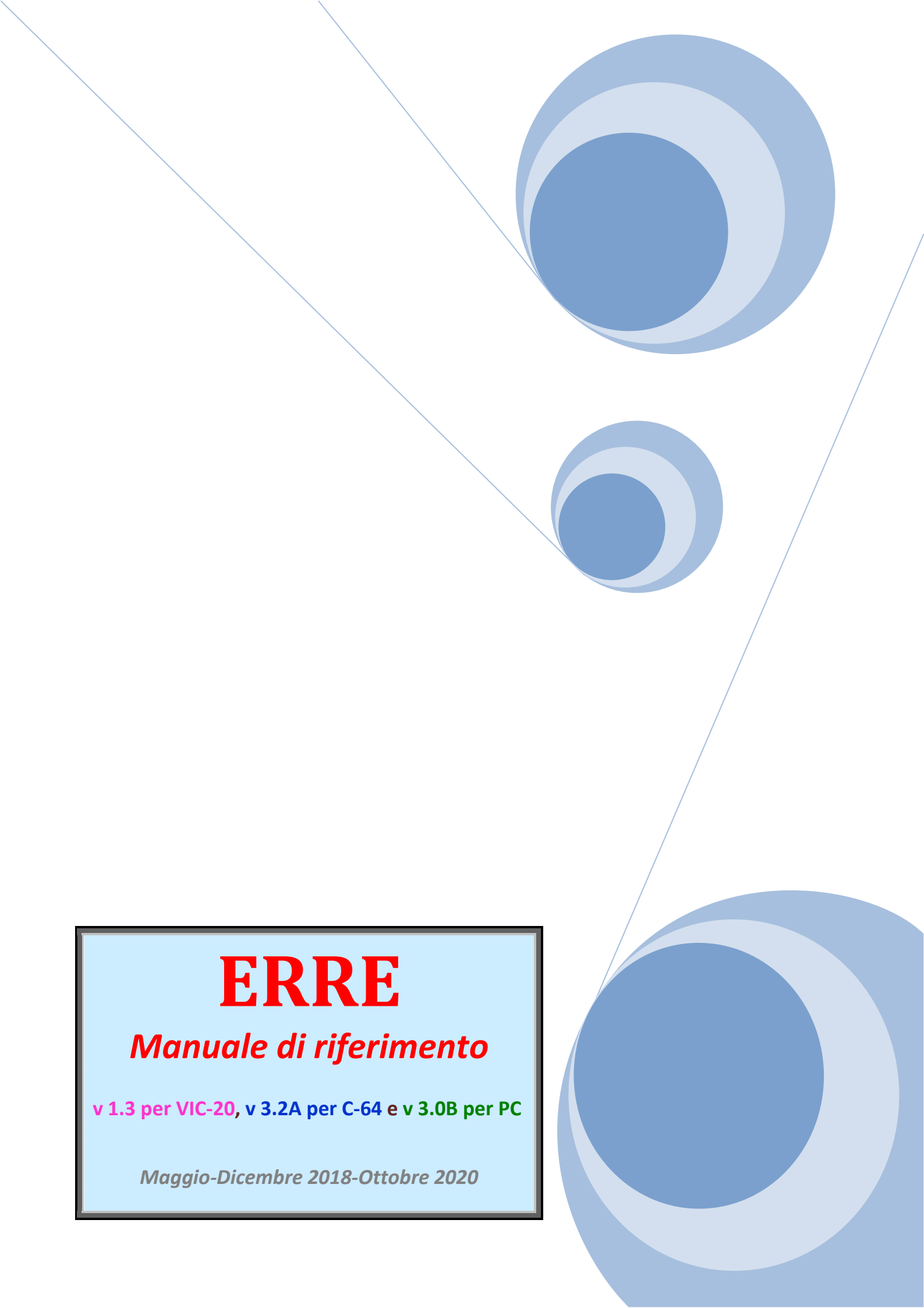


ERRE

Manuale di riferimento

v 1.3 per VIC-20, v 3.2A per C-64 e v 3.0B per PC

Maggio-Dicembre 2018-Ottobre 2020

ERRE

Manuale di Riferimento

v 1.3 per VIC-20, v 3.2A per C-64 e v 3.0B per PC

Maggio-Dicembre 2018
Ottobre 2020


erre project
64

INDICE GENERALE

INDICE GENERALE

INTRODUZIONE

Questo manuale si affianca ai manuali delle varie versioni degli ambienti di programmazione di ERRE System e al libro, ormai giunto alla settimana edizione, "PROGRAMMARE IN ERRE".

Vengono presentati, in ordine alfabetico, tutti i costrutti del linguaggio specificando piattaforma di riferimento, scopo, sintassi, esempi d'uso nonché eventuali note di chiarimento su aspetti particolari.

Ci si riferirà alle ultime versioni disponibili per ogni piattaforma ed il tutto verrà identificato da un colore e precisamente:

- **ERRE-VIC20 1.3** per il VIC-20
- **ERRE-C64 3.2** per il C-64 (**sotto-versione "A"**)
- **ERRE-PC 3.0** per il PC-IBM e compatibili (**sotto-versione "B"**)

Occasionalmente verranno fatti riferimenti ad altre versioni "storiche" (**ERRE-PC 2.6** sotto-versione “A” e **ERRE-C64 2.3**) la cui conoscenza non è comunque indispensabile per la programmazione odierna: questo a dimostrazione dell'evoluzione subita dal linguaggio nel corso del tempo.

Claudio Larini

codici di errore

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

L'interprete di R-Code fornisce, in caso di errore di esecuzione, un codice numerico che rappresenta l'errore stesso e che provoca il blocco dell'esecuzione stessa a meno che non si si sia attivato un blocco **EXCEPTION** . . **END EXCEPTION** che consente il trattamento dell'errore stesso.

I codici di errore possono essere compresi tra 1 e 255 e sono dipendenti dalla piattaforma, rappresentando così un ostacolo alla portabilità del linguaggio. A questo fine sono state implementate (solo per **ERRE-C64 3.2** e **ERRE-PC 3.0**) le cosiddette "**named exceptions**" che sono rappresentano in chiaro il codice numerico e possono essere usate solo nel blocco **EXCEPTION**.

ERRE-PC 3.0

Codice		*Messaggio*	*"Named exception"*
1	*	NEXT without FOR	
2		Syntax error	?SYNTAX
3	*	RETURN without GOSUB	
4		Out of DATA	?OUT_OF_DATA
5		Illegal function call	?ILLEGAL_FN_CALL
6		Overflow	?OVERFLOW
7		Out of memory	?OUT_OF_MEMORY
8	*	Undefined line number	
9		Subscript out of range	?BAD_SUBSCRIPT
10	*	Duplicate Definition	
11		Division by zero	?DIV_BY_ZERO
12	*	Illegal direct	
13		Type mismatch	?TYPE_MISMATCH
14		Out of string space	?OUT_OF_STRING_SPACE
15		String too long	?STRING_TOO_LONG
16		String formula too complex	?FORMULA_TOO_COMPLEX
17	*	Can't continue	
18	*	Undefined user function	
19	*	No RESUME	
20	*	RESUME without error	
21	*	Unprintable error	
22		Missing operand	?MISSING_OPERAND
23		Line buffer overflow	?LINE_BUFFER_OVERFLOW
24		Device Timeout	?DEVICE_TIMEOUT
25		Device Fault	?DEVICE_FAULT
26	*	FOR Without NEXT	
27		Out of Paper	?OUT_OF_PAPER
28	*	Unprintable error	
29	*	WHILE without WEND	
30	*	WEND without WHILE	
31-49	*	Unprintable error	
50	*	FIELD overflow	
51	*	Internal error	
52		Bad file number	?BAD_FILE_NUMBER
53		File not found	?FILE_NOT_FOUND
54		Bad file mode	?BAD_FILE_MODE
55		File already open	?FILE_ALREADY_OPEN

```
56   *   Unprintable error
57       Device I/O Error                ?DEVICE_I/O
58   *   File already exists
59-60 *  Unprintable error
61       Disk full                       ?DISK_FULL
62       Input past end                  ?INPUT_PAST_END
63       Bad record number               ?BAD_RECORD_NUMBER
64       Bad filename                    ?BAD_FILENAME
65   *   Unprintable error
66   *   Direct statement in file
67       Too many files                  ?TOO_MANY_FILES
68       Device Unavailable              ?DEVICE_NOT_PRESENT
69       Communication buffer overflow ?COM_BUFFER_OVERFLOW
70       Permission Denied               ?PERMISSION_DENIED
71       Disk not Ready                  ?DISK_NOT_READY
72       Disk media error                ?DISK_MEDIA
73   *   Advanced Feature
74   *   Rename across disks
75       Path/File Access Error          ?PATH/FILE_ACCESS
76       Path not found                  ?PATH_NOT_FOUND
77   *   Deadlock
78-255   User-defined errors
```

ERRE-C64 3.2

num. errore	descrizione	"named exceptions"
01	Too many file open	?too←many←files
02	File open	?file←already←open
03	File not open	?file←not←open
04	File not found	?file←not←found
05	Device not present	?device←not←present
06	Not input file	?not←input←file
07	Not output file	?not←output←file
08	Missing file name	?missing←filename
09	Illegal device	?illegal←device
10 *	Next without for	
11	Syntax error	?syntax
12 *	RETURN without GOSUB	
13	Out of data	?out←of←data
14	Illegal quantity	?illegal←fn←call
15	Overflow	?overflow
16	Out of memory	?out←of←memory
17 *	Undef'd statement	
18	Bad subscript	?bad←subscript
19 *	Redim'd array	
20	Divisione by zero	?div←by←zero
21 *	Illegal direct	
22	Type mismatch	?type←mismatch
23	String too long	?string←too←long
24	File data	?bad←file←data
25	Formula too complex	?formula←too←complex
26 *	Can't continue	
27 *	Undef'd function	
28 *	Verify	
29 *	Load	
30 *	Break	

Nota:

- I codici numerici di errore segnati con * non si possono verificare con un programma ERRE compilato.
- I codici numerici di errore per **ERRE-VIC20 1.3** e **ERRE-C64 3.2** sono uguali.

etichetta (label)

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

Le etichette ("labels") in ERRE servono come riferimenti per le istruzioni **GOTO** e **RESUME**.

Formato:

In **ERRE-VIC20 1.3**:

stringa. Non è necessaria alcuna dichiarazione.

In **ERRE-C64 3.2** e **ERRE-PC 3.0**:

numero_intero: dove numero_intero è compreso tra 1 e 9999.
Le etichette vanno dichiarate con l'apposita istruzione **LABEL**.

identificatore di variabile

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

In ERRE esistono dei simboli speciali che identificano il tipo delle costanti e delle variabili, sia semplici che aggregate.

identificatore	tipo	versione
(nessuno)	real	ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0
%	integer/boolean	ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0
#	double real	ERRE-PC 3.0
$	string	ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0
?	generic	ERRE-PC 3.0

Note:

- I tipi di variabile possono essere sovraccaricati ('overloaded') con le direttive di compilazione **$DOUBLE**, **$INTEGER**, **$SINGLE** e **$STRING**. (solo **ERRE-PC 3.0**)
- I tipi numerici in ERRE sono tutti dotati di segno ("signed"). In certi casi è conveniente poter usare anche il relativo "unsigned"; così ad esempio è possibile ottenere l' "unsigned integer" dal "signed integer" (e viceversa) con le seguenti funzioni:

```
! da integer ad unsigned integer
FUNCTION UI(X)
   UI=X-(X<0)*65536
END FUNCTION

! da unsigned integer a integer
FUNCTION IU(X)
   IU=X+(X>32768)*65536
END FUNCTION
```

- I valori delle variabili sono codificati secondo il formato "Microsoft Binary Format" - MBF - che assegna 16 bit alle variabili 'integer', 32 bit alle variabili 'real' (**ERRE-PC 3.0**), 40 bit alle variabili 'real' (**ERRE-VIC20 1.3 ERRE-C64 3.2**) e 64 bit alle variabili 'double real' (**ERRE-PC 3.0**).

letterale

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

I letterali in ERRE possono essere di quattro tipi diversi:

1) *letterali numerici decimali* che sono rappresentati dai classici numeri, ad esempio

34 , -17.3 , 1.23E-09, 5.11D-11

E' possibile, in caso di ambiguità, porre un '#' per identificare un letterale in doppia precisione rispetto ad uno in semplice precisione. Le lettere 'E' ed 'D' indicano "per 10 elevato alla", rispettivamente in singola e doppia precisione (quest'ultima solo per **ERRE-PC 3.0**).

2) *letterali numerici non decimali* che possono essere solo interi e sono rappresentabili in base 2, in base 8 e in base 16; per distinguerli si antepone alla costante rispettivamente il simbolo "%", "&" e "$". Ad esempio

$C000 equivale al decimale 49152
-$FF equivale invece a -255
%1000 equivale a 8
-%111 equivale a -7
&7777 equivale a 4095

I letterali numerici non decimali sono di tipo "unsigned" e sono codificati in 32 bit (**ERRE-PC 3.0**) o 24 bit (**ERRE-C64 3.2**). **ERRE-VIC20 1.3** ha solo i letterali numerici non decimali in base 16 che sono codificati su 16 bit (0..65535).

3) *letterali stringa* che sono insiemi di caratteri delimitati dal doppio apice **"**, ad esempio

"pippo" , "1+2-ABC"

4) *letterale speciale* **π** che consente di utilizzare direttamente la nota costante pigreco= 3.14159265...: in **ERRE-PC 3.0** π è una costante LONG REAL pari a 3.141592653589793 sebbene nell'ambito di una espressione a singola precisione sia utilizzato il valore 3.141593, mentre in **ERRE-VIC20 1.3** e **ERRE-C64 3.2**. è una costante REAL con valore 3.14159265. In quest'ultimo caso l'equivalente matematico 4*ATN(1) fornisce un errore di 1 sull'ultima cifra decimale rappresentabile. In **ERRE-PC 3.0** è possibile utilizzare il letterale **π'** per indicare direttamente il valore in singola precisione, a patto che sia stata utilizzata la direttiva **$IMPLICIT**.

Nota:

- I letterali non decimali e π non possono essere utilizzati come risposta all'istruzione INPUT di variabili numeriche, ma solo con variabili stringa e poi convertiti in letterali decimali.
- La doppia precisione non è disponibile su VIC-20 e C-64, anche se il tipo REAL ha una precisione migliore rispetto a quella del PC.
- π è inseribile tramite la sequenza Alt + 227 in **ERRE-PC 3.0** e direttamente in **ERRE-VIC20 1.3** e **ERRE-C64 3.2**.

operatore

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

Gli operatori in ERRE possono essere di quattro tipi diversi:

- **operatore di indirizzamento unario**
  & (ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0)
  indirizzo di memoria di una variabile
  indirizzo di un File Control Block (F.C.B.) di un file (solo ERRE-PC 3.0).

- **operatori aritmetici unari**
  + , - (ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0)
  segno per valori numerici (positivo e negativo)

- **operatori aritmetici binari**
  +, - , *, / , ^ (ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0)
  addizione-concatenazione, sottrazione, moltiplicazione, divisione ed elevamento a potenza

  DIV (ERRE-PC 3.0), **MOD** (ERRE-C64 3.2 ERRE-PC 3.0)
  divisione intera, modulo

- **operatori logici (unari e binari) e di inclusione**
  AND, OR, NOT (ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0)
  XOR (ERRE-C64 3.2 ERRE-PC 3.0)
  operatori logici

  IN, NOT IN (ERRE-PC 3.0)
  operatori di inclusione

- **operatori relazionali**
  =, >, <, >=, <=, <> (ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0)
  uguale, maggiore di, minore di, maggiore o uguale di, minore o uguale di, diverso da

Gli operatori hanno, all'interno di espressioni, una certa gerarchia:

1) operatore **IN** e/o **NOT IN**
2) operatore **&**
3) chiamate di funzioni, sia di sistema che definite dall'utente
4) operatori aritmetici in questo ordine: **^** **+** (unario) **-** (unario) ***** **/** **DIV** **MOD** **+** **-**
5) operatori relazionali (**= > < >= <= <>**)
6) operatori logici in questo ordine: **NOT** **AND** **OR** **XOR**

Questa gerarchia è alterabile tramite l'utilizzo delle parentesi '(' e ')' : in ogni caso la valutazione di una espressione avviene a partire dalle parentesi più interne.

Nota: Gli operatori **MOD** e **XOR** sono disponibili su C-64 in forma funzionale come MOD(X,Y) e XOR(X,Y) mentre l'operatore di indirizzo memoria **&** è disponibile su VIC-20 e C-64 solo in forma di assegnamento (*IND=&VAR*).

richiamo di procedure

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Consente il richiamo di procedure indicandone solo il nome, seguito dalla lista degli eventuali parametri "attuali". Questi parametri saranno distinti tra *"parametri formali di ingresso"* - costanti, variabili, espressioni o funzioni - e *"parametri formali di uscita"* (solo variabili), separati tra di loro dal simbolo "->".

Sintassi:
nome_procedura[([lista_parametri_ingresso] [->lista_parametri_uscita])]

Esempio:

Dichiarata ad esempio la procedure

```
PROCEDURE AREA_RETT(BASE,ALTEZZA->AREA)
   AREA=BASE*ALTEZZA
END PROCEDURE
```

un possibile richiamo è il seguente:

```
AREA_RETT(5,3->A)
```

che memorizzerà in A il valore 15.

Note:

- In **ERRE-VIC20 1.3** non è ammesso il passaggio parametri, che andrà quindi curato manualmente.
- In **ERRE-PC 3.0** è ammesso, in caso di procedure ricorsive, il richiamo di una procedura con il nome preceduto da '*****' che conserverà il valore delle variabili locali tra un richiamo e l'altro
- Sia in **ERRE-C64 3.2** che in **ERRE-PC 3.0** è possibile passare array con un numero massimo di tre dimensioni.

simbolo speciale

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

I simboli riconosciuti dal linguaggio ERRE appartengono all'insieme dei codici ASCII compresi tra 32 e 95, con leggere differenze tra una piattaforma e l'altra: il codice 92 nel PC è '\' e '£' nel C64/VIC20 e il codice 95 '_' è nel PC e '←' nel C64/VIC20. Nelle varie versioni di ERRE alcuni di questi simboli, o combinazioni di essi, hanno un significato speciale.

=

Scopo:
Ha due scopi e precisamente:

1) come *operatore relazionale* (vedi operatori);
2) come *operatore di assegnamento* e consente di assegnare, genericamente, il valore di una espressione ad una variabile.

Esempio:

```
IF X=Y THEN A=B

X=10.5*(Z-1)
```

Note:

1) Può essere preceduto da uno qualsiasi degli operatori binari (**ERRE-C64 3.2 ERRE-PC 3.0**) per consentire di abbreviare la scrittura di un assegnamento di variabile. Ad esempio

```
I+=1      è equivalente a scrivere   I=I+1
I*=(J-1)  è equivalente a scrivere   I=I*(J-1)
I^=2      è equivalente a scrivere   I=I^2
```

2) Nella forma ':=' consente assegnamenti multipli di variabili:

```
I,J,K,T:=30
```

assegna il valore 30 alle variabili I,J,K e T.

->

Scopo:
Ha due scopi e precisamente:

1) separare la lista dei parametri di ingresso dalla lista dei parametri in uscita nella dichiarazione e nel richiamo di una **PROCEDURE** (vedi);
2) iniziare e terminare un "case_label" nell'istruzione **CASE** (vedi)

_

Scopo:
Migliora la leggibilità degli identificatori senza modificarne lo scopo.

Esempio:

MAXCOUNT%=100

può essere scritto come

MAX_COUNT%=100

sia in **ERRE-PC 3.0** che in **ERRE-C64 3.2**.

Nota: Pur avendo graficamente lo stesso simbolo in **ERRE-PC 3.0** il codice ASCII è 95, mentre in **ERRE-C64 3.2** il codice ASCII è 164 (tasti C= @)

←

In **ERRE-VIC20 1.3** e **ERRE-C64 2.3** questo simbolo inizia e termina un "case_label". Solo in **ERRE-C64 2.3** questo simbolo è usato per separare i parametri formali (attuali) di ingresso da quelli in uscita nella dichiarazione (richiamo) di una **PROCEDURE** (vedi):

ERRE-C64 3.2 ERRE-PC 3.0	**ERRE-C64 2.3**

PROCEDURE AREA_RETT(BASE,ALTEZZA->AREA)
  AREA=BASE*ALTEZZA
END PROCEDURE
.....
AREA_RETT(5,3->A)
	PROCEDURE AREA_RETT(AREA←BASE,ALTEZZA)
	  AREA=BASE*ALTEZZA
	END PROCEDURE
	.....
	AREA_RETT(A← 5,3)

,

Scopo:
In **ERRE-PC 3.0** viene usato per specificare il richiamo del metodo relativo ad una istanza.

Sintassi:
oggetto'metodo(parametri)

Esempio:
PILA'PUSH(->NUM)

Istruzione

Scopo:
Posiziona il cursore sullo schermo senza l'utilizzo della unit relativa (*CRT.LIB* per il PC o *CURSOR.LIB* per il C-64/VIC-20).

Sintassi:
@(riga,colonna)

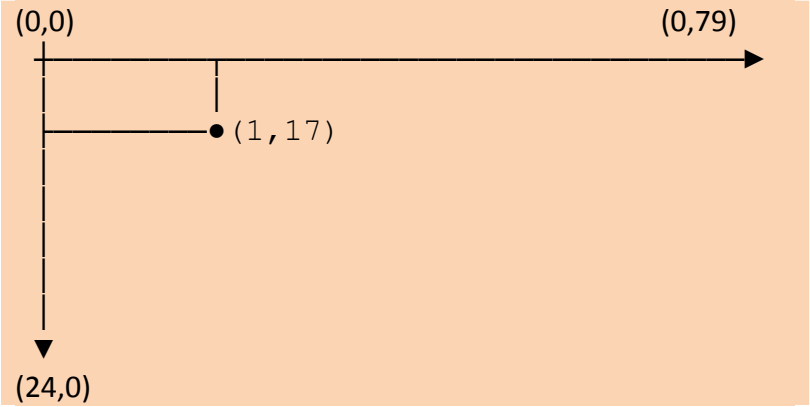
dove *riga* e *colonna* sono espressioni numeriche che rappresentano le coordinate-schermo e sono conteggiate a partire dall'estremo superiore sinistro (zero compreso). Per cui avremo:

piattaforma	*VIC-20*	*C-64*	*PC - 40 col.*	*PC - 80 col.*
riga	0 ≤ riga ≤ 22	0 ≤ riga ≤ 24	0 ≤ riga ≤ 24	0 ≤ riga ≤ 24
colonna	0 ≤ colonna ≤ 21	0 ≤ colonna ≤ 39	0 ≤ colonna ≤ 39	0 ≤ colonna ≤ 79

Esempio:

```
@(1,17)
```

posiziona il cursore in seconda riga e diciassettesima colonna.



Note:

- L'esempio va scritto come @1,17 in **ERRE-VIC20 1.3**.
- La unit **CURSOR.LIB** (**ERRE-C64 3.2**) implementa la procedure AT (equivalente della @)
- Con la unit **CRT.LIB** (**ERRE-PC 3.0**) si possono usare sia la procedure AT (equivalente della @) che la procedura LOCATE (stesso verso delle coordinate ma calcolate partendo da 1).
- In **ERRE-C64 2.3**, assieme alle istruzioni di controllo del cursore, è parte integrante del linguaggio come **AT**.

ABS(

Funzione

Scopo:
Calcola il valore assoluto di una espressione numerica.

Sintassi:
ABS(espr)

dove *espr* è una espressione numerica.

Esempio:

```
PRINT(ABS(-5.3))
```

stampa 5.3 come risultato dell'elaborazione

Nota:

- L'esempio va scritto come PRINT ABS(-5.3) in **ERRE-VIC20 1.3**.

ACCEPT(

ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Acquisisce un carattere, in maniera temporizzata, dallo standard input.

Sintassi:
ACCEPT(tempo,cod_uscita,var$)

Esempio:

```
.....
ACCEPT(3,X%,CH$)
.......
```

attende un input per 3 secondi; X% vale FALSE(0) se non si è premuto nessun tasto o TRUE(-1) se si è premuto un tasto il cui valore viene assegnato a CH$.

ACS(

Funzione

Scopo:
Calcola l'arco coseno (in radianti) di una espressione numerica.

Sintassi:
ACS(espr)

dove *espr* è una espressione numerica.

Esempio:

```
PRINT(ACS(0.5))
```

stampa 1.0471976 (60°) come risultato dell'elaborazione

Nota:

- ACS restituisce valori compresi tra 0 e π ("ramo principale")
- La funzione fornisce errore se l'argomento è, in valore assoluto, maggiore di 1.
- In **ERRE-VIC20 1.3** questa funzione può essere resa come:

```
FUNCTION ACS(X)=π/2-ATN(X/SQR(1-X*X))
```

ALL

Identificatore riservato

Scopo:
Permette di chiudere, come argomento di una istruzione CLOSE, tutti i file ancora aperti.

Sintassi:
CLOSE(ALL)

Vedi: **CLOSE**

AND

Operatore

Scopo:
Rappresenta il connettore logico "unione" all'interno di espressioni condizionali.

Sintassi:
AND

Esempio:

IF X=3 AND Y=4 THEN.....

questo test condizionale è vero se entrambe le variabili assumono i valori indicati.

Nota:

- AND può essere usato anche per operare sui singoli bit di una variabile e/o costante di tipo integer (cioè su 16 bit). Ad esempio

  63 AND 16 = 16 — 63=%00111111 e 16=%00010000, perciò applicando la tabella dell'operatore AND otteniamo %00010000 e quindi 16 in decimale

Operatore	Valore A	Valore B	Risultato
AND	V	V	V
	V	F	F
	F	V	F
	F	F	F

ASC(

Funzione

Scopo:
Restituisce un valore numerico che rappresenta il codice ASCII del primo carattere della espressione stringa usata come argomento.

Sintassi:
ASC(espr$)

dove *espr$* è una espressione stringa.

Esempio:

```
PRINT(ASC("A"))
```

stampa

```
65
```

che è il codice ASCII di "A".

Nota:

- In **ERRE-VIC20 1.3** e **ERRE-C64 3.2** se X$ è una stringa vuota viene restituito un errore: per evitare ciò basta chiedere il calcolo di ASC(come

  ```
  ASC(X$+CHR$(0))
  ```

  che fornisce 0 se X$ è nullo. Questo perché c'è una differenza sostanziale tra stringa vuota ("") di lunghezza zero e CHR$(0) che è una stringa composta di un carattere identificato nel codice ASCII come *NULL*.
- L'insieme dei codici ASCII utilizzati dal per PC ("codepage 437") e dal C-64/VIC20 ("PETSCII") è comune solo per i valori compresi tra 32 e 96: quindi attenzione all'uso di ASC per i rimanenti valori.

ASN(

Funzione

Scopo:
Calcola l'arco seno (in radianti) di una espressione numerica.

Sintassi:
ASN(espr)

dove *espr* è una espressione numerica.

Esempio:

```
PRINT(ASN(0.5))
```

stampa

0,5235988 (30°)

come risultato dell'elaborazione

Nota:

- ASN restituisce valori compresi tra -π/2 e +π/2 ("ramo principale")
- La funzione fornisce errore se l'argomento è, in valore assoluto, maggiore di 1.
- In **ERRE-VIC20 1.3** questa funzione può essere resa come:

```
FUNCTION ASN(X)=ATN(X/SQR(1-X*X))
```

ATN(

Funzione

Scopo:
Calcola l'arco tangente (in radianti) di una espressione numerica.

Sintassi:
ATN(espr)

dove *espr* è una espressione numerica.

Esempio:

```
PRINT(ATN(1))
```

stampa

```
0,7853982
```

che equivale a 45°.

Nota:

- **ATN** restituisce valori compresi tra -π/2 e +π/2 (il cosiddetto "ramo principale" della funzione arcotangente)

BEGIN

Istruzione

Scopo:
Segnala da dove iniziare l'esecuzione del modulo ERRE attualmente in memoria: è l'equivalente della *Sub Main* di Visual Basic o della *main* del linguaggio C. Insieme a **PROGRAM** e a **END PROGRAM** è uno dei tre elementi fissi di un modulo ERRE.

Sintassi:
BEGIN

Esempio:

```
PROGRAM HELLO          ◄---- intestazione del modulo
BEGIN                  ◄---- inizio esecuzione
  PRINT("HELLO WORLD")
END PROGRAM            ◄---- termine del modulo
```

CALL(

ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Fa partire un sottoprogramma scritto nel linguaggio macchina del microprocessore "target" a partire dalla locazione indicata dal valore di "var1" passando come parametri le variabili indicate nella "lista_var" (se esistono).

Sintassi:
CALL(var1[,lista_var])

dove var1 è una espressione numerica (vedi nota).

Esempio:

ERRE-PC 3.0

```
.....
SUBRT%=VARPTR(ARRAY%[0])
CALL(SUBRT%)
.....
```

Fa eseguire un routine in linguaggio macchina memorizzata nel vettore ARRAY[] a partire dall'indice 0: SUBRT% è l'indirizzo assoluto dove è memorizzato il primo elemento di ARRAY[].

ERRE-C64 3.2

```
.....
poke(214,row%)
poke(211,col%)
call(58732)  !$E56C
.....
```

Fa eseguire la routine del Kernal per il posizionamento del cursore sullo schermo.

Note:

- L'equivalente in **ERRE-VIC20 1.3** è **SYS** che va usato senza le parentesi rotonde.
- Per tutte le piattaforme le istruzioni **POKE** e **CALL**, la funzione **PEEK** nonché alcune direttive di compilazione possono indirizzare al massimo 64K, quindi il loro primo argomento (*var1* in questo caso) è di tipo "unsigned integer" (non espressamente previsto dallo standard del linguaggio) compreso tra 0 e 65535.
- Gli ambienti operativi di C64 ("64VMS") e VIC-20 mettono a disposizione parecchie ulteriori routine che possono essere utilizzate nei propri programmi tramite l'istruzione **CALL** (o **SYS**).

CASE...END CASE

Istruzione

Scopo:
Implementa un'istruzione di selezione che permette di effettuare una scelta tra più alternative, basate sul valore di "espressione".

Sintassi:

```
CASE espressione OF
   { case_label[,case_label]—>
        ┌──────────────┐
        │ B l o c c o  │
        └──────────────┘
     END —> }
   [ OTHERWISE
        ┌──────────────┐
        │ B l o c c o  │
        └──────────────┘]
END CASE
```

Il termine "case_label" si riferisce ad una espressione che sarà confrontata con "espressione" che segue CASE. Più "case_label" possono essere messe sulla stessa linea separate da una virgola se "Blocco" è uguale per tutte; la parola chiave **OTHERWISE** è facoltativa e serve per gestire esplicitamente i casi non previsti dai vari "case_label".

E' possibile inoltre specificare, come "case_label", anche degli insiemi sia numerici interi che di caratteri, usando la notazione ".." preceduta da **IN**.

Forma "standard"	Forma "compatta"
1,2,3,4,5->	1..5-> o IN 1..5->
"A","B","C","D","E"->	"A.."E"-> o IN "A.."E"

E' pure possibile usare operatori relazionali precedendoli con la parola chiave **IS** (esempio 2).

Esempi :

```
PROGRAM CASE_1
DIM NUM%
BEGIN
 INPUT(NUM%)
 CASE NUM% OF
  1-> PRINT("I";) END ->
  2-> PRINT("II";) END ->
  3-> PRINT("III";) END ->
  4-> PRINT("IV";) END ->
  5-> PRINT("V";) END ->
```

```
   6-> PRINT("VI";) END ->
   7-> PRINT("VII";) END ->
   8-> PRINT("VIII";) END ->
   9-> PRINT("IX";) END ->
   OTHERWISE
     PRINT("Lo zero non è lecito.")
 END CASE
END PROGRAM


PROGRAM CASE_2
DIM N%
BEGIN
  INPUT(N%)
  CASE N% OF
     IS >=10-> PRINT("MAGGIORE O UGUALE A 10") END ->
     IS <5-> PRINT("MINORE DI 5") END ->
   OTHERWISE
     PRINT("OUT OF RANGE")
  END CASE
END PROGRAM
```

Note:

- **ERRE-C64 3.2** non necessita delle parole chiave **IS** e **IN**.
- **ERRE-VIC20 1.3** usa ← come terminatore di un case_label (anzichè ->) e non ammette più case_label sulla stessa linea o range numerici od operatori relazionali.

CHAIN(

ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Permette di chiamare in esecuzione un altro modulo ERRE consentendo, se indicato, anche il passaggio di variabili (le cosiddette "variabili comuni") dichiarate tramite l'apposta parola chiave **COMMON**. E' la base della "gestione a moduli" di un programma ERRE.

Sintassi:
CHAIN(nome_file[,ALL])

Esempio:

ERRE-PC 3.0

```
PROGRAM COMMON1
COMMON A%,B,C$,D[]
DIM D[10]
BEGIN
  FOR I=1 TO 10 DO
    D[I]=I^2
  END FOR
  A%=255  B=1000000
  C$="OK" D$="NON PASSO"
  PRINT("Premi un tasto per proseguire")
  REPEAT
    GET(K$)
  UNTIL LEN(K$)<>0
  CHAIN("COMMON2")
END PROGRAM
```

```
PROGRAM COMMON2
COMMON A%,B,C$,D[]
!$DIMCOMMON
DIM D[10]
PROCEDURE NOTHING
END PROCEDURE
BEGIN
  NOTHING
  FOR I=1 TO 10
    PRINT(I,D[I])
  END FOR
  PRINT(A%,B,C$,D$)
  PRINT("Premi un tasto per finire ...")
  REPEAT
    GET(K$)
  UNTIL K$<>""
END PROGRAM
```

COMMON1 assegna dei valori a variabili di tipo vario, poi passa il controllo a COMMON2 che stampa questi valori. La variabile D$ di COMMON1 non viene passata, cosicché in COMMON2 D$ è la stringa vuota. Abitualmente **COMMON** accompagna **CHAIN** per la gestione delle variabili anche se ciò non è strettamente indispensabile: in questo caso le variabili potevano essere salvate su file per poi essere recuperate; l'appesantimento di questa gestione è però evidente.

Note:

- Con l'opzione **ALL** si passano tutte le variabili dal modulo chiamante a quello chiamato: in questo caso non si utilizzerà nessuna **COMMON**. Se si invece si vogliono passare solo delle variabili selezionate si dovrà utilizzare la clausola **COMMON**.
- **ERRE-C64 3.2** non ha ne l'opzione ALL ne l'istruzione COMMON: il passaggio di variabili tra moduli è diretto, ma va gestito con la direttiva di compilazione **!$VARSEG**.

CHANGE

Istruzione

Scopo:
Consente di 'scomporre' una variabile stringa "var1" nei caratteri che la costituiscono mettendo i rispettivi codici ASCII nell'array "var2". L'elemento di indice **0** dell'array contiene la lunghezza della stringa. E' possibile anche effettuare l'operazione inversa (da array a stringa).

Sintassi:
CHANGE var1 TO var2

Esempio:

```
PROGRAM CHANGE_TEST

DIM A$,C$
DIM L%[255]

BEGIN
  A$="ABCDEFGHIJKL"
  CHANGE A$ TO L%[]       ◄--- 'scompone' A$
  SWAP(L%[1],L%[3])       ◄--- scambia il I° ed il III° carattere della stringa
  CHANGE L%[] TO C$       ◄--- 'ricompone' C$
  PRINT(A$,C$)
END PROGRAM
```

stamperà

```
ABCDEFGHIJKL    CBADEFGHIJKL
```

Funzione

Scopo:
E' una funzione speciale, usata in fase di assegnamento di una variabile: se la "variabile di test" vale N alla "variabile di assegnamento" viene dato il valore della N-sima espressione di CHOOSE. Se N non rientra nei limiti, non viene effettuato nessun assegnamento ed il valore della "variabile di assegnamento" resta invariato.

Sintassi:
var_asg=CHOOSE(var_test,espressione1[,espressionej]) con $1 \leq j \leq N$

Esempio:

```
.......
   X=4
   Y=CHOOSE(X,2,4,6,8,10)
........
```

In questo caso Y vale 8 (il valore della 4ª espressione di **CHOOSE**).

Nota:

- **CHOOSE** è equivalente ad un struttura di controllo **CASE**. L'esempio precedente è dunque equivalente a:

```
CASE X OF
  1-> Y=2 END ->
  2-> Y=4 END ->
  3-> Y=6 END ->
  4-> Y=8 END ->
  5-> Y=10 END ->
  OTHERWISE
    !$NULL
END CASE
```

CHR$(

Funzione

Scopo:
Restituisce una stringa composta da un carattere il cui codice ASCII è dato dal valore dell'espressione numerica usata come argomento.

Sintassi:
CHR$(espr)

dove *espr* è una espressione numerica il cui valore deve essere compreso tra 0 e 255.

Esempio:

```
PRINT(CHR$(65))
```

stamperà

```
A
```

Nota:

- Se *espressione_numerica* è maggiore di 255 o inferiore a zero, verrà restituito un errore.
- L'insieme dei codici ASCII utilizzati dal per PC ("codepage 437") e dal C-64/VIC20 ("PETSCII") è comune solo per i valori compresi tra 32 e 96: quindi attenzione all'uso di CHR$ per i rimanenti valori.

Istruzione

Scopo:
Permette di definire un tipo astratto di dati ("classe") caratterizzato da un insieme di variabili locali e di funzioni e/o procedure ("metodi"). Queste ultime forniscono i parametri con i quali un programma esterno si interfaccia con la classe stessa, una volta definiti gli oggetti di riferimento ("istanze") tramite l'istruzione **NEW**.

Sintassi:

CLASS <nome_classe>
{dichiarazione di function e/o procedure utilizzate dalla classe con !$INTERFACE}
{dichiarazione di variabili e/o array locali alla classe con LOCAL}
[**IS CLASS** <nome_classe>]
{definizione e corpo di function e/o procedure per utilizzare la classe}
END CLASS

La clausola **IS CLASS** permette di fare "ereditare" le funzioni e i metodi di una classe precedentemente dichiarata.

Esempio:

```
PROGRAM CLASS_DEMO

 CLASS QUEUE                 <---- definizione di classe con ...

   !$INTERFACE ISEMPTY()     <---- dichiarazione dei metodi
   !$INTERFACE INIT                (è la parte che deve essere nota
   !$INTERFACE POP(->XX)           all'esterno per poter utilizzare
   !$INTERFACE PUSH(XX)            correttamente la classe).

   LOCAL SP                  <---- ... le variabili locali utilizzate dai "metodi"
   LOCAL DIM STACK[100]

   FUNCTION IS_EMPTY()       <---- qui iniziano i metodi: il programma principale
     IS_EMPTY=(SP=0)               li utilizzerà senza vedere la loro struttura
   END FUNCTION                    interna.

   PROCEDURE INIT            <---- inizializzazione della classe: gli oggetti
     SP=0                          dichiarati successivamente faranno riferimento
   END PROCEDURE                   a questo metodo per implementare il proprio
                                   "constructor"

   PROCEDURE POP(->XX)       <---- il metodo POP restituisce un valore all'esterno
     XX=STACK[SP]
```

```
    SP=SP-1
  END PROCEDURE

  PROCEDURE PUSH(XX)    <---- il metodo PUSH riceve un valore dall'esterno
    SP=SP+1
    STACK[SP]=XX
  END PROCEDURE
END CLASS

NEW PILA:QUEUE            <---- viene dichiarato l'oggetto PILA
                                riferito alla classe QUEUE

PROCEDURE STAMPA_INV      <---- inizio del programma vero e proprio

 READ(NUM)

 IF NUM<>-1 THEN  ! tappo
   PILA'PUSH(NUM)         <---- NUM viene passato per l'elaborazione
   STAMPA_INV                   all'oggetto PILA tramite il metodo PUSH
 END IF

 IF PILA'IS_EMPTY()<>0 THEN <---- idem come sopra solo con il metodo COUNT
   PILA'POP(->NUM)          <---- idem come sopra solo con il metodo POP
   PRINT(NUM)                     che restituisce NUM.
 END IF
END PROCEDURE

BEGIN
  DATA
     1,3,5,7,9,11,-1
  END DATA
  PILA'INIT                 <---- "constructor" dell'oggetto PILA
  STAMPA_INV
  PRINT("Fine")
END PROGRAM
```

Nota:

- Il richiamo dei "metodi", applicati alle relative "istanze", può essere del tipo:
  - **oggettometodo(parametri)**
  - **oggetto_metodo(parametri)**
  - **oggetto'metodo(parametri)**

  anche se, per motivi di chiarezza, l'ultima forma è quella preferibile.
- E' possibile riferirsi ad una classe solo tramite oggetti "semplici" e non "aggregati".

CLEAR

ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Mantenuta solo per motivi di compatibilità con le precedenti versioni del linguaggio:

- in **ERRE-C64 3.2** non ha nessun effetto.
- in **ERRE-PC 3.0** azzera tutte le variabili locali di una procedura.

Note:

- L'equivalente per **ERRE-VIC20 1.3** è **CLR** che cancella tutte le variabili del programma.

CLOSE(

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Consente di chiudere uno o tutti i file aperti.

Sintassi:
CLOSE(file_number|ALL)

Nota:

- In **ERRE-VIC20 1.3** si usa senza parentesi e non è ammessa l'opzione **ALL**.

Funzione

Scopo:
Funzione che si interfaccia con il DOS e ritorna la linea comandi usata per l'esecuzione del programma.

Sintassi:
CMDLINE$

Esempio:
Questo esempio verifica la congettura di Ulam per un numero N specifico: non viene richiesto l'input in fase di esecuzione se il numero è stato fornito all'atto del richiamo del programma stesso (che viene memorizzato da **CMDLINE$**).

```
PROGRAM ULAM2

DIM X,COUNT%

BEGIN
  IF CMDLINE$="" THEN
    PRINT("Imposta un numero intero";)
    INPUT(X)
   ELSE
    X=VAL(CMDLINE$)
    PRINT("N=";X)
  END IF
  REPEAT
    IF X/2<>INT(X/2) THEN
       X=X*3+1
     ELSE
       X=X/2
    END IF
    PRINT(X,)
    COUNT%+=1
  UNTIL X=1
  PRINT
  PRINT("Ok, il programma si è fermato dopo";COUNT%;"passaggi!")
END PROGRAM
```

COMMON

Istruzione

Scopo:
In una gestione a moduli di un programma ERRE, dichiara le variabili comuni da passare al modulo chiamato.

Sintassi:
COMMON lista_var

Esempio:

Vedi istruzione **CHAIN**

Nota:

- L'istruzione nel modulo chiamato deve avere la propria dichiarazione COMMON le cui variabili devono coincidere in tipo (ma non in nome) con quelle COMMON del modulo chiamante.
- La direttiva **!$DIMCOMMON** deve essere usata nel caso in cui si siano passati variabili aggregate (array) con lo stesso nome.

CONST

Istruzione

Scopo:
Permette di dichiarare costanti, sia numeriche che alfanumeriche.

Sintassi:
CONST $c_1=v_1,c_2=v_2,.....,c_j=v_j$

Esempio:

```
CONST ALFA=100,PLUTO$="XXXXX",INIZ%=5.1
```

definisce la costante "real" ALFA come 100, la costante stringa PIPPO$ come "XXXXX" e la costante "integer" INIZ% come 5 (le costanti di tipo "integer" vengono comunque arrotondate).

```
CONST ALFA=100,DUE_ALFA=(ALFA*2),ESCKEY$=CHR$(27)
```

"Dichiarazione estesa di costante". Si possono usare la funzione predefinita **CHR$(** per le costanti di tipo stringa, per quelle numeriche la definizione va messa tra parentesi rotonde e deve comprendere solo operatori aritmetici e/o costanti numeriche. Se si utilizza una costante nella dichiarazione di un'altra, la prima dev'essere già stata dichiarata (vedi ALFA e DUE_ALFA dell'esempio 2).

CONTINUE

ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Usata all'interno di un ciclo **FOR**, **WHILE**, **REPEAT** o **LOOP** trasferisce il controllo all'iterazione successiva del ciclo. Nel caso siano presenti più cicli annidati dello stesso tipo, **CONTINUE** trasferisce il controllo all'iterazione successiva del ciclo più interno.

Sintassi:
CONTINUE FOR|WHILE|REPEAT|LOOP

Esempio:

```
FOR I=1 TO 9 DO
   PRINT(I,)
   IF I>4 AND I<9 THEN CONTINUE FOR END IF
END FOR
```

stamperà

```
1        2        3        4        9
```

dopo la quarta iterazione (e fino alla ottava) **CONTINUE** trasferisce sempre il controllo all'iterazione successiva fino a fine ciclo.

COS(

Funzione

Scopo:
Calcola il coseno di una espressione numerica (in radianti).

Sintassi:
COS(espr)

dove *espr* è una espressione numerica.

Esempio:

```
PRINT(COS(π/2))
```

stampa

```
0
```

Nota:

- Per calcolare il coseno di un angolo in gradi sessagesimali, moltiplicare l'argomento per "180/π"

DATA(

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Permette di avere dei dati "in line" all'interno di un programma, anziché su supporto esterno. Questi dati possono essere letti ed assegnati ad una variabile dall'istruzione **READ.**

Sintassi:
DATA(lista_costanti)

oppure

DATA
lista_costanti
END DATA

Esempio:

ERRE-PC 3.0

```
.....
DATA
     1,2,3,4
END DATA
.....
READ(X,Y,Z,W)
PRINT(X,Y,Z,W)
.....
```

ERRE-C64 3.2

```
.....
DATA(1,2,3,4)
.....
READ(X,Y,Z,W)
PRINT(X,Y,Z,W)
.....
```

stamperà in entrambi i casi
1 2 3 4

Nota:

- In **ERRE-VIC20 1.3** l'istruzione DATA va senza parentesi e può essere posizionate ovunque.
- Il blocco **DATA..END DATA** (o le istruzioni **DATA**) vanno messe immediatamente dopo una dichiarazione di **PROCEDURE** o **BEGIN** (**ERRE-C64 3.2** e **ERRE-PC 3.0**) – la cosiddetta "*data section*".
- E' possibile mescolare dati numerici a dati stringa.
- Nel caso si debbano assegnare dati ad un array (vettore o matrice) risulta più conveniente effettuare un assegnamento diretto (solo **ERRE-C64 3.2** e **ERRE-PC 3.0**):

```
DIM A[5]
......
DATA                \
   1,2,3,4,5         | DATA(1,2,3,4,5) per ERRE-C64 3.2
END DATA            /
```

```
FOR I=1 TO 5
   READ(A[I])
END FOR
......
```

equivale a

```
DIM A[5]
......
A[]=(0,1,2,3,4,5)
…..
```

DATE$

ERRE-C64 3.2 ERRE-PC 3.0

Funzione

Scopo:
Legge o assegna la data di sistema.

Sintassi:
DATE$=

=DATE$

Il formato usato è quello MM-GG-AAAA con separatore '-'. Altri formati e/o separatori diversi possono essere ottenuti manipolando **DATE$** con le usuali funzioni di stringa. Ad es., per avere il formato GG-MM-AAAA si può fare

```
DATE$=MID$(DATE$,4,3)+LEFT$(DATE$,2)+RIGHT$(DATE$,5)
```

o se si vuole avere un formato GG/MM/AAAA

```
DATE$=MID$(DATE$,4,2)+"/"+LEFT$(DATE$,2)+"/"+RIGHT$(DATE$,4)
```

Esempio:

```
PRINT(DATE$)

02-11-2019
```

Nota:

- Assieme alla **MID$(** ed alla **TIME$** è l'unica funzione che può essere usata a sinistra del segno di assegnamento.
- **DATE$** non è implementata nelle versioni **ERRE-VIC20 1.3** e **ERRE-C64 3.2:** in quest'ultimo caso può essere ricavata dalla prima riga del file *SYSTEM.DAT* di *64VMS* (che è nel formato GGMMAAAA).

DIM

Istruzione

Scopo:
Permette la dichiarazioni di variabili, sia semplici che aggregate (array e record).

Sintassi:
DIM a_1,a_2,.....,a_j

dove "a_1","a_2",..."a_j" sono delle variabili semplici, ognuna col il proprio tipo.

DIM $a_1[d_1,...,dN_1],a_2[d_1,...,dN_2],.....,a_j[d_1,...,dN_j]$

dove "a_1","a_2",..."a_j" sono delle variabili di tipo ARRAY, "N_1", "N_2",..,"N_j" rappresentano il numero delle dimensioni di ogni array dichiarato; "d_1","d_2", ...,"d_n" sono gli estremi superiori degli indici degli array in quanto l'estremo inferiore parte obbligatoriamente da zero: perciò gli indici indirizzabili vanno da 0 a d_j.

DIM var_record$_1$:tipo_record$_1$[,var_record$_N$:tipo_record$_N$]

dove "var_record$_j$" è una variabile record di tipo "tipo_record$_j$"

Esempio:

```
DIM X,Z%,W#,Y$
```

dichiara le variabili semplici X (di tipo real), Z% (integer), W#(doppia precisione) e Y$ (stringa)

```
DIM A$[10],B%[5],F2[10,10]
```

dimensiona il vettore di stringhe A (con 11 elementi con indice da 0 a 10),il vettore di interi B (con 6 elementi) e la matrice di reali F2 (con 121 elementi, da F2(0,0) a F2(10,10)).

Se

```
TYPE CUSTOMER IS (NOME$,INDIRIZZO$,NUMTEL$)
```

definisce il tipo "CUSTOMER" composto da tre campi stringa ("NOME$","INDIRIZZO$" e "NUMTEL$"). Tramite una DIM successiva ci si potrà riferire a "CUSTOMER"

```
DIM WORK:CUSTOMER,ELENCO[10]:CUSTOMER
```

creando così il record semplice WORK e l'array ELENCO composto di undici record.

Note:

- La dichiarazione delle variabili semplici, seppure facoltativa, è assolutamente consigliata per ottenere la migliore allocazione possibile dello spazio di memoria del programma.
- **ERRE-VIC20 1.3** e **ERRE-C64 3.2** non hanno le variabili di tipo record e la relativa istruzione **TYPE**.

DIV

Operatore

Scopo:
Operatore di divisione intera.

Sintassi:
DIV

Esempio:

```
PRINT 13 DIV 4
```

stampa

```
3
```

Nota:

- Prima di effettuare l'operazione gli operandi vengono arrotondati all'intero superiore.
- Gli operandi devono essere in valore assoluto minori di **MAXINT** (32767), altrimenti viene segnalato un errore.
- In **ERRE-VIC20 1.3** e **ERRE-C64 3.2** DIV viene resa con la funzione INT e l'operatore di divisione. L'esempio precedente può essere reso come `PRINT INT(13/4).`

DO

Identificatore riservato

Scopo:
Termina una riga **FOR** e/o **WHILE**.

Sintassi:
Vedi **FOR** e **WHILE**

Esempio:
Vedi **FOR** e **WHILE**

Nota:

- In **ERRE-VIC20 1.3** **DO** è sostituita dal <RETURN> di fine riga.

DS e DS$

Variabile riservata

Scopo:
Segnala un errore di disco in caso di controllo con l'istruzione **EXEC(">")**.

Sintassi:
DS e **DS$**

Esempio:

```
EXEC(">")
PRINT(DS,DS$)
```

se non c'è nessun errore viene stampato il messaggio

```
 00,OK, 00, 00
```

Nota: Queste variabili sono attivate solo in caso venga eseguita l'istruzione **EXEC(">")**. In caso contrario **DS** e **DS$** possono essere liberamente utilizzate. Per questo motivo il compilatore non esegue nessun controllo, contrariamente alle altre variabili riservate **TIME** e **STATUS**.

ELSE

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Ramo “false” finale di una istruzione IF.

Sintassi:
ELSE

Esempio:
Vedi **IF**

ELSIF

Istruzione

Scopo:
Ramo “false” intermedio di una istruzione IF.

Sintassi:
ELSIF condizione

Esempio:
Vedi **IF**

EMPTY

Costante predefinita

Scopo:
Azzera una variabile di tipo **VARIANT**. Può assumere il valore 0 se la variabile è intesa successivamente come numerica, in caso contrario vale stringa nulla ("").

Sintassi:
EMPTY

Nota:

- **EMPTY** può essere utilizzata solo entro una procedure di tipo VARIANT.

END ->

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Termina una “case_label”

Sintassi:
END ->

Esempio:
Vedi **CASE**.

Equivalente: (ERRE-VIC20 1.3) **END←**

END CASE

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Termina un blocco **CASE**.

Sintassi:
END CASE

Esempio:
Vedi **CASE**

Abbreviazione: (ERRE-VIC20 1.3) **ENDC**

Equivalente: (ERRE-VIC20 1.3) **ENDCASE**

END CLASS

Istruzione

Scopo:
Termina una dichiarazione **CLASS**.

Sintassi:
END CLASS

Esempio:
Vedi **CLASS**

END EXCEPTION

ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Termina il blocco di istruzioni che gestisce le eccezioni in un modulo ERRE.

Sintassi:
END EXCEPTION

Esempio:
Vedi **EXCEPTION**

END FOR

ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Termina un ciclo FOR.

Sintassi:
END FOR

Esempio:
Vedi **FOR**

Equivalente: **NEXT** (**ERRE-VIC20 1.3**)

END FOREACH

ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Termina un ciclo **FOREACH**.

Sintassi:
END FOREACH

Esempio:
Vedi **FOREACH**

END FUNCTION

ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Termina il corpo di una function.

Sintassi:
END FUNCTION

Esempio:
Vedi **FUNCTION**

Nota:

- In **ERRE-VIC20 1.3** questa istruzione non serve in quanto dichiarazione e corpo di funzione devono essere scritte su una sola riga.
- Per lo stesso scopo in **ERRE-C64 2.3** si usa **ENDFUNCTION** (abbreviabile in **ENDF**).

END IF

ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Termina un blocco **IF**.

Sintassi:
END IF

Esempio:
Vedi **IF**

Nota:
Solo per **ERRE-PC 3.0** **END IF** può essere omesso nel caso si utilizzi una istruzione **IF** a riga singola. Ad esempio

```
IF X=1 THEN
        Y=X*X
    ELSE
        Y=X+X
END IF
```

può essere scritto come

```
IF X=1 THEN  Y=X*X ELSE Y=X+X
```

Chiaramente da evitare se si vuole conservare la portabilità di un programma.

Equivalente: **ENDIF** (**ERRE-VIC20 1.3**)

END LOOP

Istruzione

Scopo:
Termina un ciclo **LOOP**.

Sintassi:
END LOOP

Esempio:
Vedi **LOOP**

END PROCEDURE

ERRE-C64 3.2 **ERRE-PC 3.0**

Istruzione

Scopo:
Termina il corpo di una procedure.

Sintassi:
END PROCEDURE

Esempio:
Vedi **PROCEDURE**

Equivalente: **ENDPROCEDURE** (**ERRE-VIC20 1.3**)

END PROGRAM

ERRE-C64 3.2 **ERRE-PC 3.0**

Istruzione

Scopo:
Chiude un modulo ERRE.

Sintassi:
END PROGRAM

Esempio:
Vedi **PROGRAM**

Equivalente: **END.** (**ERRE-VIC20 1.3**)

Nota: **ERRE-C64 3.2** ed **ERRE-PC 3.0** riconoscono **END.** come sostituto di **END PROGRAM**.

END WHILE

ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Chiude un ciclo **WHILE...DO**

Sintassi:
END WHILE

Esempio:
Vedi **WHILE**

Equivalente: **ENDWHILE** (**ERRE-VIC20 1.3**)

END WITH

Istruzione

Scopo:
Chiude un blocco di istruzioni nel quale è possibile indicare le variabili di tipo record in maniera abbreviata.

Sintassi:
END WITH

Esempio:
Vedi **WITH**

END.

Istruzione

Scopo:
Chiude un modulo ERRE (vedi **END PROGRAM)**

Nota:
Viene utilizzato anche da **ERRE-C64 2.3** con lo stesso scopo.

ENDCASE

Istruzione

Scopo:
Termina una istruzione **CASE...OF**.

Abbreviazione: **ENDC**

Vedi **END CASE**

ENDIF

Istruzione

Scopo:
Termina una istruzione **IF**.

Vedi **END IF**

ENDPROCEDURE

Istruzione

Scopo:
Termina una PROCEDURE.

Abbreviazione: **ENDP**

Vedi **END PROCEDURE**

ENDWHILE

Istruzione

Scopo:
Termina un ciclo WHILE.

Abbreviazione: **ENDW**

Vedi **END WHILE**

Istruzione

Scopo:
Termina una “case_label”

Vedi **END ->**

EOF(

Funzione

Scopo:
Segnala la fine di un file precedentemente aperto.

Sintassi:
EOF(num_file)

Esempio:

```
REPEAT
.........
UNTIL EOF(1)
```

legge dal file aperto come #1 fino al termine dello stesso.

Nota:

- La stessa funzione viene svolta da **STATUS** su **ERRE-VIC20 1.3** e **ERRE-C64 3.2**.

ERR

Variabile riservata

Scopo:
Variabile riservata contenente un valore numerico, compreso tra 1 e 255, che rappresenta il tipo di errore incontrato durante l'esecuzione di un programma. **ERR** viene utilizzata in una procedura **EXCEPTION** per gestire correttamente gli errori di esecuzione.

Sintassi:
ERR

Esempio:

```
.....
IF ERR=53 THEN
  PRINT("FILE NON TROVATO!")
END IF
.....
```

Equivalente (**ERRE-VIC20 1.3** **ERRE-C64 3.2**): **STATUS**

Nota:

- I codici di errore per PC e C64 sono diversi (vedi **EXCEPTION**), inoltre la variabile riservata **STATUS** è valida anche per gli errori di I/O.
- Per migliorare la leggibilità del codice ed aumentare la portabilità è possibile usare anziché i codici numerici di errore delle costanti, dette "**named exceptions**", che hanno la caratteristica di iniziare con il carattere '?'. Tali costanti hanno significato solo all'interno di un blocco **EXCEPTION**.
- Sebbene non prevista dallo standard, è liberamente utilizzabile anche la variabile riservata di R-Code **ERL**, che fornisce il numero di istruzione dove si è verificato l'errore (**ERRE-PC 3.0** e **ERRE-PC 2.6**). L'equivalente per **ERRE-C64 3.2** e **ERRE-C64 2.3** è la variabile **LN%**, gestita direttamente dalla routine di "trapping" degli errori di 64VMS.

Istruzione

Scopo:
Attiva, disattiva o blocca temporaneamente un evento.

Sintassi:
EVENT("messaggio")

"messaggio" è una stringa di comando che deve essere risolta a livello di interprete di R-Code: è quindi dipendente dall'implementazione; mentre <azione> è il nome di una procedura che specifica quale deve essere il trattamento dell'evento una volta che questo sia stato "intercettato" ("trapping" in inglese): l'istruzione **EVENT** determina lo stato dell'evento (attivato, disattivato o fermo) mentre **IF EVENT** determina l'azione da svolgere.
Nel caso del PC IBM tali "messaggi" possono comprendere i seguenti casi:

TIMER(n)	legato all'orologio di sistema (n: in secondi)
KEY(n)	legato alla tastiera (n: tipo tasto)
COM(n)	legato alla porta seriale (n: porta 1 o 2)
PEN	legato alla penna ottica
PLAY(n)	legato all'esecuzione di musica di sottofondo (n: num. note di sottofondo)
STRIG(n)	legato ai joystick (n: tasto premuto) *Nota:* può essere usato anche con il mouse se questo lavora in modalità di emulazione joystick.
UEVENT	evento "utente" personalizzabile. Una volta definito lo stato dell'evento e l'azione da svolgere, questi va verificato in modo diretto tramite la direttiva di compilazione "**$SETUEVENT**", contrariamente agli altri tipi di eventi che sono legati direttamente all'hardware della piattaforma.

Esempio:

```
PROGRAM EVENT

USES CRT

PROCEDURE SHOW_TIME
  OLDROW=CSRLIN                ! salva la posizione del cursore
  OLDCOL=POS(0)                ! (riga e colonna)
  LOCATE(1,1) PRINT(TIME$)
  LOCATE(OLDROW,OLDCOL)   ! ripristina cursore all'uscita
END PROCEDURE

BEGIN
  CLS
  IF EVENT("TIMER(2)") THEN SHOW_TIME    ◄--- ogni due secondi richiama
```

```
                                        la procedura SHOW_TIME
  EVENT("TIMER ON")          <--- abilita l'evento TIMER
                                  con l'apposito "messaggio"
  FOR I=1 TO 10000 DO  \
    LOCATE(10,1)       | ognuna di queste istruzioni può "subire"
    PRINT(I,I*I)       | l'evento che provoca il salto alla SHOW_TIME
  END FOR              /

  EVENT("TIMER OFF")         <--- disabilita l'evento TIMER
END PROGRAM                       con l'apposito "messaggio"
```

Nota:

- Sul C-64 un esempio come il precedente può essere ottenuto solo a livello di programmazione in linguaggio macchina (vedi comando CLOCK di MCR).

EXCEPTION

ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Routine di gestione errori di un modulo ERRE: tale routine va posta prima dell'inizio del main program.

Sintassi:

```
EXCEPTION
    ┌──────────────┐
    │  B l o c c o │
    └──────────────┘
END EXCEPTION
```

Esempio:

```
PROGRAM LETTURA

 DIM FERROR%,FILE$,CH$

 EXCEPTION
   FERROR%=TRUE                ! si è verificata l'eccezione !!!
   PRINT("Il file richiesto non esiste .....")
 END EXCEPTION

 BEGIN
   FERROR%=FALSE
   PRINT("Nome del file";)
   INPUT(FILE$)                ! chiede il nome del file
   OPEN(#1,FILE$,"SEQ,INPUT","READ") ! apre un file sequenziale in lettura
     IF NOT FERROR% THEN
       REPEAT
         GET(#1,CH$)           ! legge un carattere dal file #1....
        IF CH$<>CHR$(10) THEN
           PRINT(CH$;)         ! lo stampa sullo standard output...
        END IF
       UNTIL EOF(1)            ! fine a fine file
     END IF
     PRINT
  CLOSE(#1)                    ! chiude il file
END PROGRAM
```

Se il nome del file dato in input non esiste (e quindi c'è un errore) si salta direttamente al blocco **EXCEPTION** che gestisce l'errore stampando l'apposito messaggio di avviso e ritorna, mancando un'istruzione **RESUME**, all'istruzione che segue quella che ha provocato l'errore. La variabile FERROR% attivata segnala al main di chiudere subito il file e terminare l'elaborazione.

Note:

- Se **EXCEPTION** è assente o è stata disabilitata, in caso di errore quando si blocca l'elaborazione viene stampato il messaggio:

  [Runtime error xxx]

  dove xxx è il numero di errore il cui significato è ricavabile dalle tabelle sotto la voce "**codici di errore**". Si noti che i codici 01-18 della versione PC corrispondono ai codici 10-27 della versione C-64.
- Il codice numerico di errore vene automaticamente posto nella variabile riservata **ERR** (**ERRE-PC 3.0**) o **STATUS** (**ERRE-C64 3.2**) al verificarsi dell'eccezione stessa: perciò ERR/STATUS può essere consultata per scoprire l'origine dell'errore e per porvi rimedio.
- E' possibile disabilitare una dichiarazione di EXCEPTION tramite la direttiva di compilazione **!$NOEXCEPTION** (solo dal blocco EXCEPTION e dal main).
- In **ERRE-VIC20 1.3** **EXCEPTION** è vista come una **PROCEDURE** e gestisce gli stessi codici di errore della versione per C-64.

```
program trapping
procedure exception
   resume *riprendi*
endp
begin
   print chr$(147);
   repeat
*riprendi*
     input n
     print n"is ok"
     if n=0
       then
          goto *fine*
     endif
   until false
*fine*
end.
```

Istruzione

Scopo:
Scambia tra di loro i contenuti di due variabili di tipo record (che fanno riferimento allo stesso TYPE).

Sintassi:
EXCHANGE(var_rec1,var_rec2)

Esempio:

```
.....
TYPE CUSTOMERTYPE=(NAME$,ADDRESS$,PHONENUM$)
DIM CUSTOMER[25]:CUSTOMERTYPE,WORKSPACE:CUSTOMERTYPE
.....
EXCHANGE(CUSTOMER[I%],CUSTOMER[J%])
......
```

scambio il I%-simo record di CUSTOMER con il J%-simo, senza doverlo fare campo a campo.

EXEC(

ERRE-C64 3.2 **ERRE-PC 3.0**

Istruzione

Scopo:
Richiama il Sistema Operativo "target" per eseguire il comando identificato da "espr$".

Sintassi:
EXEC(espr$)

- **ERRE-PC 3.0**

espr$ segue la sintassi di MS-DOS (o del Prompt dei Comandi di Windows) ed, oltre agli abituali comandi del S.O. - DIR, REN, COPY ecc..., può far eseguire un qualsiasi programma applicativo

- **ERRE-C64 3.2**

espr$ segue la sintassi del programma "*C-64 DOS Wedge*" e può assumere i seguenti valori:

>	**>I0:**
status disco (nel drive corrente). Restituisce le variabili riservate **DS** e **DS$**	inizializza disco (nel drive corrente)
>$[0:<*pattern*>]	**>S0:filename**
Directory disco (*pattern* facoltativo	cancella *filename* ('?' e '*' ammessi)
>#[<*num_device*>]	**>C0:*newfile*=*oldfile*[1+*oldfile2*+..+*oldfile4*]**
scegli device (8 o 9).Se manca *device* è scelto il disco utente.	copia *oldfile* in *newfile* o concatena *newfile* unendo i vari *oldfile*.
>N0:*name*,*id*	**>R0:*newfile*=*oldfile***
formatta disco (nel drive corrente). Se manca *id* formattazione rapida.	Rinomina *oldfile* in *newfile*
>V0:	
convalida disco (nel drive corrente)	

Esempio (**ERRE-PC 3.0**):

```
.....
EXEC("DIR /W > ELENCO.LST")
.....
```

richiama il comando DIR di MS-DOS e salva il risultato sul file ELENCO.LST (nella directory corrente), tornando poi all'istruzione seguente del programma.

Esempio (**ERRE-C64 3.2**)

```
.....
EXEC("S0:Z/*")
.....
```

cancella i file che iniziano con 'Z/' dal disco utente (comando del DOS Commodore).

Note:

- Non viene effettuato nessun controllo sulla sintassi di "espressione_stringa".
- In caso di operazioni su file resta valido il significato dei metacaratteri '?' e '*'
- Su **ERRE-PC 3.0** l'esecuzione di questo comando genera un cosiddetto "processo figlio".
- Su **ERRE-C64 3.2** **EXEC** restituisce le variabili **DS** e **DS$** che identificano rispettivamente il numero dell'errore e la sua spiegazione in chiaro degli errori su disco. Per gli errori generici di esecuzione far riferimento a **EXCEPTION..END EXECEPTION**, **RESUME** e **STATUS**.
- Fino a **ERRE-PC 2.6** **EXEC** era scritta come **SHELL**.

EXIT

Istruzione

Scopo:
Consente l'uscita incondizionata da un ciclo FOR, REPEAT, WHILE o LOOP: nel caso di cicli nidificati l'uscita avviene dal ciclo più interno.

Sintassi:
EXIT

Esempio:

```
FOR I=1 TO 3 DO
    FOR J=1 TO 5 DO
       PRINT(I,J)
       EXIT                  ◄--- esce subito dal ciclo interno (J)
   END FOR !J
END FOR !I
```

stampa

```
1       1
2       1
3       1
```

EXIT IF

ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Consente l'uscita condizionata da un ciclo **FOR**, **REPEAT**, **WHILE** o **LOOP**: nel caso di cicli nidificati l'uscita avviene dal ciclo più interno.

Sintassi:
EXIT IF condizione

Esempio:

```
I=0
LOOP
   I=I+1
   PRINT(I;)
   EXIT IF I>5
END LOOP
```

LOOP...**ENDLOOP** è normalmente un *ciclo infinito*: **EXIT IF** ne consente l'uscita trasformandolo così in un *ciclo a condizione intermedia*.

EXIT PROCEDURE

ERRE-C64 3.2 **ERRE-PC 3.0**

Istruzione

Scopo:
Consente l'uscita anticipata da una procedure.

Sintassi:
EXIT PROCEDURE

Nota:

- L'equivalente **ERRE-VIC20 1.3** è **EXITPROC.**

EXITPROC

Istruzione

Scopo:
E' l'equivalente della **EXIT PROCEDURE** per le altre versioni ERRE (vedi).

EXIT ->

ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Consente l'uscita incondizionata da un "case-label".

Sintassi:
EXIT ->

EXP(

Funzione

Scopo:
Calcola la funzione esponenziale.

Sintassi:
EXP(X)

dove X è una variabile numerica.

Esempio:

```
PRINT EXP(1)
```

stampa

```
 2,71828183
```

Nota:

- X deve essere minore di 88.02969, altrimenti abbiamo un errore. In **ERRE-VIC20 1.3** **ERRE-C64 3.2** l'esecuzione si blocca, mentre in **ERRE-PC 3.0** alla funzione viene attribuito l'infinito di macchina (MAXREAL o MAXREAL#) e l'esecuzione prosegue.

EXTERNAL

ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Dichiara una procedura come esterna, cioè scritta direttamente in R-Code.

Sintassi:
PROCEDURE nome_proc[([param.ingresso][->param.uscita])] EXTERNAL,costante_stringa

Esempio:

```
PROGRAM EXTERNAL

DIM N%,I%,TEMPO

PROCEDURE ADDONE(I%->I%) EXTERNAL,"ADDONE.EXT"

PROCEDURE DELAY(TEMPO) EXTERNAL,"DELAY.EXT"

BEGIN
   PRINT(CHR$(12);)
   INPUT("NUMERO",N%)
   ADDONE(N%->N%)
   PRINT(N%)
   DELAY(10)
END PROGRAM
```

I corpi delle due procedure EXTERNAL sono contenuti rispettivamente nei file "ADDONE.EXT" e "DELAY.EXT"

```
---------- ADDONE.EXT
5000 I%=I%+1:RETURN

---------- DELAY.EXT
6000 T1!=TIMER:WHILE TIMER-T1!<TEMPO:WEND
6010 RETURN
```

Note:

- Il codice contenuto nei file esterni non viene ovviamente controllato dal compilatore ERRE e quindi va preventivamente fatto eseguire dall'interprete di R-Code per verificarne la funzionalità.

FACT(

Funzione

Scopo:
Calcola il fattoriale dell'argomento.

Sintassi:
FACT(espr)

Esempio:

```
PRINT FACT(8)
 40320
```

Nota:

- Se l'argomento X non è intero, FACT viene calcolata come se fosse FACT(INT(X)).
- Il valore dell'argomento deve essere compreso tra 0 e 33: in caso contrario viene data segnalazione di errore.

FALSE

Costante predefinita

Scopo:
Costante booleana cui viene attribuito convenzionalmente il valore **0**.

Sintassi:
FALSE

Esempio:

```
X=(Y>2)
PRINT(X)
```

se Y vale -100, verrà stampato 0 (**FALSE**), perché l'espressione (Y>2) è falsa.

Istruzione

Scopo:
Definisce la struttura di un file ad accesso casuale (“random”) in modo esplicito o tramite una istruzione di record TYPE.

Sintassi:
FIELD(#num_file,lista_campi)

num_file fa riferimento ad un file aperto precedentemente con l’istruzione OPEN.
lista_campi è costituita da n coppie *lung_campo,nome_campo* se definita esplicitamente o n *lung_campo* se derivata da una struttura TYPE, con cui deve corrispondere – vedi esempio.

Esempio:

```
.......
OPEN(#1,"CLIENTI.DAT","R","READ WRITE",LEN=82)
FIELD(#1,27,CLRSOC$,27,CLINDIRI$,23,CLCITTA$,5,CLCAP$)
.......
```

viene aperto il file CLIENTI.DAT con lunghezza record 82 e composto da 4 campi ognuno con la propria lunghezza. (27+27+23+5=82).

Se

```
TYPE DIPENDENTE IS (NOME$,MATR$,ORARIA%,SALARIO#)
```

si può dichiarare un file ad accesso diretto basato su questa struttura record come

```
.......
OPEN(#1,"PAYROLL.DAT","R","READ WRITE",^DIPENDENTE)
                                          ▲
                                          \__  '^' indica puntatore al record
                                               DIPENDENTE

FIELD(#1,27,11,2,8)                       ◄--- In questo caso nella FIELD ba-
                                               sta indicare solo le lunghezze
                                               dei campi: i campi numerici
                                               hanno lunghezza fissa (2 per
                                               INTEGER, 4 per REAL e 8 per
                                               LONG REAL) mentre la lunghezza
                                               dei campi STRING (da 1 a 255)è
                                               in funzione dell'applicazione.
.......
```

Nota:

- La somma delle lunghezze dei campi dell'istruzione "FIELD" non deve superare la lunghezza record dichiarata nella "OPEN".
- Se la lista dell'istruzione FIELD è troppo lunga, o anche solo per motivi di chiarezza, è possibile spezzarla su più istruzioni FIELD consecutive.
- Per motivi dipendenti dall'applicazione (ad esempio per un'operazione di ordinamento) la struttura di un file ad accesso diretto può anche essere "ridefinita" all'interno di uno stesso modulo con delle istruzioni FIELD diverse: per fare ciò è necessario chiudere il file definito precedentemente e riaprirlo con le nuove definizioni.
- **ERRE-C64 3.2** non ha una istruzione FIELD ed la struttura record va gestita esplicitamente con variabile stringa.

FILEATTR$

Funzione

Scopo:
Permette di conoscere il tipo di un file, una volta assegnato il numero_file di apertura.

Sintassi:
FILEATTR$(X%)

se X% è il numero_file di apertura, i valori di FILEATTR$ sono i seguenti:

- "I","O","A" sequenziali in input/output/append
- "R" ad accesso casuale (su disco)
- "C" ad accesso casuale (porta COMx:)
- "0" file non aperto

Esempio:

```
OPEN(#1,"ESEMPIO.TXT","SEQ,INPUT","READ")
......
PRINT(FILEATTR$(1))
```

verrà stampato

```
I
```

cioè il file #1 è stato aperto come sequenziale in lettura.

FOR

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Implementa un ciclo a conteggio in cui la variabile di controllo assume, consecutivamente, tutti i valori compresi tra *espressione1* ed *espressione2* con passo *espressione3* (se utilizzato, altrimenti vale 1).

Sintassi:

```
FOR var=espressione1 TO espressione2 [STEP espressione3] DO
    ┌──────────┐
    │ B l o c c o │
    └──────────┘
END FOR
```

Esempio:

Con il ciclo

```
FOR NUMERO%=2 TO 100 STEP 2   DO     ! il passo del ciclo è 2
   PRINT(NUMERO%,NUMERO%^2)
END FOR
```

si otterrà

```
 2        4
 4        16
.....................
 100      10000
```

cioè la tabella dei quadrati dei numeri pari fino a 100.

Invece con

```
FOR NUMERO%=10 TO 1 STEP -1   DO     ! il passo del ciclo è -1 (conteggio all'indietro)
   PRINT(NUMERO%;)
END FOR
```

si otterrà:

```
 10  9  8  7  6  5  4  3  2  1
```

Nota:

- In **ERRE-VIC20 1.3** la parola chiave **DO** viene omessa ed al posto di **END FOR** si utilizza la parola chiave **NEXT**. Inoltre il ciclo viene eseguito sempre <u>almeno</u> una volta poiché il controllo di ciclo avviene alla fine.
- I valori delle varie espressioni sono modificabili nel corso dell'esecuzione del ciclo e queste modifiche si ripercuotono in modo dinamico sul ciclo: per motivi di chiarezza programmativa si consiglia comunque di utilizzare sempre costanti. Ad esempio sono equivalenti le

```
FOR NUMERO%=2 TO 100 STEP 2 ! passo 2
    PRINT(NUMERO%,NUMERO%^2)
END FOR
```

```
FOR NUMERO%=2 TO 100    ! passo 1
    PRINT(NUMERO%,NUMERO%^2)
    NUMERO%=NUMERO%+1
END FOR
```

ma la prime è preferibile per chiarezza e funzionalità.

FOREACH

ERRE-C64 3.2 **ERRE-PC 3.0**

Istruzione

Scopo:
Implementa un ciclo enumerativo in cui la variabile di controllo *var1* assume, consecutivamente, tutti i valori racchiusi in una lista o in un vettore *var2*.

Sintassi:

```
FOREACH var1 IN (lista)|var2[] DO
    ┌──────────────┐
    │  B l o c c o │
    └──────────────┘
END FOREACH
```

lista è un insieme di valori costanti (dello stesso tipo: o numerici o alfanumerici) separati tra di loro con virgole e non può essere vuota.

Esempio:

Da

```
FOREACH I$ IN  ("PIPPO","PLUTO","PAPERINO") DO
   PRINT("Adesso guardo -> ";I$)
END FOREACH
```

si ottiene come risultato:

```
Adesso guardo -> PIPPO
Adesso guardo -> PLUTO
Adesso guardo -> PAPERINO
```

Nota:

- Nella versione per C-64 la sintassi è leggermente diversa, più simile a quella delle versioni 2.X di ERRE per PC:

  ```
  FOREACH var=(lista) DO
      ┌──────────────┐
      │  B l o c c o │
      └──────────────┘
  END FOREACH
  ```

- Nella versione **ERRE-VIC20 1.3** tale struttura di controllo può essere simulata ricorrendo ad un ciclo a conteggio, ove i valori di *lista* sono contenuti in un array.

Funzione

Scopo:
Assegna ad una variabile il risultato di una istruzione WRITE.

Sintassi:
FORMAT$(formato,variabile)

Esempio:

```
.....
A=5.443E-1
B$=FORMAT$("**##.####^^^^",A)
PRINT(B$)
.......
```

allora sarà stampato

```
***5.4430E-01
```

Note:

- **FORMAT$** è l'equivalente funzionale di **WRITE**.

FORWARD

Istruzione

Scopo:
Consente di alterare la gerarchia delle procedure (ad esempio in caso di mutua ricorsione). FORWARD termina una dichiarazione di procedura.

Sintassi:
PROCEDURE nome_proc([param.ingresso][->param.uscita]) FORWARD

Esempio:
La procedura UP è dichiarata **FORWARD** per poter essere utilizzata dalla successiva DOWN a sua volta richiamata nel corpo della procedura UP.

```
!-----------------------------------------------------------------------------------------
! CATCH 22 :  programma di esempio sull'uso della clausola FORWARD
!-----------------------------------------------------------------------------------------

PROGRAM CATCH22

!$STACKSIZE=4096

DIM X,I

FUNCTION ODD(X)
  ODD=FRAC(X/2)<>0
END FUNCTION

PROCEDURE UP(I->I)
  FORWARD

PROCEDURE DOWN(I->I)
  I=INT(I/2)
  PRINT(I,)
  IF I<>1
    THEN
      UP(I->I)
  END IF
END PROCEDURE

PROCEDURE UP
  WHILE ODD(I) DO
    I=I*3+1
    PRINT(I,)
  END WHILE
  DOWN(I->I)
END PROCEDURE
```

```
BEGIN
  PRINT("Imposta un numero intero ....";)
  INPUT(X)
  UP(X->X)
  PRINT(CHR$(7))
  PRINT("Ok, il programma si è fermato.")
  REPEAT
    GET(CH$)
  UNTIL CH$<>""
END PROGRAM
```

Note:

- Una procedura non può essere dichiarata **FORWARD** se non presenti dei parametri di ingresso e/o di uscita.
- La mutua ricorsione occupa molto spazio di stack: l'utilizzo della direttiva di compilazione **$STACKSIZE** permette di dimensionarlo (solo **ERRE-PC 3.0**).
- Lo spazio di stack non può essere ridimensionato in **ERRE-C64 3.2**

FPRINT(

Istruzione

Scopo:
Stampa sullo standard output "lista_espressioni" secondo la seguente formattazione: le espressioni numeriche vengono stampate senza l'eventuale spazio bianco iniziale e le espressioni di tipo stringa vengono stampate racchiuse tra doppi apici usando "," come separatore.

Sintassi:
FPRINT([#numero_file,]lista_espressioni)

Esempio:

Se A%=80, B=-90.5, C$="That's all." e D=-1E-13 allora le istruzioni

```
FPRINT(A%,B,C$,D)
PRINT(A%,B,C$,D)
```

stamperanno rispettivamente

```
80,90.5,"That's all.",-1E-13
 80          -90.5         That's all.   -1E-13
```

Note:

- FPRINT consente di preparare file dati molto compatti utili per l'importazione in altri programmi (ad esempio il formato CSV - Comma Separated Values utilizzato anche da Microsoft Excel).

FRAC(

Funzione

Scopo:
Restituisce la parte frazionaria (decimale) di una espressione numerica.

Sintassi:
FRAC(espr)

Esempio:

```
PRINT(FRAC(1.5))
```

stampa

```
.5
```

che è la parte decimale di 1.5.

Note:

- Nelle precedenti versioni di ERRE (sia per PC che per C-64) FRAC era scritta come FRC.

FREEMEM e FRE(

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

Funzione

Scopo:
Calcola la memoria disponibile per il modulo ERRE in esecuzione ed ottimizza lo spazio occupato dalle variabili di tipo string.

Sintassi:
FREEMEM o per **ERRE-VIC20 1.3** **FRE(X)**

Esempio:

```
PRINT FREEMEM
```

stampa la memoria disponibile.

```
PRINT FRE(0)
```

Nota:

- X è un argomento fittizio. Su **ERRE-VIC20 1.3** il risultato della funzione è di tipo INTEGER, per cui valori negativi della funzione vanno corretti aggiungendo 65535. Per **ERRE-C64 3.2** e **ERRE-PC 3.0** **FREEMEM** fornisce invece il risultato corretto a 16 bit senza segno e quindi compreso tra 0 e 65535.

FREEFILE

Funzione

Scopo:
Fornisce il primo numero di file disponibile.

Sintassi:
FREEMEM

Esempio:
```
PRINT(FREEFILE)
```

stampa il primo numero di file disponibile.

Nota:

- In **ERRE-PC 3.0** il numero massimo di file apribili contemporaneamente è determinato dalla variabile riservata **MAXFILES** e va da 3 a 15 e comunque fa sempre riferimento al comando FILES=mmm del CONFIG.SYS del DOS. Per ottenere queste informazioni in **ERRE-VIC20 1.3** e **ERRE-C64 3.2** riferirsi alla tabella file del Kernal allocata a 601-630 ($259-$276) il cui puntatore è in 152 ($98): in ogni caso il numero massimo di file apribili è 10..

FUNCTION

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Permette la dichiarazione di una funzione a riga singola scritta dall'utente, seguita dal calcolo effettivo ("corpo della funzione").

Sintassi:

ERRE-PC 3.0
```
[VARIANT] FUNCTION nome_funz([var1[,varN]])
   nome_funz=espr
END FUNCTION
```

ERRE-C64 3.2
```
FUNCTION nome_funz([var])
   nome_funz=espr
END FUNCTION
```

ERRE-VIC20 1.3
```
FUNCTION nome_funz(var)=espr
```

Esempio:
La funzione $y(x)=x^2+5x+6$ può essere scritta come:

```
FUNCTION PARABOLA(X)
      PARABOLA=X*X+5*X+6
END FUNCTION
```

ed un richiamo possibile è:

```
ZETA=X-Y+PARABOLA(X+Y)
```

Nota:

- Le function sono composte da una sola linea e restituiscono sempre un valore. Tale valore è numerico o stringa in **ERRE-PC 3.0** e solo numerico in **ERRE-VIC20 1.3** e **ERRE-C64 3.2**.
- L'opzione **VARIANT** (**ERRE-PC 3.0**) consente di richiamare una funzione con lo stesso nome, a prescindere della precisione (real o double).
- Gli argomenti della funzione sono **sempre** variabili locali.
- Il numero di argomenti che possono essere passati ad una function è:
  - uno di tipo numerico per **ERRE-VIC20 1.3;**
  - al massimo uno (numerico) per **ERRE-C64 3.2**
  - più di uno (numerici e/o stringa) per **ERRE-PC 3.0.**
- In **ERRE-C64 2.3** la sintassi è leggermente diversa:

```
FUNCTION nome_funz
    nome_funz([var])=espr
END FUNCTION
```

GET(

Istruzione

Scopo:
Acquisisce un carattere dallo standard input o da un file.

Sintassi:
GET([#numero_file,]lista_var)

Esempio:

```
.....
REPEAT
    GET(CH$)
UNTIL CH$<>""
.......
```

è un ciclo di attesa: si prosegue battendo un tasto.

Note:

- In **ERRE-PC 2.6** **GET** è l'equivalente di **INPUT$**(1) a meno che non sia preceduta dalla direttiva di compilazione **!$KEY**.

GETKEY$

ERRE-C64 3.2 **ERRE-PC 3.0**

Funzione

Scopo:
Equivalente funzionale dell'istruzione **GET**.

Sintassi:
GETKEY$

Esempio:

```
.....
REPEAT
UNTIL LEN(GETKEY$)<>0
.......
```

Nota:

- Contrariamente all'istruzione **GET**, **GETKEY$** si applica solo allo standard input.
- In **ERRE-PC 2.6** si può utilizzare liberamente la funzione **INKEY$** dell'interprete di R-Code.

GOTO

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Dirige l'esecuzione del programma verso un punto prefissato, segnato tramite una "label" ("etichetta")

Sintassi:
GOTO etichetta

Esempio:

ERRE-C64 3.2 ERRE-PC 3.0

```
PROGRAM GOTO_ESEMPIO
LABEL 99
BEGIN
    INPUT("NUMERO",I)
    IF I>100 THEN
       A$="SUBITO"
       GOTO 99     ← salto
    END IF
    PRINT("NUMERO <=100 !")
    A$="DOPO"
99:                 ← etichetta
    PRINT("ETICHETTA RAGGIUNTA ";A$;"!")
END PROGRAM
```

ERRE-VIC20 1.3

```
program goto←test
begin
print chr$(147);
!ciclo loop..endloop
while true
   input "Numero";n
   if n>100
    then
      goto *break*  ← salto
   endif
endw
*break*            ← etichetta
end.
```

Nota:

- E' possibile effettuare salti solo all'interno di una stessa procedure o del main (si dice che 'GOTO' è un'istruzione locale).
- Per la gestione delle etichette nella varie versioni vedi la voce "etichetta (label)".

IF

Istruzione

Scopo:
Implementa l'istruzione decisionale che permette di effettuare una scelta tra due alternative (o più se viene utilizzata la clausola **ELSIF**): se il valore di "espressione_if" è TRUE viene eseguito il blocco che segue **THEN**, altrimenti vengono valutate in sequenza le "espressione_elsif" dei vari rami **ELSIF** (se sono usati) ed eseguito il blocco corrispondente della prima che risulti TRUE altrimenti se anche tutte queste sono FALSE viene eseguito il blocco che segue **ELSE** (che può anche essere facoltativa).

Sintassi:

```
IF espressione_if
   THEN
      ┌─────────────────┐
      │  B l o c c o    │
      └─────────────────┘
  { ELSIF espressione_elsif
      ┌─────────────────┐
      │  B l o c c o    │
      └─────────────────┘
  }
   ELSE
      ┌─────────────────┐
      │  B l o c c o    │
      └─────────────────┘
END IF
```

Esempio:

Per vedere se un carattere assegnato è una lettera maiuscola si può scrivere:

```
.....
  IF A$>="A" AND A$<="Z"
   THEN
     PRINT("E' una lettera maiuscola!!")
   ELSIF A$>="a" AND A$<="z"
     PRINT("Non è una lettera maiuscola!!")
   ELSE
     PRINT("Non è proprio una lettera!!")
  END IF
.....
```

Per calcolare, invece, il numero dei giorni di un mese (M%) assegnato un anno qualsiasi (A%) si può scrivere:

```
.....
```

```
  IF M%=4 OR M%=6 OR M%=9 THEN
     GG%=30
   ELSIF M%=2 AND BISEST% THEN    <--- BISEST% è un flag che indica se
     GG%=29                           l'anno A% è bisestile
   ELSIF M%=2 AND NOT BISEST% THEN
     GG%=28
   ELSE
     GG%=31
  END IF
.....
```

Nota:

- **ERRE-VIC20 1.3** non ha l'opzione **ELSIF**.
- Solo per **ERRE-PC 3.0** è possibile, nel caso che "Blocco" sia composto da una sola istruzione e non sia presente ELSIF, omettere "END IF" e scrivere tutto su una sola riga.

```
IF A=3 THEN
     B+=1
   ELSE
     B-=1
END IF
```

può essere scritto come

```
IF A=3 THEN B+=1 ELSE B-=1
```


L'uso di questa opzione fa perdere la compatibilità con **ERRE-C64 3.2** che esige sempre la fine dell'istruzione IF con **END IF**.

IF EVENT

Istruzione

Scopo:
Determina l'azione da svolgere in seguito al verificarsi di un evento.

Sintassi:
IF EVENT("messaggio") THEN "azione"

Esempio:
Vedi istruzione **EVENT**

Nota:

- "azione" è il richiamo di una procedure che gestisce il "messaggio" collegato all'evento.

Funzione speciale

Scopo:
Assegna il risultato di un confronto condizionale ad una variabile.

Sintassi:
var=IIF(condizione,se_vera, se_falsa)

Esempio:

```
RES$=IIF(N>=0,"Positivo","Negativo")
   ▲       ▲   ▲          ▲
   |       |   |          └─ usata se condizione è FALSE
variabile  |   └─ usata se condizione è TRUE
assegnam.  └─ condizione
```

che è equivalente a

```
IF N>=0 THEN
   RES$="Positivo"
 ELSE
   RES$="Negativo"
END IF
```

Nota:

- E' possibile utilizzare IIF (limitata alle espressioni numeriche) nella definizione di una funzione. Ad esempio è possibile scrivere una funzione MAX a due argomenti come segue:

```
FUNCTION MAX(X,Y)
    MAX=IIF(X>=Y,X,Y)
END FUNCTION
```

IN e NOT IN

Operatore

Scopo:
Operatore di inclusione.

Sintassi:
IN range

NOT IN range

Esempio:

```
IF I IN 1..5 THEN
```

equivale a

```
IF I>=1 AND I<=5 THEN
```

Nota:

- **IN** deve essere usato in una istruzione **CASE** nel caso di "case_label" con operatori relazionali.
- **NOT IN** implementa l'operatore di esclusione.

INPUT(

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Acquisisce uno o più valori dallo standard input o da un file separati, eventualmente, da ",", attribuendoli alle variabili di "lista_var".

Sintassi:
INPUT(["prompt",|;][#numero_file,]lista_var[;]) **ERRE-PC 3.0**

INPUT(["prompt",][#numero_file,]lista_var) **ERRE-VIC20 1.3 ERRE-C64 3.2**

Esempi:

INPUT(A$) → ? _
INPUT("Numero";N%) → Numero? _
INPUT("Numero",N%) → Numero _
INPUT(#1,A$) → legge dati dal file #1 (fino alla prima "," o fine riga e li attribuisce ad A$

Note:

- INPUT genera, se non diversamente previsto, un "?" come segnalazione di ingresso (solo da standard input).
- Il "prompt" esplicito viene usato solo per l'acquisizione da standard input.
- Il ";" finale non va a capo dopo una INPUT (solo per l'acquisizione da standard input – solo per **ERRE-PC 3.0**).
- **INPUT$** è l'equivalente funzionale della INPUT (solo da standard input per **ERRE-C64 3.2**).
- **LINPUT** acquisisce dati fino al primo RETURN e li attribuisce ad una singola variabile stringa.
- I dati acquisiti devono concordare in numero e tipo con le variabili elencate in "lista_var": l'interprete di R-Code effettua questo controllo in esecuzione. Le differenze riguardo la gestione di questo controllo sono diverse tra le varie piattaforme e sono elencate nella tabella seguente.

ERRE-VIC20 1.3 ERRE-C64 3.2	ERRE-PC 3.0
Se i dati introdotti sono insufficienti, l'interprete di R-Code visualizza un doppio punto interrogativo (**??**) e richiede il dato o i dati mancanti, mentre se i dati immessi sono di più di quelli richiesti, viene visualizzato il messaggio "**?Extra ignored**" (i dati in sovrappiù vengono quindi ignorati) e ripresa l'esecuzione del programma. L'immissione di dati non coerenti con le variabili dichiarate comporta la comparsa del messaggio "**?Redo from start**" (rifare da capo) e la riesecuzione dell' istruzione.	Troppi o troppo pochi dati, oppure un tipo sbagliato di valori (per esempio, numerico invece di stringa), causano la stampa del messaggio "**?Redo from start**". Non viene fatto nessun assegnamento di valori per le variabili di input finché non viene data una riposta accettabile.

INPUT$(

ERRE-C64 3.2 **ERRE-PC 3.0**

Istruzione

Scopo:
Acquisisce un certo numero di caratteri dallo standard input o da un file secondo il valore dell'argomento (quest'ultimo solo per **ERRE-PC 3.0**).

Sintassi:
var=INPUT$(expr,#nfile)

Esempio:

```
.....
A$=INPUT$(5)
.....
```

attribuisce ad A$ i primi 5 caratteri ricevuti dallo standard input.

Note:

- Contrariamente ad INPUT, non viene presentato nessun prompt.

INSTR(

Funzione

Scopo:
Trova la posizione di una stringa all'interno di un'altra stringa.

Sintassi:
INSTR(espr,espr1$,espr2$)

Esempio:

```
.....
PRINT(INSTR("PIPPO","PP"))
.....
```

stampa

```
3
```

perché la stringa "PP" inizia al terzo carattere della stringa "PIPPO"

Nota:
Assumendo che *espr* sia uguale a N%, X$ a *espr1$* e Y$ a *espr2$*, allora:

- Se non viene specificato N% la ricerca inizia dalla prima posizione di X$
- Se N% è maggiore della lunghezza di X$ oppure se Y$ non viene trovata entro X$, la funzione fornisce come risposta 0.
- Se N% è negativo si verifica un errore.

INT(

Funzione

Scopo:
Fornisce il numero intero più grande minore di espr (*"floor"*).

Sintassi:
INT(espr)

Esempio:

```
.....
PRINT(INT(2.5))
.....
```

stampa

```
2
```

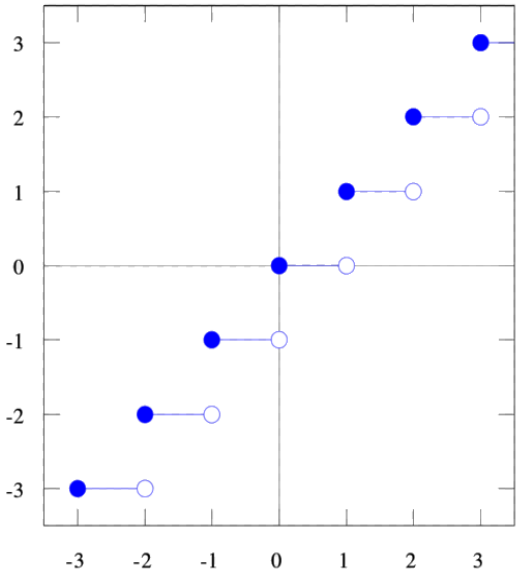
Nota:

- Per avere effettivamente la parte troncata (col proprio segno) di *espr* usare

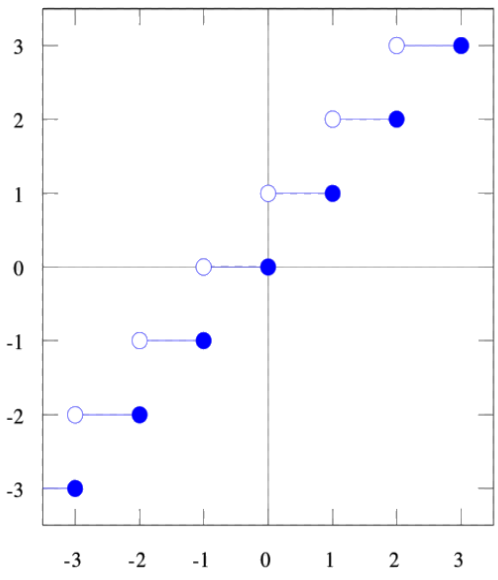
  ```
  SGN(espr)*INT(ABS(espr))
  ```

- Per ottenere invece il numero intero più piccolo maggiore di espr (funzione *"ceil"*) usare

  ```
  INT(espr)-(espr-INT(espr)>0)
  ```



Funzione "floor" - INT



Funzione "ceil"

LABEL

ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Dichiara le etichette usate nel modulo.

Sintassi:
LABEL etichetta1[,etichettaN]

Esempio:

```
LABEL 90,110,700
```

dichiara le etichette 90, 110 e 700. Queste potranno essere utilizzate da una istruzione **GOTO** o **RESUME**.

Nota:

- In **ERRE-VIC20 1.3** la dichiarazione di etichetta non è obbligatoria.

LBOUND(

Funzione

Scopo:
Fornisce l'estremo inferiore di una qualsiasi delle dimensioni di un array.

Sintassi:
LBOUND(var_array,espr)

Esempio:

```
.....
DIM A[5,10]
PRINT(LBOUND(A,2)
.....
```

stampa

```
0
```

l'indice minimo della seconda dimensione.

Nota:

- E' sempre uguale a 0: in **ERRE-VIC20 1.3** **ERRE-C64 3.2** non è stata implementata per questo motivo.
- Se espr è uguale a 0 viene fornito il numero di dimensioni dell'array.
- In **ERRE-PC 2.6** essendo attiva la direttiva **!$BASE**, **LBOUND** può assumere anche il valore 1.

LEFT$(

Funzione

Scopo:
Seleziona i caratteri iniziali di una stringa secondo quando indicato dall'argomento.

Sintassi:
LEFT$(espr$,espr)

Esempio:

```
.....
PRINT(LEFT$("PLUTO",2))
.....
```

stampa

```
PL
```

Cioè i primi 2 caratteri iniziale della stringa "PLUTO".

Nota:

- Se il valore di *espr* è negativa viene fornito un errore
- Se il valore di *espr* è nulla viene fornita la stringa vuota ""
- Se il valore di *espr* supera la lunghezza di espr$ viene fornita espr$

LEN(

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

Funzione

Scopo:
Fornisce la lunghezza di una stringa.

Sintassi:
LEN(espr$)

Esempio:

```
…..
A$=”HO VISTO IL SOLE”
PRINT(LEN(A$))
…..
```

stampa

16

Nota:

- **LEN** conteggia tutti i caratteri della stringa compresi spazi e caratteri non stampabili.
- **LEN=** può essere usata per evidenziare la lunghezza record di un file ad accesso diretto o di un file di comunicazione.

LET

Istruzione

Scopo:
Copia il contenuto di un record in un altro record (che fanno riferimento allo stesso TYPE).

Sintassi:
LET var_rec1=var_rec2

Esempio:

```
.....
TYPE CUSTOMERTYPE=(NAME$,ADDRESS$,PHONENUM$)
DIM CUSTOMER[25]:CUSTOMERTYPE,WORKSPACE:CUSTOMERTYPE
.....
LET CUSTOMER[I%]=WORKSPACE
......
```

Assegna il contenuto di WORKSPACE al I%-simo record di CUSTOMER, senza doverlo fare campo a campo.

Identificatore riservato

Scopo:
Modifica il comportamento dell'istruzione **RESUME.**

Sintassi:
RESUME LINE

Esempio:

```
RESUME LINE
```

Riprende l'esecuzione, dopo un errore gestito da **EXCEPTION**, dall'istruzione che ha provocato l'errore stesso.

Vedi:
RESUME

LINPUT(

Istruzione

Scopo:
Acquisisce una stringa (nel limite dei 255 caratteri) dallo standard input o da un file attribuendola alle variabili "var_stringa".

Sintassi:
LINPUT(["prompt",][#numero_file,] var_stringa[;])

Esempi:

LINPUT(A$) → _

LINPUT("Stringa?") → Stringa? _

LINPUT(#1,A$) → legge una stringa dal file #1 (fino al primo RETURN) e li attribuisce ad A$

Note:

- LINPUT non genera, contrariamente ad INPUT, un "?" come segnalazione di ingresso (solo usata con lo standard input).
- Il "prompt" esplicito viene usato solo per l'acquisizione da standard input.
- Il ";" finale non va a capo dopo una LINPUT (solo per l'acquisizione da standard input).
- Per simulare questa istruzione (solo da standard input) su **ERRE-C64 3.2** bisogna utilizzare la sequenza **POKE(198,1) POKE(631,13) INPUT(A$)**.

Funzione

Scopo:
Restituisce la posizione all'interno di un file aperto con il valore numerico dato dall'argomento.

Sintassi:
LOC(espr)

Esempio:

```
PRINT(LOC(1))
```

stampa la posizione all'interno del file #1.

Nota:

- Se il file è ad accesso diretto viene restituito l'ultimo record letto o scritto, se è sequenziale il numero di blocchi da 128 byte letti o scritti, se è di tipo COM il numero dei caratteri che attendono di essere letti.

LOCAL e LOCAL DIM

ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Permette di definire variabili locali in una procedura o in una dichiarazione di classe (solo PC).

Sintassi:

LOCAL var_1[,var_N]

LOCAL DIM $var_1[d_1,...,d_{N1}]$,$var_2[d_1,...,d_{N2}]$,.....,$a_N[d_1,...,d_{Nj}]$ (**ERRE-PC 3.0**)

Esempio:

```
PROCEDURE FACTORIAL(X%->F)
    LOCAL I%
    F=1
    IF X%<>0 THEN
      FOR I%=X% TO 2 STEP -1
        F=F*X%
      END FOR
    END IF
END PROCEDURE
```

I% è una variabile locale ed è diversa dalla I% usata eventualmente in altre procedure o nel main.

Nota:

- Le variabili dichiarate con **LOCAL** vengono allocate con valore zero o stringa nulla all'ingresso nella procedura e vengono cancellate all'uscita. L'istruzione **CLEAR** permette di azzerarle in un qualsiasi altro punto all'interno della procedura (**ERRE-PC 3.0**).
- Non è possibile dichiarare variabili locali aggregate in **ERRE-C64 3.2.**

Funzione

Scopo:
Restituisce la lunghezza del file aperto con *numero_file* uguale al valore dell'argomento.

Sintassi:
LOF(espr)

Esempio:

```
PRINT(LOF(2))
```

restituisce la lunghezza del file aperto come #2.

LOG(

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

Funzione

Scopo:
Calcola il logaritmo naturale (in base *e=2.7182818…..*) dell'argomento

Sintassi:
LOG(espr)

Esempio:

```
…..
PRINT(LOG(5))
…..
```

stampa

```
 1.609438
```

Nota:

- Se *espr* è minore a 0, la funzione fornisce un errore.
- Se si vuole avere il logaritmo di una espressione in un base qualsiasi (positiva) basterà scrivere (**ERRE-PC 3.0**):

```
VARIANT FUNCTION LOGN?(X),N)
        LOGN?=LOG(X?)/LOG(N)
END FUNCTION
```

Nel caso di **ERRE-C64 3.2** la base andrà esplicitata (potendo avere solo funzioni di una variabile):

```
FUNCTION LOG10(X)
        LOG10=LOG(X)/LOG(10)
END FUNCTION
```

oppure (**ERRE-VIC20 1.3**)

```
FUNCTION LG10(X)=LOG(X)/LOG(10)
```

LOOP

ERRE-C64 3.2 **ERRE-PC 3.0**

Istruzione

Scopo:
Implementa un ciclo infinito che diventa a condizione intermedia usando l'istruzione **EXIT**.

Sintassi:

```
LOOP
    ┌──────────┐
    │ B l o c c o │
    └──────────┘
END LOOP
```

Esempio:

```
…..
I=0
LOOP
   I=I+1
   PRINT(I)
END LOOP
…..
```

prosegue indefinitamente …….

Nota:

- Senza l'utilizzo di **EXIT** si ottiene un ciclo infinito! Si può uscire solo con Ctrl Break (PC) o Run/Stop (C64 e VIC-20).

MACHINE$

ERRE-C64 3.2 ERRE-PC 3.0

Variabile riservata

Scopo:
E' una variabile riservata ("di sistema") che serve ad identificare la macchina sulla quale viene eseguito ERRE System. E' strettamente collegato alle direttive di "compilazione condizionale" **!$IF** condizione **$THEN** .... **$ELSE** ..... **$ENDIF**.

Nota:
Al momento può assumere due soli valori:

- **PCIBM** per computer PC IBM e compatibili.
- **CBM64** per computer Commodore 64.

MAXFILES

Variabile riservata

Scopo:
E' il numero massimo di file apribili contemporaneamente durante l'esecuzione di un programma.

Nota:

- **MAXFILES** va da 3 a 15 e comunque fa sempre riferimento al comando FILES=mmm del CONFIG.SYS del DOS.
- Per ottenere queste informazioni in **ERRE-VIC20 1.3** e **ERRE-C64 3.2** riferirsi alla tabella file del Kernal allocata a 601-630 ($259-$276) il cui puntatore è in 152 ($98): in ogni caso il numero massimo di file apribili è 10..

MAXINT

ERRE-C64 3.2 ERRE-PC 3.0

Variabile riservata

Scopo:
E' il massimo numero rappresentabile per il tipo INTEGER.

Nota:

- Poiché il tipo INTEGER è codificato a 16 bit, il valore di MAXINT è 32767 (2^{15}-1)
- Le istruzioni **POKE** e **CALL** e la funzione **PEEK** nonché alcune direttive di compilazione lavorano usando il tipo "unsigned integer" non espressamente codificato nel linguaggio ERRE: gli estremi di questo tipo sono compresi tra 0 e 65535 (=2*MAXINT+1).

MAXREAL

ERRE-C64 3.2 ERRE-PC 3.0

Variabile riservata

Scopo:
E' il massimo numero rappresentabile per il tipo REAL, in accordo con la precisione numerica richiesta.

Nota:

- Su PC il suo valore è +1.701412E+38 (codificato in floating point su 32 bit), mentre su C-64 è +1.70141183E+38 (codificato in floating point su 40 bit).
- Il minimo numero rappresentabile (a parte lo zero) è 2.938736E-39 su PC e 2.93873588E-39 su C-64.
- Su C-64 è possibile, sfruttando il tipo REAL, una implementazione del tipo LONG INTEGER a 24 bit.

MAXREAL#

Variabile riservata

Scopo:
E' il massimo numero rappresentabile per il tipo LONG REAL, in accordo con la precisione numerica richiesta.

Note:

- Viene codificato su 64 bit. Vedi **MAXREAL** per gli estremi.
- Sfruttando opportunamente il tipo LONG REAL è possibile anche una implementazione del tipo LONG INTEGER a 32 bit.
- In **ERRE-PC 2.6** viene utilizzato **MAXLONGREAL**.

MID$(

Funzione

Scopo:
Permette di selezionare parti di una stringa.

Sintassi:

MID$(espr$,espr1[,espr2])

oppure

MID$(espr1$,espr1[,espr2])=espr2$

Esempio:

```
…..
A$="1234567890"
PRINT(MID$(A$,4,1))
…..
MID$(A$,4,2)="DE"
PRINT(A$)
.....
```

stampa rispettivamente

4

e

123DE67890

Nota:

- MID$, assieme a TIME e DATE$ è l'unica funzione che può essere messa a sinistra del segno di assegnamento.
- Se *espr1* o *espr2* sono negativi viene fornito un errore.
- MID$ può fornire come risultato anche una stringa vuota "" assegnando valori particolari a *espr1* o *espr2*.

MOD

ERRE-C64 3.2 ERRE-PC 3.0

Operatore/Funzione

Scopo:
Operatore che fornisce il resto di divisione intera.

Sintassi:
MOD

Esempio:

```
PRINT 13 MOD 4
```

stampa

```
1
```

Nota:

- Prima di effettuare l'operazione gli operandi vengono arrotondati all'intero superiore.
- In **ERRE-PC 3.0** e **ERRE-PC 2.6** gli operandi devono essere in valore assoluto minori di **MAXINT** (32767), altrimenti c'è un errore.
- In **ERRE-VIC20 1.3** e **ERRE-C64 2.3** l'operatore MOD deve venire reso con la sua definizione matematica e cioè **X MOD Y** => X **– Y*INT(X/Y),** mentre in **ERRE-C64 3.2** esiste come forma funzionale **MOD(X,Y)**.

NEW

Istruzione

Scopo:
Consente di dichiarare oggetti (semplici) che fanno riferimento ad una certa classe.

Sintassi:
NEW var_1:$classe_1$[,var_N:$classe_N$]

Esempio:
Vedi **CLASS**

Nota:
- Al momento non è possibile dichiarare array di oggetti.

NEXT

Istruzione

Scopo:
Istruzione finale di un ciclo FOR.

Sintassi:
NEXT

Esempio
Vedi **END FOR**

Nota:
Utilizzato anche da **ERRE-C64 2.3**.

NEXT

Identificatore riservato

Scopo:
Modifica il comportamento dell'istruzione **RESUME.**

Sintassi:
RESUME NEXT

Esempio:

```
RESUME NEXT
```

Riprende l'esecuzione, dopo un errore gestito da **EXCEPTION**, dall'istruzione che segue quella che ha provocato l'errore stesso.

Vedi:
RESUME

NOT e NOT(

Operatore

Scopo:
Rappresenta il connettore logico "negazione" all'interno di espressioni condizionali.

Sintassi:
NOT

oppure

NOT(

Esempio:

IF NOT X=3 THEN.....

questo test condizionale è vero se la variabile X è diversa da 3.

Nota:

- **NOT** può essere usato anche per operare sui singoli bit di una variabile e/o costante di tipo integer (cioè su 16 bit). Ad esempio

  NOT 15=-16 15=%00001111 perciò applicando la tabella delle verità del NOT si ottiene %11110000 e quindi -16 (usando il complemento a due). Vale in generale la relazione **NOT X=-(X+1)**.

Operatore	Valore A	Valore B	Risultato
NOT	V		F
	F		V

OF

Identificatore riservato

Scopo:
Termina una riga CASE.

Sintassi:
OF

Esempio:
Vedi **CASE**

OPEN(

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Permette l'apertura di un file.

Sintassi:
OPEN(lista_parametri)

A causa delle numerose opzioni possibili, esaminiamo la **OPEN** per ognuna delle tre piattaforme.

- **ERRE-PC 3.0**

OPEN(#numero_file,nome_file,tipo_file,tipo_accesso,[[LEN=]lung_record])

"#numero_file": questo numero associa il file ad un buffer di I/O; il suo valore dipende strettamente dalla piattaforma (nel caso del PC varia tra 1 e 255). Un buffer di I/O può essere associato solo ad un file alla volta: la funzione **FREEFILE** permette di conoscere il primo buffer di I/O libero. La variabile riservata **MAXFILES** permette invece di conoscere il numero massimo di file che possono essere aperti contemporaneamente. # è obbligatorio.

"nome_file": indica il nome attribuito al file e dipende strettamente dalla piattaforma; nel caso del PC può essere un file su disco (il cui nome segue quindi la sintassi DOS/Windows) oppure un dispositivo. Abbiamo a disposizione i seguenti dispositivi:

SCRN:	schermo	WRITE	
CONS:	schermo	WRITE	
KYBD:	tastiera	READ	
LPTx:	porta parallela	WRITE	x vale 1, 2 o 3
COMx:	porta seriale	READ WRITE	x vale 1 o 2

Egualmente possono essere utilizzati come "nome_file" anche i device standard del DOS:

CON	tastiera/schermo	READ o WRITE	
PRN	porta parallela 1	WRITE	
LPTx	porta parallela	WRITE	x vale 1, 2 o 3
AUX	porta seriale 1	READ WRITE	
COMx	porta seriale	READ WRITE	x vale 1, 2, 3 o 4
NUL	"null device"	READ WRITE	per test

Nel caso delle porte seriali al nome del dispositivo (COM1: o COM2:) si associano i parametri della porta utilizzata e cioè la velocità, il tipo di parità, il numero di bit di dati ed il numero di bit di stop nonché eventuali altre opzioni (un esempio possibile è *"COM1:2400,N,8,1"* che apre la prima seriale con una velocità di 2400 bps, nessuna parità, 8 bit di dati ed 1 bit di stop).

"tipo_file": il file può essere sequenziale (**"S,I"**, **"S,O"** e **"S,A"** in lettura, scrittura o in accodamento), ad accesso diretto (**"R"** - "random"), o di comunicazione (**"C"**). Per motivi di leggibilità sono ammesse anche le forme estese **"SEQ,INPUT"**, **"SEQ,OUTPUT"**, **"SEQ,APPEND"**, **"REL"** e **"COM"**. La funzione **FILEATTR$** permette di conoscere il tipo del file una volta assegnato "numero_file".

"tipo_accesso": indica la modalità di accesso al file (**READ**, **WRITE**, **READ WRITE)** ed il tipo di gestione del file da parte di altri processi in ambiente distribuito (**SHARED**, **LOCK READ**, **LOCK WRITE** e **LOCK READ WRITE**). Questa seconda caratteristica è facoltativa: se manca si assume la proprietà del file da parte del processo che lo utilizza.

"lung_record" indica la grandezza del buffer di I/O (compreso tra 128 e 1024) e va utilizzato solo con i file di tipo "R" o di tipo "C". L'identificatore **LEN=** è obbligatorio.

- **ERRE-C64 3.2**

OPEN(file_number,device_number,secondary_address,command_string)

dove:

"*file_number*" è il numero del buffer utilizzato dal file (tra 1 e 127).

"device_number" indica un file su disco (**8**,**9**,**10**,**11**) o su nastro (**1**), lo schermo (**3**), la tastiera (**0**), una stampante (**4**,**5**), il plotter (**6**) o la porta RS-232 (**2**). Valori di "*device_number*" superiori a 31 daranno errore in fase di esecuzione. E' possibile, inoltre, indicare anche un nome di periferica per una maggior chiarezza secondo la seguente tabella:

NOME DELLA PERIFERICA	NUMERO
"keyboard"	0
"tape"	1
"rs232"	2
"video"	3
"printer"	4
"printer2"	5
"plotter"	6
"ramdisk"	7
"disk"	8
"disk2"	9
"disk3"	10
"disk4"	11

"secondary_address" indica l'indirizzo secondario: nel caso del disco è il numero di canale che il Sistema Operativo apre con il dispositivo, nel caso del nastro indica lettura o scrittura, nel caso della stampante tipo di font utilizzato, ecc...

"command_string" è la stringa comandi (non obbligatoria): nel caso di disco o nastro va specificato il nome del file. Solo nel caso del disco vanno date le caratteristiche del file:

- **,S,R** indica "sequenziale in lettura"

- **,S,W** indica "sequenziale in scrittura"
- **,S,A** indica "sequenziale in scrittura come append"
- **,L** indica "relativo – ad accesso diretto", seguito da CHR$(lung_record)
- **,U,R** indica "user in lettura"
- **,U,W** indica "user in scrittura"

Possono essere usate anche modalità speciali, come indicato nei manuali delle unità disco.

Nota: La parentesi dopo OPEN e a fine istruzione, il device_number espresso sotto forma di stringa non sono presenti in **ERRE-VIC20 1.3**.

Esempi:

OPEN(#1,"CLIENTI.DAT","REL",”READ WRITE”,LEN=82) **ERRE-PC 3.0**

apre come numero 1 sull'unità corrente un file (CLIENT.DAT) ad accesso diretto ("REL") in lettura/scrittura con lunghezza record = 82;

OPEN(#2,"DISK",2,"TRIS.SCR,S,R") **ERRE-C64 3.2**

apre un file (TRIS.SCR) sequenziale in lettura (,S,R) come numero 2 sulla prima unità disco (DISK --> 8).

OPEN 2,8,2,"TRIS.SCR,S,R" **ERRE-VIC20 1.3**

apre lo stesso file sequenziale in lettura (,S,R) come numero 2 sull'unità disco 8.

OR

Operatore

Scopo:
Rappresenta il connettore logico "intersezione" all'interno di espressioni condizionali.

Sintassi:
OR

Esempio:

```
IF X=3 OR Y=4 THEN.....
```

questo test condizionale è vero se almeno una delle due variabili assumono i valori indicati.

Nota:

- OR può essere usato anche per operare sui singoli bit di una variabile e/o costante di tipo integer (cioè su 16 bit). Ad esempio

  4 OR 2 = 6 — 4=%00000100 e 2=%000000010, perciò applicando la tabella dell'operatore OR otteniamo %00000110 e quindi 6 in decimale

Operatore	Valore A	Valore B	Risultato
OR	V	V	V
	V	F	V
	F	V	V
	F	F	F

OTHERWISE

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

Identificatore riservato

Scopo:
Ramo alternativo di un'istruzione **CASE**: quando nessuna delle opzione elencate con '->' è soddisfatta, viene eseguito il codice che segue **OTHERWISE**.

Sintassi:
OTHERWISE

Esempio:
Vedi **CASE**

PAUSE

ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Arresta temporaneamente il programma per un numero di secondi indicato dall'argomento numerico. Viene inoltre inviato allo "standard output" il messaggio indicato, se presente.

Sintassi:
PAUSE("messaggio",]espressione_numerica) (**ERRE-PC 3.0**)
PAUSE(espressione_numerica) (**ERRE-C64 3.2**)

Esempi:

```
PAUSE(1)
```

il programma viene bloccato per 1 secondo senza messaggio esplicito di avviso.

```
PAUSE("Premi un tasto per continuare....",0)
```

il programma, dopo aver visualizzato il messaggio "Premi un tasto per continuare....." viene bloccato fino alla pressione di un tasto qualsiasi.

Nota:

- In **ERRE-PC 3.0** se "espressione_numerica" è esplicitamente uguale a 0 il programma resta in attesa di un tasto: quindi PAUSE(0) equivale ad un ciclo di attesa, ad esempio REPEAT ... UNTIL LEN(GETKEY$)<>0. In **ERRE-C64 3.2** PAUSE(0) equivale a REPEAT ... UNTIL GETKEY$<>CHR$(0).
- In **ERRE-C64 3.2** una routine di 64VMS il cui entry point è a 51190 ($C7F6) ha lo stesso scopo. Il richiamo avviene tramite l'istruzione ERRE

  ```
  CALL($C7F6,durata)
  ```

  dove *durata* è misurata in decimi di secondo. Può comunque essere interrotta premendo un tasto.

PEEK(

Funzione

Scopo:
Legge una locazione di memoria nel segmento di 64K attualmente indirizzato.

Sintassi:
PEEK(espr)

legge la locazione di memoria *espr*.

Esempio:

```
....
PRINT(PEEK(1024))
....
```

legge la locazione 1024 dell'attuale segmento di memoria.

Nota:

- Nel PC la direttiva di compilazione **$SEGMENT** fissa il segmento attuale, mentre sui C-64/VIC-20 (che sono macchine ad 8 bit) si fa riferimento direttamente alla memoria indirizzabile: in entrambi i casi il massimo valore leggibile è 255.

- E' possibile implementare una funzione DPEEK che permette di leggere 16 bit (nel formato LOW-HIGH)

```
FUNCTION DPEEK(X)
  DPEEK=PEEK(X)+256*PEEK(X+1)
END FUNCTION
```

- Le locazioni utili dal punto di vista programmativo (ad es. memoria video) possono essere ottenute dalle mappe di memoria delle singole piattaforme.

POKE(

ERRE-VIC20 1.3 **ERRE-C64 3.2** **ERRE-PC 3.0**

Istruzione

Scopo:
Scrive (su 8 bit) in una locazione di memoria nel segmento (64K) attualmente indirizzato.

Sintassi:
POKE(espr1,espr2)

scrive *espr2* nella cella di memoria *espr1.*

Esempio:

ERRE-PC 3.0
```
!$SEGMENT=$B800
POKE(0,65)
```

oppure

ERRE-C64 3.2
```
poke(1024,1)
```

oppure

ERRE-VIC20 1.3
```
poke 4096,1
```

scrivono tutte una 'A' nell'angolo in alto a sinistra dello schermo.

Nota:
- *espr1* è compresa tra 0 e 65535 ("unsigned integer"), mentre *espr2* tra 0 e 255.
- **ERRE-C64 3.2** può usare la routine di 64VMS **DPOKE** che scrive su 16 bit.
- Molte routine di 64VMS possono essere personalizzate con opportune "poke" in certe locazioni di memoria.

POLY(

Funzione

Scopo:
Permette di scrivere in forma compatta una espressione polinomiale ad una variabile.

Sintassi:
POLY(lista)

Esempio:

```
……
PRINT POLY(X,1,2,-3,5)
…..
```

è l'equivalente di

```
……
PRINT X^3+2*X^2-3*X+5
…..
```

Il primo elemento della lista (variabile o costante) è la variabile indipendente del polinomio.

POS(

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

Funzione

Scopo:
Ritorna la colonna attuale su cui è posizionato il cursore.

Sintassi:
POS(espr)

Esempio:

```
……
PRINT(“ABCDE”;POS(0);)
…..
```

stampa

```
6
```

valore che sarà utilizzato dall’istruzione **PRINT** seguente.

Nota:

- *espr* è un argomento fittizio, ma deve essere compreso tra 0 e 255 per non generare un errore.
- In **ERRE-PC 2.6** e in **ERRE-PC 3.0** si può utilizzare liberamente la funzione **CSRLIN** dell'interprete di R-Code, che fornisce il numero di riga attuale su cui è posizionato il cursore. In **ERRE-VIC20 1.3** e **ERRE-C64 3.2** può essere impiegato per lo stesso scopo **PEEK(214)**.

PRINT(

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo :
Stampa sullo standard output o su un file "lista_espressioni" secondo la formattazione indicata. Questa può venire assegnata tramite le funzioni di tabulazione **SPC** e **TAB** o tramite i separatori ":", ";" e "," che indicano rispettivamente stampa a capo, stampa di seguito e stampa nel prossimo campo: la lunghezza di ogni campo dipende dalla larghezza dello schermo (40 o 80 colonne a disposizione). Tutte le variabili di tipo INTEGER, BOOLEAN e REAL occupano un campo, le variabili di tipo LONG REAL possono occupare due campi, mentre per le variabili di tipo STRING conta la lunghezza delle stringhe stesse.

Sintassi:
PRINT([#numero_file,]lista_espressioni)

Esempio:

se I%=3 e J%=7.55

allora

```
PRINT(I%,J,I%,J)
PRINT(I%;J;I%;J)
```

stampano rispettivamente

```
  campo n.1      campo n.2      campo n. 3  campo n. 4    campo n. 5
-------------==============-------------==============-------------
12345678901234123456789012341234567890123412345678901234123456789012341234
 3              7.55           3              7.55
▲              ▲              ▲              ▲
└------------ spazi per il segno del numero ----------------┘

-3 -7.55 -3 -7.55
  ▲      ▲  ▲
```

spazi dopo il numero

Note:

- In **ERRE-VIC20 1.3** PRINT si usa senza le parentesi ed il separatore ":" non è ammesso.
- Fino alla versione **ERRE-PC 2.6** PRINT era abbreviabile con '**?**'. Ad esempio ?(X) era equivalente a PRINT(X).
- I caratteri della tabella dei codici ASCII possono essere idealmente divisi in "caratteri stampabili" e "caratteri non stampabili": questi ultimi servono per eseguire delle funzioni speciali che dipendono strettamente dalla piattaforma. A questo scopo si usa la sequenza **PRINT(CHR$(n))**, dove n è inferiore a 32, così nel caso del PC è possibile avere:

ERRE-PC 3.0

n	risultato
7	beep
9	tabulazione
10	riga a capo (come 13)
11	cursore in alto a sinistra
12	cancella lo schermo
13	riga a capo (come 10)
28	sposta cursore a destra
29	sposta cursore a sinistra
30	sposta cursore su
31	sposta cursore giù
altri	stampano il carattere corrispondente

ERRE-C64 3.2

n	risultato
5	colore Bianco
28	colore Rosso
30	colore Verde
31	colore Blu
129	colore Arancio
144	colore Nero
150	colore Rosso 2
151	colore Grigio 1
152	colore Grigio 2
153	colore Verde 2
154	colore Blu 2
155	colore Grigio 3
156	colore Porpora
158	colore Giallo
159	colore Azzurro
8	disabilita passaggio maiuscolo/minuscolo
9	abilita passaggio maiuscolo/minuscolo
13	riga a capo
14	passaggio al minuscolo
142	passaggio al maiuscolo
17	sposta cursore su
145	sposta cursore giù
18	attiva "reverse"
146	disattiva "reverse"
19	cursore in alto a sinistra
147	cancella lo schermo
20	cancella carattere a sinistra
29	sposta cursore a destra
157	sposta cursore a sinistra
altri	stampano il carattere corrispondente

In entrambi i casi gli stessi risultati si possono ottenere usando la unit "CRT.LIB" richiamando le opportune procedure.

PROCEDURE

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Dichiara una procedura assegnando un nome, una eventuale lista di parametri "formali" (di ingresso e/o di uscita) e la tipologia.

Sintassi:
[VARIANT] PROCEDURE nome_proc[([param.ingresso][->param.uscita])] _
_ [FORWARD | EXTERNAL,costante]

Esempio:

```
PROGRAM PROC_TEST

DIM X                    ◄--- variabile globale X
DIM A%[10]               ◄--- array globale A%[]

PROCEDURE MY_PROC  ◄--- procedure senza lista parametri
      LOCAL X, I%              ◄--- variabili locali X e I%  \ distinte dalle
      LOCAL DIM A%[5]          ◄--- array locale A%[]        / precedenti
      X=99
      FOR I%=0 TO 5 DO
        A%[I%]=I%*3
      END FOR
      PRINT("MYPROC =>";X,A%[5])
END PROCEDURE

 BEGIN
    X=11
    A%[5]=999
    MY_PROC
    PRINT("MAIN =>";X,A%[5])
END PROGRAM
```

Nota:

- Per i parametri formali di ingresso e di uscita si utilizza come separatore tra gli elementi ','.
- Una procedura può avere le proprie variabili locali (se aggregate solo per **ERRE-PC 3.0**)
- L'istruzione **VARIANT** definisce una procedura generica, indipendente dal tipo di dati che tratta. (**ERRE-PC 3.0**)
- L'istruzione **FORWARD** permette la mutua versione (**ERRE-C64 3.2** e **ERRE-PC 3.0**)
- L'istruzione **EXTERNAL** dichiara una procedura scritta in R-Code e residente su un file esterno. (**ERRE-PC 3.0**)
- Sono ammesse procedure senza parametri (**ERRE-C64 3.2** e **ERRE-PC 3.0**) o con soli parametri di ingresso o di uscita: i "parametri formali" possono essere semplici, aggregati,

funzioni o record (**ERRE-PC 3.0**). Nel caso di variabili aggregate, il numero massimo di dimensioni è di 3.

- In **ERRE-PC 2.6** non è disponibile l'istruzione **EXTERNAL**.
- In **ERRE-VIC20 1.3** non è ammesso il passaggio dei parametri.
- In **ERRE-C64 2.3** la dichiarazione dei parametri è invertita e si usa come separatore tra gli elementi ';':

PROCEDURE nome_proc[([param.uscita][←param.ingresso])]

PROGRAM

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Identifica l'inizio di un modulo ERRE. E' una delle tre istruzioni sempre presenti in un modulo ERRE.

Sintassi:
PROGRAM identificatore

Esempio:

```
PROGRAM HELLO          ◄---- intestazione del modulo
BEGIN                  ◄---- inizio esecuzione
  PRINT("HELLO WORLD")
END PROGRAM            ◄---- termine del modulo
```

RANDOMIZE(

ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Inizializza il generatore di numeri casuali usando l'argomento come seme ("seed").

Sintassi:
RANDOMIZE(espr)

Esempio:

```
.....
RANDOMIZE(16)
.....
```

Inizializza il generatore di numeri casuali usando 16 come generatore.

Note:

- Per avere ogni volta una sequenza diversa utilizzare la variabile di sistema **TIMER**, altrimenti ad ogni nuova esecuzione verranno generati gli stessi numeri casuali.
- L'equivalente per **ERRE-VIC20 1.3** è

```
X=RND(-TIME)
```

READ(

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Legge i dati contenuti nelle righe **DATA** e li attribuisce a variabili.

Sintassi:
READ(var$_1$[,...,[var$_N$]])

Esempio:

```
.....
DATA(1,2,3,4)
.....
READ(X,Y,Z,W)
PRINT(X,Y,Z,W)
.....
```

stamperà

```
1       2       3       4
```

Nota:

- Ci deve corrispondenza (in numero e tipo) tra i dati delle righe **DATA** e le variabili lette da **READ**, altrimenti viene generato un errore.

REDIM

Istruzione

Scopo:
Permette di ridimensionare uno o più array, conservandone o meno i valori originali: non è possibile modificarne il numero di dimensioni.

Sintassi:
REDIM var_array$_1$(dimensioni)[, ..., var_array$_N$(dimensioni)]

Esempio:

```
PROGRAM REDIM_TEST

DIM P%,I%,DONE%,UB%,A$
DIM X%[15]

PROCEDURE MAT_PRINT(X%[],UB%)
LOCAL I%
  FOR I%=1 TO UB% DO
    PRINT(X%[I%];)
  END FOR
  PRINT
END PROCEDURE

BEGIN
  DATA(23,-45,6,5,1,12,7,1,-9,10)
  PRINT("Items in list:";)
  FOR I%=1 TO 10 DO
    READ(X%[I%])
  END FOR
  MAT_PRINT(X%[],UBOUND(X%,1))

  P%=10 DONE%=FALSE

  REPEAT
    INPUT("Item to add to list (ENTER to finish)",A$)
    IF LEN(A$)>0 THEN
      P%+=1
      IF P%>15 AND NOT DONE% THEN
          !$PRESERVE
          REDIM X%[30]
          DONE%=TRUE
      END IF
      EXIT IF P%>30   ! 30 items max
      X%[P%]=VAL(A$)
    END IF
```

```
  UNTIL LEN(A$)=0
  MAT_PRINT(X%[],UBOUND(X%,1))
END PROGRAM
```

Nota:

- L'istruzione **REDIM** viene risolta in modo statico **prima dell'esecuzione** per cui va prestata attenzione all'uso della funzione **UBOUND** applicata ad un array ridimensionato. La procedura MAT_PRINT viene richiamata su X% due volte: la prima volta l'indice massimo di X% vale 15 (quello dichiarato nel **DIM**), la seconda volta vale 30 (il valore del **REDIM** anche se questo non venisse eseguito).
- Le direttive di compilazione **$PRESERVE** e **$NOPRESERVE** specificano l'azione dell'istruzione **REDIM** che segue. Ad esempio, se si vuole ridimensionare l'array A% con il suo contemporaneo azzeramento, basterà fare:

  ```
  .....
  !$NOPRESERVE
  REDIM A%[150]
  .....
  ```
- E' possibile ridimensionare array con al massimo tre dimensioni.
- Altre istruzioni legate alla dichiarazione di array sono le direttive di compilazione **!$DYNAMIC, !$DIM, !$DIMCOMMON** e !**$ERASE.**

REPEAT

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Implementa un ciclo a condizione finale.

Sintassi:

```
REPEAT
   ┌─────────────┐
   │ B l o c c o │
   └─────────────┘
UNTIL espressione
```

Esempio:
Per calcolare l'espressione 1 + 1/2 + 1/3 + 1/4 +... fino a che la somma non superi un certo limite LIM prefissato si scrive:

```
SOMMA=0
NUM%=0
REPEAT
   NUM%=NUM%+1
   SOMMA=SOMMA+1/NUM%    ! conversione automatica di tipo
UNTIL SOMMA>LIM
```

Nota:

- Per la sua struttura, il ciclo viene eseguito sempre almeno una volta.

RESTORE

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Riposizione il puntatore all'inizio delle istruzioni **DATA** (distinguendo se sono nel main o in una procedure).

Sintassi:
RESTORE

Esempio:

```
PROGRAM TEST_RESTORE

PROCEDURE P1
LOCAL I%,A% ◄--- mettere come commento nel C-64
  DATA(5,6,7,8)
  RESTORE     ◄--- posiziona il puntatore al primo DATA all'interno della
                   procedure *corrente* (cioè sul dato "5")
  FOR I%=1 TO 4 DO
    READ(A%)
    PRINT(A%;)
  END FOR
  PRINT
END PROCEDURE

BEGIN
DATA(1,2,3,4)
  FLAG%=FALSE
  P1            ◄--- P1 riposizionerà il proprio puntatore
  RESTORE       ◄--- posiziona il puntatore al primo DATA all'interno del main
                     (dato "1")
  FOR I%=1 TO 4 DO
    READ(A%)
    PRINT(A%;)
  END FOR
  PRINT
  P1            ◄--- P1 riposizionerà di nuovo il proprio puntatore
  PRINT(FLAG%)
  RESTORE       ◄--- riporta il puntatore sul dato "1" del main
END PROGRAM
```

stampa:

```
5 6 7 8
1 2 3 4
5 6 7 8
```

Dopo l'ultimo **RESTORE**, una **READ** ulteriore rileggerebbe il dato "1" del main.

Nota:

- Assieme all'istruzione **GOTO**, **RESTORE** è un'istruzione di tipo "locale", cioè agisce nella sezione di programma in cui viene incontrata.

RESUME

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Indirizza l'esecuzione di un programma dopo la gestione delle eccezioni.

Sintassi:
RESUME etichetta|LINE|NEXT

Esempio:

RESUME 100

riprende l'esecuzione dalla label 100

RESUME LINE

riprende l'esecuzione dalla stessa istruzione che ha provocato l'errore.

RESUME NEXT

riprende l'esecuzione dalla istruzione che segue quella che ha provocato l'errore.

Nota:

- Se manca una istruzione **RESUME** al termine del blocco **EXCEPTION** avviene il ritorno all'istruzione che segue quella che ha provocato l'eccezione.

RIGHT$(

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

Funzione

Scopo:
Seleziona i caratteri finali di una stringa secondo quando indicato dall'argomento.

Sintassi:
RIGHT$(espr$,espr)

Esempio:

```
.....
PRINT(RIGHT$("PLUTO",2))
.....
```

stampa

```
TO
```

cioè gli ultimi 2 caratteri iniziale della stringa "PLUTO".

Nota:

- Se il valore di *espr* è negativa viene fornito un errore.
- Se il valore di *espr* è nulla viene fornita la stringa vuota "".
- Se il valore di *espr* supera la lunghezza di *espr$* viene fornita *espr$*.

RND(

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

Funzione

Scopo:
Genera un numero casuale compreso tra 0 e 1 se l'argomento è positivo, con argomento nullo o ripete l'ultimo numero generato (**ERRE-PC 3.0**) o genera un numero tramite l'orologio di sistema (**ERRE-VIC20 1.3 ERRE-C64 3.2**). Con un argomento negativo la sequenza di numeri casuali viene reinizializzata ogni volta.

Sintassi:
RND(espr)

dove *espr* è una espressione numerica

Esempio:

```
FOR I=1 TO 5 DO
   PRINT(RND(1);)
END FOR
```

stampa cinque numeri casuali (il generatore viene inizializzato in modo standard se non viene previsto altrimenti):

ERRE-PC 3.0
.1213501 .651861 .8688611 .7297625 .798853

ERRE-VIC20 1.3 ERRE-C64 3.2
.185564016 .0468986348 .827743801 .554749226 .897233831

Nota:

- Solo il segno dell'argomento determina il comportamento della funzione RND.
- L'istruzione **RANDOMIZE** - se supportata - permette di inizializzare il generatore dei numeri casuali all'inizio del programma: si può usare **RANDOMIZE(TIMER)** per **ERRE-PC 3.0** o **RANDOMIZE(TIME)** per **ERRE-C64 3.2** o **var=RND(-TIME)** per **ERRE-VIC20 1.3 .**
- L'espressione INT(RND(1)*N+1) genera numeri casuali compresi tra 1 e N.

Istruzione

Scopo:
Si posiziona sul numero record richiesto di un file ad accesso diretto: deve sempre essere seguito da una lettura o da una scrittura.

Sintassi:
SEEK(numero_file,numero_record)

Esempio:

```
.....
OPEN("R",1,"PROVA.REL","READ WRITE",LEN=256)
    FIELD(#1,48,C1$,60,C2$)
    FIELD(#1,128,C3$)
    A%=5
    B#=67.1256#
    F$="fffffff"
    ! scrive dati
    SEEK(#1,1)
    PRINT(#1,A%,B#,F$)
    ! legge dati
    SEEK(#1,1)
    INPUT(#1,ALFA%,BETA#,F$)
CLOSE(1)
```

scrive e poi legge i dati nel primo record di PROVA.REL, aperto come #1.

Nota:

- Le unità disco 1541/1581 del VIC-20 e del C-64 dispongono dell'istruzione diretta PRINT(#15,"P".....) per posizionarsi sul record desiderato ("P"=POSITION) dalla sintassi piuttosto contorta. Una procedure equivalente all'istruzione SEEK da utilizzare in **ERRE-C64 3.2** può essere la seguente:

```
procedure seek(nf,re)
    rh=int(re/256) rl=re-256*rh
    print(#15,"p"+chr$(96+nf)+chr$(rl)+chr$(rh)+chr$(1))
    checkdisk
end procedure
```

dove **nf** è il numero file e **re** il record su cui ci si vuole posizionare.

SGN(

Funzione

Scopo:
Calcola la funzione "segno" per un valore numerico

Sintassi:
SGN(espr)

vale 0 se espr=0, 1 se espr>0 e -1 se espr<0

Esempio:

```
PRINT SGN(-5)
```

stamperà

```
-1
```

perché -5 è negativo.

SIN(

Funzione

Scopo:
Calcola il seno di una espressione numerica (in radianti).

Sintassi:
SIN(espr)

Esempio:

```
PRINT(SIN(π))
```

stampa

```
-1
```

Nota:

- Per calcolare il seno di un angolo in gradi sessagesimali, moltiplicare l'argomento per "180/π"

SPC(

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

Funzione

Scopo:
Stampa un numero specificato di spazi all'interno di una istruzione PRINT.

Sintassi:
SPC(espr)

dove *espr* è una espressione numerica compresa tra e 255.

Esempio:

```
PRINT("ABC";SPC(5);"CDEF")
```

stampa

```
ABC.....CDEF
   ▲
```

5 spazi inseriti da SPC

Nota:

- **SPC** è strettamente collegata alla funzione **TAB**: la prima lavora in modalità relativa (gli spazi vengono stampati a partire dalla posizione corrente) mentre la seconda in modalità assoluta (si posiziona direttamente sulla colonna di stampa desiderata).
- Analoga alla **SPC** ma usabile con qualsiasi istruzione è la funzione **STRING$**.

SQR(

Funzione

Scopo:
Calcola la radice quadrata del valore di una espressione numerica.

Sintassi:
SQR(espr)

dove *espr* è una espressione numerica

Esempio:

```
PRINT SQR(16)
```

stampa

```
4
```

Nota:

- Da errore se il valore di *espr* è < 0.
- Per ottenere la radice cubica, una funzione utile può essere:

```
FUNCTION CUBRT(X)
   CUBRT=SGN(X)*ABS(X)^(1/3)
END FUNCTION
```

STATUS

ERRE-VIC20 1.3 ERRE-C64 3.2

Variabile riservata

Scopo:
Segnala un errore di esecuzione (viene aggiornato dalla routine di controllo errori di 64VMS) o un errore di I/O.

Sintassi:
STATUS

Esempio:
Questo esempio visualizza un file sequenziale sullo schermo: **STATUS** controlla la fine del file.

```
program lettura
   begin
     input(file$)                ! chiede il nome del file
     open(2,8,2,file$+",seq,read")  ! apre un file (#2),  se-
                                    ! quenziale in lettura
                                    ! (,seq,read) dal device
                                    ! num. 8 (unità disco)
     repeat
       get(#2,ch$)               ! legge un carattere ....
       if not status
         then print(ch$;)        ! ... lo stampa ...
       end if
     until status                ! ... fine a fine file
     print
     close(2)                    ! chiude il file
end program
```

Nota:

- **STATUS** è il valore della locazione 144. Può assumere i seguenti valori significativi:

Bit	Valore	Unità nastro Datasette	Bus Seriale (dischi)	RS-232
0	1	-	In caso di time-out: =0 in lettura =1 in scrittura.	Errore di parità
1	2	-	Errore di time-out	Errore di frame
2	4	Blocco troppo corto (< 192 bytes)	-	Ricevuto buffer sovraccarico
3	8	Blocco troppo lungo (>192 bytes)	-	Ricevuto buffer vuoto
4	16	Errore di VERIFY	Errore di VERIFY	Manca CTS
5	32	Errore di checksum	-	-
6	64	Fine file (in lettura)	Fine file (in lettura)	Manca RTS
7	128	-	Dispositivo non presente	Rilevato BREAK

STEP

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

Identificatore riservato

Scopo:
Specifica il passo (cioè il valore dell'incremento della variabile di controllo) per un ciclo a conteggio.

Vedi: **FOR**

Nota:

- Se **STEP** non è presente si assume un passo unitario.
- Se è il passo è esplicitamente pari a 0, il compilatore segnala errore (**ERRE-C64 3.2 ERRE-PC 3.0**)

STR$(

Funzione

Scopo:
Converte una espressione numerica in una stringa.

Sintassi:
STR$(espr)

dove *espr* è una espressione numerica

Esempio:

```
PRINT(STR$(2*3))
```

stamperà

```
 6
```

Nota:

- **STR$** riserva sempre il primo carattere della stringa per il segno dell'espressione numerica: " " se positiva, "-" se negativa.
- **VAL** è la funzione inversa di **STR$**. La conversione è da stringa a numero: la valutazione si ferma al primo carattere non numerico. Per una valutazione completa si usa invece la funzione EVAL.

STRING$(

Funzione

Scopo:
Restituisce una stringa formata da un certo numero di elementi del primo carattere di un'altra stringa.

Sintassi:
STRING$(espr,espr$)

Esempio:

```
.....
STRING$(5,"AB")
.....
```

stampa

"AAAAA".

Nota:

- *espr* al massimo può valere 255 (massima lunghezza ammissibile per una stringa)

SWAP(

ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Scambia tra di loro i contenuti di due variabili (semplici o elementi di variabile aggregate) dello stesso tipo.

Sintassi:
SWAP(var1,var2)

Esempio:

```
.....
A=3
B=5
SWAP(A,B)
PRINT(A,B)
......
```

stampa

```
 5         3
```

Nota:

- In **ERRE-VIC20 1.3** bisogna usare una variabile temporanea (di tipo opportuno). L'esempio precedente diventa:

```
A=3
B=5
TEMP=B   \
B=A          |  swap effettivo
A=TEMP   /
PRINT(A,B)
```

con lo stesso risultato di prima.

=SWITCH(

Funzione speciale

Scopo:
Consente di effettuare un assegnamento scegliendo tra una serie di coppie (condizione,valore): la prima coppia che verifica la propria condizione assegna il suo valore alla variabile risultato.

Sintassi:
var=SWITCH(condizione1,valore1,[condizioneN, valoreN])

Esempio:

```
MAX=SWITCH(A>=B,A,A<B,B)
 ▲          ▲   ▲  ▲  ▲
 |          |   |  └──┘ coppie formate da "condizione,valore": la prima
variabile   └───┴── condizione che è TRUE assegna il proprio valore alla
assegnam.         variabile di assegnamento. Se nessuna è TRUE non c'è
                  assegnamento.
```

Se A=17 e B=11, MAX vale A e cioè 17.

Nota:

- SWITCH è equivalente a:

```
IF A>=B THEN
    MAX=A
 ELSIF A<B THEN
    MAX=B
 ELSE
    !$NULL
END IF
```

Come si vede, se nessuna condizione si verifica l'assegnamento non viene effettuato.

TAB(

Funzione

Scopo:
Si posiziona sulla colonna definita dall'argomento in una istruzione **PRINT(**.

Sintassi:
TAB(espr)

Esempio:

```
.....
PRINT(TAB(20);"TEST")
.....
```

stampa la stringa TEST sulla ventesima colonna.

Nota:

- *espr* può andare da 0 a 255. Il posizionamento avviene solo se il numero di colonna attuale (ricavabile tramite la funzione **POS**) è inferiore a quanto richiesto dall'argomento della **TAB(**.
- **TAB(** lavora in modo assoluto, contrariamente alla funzione **SPC(** che è di tipo relativo.

TAN(

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

Funzione

Scopo:
Calcola la tangente di un angolo in radianti

Sintassi:
TAN(espr)

dove **espr** è una espressione numerica.

Esempio:

```
PRINT(TAN(π/4))
```

stampa

```
1
```

Nota:

- **TAN** può fornire risultati diversi rispetto alla sua omonima matematica nei pressi degli asintoti. La funzione non esiste per tutti i multipli dispari di π/2, così mentre, ad esempio, TAN(±π/2) su **ERRE-VIC20 1.3** e **ERRE-C64 3.2** dà correttamente un errore, TAN(±3*π/2) no. In **ERRE-PC 3.0** **TAN** non fornisce mai errore, bensì risultati molto grandi in valore assoluto. In conclusione, i risultati forniti da **TAN** nei pressi dei punti critici vanno valutati con attenzione.
- Per calcolare la tangente di un angolo in gradi sessagesimali, moltiplicare l'argomento per "180/π"

THEN

Identificatore riservato

Scopo:
Inizia il ramo "true" di una istruzione **IF**.

Vedi **IF**

TIME

Variabile riservata

Scopo:
Restituisce il numero di sessantesimi di secondo ("jiffies") trascorsi dalla mezzanotte o dalla reimpostazione del sistema.

Sintassi:
var=TIMER

Esempio:

```
PRINT(TIME)

 70218
```

Nota:

- Il valore massimo per **TIME** è 5184000 (24 ore x 60 minuti x 60 secondi x 60 sessantesimi).
- **TIME** può essere ricavato come 65536*PEEK(160) + 256*PEEK(161) + PEEK(162)
- Il valore di **TIME** non è accurato dopo un I/O da nastro.

TIME$

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

Variabile riservata

Scopo:
Permette di impostare o leggere l'ora.

Sintassi:
TIME$=espr (per impostare)

oppure

var=TIME$ (per leggere)

Il formato in **ERRE-PC 3.0** è *HH:MM:SS*, mentre in **ERRE-VIC20 1.3** e **ERRE-C64 3.2** è *HHMMSS* dove HH è compresa tra 0 e 23, MM tra 00 e 60 e SS tra 00 e 60.

Esempio:

```
PRINT(TIME$)
```

19:19:05 **ERRE-PC 3.0**

o

191905 **ERRE-VIC20 1.3** e **ERRE-C64 3.2**

stampa l'ora corrente.

Nota:

- Se l'elaboratore non possiede una scheda orologio in tempo reale, all'accensione **TIME$** varrà "00:00:00" o "000000".

TIMER

Variabile riservata

Scopo:
Restituisce il numero di secondi trascorsi dalla mezzanotte o dalla reimpostazione del sistema.

Sintassi:
var=TIMER

Esempio:

```
PRINT(TIMER)

 70218.93
```

Nota:

- Come si vede dall'esempio vengono considerati anche i centesimi di secondo.
- Il valore massimo che può assumere **TIMER** è 86400 (24 ore x 60 minuti x 60 secondi), dato che a mezzanotte la variabile si azzera.

TO

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

Identificatore riservato

Scopo:

- Specifica il valore finale per la variabile di un ciclo a conteggio (**FOR**).
- Costituisce l'elemento separatore in una istruzione **CHANGE**. (solo **ERRE-PC 3.0**)

Vedi: **FOR**
CHANGE

TRUE

Costante predefinita

Scopo:
Costante booleana cui viene attribuito convenzionalmente il valore **-1**.

Sintassi:
TRUE

Esempio:

```
X=(Y>2)
PRINT(X)
```

se Y vale 100, verrà stampato -1 (**TRUE**), perché l'espressione (Y>2) è vera.

TRUNC$(

Funzione

Scopo:
Toglie caratteri finali della stringa secondo quando indicato dal secondo argomento.

Sintassi:
TRUNC$(espr$,espr)

Esempio:

```
.....
PRINT(TRUNC$("PLUTO",2))
.....
```

stampa

```
PLU
```

cioè gli ultimi 2 caratteri finali della stringa "PLUTO" sono stati tolti.

Nota:

- Un equivalente per **ERRE-VIC20 1.3** **ERRE-C64 3.2** è
  LEFT$(espr$,LEN(espr$)-espr)
- Se il valore di *espr* è nullo viene fornita la stringa stessa.
- Se il valore di *espr* supera la lunghezza di espr$ viene fornita la stringa vuota "".

TYPE

Istruzione

Scopo:
Permette di definire dei nuovi tipi di variabile, essenzialmente strutture per record o formati per file ad accesso diretto.

Sintassi:

TYPE nome_campo IS (lista_campi)

Esempio:

```
TYPE CUSTOMER IS (NOME$,INDIRIZZO$,NUMTEL$)
```

definisce il tipo "CUSTOMER" composto da tre campi stringa ("NOME$","INDIRIZZO$" e "NUMTEL$"). Tramite una DIM successiva ci si potrà riferire a "CUSTOMER"

```
DIM WORK:CUSTOMER,ELENCO[10]:CUSTOMER
```

creando così il record semplice WORK e l'array ELENCO composto di undici record.

Nota:

- "lista_campi" può comprendere sia campi numerici che campi stringa.
- In **ERRE-PC 2.6** la sintassi dell'istruzione è **TYPE nome_campo=(lista_campi)**

UBOUND(

ERRE-C64 3.2 ERRE-PC 3.0

Funzione

Scopo:
Fornisce l'estremo superiore di una qualsiasi delle dimensioni di un array.

Sintassi:
UBOUND(var_array,espr)

Esempio:

```
.....
DIM A[5,10]
PRINT(UBOUND(A,2)
.....
```

stampa

```
10
```

l'indice massimo della seconda dimensione.

Nota:

- In **ERRE-VIC20 1.3** non è stata implementata.
- Se *espr* è uguale a 0 viene fornito il numero di dimensioni dell'array.
- Se *espr* è maggiore di zero, per un array dinamico **UBOUND** fornisce sempre zero.

UNTIL

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Ultima istruzione di un ciclo REPEAT, dove viene effettuato il test sulla condizione finale.

Vedi: **REPEAT**

USES

ERRE-C64 3.2 **ERRE-PC 3.0**

Istruzione

Scopo:
Segnala al compilatore di includere la "unit" che segue. Una "unit" è un insieme di funzioni o procedure organizzate in modo tematico.

Sintassi:
USES nome_unit

Esempio:

USES CRT **ERRE-PC 3.0**

USES CURSOR **ERRE-C64 3.2**

Entrambe includono, sulla propria piattaforma, le procedure per il controllo dello schermo in modo testo.

Nota:

- La unit **SYSTEM.LIB** si intende automaticamente inclusa.
- Per una descrizione delle "unit" vedi la singola pagina descrittiva.
- **USES** sostituisce la direttiva **$INCLUDE** delle precedenti versioni di ERRE che aveva lo stesso scopo.

USR(

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

Funzione

Scopo:
Cede il controllo ad una routine in linguaggio macchina e restituisce un valore utilizzabile nell'ambito di una espressione.

Sintassi:
USR(espr)

Esempio:

```
A=USR(1)*3
```

la routine in linguaggio macchina fornisce un valore che, moltiplicato per 3, viene memorizzato nella variabile A.

Nota:

- Su PC è stata mantenuta per compatibilità con **ERRE-VIC20 1.3** e **ERRE-C64 3.2.**
- Prima di poterla utilizzare va definito il punto della routine in linguaggio macchina: si usa a tale scopo la direttiva di compilazione **$USERPTR.** Per **ERRE-VIC20 1.3** vanno riempite invece direttamente le tre locazioni 0 (=76 fisso),1 (parte bassa indirizzo) e 2 (parte alta indirizzo).

VAL(

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

Funzione

Scopo:
Converte una stringa in un numero.

Sintassi:
VAL(espr$)

Esempio:

```
.....
PRINT(VAL("1.2E-3"))
.....
```

stampa

```
 .0012
```

Nota:

- La valutazione della stringa si ferma appena viene trovato un carattere non pertinente ad una costante numerica. I caratteri accettati sono quelli numerici, il punto, **+**, **-** ed il segno di esponente (**E** o **D**). Quindi

  ```
  VAL("12AB")
  ```

  fornisce

  ```
  12
  ```

- La funzione **VAL** è una forma ridotta della funzione **EVAL** (non implementata nel linguaggio ERRE) che valuta completamente una stringa: VAL("12*15+1") fornisce 12, mentre EVAL("12*15+1") fornisce 181. **ERRE-C64 3.2** può utilizzare a tale scopo la routine EVAL di 64VMS, mentre **ERRE-PC 3.0** può ricorrere a routine apposite richiamate tramite un comando **EXEC**.

Istruzione

Scopo:
Definisce una procedure o una function come VARIANT, ovvero una parte di codice che è indipendente dal tipo di variabile trattato.

Sintassi:

- **VARIANT PROCEDURE nome_proc[(param.ingresso->param.uscita)]**
- **VARIANT FUNCTION nome?(param.ingresso)**

dove *param.ingresso* e *param.uscita* possono contenere variabili di tipo generico, prefissate cioè con l'identificatore di tipo '?'.

Esempio:
La seguente procedure ordina un vettore (numerico o stringa) di al massimo 100 elementi:

```
DIM A?[100]

VARIANT PROCEDURE SORT(A?[],N%)
   LOCAL I%,FLIPS%
   FLIPS%=TRUE
   WHILE FLIPS% DO
     FLIPS%=FALSE
     FOR I%=1 TO N%-1 DO
       IF A?[I%]>A?[I%+1] THEN
          SWAP(A?[I%],A?[I%+1])
          FLIPS%=TRUE
       END IF
     END FOR
   END WHILE
END PROCEDURE
```

Una chiamata a questa procedure può essere

```
SORT(A[],50)
```
o
```
SORT$(A$[],100)
```

La seguente function calcola invece il seno iperbolico dell'argomento.

```
VARIANT FUNCTION SINH?(X?)
    SINH?=(EXP(X?)-EXP(-X?))/2
END FUNCTION
```

Una chiamata a questa funzione può essere

```
PRINT(SINH(0.5))
```
in singola precisione

oppure

```
PRINT(SINH#(0.5))
```
in doppia precisione

Nota:

- Una *variant procedure* non può richiamare un'altra *variant procedure*.
- Le *variant function* possono essere solo di tipo numerico.

VARPTR(

ERRE-C64 3.2 **ERRE-PC 3.0**

Funzione

Scopo:
Fornisce l'indirizzo di memoria di una variabile o l'indirizzo del "File Control Block" (F.C.B.) di un file.

Sintassi:
VARPTR(X) dove X è una variabile qualsiasi.

VARPTR(#N%) dove N% è un numero di file aperto.

Esempio:

```
Y=VARPTR(X%)
```

assegna a Y il valore dell'indirizzo di memoria della variabile intera X%.

Nota:

- Su **ERRE-C64 3.2** **VARPTR** è utilizzabile solo in forma di assegnamento IND=VARPTR(VAR) e non può essere applicata ad un file.
- E' l'equivalente funzionale dell'operatore **&**.

VARPTR$(

Funzione

Scopo:
Restituisce l'indirizzo della variabile fornita come argomento sotto forma di una stringa di tre caratteri. Il primo carattere identifica il tipo della variabile, il secondo e il terzo, in forma codificata, il suo indirizzo.

Sintassi:
VARPTR$(X) dove X è una variabile qualsiasi.

Esempio:

```
Y$=VARPTR$(X%)
```

assegna a Y$ il valore codificato del tipo e dell'indirizzo di memoria della variabile intera X%.

Nota:

- Nella forma ASC(LEFT$(VARPTR$(X),1)) è l'equivalente della funzione *sizeof* del linguaggio C.

VERSION$

ERRE-C64 3.2 ERRE-PC 3.0

Costante predefinita

Scopo:
Identifica la versione in uso del linguaggio ERRE. E' strettamente collegata alle direttive di "compilazione condizionale" **!$IF** condizione **$THEN** .... **$ELSE** ..... **$ENDIF**.

Sintassi:
VERSION$

Esempio:

```
PRINT VERSION$
```

stampa 32 su C-64 e 30 su PC-IBM

Nota:

- **ERRE-VIC20 1.3** non dispone di questa costante predefinita.
- Può assumere anche il valore 26 su **ERRE-PC 2.6**.

WHILE... END WHILE

ERRE-VIC20 1.3 ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Implementa un ciclo a condizione iniziale.

Sintassi:

```
WHILE espressione DO
    ┌─────────────┐
    │ B l o c c o │
    └─────────────┘
END WHILE
```

Blocco viene ripetuto fintantoché *espressione* resta TRUE.

Esempio:

```
!BUBBLE SORT SU ARRAY A$[]
FLIPS%=1
WHILE FLIPS% DO
  FLIPS=0
  FOR N=1 TO J-1 DO
    IF A$[N]>A$[N+1] THEN
      SWAP(A$[N],A$[N+1])
      FLIPS=1
    END IF
  END FOR
END WHILE
```

implementa una routine di ordinamento ("bubble sort") sui primi J elementi dell'array A$.

Note:

- Per **ERRE-VIC20 1.3** non va indicato **DO** ed **END WHILE** è scritto tutto unito (abbreviabile in **ENDW**.)

WITH...END WITH

Istruzione

Scopo:
Permette di indicare, nel blocco **WITH..END WITH**, una variabile di tipo record con il solo nome di campo preceduto da ".".

Sintassi:

```
WITH record_var DO
     ┌─────────────┐
     │ B l o c c o │
     └─────────────┘
END WITH
```

Esempio:

```
.....
WITH CUSTOMER DO
      PRINT(.NOME$,.INDIRIZZO$,.CITTA$)
END WITH
.....
```

La forma estesa sarebbe stata la seguente:

```
PRINT(CUSTOMER.NOME$,CUSTOMER.INDIRIZZO$,CUSTOMER.CITTA$)
```

WRITE(

ERRE-C64 3.2 ERRE-PC 3.0

Istruzione

Scopo:
Permette la formattazione, secondo un formato indicato, di una qualsiasi espressione.

Sintassi:
WRITE([#numero_file,]formato,lista_espr) (**ERRE-PC 3.0**)
WRITE(var,lungh_campo,num_dec) (**ERRE-C64 3.2**)

- **ERRE-PC 3.0**

Per specificare il formato si usano i seguenti simboli ("metacaratteri"):

 # . , ^ * $ + - \ & ! _

Gli altri caratteri verranno stampati così come sono. I formati possibili sono di due tipi:

- numerici: si indica con '#' una cifra da stampare, con '.' la posizione del punto decimale e con '^^^^' l'eventuale esponente. Sono previste possibilità aggiuntive usando gli altri metacaratteri: con ',.' si ottiene la separazione delle migliaia, con '**' si riempie con asterischi la parte iniziale, con '$$' iniziale viene stampato il simbolo del $ all'inizio del numero, con '**$' si combinano gli ultimi due effetti, con '**+**' si può specificare la posizione della stampa del segno algebrico (all'inizio o alla fine del numero), idem con '-' ma solo per i numeri negativi.

- stringa : si indica con "\" + (n-2) spazi + "\" la lunghezza della stringa da stampare. Se si indica '&' viene stampata l'intera stringa, con '!' si stampa solo il primo carattere.

Esempio:

Se A=5.443E-1, allora

```
WRITE("##.####^^^^";A)
```

stamperà

```
5.4430E-01
```

Se A$="PIPPOPLUTO", allora

```
WRITE("\····\";A$)
          ▲
       4 spazi
```

stamperà

PIPPOP (6 caratteri)

Se A$="NUM:", A=5.443 e B=18.4768

```
WRITE("\ \##.####  ###.## ;A$;A;B)
```

stamperà

```
NUM=  5.4430   18.48
```

Note:

- Se un valore non rispetta il formato numerico impostato, viene visualizzato con un '**%**' all'inizio.
- In tutti i casi se i metacaratteri di cui sopra sono preceduti da '**_**' la loro funzione viene annullata.
- "formato" deve concordare in tipo con i componenti di "lista_espressioni".

● **ERRE-C64 3.2**

Possono solo essere formattate variabili numeriche, specificando la lunghezza totale del campo e il numero di decimali richiesti. Non possono essere formattati direttamente numeri in formato esponenziale.

Se A=0.544311, allora

```
WRITE(A,10,4)
```

stamperà

```
      0.5443
. . . . . .
```

10 posizioni

Note:

- WRITE è *incompatibile* con le routine HGR di 64VMS.

XOR

ERRE-C64 3.2 **ERRE-PC 3.0**

Operatore

Scopo:
Permette l'OR esclusivo all'interno di espressioni condizionali.

Sintassi:
XOR

Esempio:

```
Y%= Y% XOR 255          ERRE-PC 3.0

Y%=XOR(Y%,255)          ERRE-C64 3.2
```

questo assegnamento permette il cosiddetto "bit-flipping" della variabile Y%: se un bit è vero diventa falso e viceversa.

Nota:

- **XOR** può essere usato anche per operare sui singoli bit di una variabile e/o costante di tipo integer (cioè su 16 bit). Ad esempio

  14 XOR 255 = 241 — 4=%00001101 e 255=%11111111, perciò applicando la tabella dell'operatore XOR otteniamo %11110010 e quindi 241 in decimale

- In **ERRE-C64 3.2** è disponibile solo sotto forma funzionale nella forma **XOR(X,Y)**.

Si ricorda comunque che l'operatore XOR può essere reso anche tramite i tre operatori fondamentali:

A XOR B ==> (A AND NOT B) OR (B AND NOT A)

Questo è esattamente il caso per **ERRE-VIC20 1.3** che non dispone di questo operatore.

Operatore	Valore A	Valore B	Risultato
XOR	V	V	F
	V	F	V
	F	V	V
	F	F	F

Istruzione

La versione 1.3 del linguaggio ERRE su VIC-20 può incorporare alcune istruzioni tipiche del BASIC 2.0 Commodore al cui manuale si rimanda. Si ricorda che istruzioni non proprie di ERRE possono essere messe sulla stessa riga separate da "**:**".

CLR

Scopo: Cancella tutte le variabili, semplici ed aggregate, di un programma. L'equivalente C-64 e PC ha uno scopo completamente diverso.

Sintassi:
CLR

Equivalente (**ERRE-C64 3.2** **ERRE-PC 3.0**): **CLEAR**

CMD

Scopo: Consente di redirigere lo standard output verso un file.

Sintassi:
CMD num_file,"messaggio"

Equivalente (**ERRE-C64 3.2** **ERRE-PC 3.0**): **$REDIR**

RUN

Scopo:
Consente di riavviare l'esecuzione di un programma.

Sintassi:
RUN

Equivalente (**ERRE-C64 3.2** **ERRE-PC 3.0**): **!$EXECUTE**

STOP

Scopo:
Consente di bloccare temporaneamente l'elaborazione del programma che può essere ripresa con l'istruzione R-Code 'CONT'.

Sintassi:

STOP

Equivalente (**ERRE-C64 3.2 ERRE-PC 3.0**): **!$STOP**

SYS

Scopo:
Fa partire un sottoprogramma scritto nel linguaggio macchina del microprocessore "target" a partire dalla locazione indicata dal valore di "var1" passando come parametri le variabili indicate nella "lista_var" (se esistono).

Sintassi:
SYS var1[,lista_var]

dove var1 è una variabile numerica intera (compresa tra 0 65535).

Equivalente (**ERRE-C64 3.2 ERRE-PC 3.0**): **CALL(var1[,lista_var])**

WAIT

Scopo: Blocca l'elaborazione finché il valore di una locazione di memoria non soddisfa certi requisiti.

Sintassi
WAIT parametri

Nota: **parametri** è: **indirizzo, mask[,invert]**
Blocca l'elaborazione fintantoché (PEEK(*indirizzo*) XOR *invert* AND *mask*)<>0. Se *invert* manca lo si assume uguale a 0

Equivalente (**ERRE-C64 3.2 ERRE-PC 3.0**): **!$RCODE="WAIT** parametri**"**

!

ERRE-VIC20 1.3 **ERRE-C64 3.2** **ERRE-PC 3.0**

Simbolo speciale

Scopo:
Segnala l'inizio di un commento o di una sequenza speciale.

Esempio:

```
PROGRAM REMARK

!$ ..................  ◄--- direttiva di compilazione

BEGIN

! ...................  ◄--- commento "monolinea"

!!                     ◄--- inizio commento "multilinea"
.....................
.....................
!!                     ◄--- fine commento "multilinea"

END PROGRAM
```

Note:

- Un commento "monolinea" termina a fine riga.
- La sequenza '**!$**' indica una direttiva di compilazione (solo per **ERRE-C64 3.2** **ERRE-PC 3.0**)
- La sequenza **'!!'** indica l'inizio di un commento '*multilinea*': tutta la parte di codice fino al **'!!'** seguente è considerata commento. Questa caratteristica è utile per escludere dalla compilazione parti di codice utilizzate a scopo di debug, senza rimuoverle fisicamente (solo per **ERRE-PC 3.0**).
- Poiché **ERRE-VIC20 1.3** non accetta righe vuote è possibile usare **!** per simularne la presenza.

!$IF condizione $THEN .... $ELSE .... $ENDIF

Direttiva

Scopo:
Permettono la "compilazione condizionale" consentendo di inserire parti di codice ERRE specifiche di una certa piattaforma o di una certa versione del linguaggio: al momento "condizione" può essere testata sulle due variabili di sistema **MACHINE$** e **VERSION$** che possono assumere rispettivamente i valori "CBM64" e "PCIBM" oppure "30" e "26".

Esempio:
Per cancellare lo schermo si potrà scrivere:

```
.....
   !$IF MACHINE$="CBM64" $THEN
     PRINT(CHR$(147);)
     A$="SOLO PER IL C-64"
   !$ELSE
     PRINT(CHR$(12);)
     A$="SOLO PER IL PC"
   !$ENDIF
.....
```

} righe sostituite con istruzioni $NULL

} parte effettivamente compilata

Come indicato, il compilatore (per PC) processerà solo la parte di codice compresa tra **!$ELSE** ed **!$ENDIF**.

!$ABORT

Direttiva

Scopo:
Abortisce l'esecuzione di un programma ritornando all'ambiente operativo.

Sintassi:
!$ABORT

Esempio:

```
.....
IF NUMBER>100 THEN
  !$ABORT
END IF
......
```

se il valore di NUMBER supera 100 il programma termina (in modo non naturale.....)

!$DIM

Direttiva

Scopo:
Consente di allocare un array dinamico (dichiarato precedentemente con **$DYNAMIC**). Questa direttiva può essere usata solo nel main.

Sintassi:
!$DIM $a_1[d_1,...,dN_1],a_2[d_1,...,dN_2],.....,a_j[d_1,...,dN_j]$

Esempio:

```
.....
!$DYNAMIC       ◄--- dichiara il DIM seguente come array dinamico
DIM Z[<>,<>]    ◄--- i '<>' servono per definire il numero delle dimensioni
                     (in questo caso Z è un'array bidimensionale)
.....
INPUT(N)        ◄--- nel corso dell'esecuzione richiedi N
!$DIM Z[N,N]    ◄--- allocazione effettiva di Z, dipendente da N.
                     Se l'utente imposta per N il valore 15, la matrice sarà
                     un 15x15, se N=20 avremo una 20x20 ecc..
.....
```

Note:

- Questa direttiva risulta utile per allocare solo la memoria effettivamente richiesta per l'elaborazione (ad esempio calcoli matriciali).

!$DIMCOMMON

Direttiva

Scopo:
Usato nella gestione a moduli di un programma ERRE. **$DIMCOMMON** deve essere usato prima di dichiarare con una DIM le variabili aggregate passate tramite una clausola COMMON, pena un errore in fase di esecuzione, solo nel modulo chiamato.

Sintassi:
!$DIMCOMMON

Esempio:

Se abbiamo nel modulo chiamante:

```
COMMON A%,B,C$,D[]
DIM D[10]
```

dovremo usare nel modulo chiamato

```
COMMON A%,B,C$,D[]
!$DIMCOMMON
DIM D[10]
```

perché l'interprete di R-Code considera già implicitamente dimensionata la variabile aggregata D[] a livello di COMMON e quindi la DIM successiva genera un errore. Questo problema non sussiste se nella clausola COMMON si utilizza un nome diverso di variabile: è corretto perciò scrivere nel modulo chiamato

```
COMMON A%,B,C$,E[]
DIM E[10]
```

senza la direttiva !$DIMCOMMON.

Direttiva

Scopo:
Dichiara come reali a doppia precisione tutte le variabili (semplici ed aggregate) e le funzioni del modulo. Una variabile con il proprio identificatore di tipo non viene influenzata da questa direttiva.

Sintassi:
$DOUBLE

Esempio:

```
.....
!$DOUBLE
.....
A=100/33        ! A è in doppia precisione
B%=15           ! B% resta di tipo intero, nonostante la direttiva
```

Note:

- Questa direttiva va messa prima della parte dichiarativa per essere sicuri della sua efficacia.

!$DYNAMIC

ERRE-C64 3.2 ERRE-PC 3.0

Direttiva

Scopo:
Specifica che il DIM seguente dichiara (senza allocare) array dinamici.

Sintassi:
!$DYNAMIC

Esempio:
vedi **!$DIM**

!$ERASE

ERRE-C64 3.2 **ERRE-PC 3.0**

Direttiva

Scopo:
Permette la cancellazione di array di tipo dinamico (dichiarati con **$DYNAMIC**), per poterli poi eventualmente ridimensionare.

Sintassi:
!$ERASE var_array$_1$[,var_array$_N$]

Esempio:
Se ad esempio abbiamo, come nell'esempio precedente

```
.....
!$DYNAMIC
DIM Z[<>,<>]
.....
INPUT(N)
!$DIM Z[N,N]
.....
```

una volta usato l'array Z potremo eliminarlo con un

```
!$ERASE Z
```

e rieffettuare ridimensionare Z con un'altra !$DIM più avanti.

Note:

- Nello specificare il nome di una variabile array non va usata la forma [] come in altri casi.
- In **ERRE-C64 3.2** **$ERASE** cancella tutti gli array.
- In **ERRE-PC 2.6** uno ‘0’ sostituisce ‘<>’ nella dichiarazione di array dinamico.

Direttiva

Scopo:
Permette di simulare degli errori di esecuzione "standard" o di definirne di propri, gestendoli poi in maniera opportuna tramite l'abituale EXCEPTION...END EXCEPTION

Sintassi:
!$ERROR=nnn

Esempio:

```
PROGRAM MY_ERROR

 EXCEPTION
   IF ERR=200 THEN PRINT("Troppo alto!") END IF  ! errore personalizzato
 END EXCEPTION                                   ! num. 200

 BEGIN
   PRINT(CHR$(12);)
   LOOP
    INPUT("Imposta numero minore di 1000 (0=esci)",A%)
    EXIT IF A%=0
    IF A%>=1000 THEN
      !$ERROR=200
     ELSE
      PRINT(A%)
    END IF
  END LOOP
END PROGRAM
```

Nota:

- nnn può essere al massimo 255: poiché i numeri di errore che vanno da 1 a 76 sono quelli degli errori standard, si consiglia di usare valori più alti per nnn.

!$EXCEPTION

Direttiva

Scopo:
Segna l'inizio di una routine di gestione delle eccezioni locale ad una procedura.

Sintassi:
!$EXCEPTION

Esempio:
Vedi **!$LOCAL EXCEPTION**

!$EXECUTE

Direttiva

Scopo:
Permette di rieseguire un programma come se fosse stato appena caricato.

Sintassi:
!$EXECUTE

Esempio:

```
.....
INPUT("Giochi ancora";ANS$)
IF ANS$="S" THEN
    !$EXECUTE
END IF
.....
```

Nota:

- Non va usato in una gestione a moduli di un programma ERRE perché potrebbe cancellare elementi importanti di memoria (ad es. variabili comuni).

!$FREE

Direttiva

Scopo:
Effettua il "*garbage collection*" delle variabili di tipo stringa per ottimizzare la memoria disponibile. Questa operazione può richiedere un certo periodo di tempo e non è interrompibile.

Sintassi:
!$FREE

Esempio:

```
.....
!$FREE
.....
```

Note:

- **FREEMEM** (o la funzione **FRE(X)** per **ERRE-VIC 1.3**), che fornisce la memoria disponibile, effettua la stessa operazione.

!$HALT

Direttiva

Scopo:
Abortisce l'esecuzione di un programma ritornando all'ambiente operativo e restituendo un codice numerico di uscita ("exit code").

Sintassi:
!$HALT[=nnn]

Esempio:

```
.....
IF NUMBER>100 THEN
  !$HALT=100
END IF
......
```

se il valore di NUMER supera 100 termina il programma restituendo 100 come "exit code".

Note:

- **nnn** può andare da 0 a 255: 0 è il codice di uscita predefinto.
- Il valore di **nnn** è memorizzato nella cosiddetta "*Inter Application Communication Area*" all'indirizzo assoluto 04FF:0000. Tale zona, di 16 byte, viene usata anche dall'ambiente di programmazione per scopi diversi.

!$HIMEM

Direttiva

Scopo:
Fissa l'indirizzo più alto per la memorizzazione dei dati utilizzati da un programma ERRE, tramite un numero di pagina.

Sintassi:
!$HIMEM=nnn

Esempio:

```
.....
!$HIMEM=159
.....
```

fissa l'indirizzo più alto per la memorizzazione dei dati di un programma ERRE alla locazione 40704 (159*256). Quindi i 256 bytes allocati tra 40705 e 40959 sono disponibili per caricare, ad esempio, routine in linguaggio macchina.

!$IMPLICIT

Direttiva

Scopo:
Consente di utilizzare le caratteristiche estese del letterale π (vedi esempio).

Sintassi:
!$IMPLICIT

Esempio:

```
PROGRAM AREA

!$IMPLICIT                          ◄------ attiva le caratteristiche estese del letterale π
DIM R,S

BEGIN
   INPUT("Raggio ?",R)
   PRINT("Circonferenza=";2πR)  ◄------ moltiplicazione implicita per una costante numerica
   PRINT(π')                    ◄------ stampa π in singola precisione (3.141593)

!$NOIMPLICIT                        ◄------ disattiva le caratteristiche estese del letterale π
   S=π'                       -----► errore di compilazione per π' a causa della direttiva precedente.
END PROGRAM
```

Questo esempio è stato tratto dalla distribuzione ($IMPLICT.R)

!$INTEGER

Direttiva

Scopo:
Dichiara come intere tutte le variabili (semplici ed aggregate) e le funzioni del modulo. Una variabile con il proprio identificatore di tipo non viene influenzata da questa direttiva.

Sintassi:
!$INTEGER

Esempio:

```
.....
!$INTEGER
.....
A=100/33            ! A è intera e vale 3
B$="PIPPO"          ! B$ resta di tipo stringa, nonostante la direttiva
```

Note:

- Questa direttiva va messa prima della parte dichiarativa per essere sicuri della sua efficacia.

!$INTERFACE

Direttiva

Scopo:
Dichiara i metodi utilizzati in una classe permettendone così la visibilità all'esterno: la conoscenza delle dichiarazioni $INTERFACE consente l'utilizzo corretto della classe in un programma.

Sintassi:
!$INTERFACE metodo

Esempio:
Vedi esempio istruzione **CLASS.**

!$LOCAL EXCEPTION

Direttiva

Scopo:
Dichiara una routine di gestione delle eccezioni locale ad una procedura.

Sintassi:
!$LOCAL EXCEPTION

Esempio:

```
PROGRAM LOCAL_ERROR

DIM A[10]

PROCEDURE ASSIGN
LOCAL I%
!$LOCAL EXCEPTION                    (dichiarazione)
  LOOP
    INPUT("Indice (max. 10)";I%)
    EXIT IF I%<0
    A[I%]=-5
  END LOOP
  EXIT PROCEDURE
  !$EXCEPTION
  PRINT("INDICE ERRATO")             (corpo della routine di gestione)
  RESUME NEXT
END PROCEDURE

BEGIN
   ASSIGN
   FOR I%=LBOUND(A,1) TO UBOUND(A,1) DO
      PRINT(A[I%];)
   END FOR
   PRINT
END PROGRAM
```

In questo esempio viene dichiarata all'interno della procedure ASSIGN una routine di gestione delle eccezioni, collocata al termine della procedura stessa. L'istruzione **EXIT PROCEDURE** viene messa per separare il corpo della procedure dalla routine di gestione delle eccezioni.

!$LOCK .... !$UNLOCK

Direttiva

Scopo:
Permettono di gestire risorse condivise (file) in un ambiente distribuito, rispettivamente bloccando e sbloccando file (o record di file).

Sintassi:
$LOCK|UNLOCK #numfile[,rec_iniziale..rec..finale]

Esempio:

```
!$LOCK #1,10
```

blocca il record 10 del file aperto come numero 1 (ad esempio per scrivere informazioni)

```
!$UNLOCK #1,10
```

sblocca il record 10 e rendilo disponibile per la modifica anche da parte degli gli altri processi

Nota:

- I parametri di $LOCK ed $UNLOCK devono essere uguali, altrimenti si verifica un errore durante l'esecuzione.

!$MATRIX

Direttiva

Scopo:
Abilita le operazioni estese sulle variabili aggregate (array).

Sintassi:
!$MATRIX

Esempio:

```
.....
!$MATRIX
.....
```

Note:

- Le operazioni possibili (con i relativi tipi di variabile) sono le seguenti:

Operazione	Tipi ammessi	Possibile senza !$MATRIX
A[]=k	Numerico e Stringa	no
A[]=B[]	Numerico e Stringa	si
A[]=B[]+C[]	Numerico e Stringa	no
A[]=B[]-C[]	Numerico	no
A[]=k*B[] o B[]*k	Numerico	no

- La versione **ERRE-C64 3.2** non dispone di questa direttiva e quindi l'unica operazione possibile è l'assegnamento tra variabili aggregate.

!$NOEXCEPTION

Direttiva

Scopo:
Disabilita gli effetti di una **EXCEPTION**: il uso provoca la disabilitazione del trapping degli errori e quindi un qualsiasi errore successivo causa la fine forzata del programma.

Sintassi:
!$NOEXCEPTION

Esempio:

```
.....
EXCEPTION
......
!$NOEXCEPTION
END EXCEPTION
```

In questo caso una volta gestito il primo errore, con questa direttiva attivata al secondo errore si esce dal programma e si torna all'ambiente operativo.

!$NOIMPLICIT

Direttiva

Scopo:
Disabilita le caratteristiche estese del letterale π, che è la condizione iniziale assunta dal compilatore.

Sintassi:
!$NOIMPLICIT

Esempio:
Vedi l'esempio riportato sotto **$IMPLICIT**.

!$NOPRESERVE

Direttiva

Scopo:
Azzera i valori degli array oggetto delle istruzioni REDIM seguenti.

Sintassi:
!$NOPRESERVE

Esempio:

```
PROGRAM REDIM_TEST

DIM A[10],B$[5,5]

BEGIN
  FOR I=0 TO 10 DO          ◀------ assegna i valori agli array A[] e B$[]
    A[I]=I*2
  END FOR

  FOR I=0 TO 5 DO
    FOR J=0 TO 5 DO
      B$[I,J]=CHR$(64)
    END FOR
  END FOR

  !$PRESERVE                ◀------ conserva i valori dell'array A[] dopo il ridimensionamento
  REDIM A[20]

  !$NOPRESERVE              ◀------ azzera i valori dell'array B$[] dopo il ridimensionamento
  REDIM B$[10,10]

  FOR I=0 TO 10 DO
    PRINT(A[I];)
  END FOR
  PRINT(::)

  FOR I=0 TO 5 DO
    FOR J=0 TO 5 DO
      PRINT(B$[I,J];)
    END FOR
  END FOR
  PRINT(::"OK")

END PROGRAM
```

!$NULL

ERRE-C64 3.2 **ERRE-PC 3.0**

Direttiva

Scopo:
Implementa l'istruzione **NULL** non prevista dagli standard del linguaggio.

Sintassi:
!$NULL

Esempio:

```
.....
FOR I%=1 TO 100 DO
  !$NULL
END FOR
......
```

è un ciclo di attesa.

Note:

- Un equivalente per **ERRE-VIC20 1.3** può essere il simbolo di commento '!'

!$PRESERVE

Direttiva

Scopo:
Conserva i valori degli array oggetto delle istruzioni REDIM seguenti(nei limiti imposti dal nuovo dimensionamento)

Sintassi:
!$PRESERVE

Esempio:

```
PROGRAM REDIM_TEST

DIM A[10],B$[5,5]

BEGIN
  FOR I=0 TO 10 DO          ◀------ assegna dei valori agli array A[] e B$[]
    A[I]=I*2
  END FOR

  FOR I=0 TO 5 DO
    FOR J=0 TO 5 DO
      B$[I,J]=CHR$(64)
    END FOR
  END FOR

  !$PRESERVE                ◀------ conserva i valori dell'array A[] dopo il ridimensionamento
  REDIM A[20]

  !$NOPRESERVE              ◀------ azzera i valori dell'array B$[] dopo il ridimensionamento
  REDIM B$[10,10]

  FOR I=0 TO 10 DO
    PRINT(A[I];)
  END FOR
  PRINT(::)

  FOR I=0 TO 5 DO
    FOR J=0 TO 5 DO
      PRINT(B$[I,J];)
    END FOR
  END FOR
  PRINT(::"OK")
END PROGRAM
```

!$REDEFINE

ERRE-C64 3.2 ERRE-PC 3.0

Direttiva

Scopo:
Consente di ridefinire (solo dal main) una funzione già dichiarata: da questo momento in poi, nel corso dell'esecuzione si farà riferimento alla nuova dichiarazione.

Sintassi:

!$REDEFINE nome_funzione(var,[,var])=espressione

Esempio:

Se si è dichiarata la funzione

```
.....
FUNCTION P(X)
   P=X*X-2*X+3
END FUNCTION
......
```

nel main posso ridefinire P(X) e scrivere

```
.....
   PRINT(P(0))
   !$REDEFINE P(X)=X*X+5*X-11
   PRINT(P(0))
......
```

ottenendo come risultati

```
 3
-11
```

!$REDIR... !$NOREDIR

Direttiva

Scopo:
!$REDIR abilita l'uscita delle istruzioni **PRINT** e **WRITE** che seguono verso la stampante collegata a LPT1:
!$NOREDIR disabilita una precedente !**$REDIR**.

Sintassi:
!$REDIR
!$NOREDIR

Esempio:

```
.......
PRINT("HELLO WORLD !")
!$REDIR
PRINT("HELLO WORLD !")
!$NOREDIR
PRINT("HELLO WORLD !")
.....
```

fa stampare *HELLO WORLD !* tre volte: la prima e la terza a video e la seconda sulla stampante collegata a LPT1:

Nota:

- In **ERRE-PC 3.0** **$REDIR** sfrutta il terzo "handle" che l'interprete attiva automaticamente all'inizio dell'esecuzione: i primi due sono lo "standard input" e lo "standard output". Da qui si vede che il numero minimo di file gestibili dall'interprete di R-Code è tre che sommati ai tre "handle" automaticamente aperti fanno sei, proprio il numero gestito dal DOS in mancanza di un comando FILES in CONFIG.SYS.

!$REDIR

Direttiva

Scopo:
Consente di redirigere l'istruzione PRINT sul file, preventivamente aperto, identificato da <num.file>.

Sintassi:
!$REDIR <num.file>

Esempio:

```
.......
PRINT("HELLO WORLD !")
OPEN(#1,8,8,"HELLO.SEQ,S,R")
!$REDIR 1
PRINT("HELLO WORLD !")
PRINT (#1)
CLOSE(1)
PRINT("HELLO WORLD !")
.....
```

fa stampare *HELLO WORLD !* tre volte: la prima e la terza volta a video, la seconda volta sul file #1.

Note:

- Il gruppo di istruzioni *PRINT(#num.file) CLOSE(num.file)* termina la redirezione.

!$RCODE

ERRE-C64 3.2 **ERRE-PC 3.0**

Direttiva

Scopo:
Informa il compilatore di considerare la stringa tra apici come scrittura diretta in R-Code, cioè da inserire tale e quale nel codice del programma ERRE compilato.

Sintassi:
!$RCODE="stringa"

Esempio:

```
!$RCODE="BEEP"
```

Nel PC fa suonare lo speaker: senza la direttiva avremmo dichiarare la relativa *unit* con **USES SOUND** e richiamato l'apposita procedure BEEP.

Nota:

- "stringa" non può contenere un numero di riga iniziale.
- E' possibile usare, come separatore, al posto di " un qualsiasi altro carattere (solo in **ERRE-PC 3.0**). L'esempio precedente può essere scritto anche come:
  `!$RCODE='BEEP'` (usando ' come separatore)
- E' anche disponibile la forma abbreviata **!$·** (spazio obbligatorio) cosicché l'esempio precedente diventa:
  `!$ BEEP` (forma abbreviata)
- Un altro modo (solo in **ERRE-PC 3.0**) per inserire direttamente codice R-Code in un programma ERRE è dato dalla dichiarazione di procedure **EXTERNAL**.
- In **ERRE-VIC20 1.3** è possibile inserire direttamente istruzioni R-Code.

!$RETURN

Direttiva

Scopo:
Indirizza l'uscita da una procedure verso l'etichetta <label> anziché verso l'istruzione che segue il richiamo della procedura stessa (come avviene con END PROCEDURE o EXIT PROCEDURE): l'uso di questa direttiva è rivolto essenzialmente a procedure gestite da eventi.

Sintassi:
$RETURN <label>

Esempio:

```
PROGRAM RETURN

LABEL 1000

PROCEDURE TEST
  LOCAL J,I
  PRINT("Sono nella procedure Test")
  J=1
  FOR I=1 TO 10 DO
    J=J*I
  END FOR
  !$RETURN 1000   <-- con questa direttiva si chiude la procedura
                      e si salta alla label 1000. Se la $RETURN
                      non ci fosse,  la chiusura "regolare" con
                      END PROCEDURE farebbe ritornare a

END PROCEDURE

BEGIN
  TEST
  PRINT("Non verrò mai stampato con la $RETURN!")
1000:
  PRINT("Invece io sarò stampato in tutti i casi!")
END PROGRAM
```

Note:

- Indirizzo è numero “unsigned integer” e quindi può andare da 0 a 65535. Può essere espresso anche in base numerica diversa da 10.

!$SEGMENT

Direttiva

Scopo:
Seleziona un segmento di memoria (da 64K) al quale si potrà poi accedere con le istruzioni POKE e CALL e la funzione PEEK.

Sintassi:
!$SEGMENT[=indirizzo]

Esempio:

```
!$SEGMENT=$B800
POKE(0,65)
POKE(1,112)
```

scrive la lettera 'A' nell'angolo in alto a sinistra (in nero su sfondo bianco). $B800 è l'indirizzo di segmento relativo all'area di memoria video colore (CGA/EGA/VGA).

Note:

- Indirizzo è numero "unsigned integer" e quindi può andare da 0 a 65535. Può essere espresso anche in base numerica diversa da 10.
- Indirizzi notevoli sono quelli relativi alla memoria della scheda video: in particolare **$B000** per la scheda MDA/Hercules monocromatiche, **$B800** per la schede CGA/EGA/VGA color in modo testo e in modo grafico per le risoluzioni inferiori, **$A000** per le risoluzioni superiori delle schede EGA/VGA in modo grafico.
- Se indirizzo manca si assume l'accesso all'area dati dell'interprete di R-Code.

Direttiva

Scopo:
Dichiara come intere tutte le variabili (semplici e aggregate) e le funzioni del modulo ed a singola precisione tutte le variabili e le funzioni dichiarate esplicitamente a doppia precisione. Una variabile con identificatore intero o stringa non viene influenzata da questa direttiva.

Sintassi:
!$SINGLE

Esempio:

```
.....
!$SINGLE
.....
A=100/33        ! A è intera e vale 33
B#=15/7         ! B# viene calcolata in singola precisione e vale 2.142857
.....
```

Note:

- Questa direttiva va messa prima della parte dichiarativa per essere sicuri della sua efficacia.
- Questa direttiva viene utilizzata dal programma di esempio R-SPREAD.

!$STACKSIZE

Direttiva

Scopo:
Consente di ridimensionare lo spazio di stack utilizzato dall'interprete R-Code per PC.

Sintassi:
!$STACKSIZE=dimensione

Esempio:

```
.....
!$STACKSIZE=$1000
.....
```

fissa lo spazio di stack a 4096 bytes.

Nota:

- Di default lo stack occupa 512 bytes, ma tale valore si può rilevare insufficiente in caso di procedure ricorsive.
- Dimensione può andare tra 512 e 8192 bytes. In **ERRE-VIC20 1.3** e **ERRE-C64 3.2** la dimensione dello stack è fissata in 192 bytes (non modificabile).

!$STOP

Direttiva

Scopo:
Consente di bloccare temporaneamente l'elaborazione del programma che può essere ripresa con l'istruzione R-Code 'CONT'.

Sintassi:
$STOP

Esempio:

Nota:

- In **ERRE-VIC20 1.3** può essere sostituita con **STOP**.

!$STRING

Direttiva

Scopo:
Dichiara come stringhe tutte le variabili (semplici ed aggregate) e le funzioni del modulo. Una variabile con il proprio identificatore di tipo non viene influenzata da questa direttiva.

Sintassi:

!$STRING

Esempio:

```
!$STRING
......
A="PIPPO"    ! A è di tipo stringa
B%=15        ! B% resta di tipo intero, nonostante la direttiva
```

Note:

- Questa direttiva va messa prima della parte dichiarativa per essere sicuri della sua efficacia.

!$TRACE... !$NOTRACE

Direttiva

Scopo:
Stampa, in fase di esecuzione, il numero dell'istruzione corrente, permettendo così di seguire il flusso del programma.

Sintassi:

!$TRACE

!$NOTRACE

Nota:
$NOTRACE disabilita una precedente $TRACE.

!$USERPTR

ERRE-C64 3.2 **ERRE-PC 3.0**

Direttiva

Scopo:
Definisce l'indirizzo di partenza ("entry point") di una routine scritta per il microprocessore "target" e richiamabile tramite la funzione USR.

Sintassi:
!$USERPTR=xxxx

Esempio:

```
.....
!$SEGMENT
!$USERPTR=$F000
.....
```

stabilisce come "entry-point" della routine in linguaggio macchina, l'indirizzo $F000 (61440 decimale) del segmento corrente.

Note:

- In **ERRE-C64 3.2** la funzione **USR** viene usata da alcune routine di 64VMS, il cui "entry point" deve essere fissato con questa direttiva.
- In **ERRE-VIC20 1.3** bisogna settare l' "entry point" della routine in modo esplicito con delle istruzioni POKE nelle locazioni di memoria 1 (LSB) e 2 (MSB).

!$VARSEG

Direttiva

Scopo:
Fissa la pagina iniziale della memorizzazione delle variabili.

Sintassi:
!$VARSEG=nnn

Esempio:

```
.....
!$VARSEG=60
.....
```

fissa l'inizio delle variabili alla pagina 60 (locazione 60*256=15360).

Note:

- Questa direttiva deve essere usata in unione con l'istruzione CHAIN per consentire il passaggio delle variabili tra moduli e va scelta basandosi sul programma di dimensione maggiore tra quelli coinvolti nelle CHAIN. **nnn** può andare da 8 a 160, ma non viene comunque eseguito nessun controllo sulla sua congruità.

!$BASE e !$KEY

Direttiva

Scopo:
!$BASE=n fissa l'indice inferiore di tutti gli array dichiarati con l'istruzione DIM a n, dove n può essere 0 (standard) o 1 (per compatibilità con il calcolo matriciale)

Sintassi:
!$BASE=0|1

Scopo:
!$KEY modifica il comportamento dell'istruzione **GET:** nelle versioni antecedenti **ERRE-PC 3.0** GET(var) è equivalente a var=INPUT$(1), usando questa direttiva si ottiene un comportamento uguale a quello delle versioni di ERRE per C-64.

Sintassi:
!$KEY

Le equivalenze con la versione **ERRE-PC 3.0** sono le seguenti:

	ERRE-PC 2.6	ERRE-PC 3.0
blocca la tastiera	GET(var$)	var$=INPUT$(1)
non blocca la tastiera	!$KEY GET(var$) var$ = INKEY$ (dall'interprete di R-Code - non previsto dallo standard)	GET(a$) var$ = GETKEY$ (forma funzionale)

!$INCLUDE

Direttiva

Scopo:
Segnala al compilatore di includere il file che segue (abitualmente una libreria).

Sintassi:
!$INCLUDE="nome_file"

Esempio:

```
!$INCLUDE="PC.LIB"
```

include, in fase di compilazione, la libreria di sistema PC.LIB.

Nota:

- E' stata parzialmente sostituita dall'istruzione **USES** nelle versioni successive di ERRE per ciò che riguarda la gestione delle librerie di sistema, sebbene $INCLUDE abbia un ambito di applicazione più vasto potendo includere file sorgente di tipo qualsiasi.

Unit

Scopo:
Controllo dello schermo in modo testo.

Procedure disponibili:

```
! Gestione cursore
! per ERRE 1.3 su
! VIC-20.
procedure home
!
procedure cls
!
procedure up
!
procedure down
!
procedure left
!
procedure right
!
procedure at
```

Note:

- A causa della mancanza del passaggio parametri nelle procedure in **ERRE-VIC20 1.3** le chiamate alle routine sono le seguenti:

  HOME, CLS → *<nome_procedura>*
  UP, LEFT, RIGHT, DOWN → *ARG%=<valore>* *<nome_procedura>*
  AT → *ROW%=<valore>:COL%=<valore>* *AT*

- La procedure 'AT' è stata sostituita dall'istruzione '@'.
- Va inserita nel programma nel punto desiderato tramite il comando **MERGE** di EDITOR13.

Unit

Scopo:
Controllo dello schermo in modo grafico (160x152).

Procedure disponibili:

```
!
! routine hires (160x152)
!
procedure hgrin
!
procedure hgrout
!
procedure plot
```

Note:

- A causa della mancanza del passaggio parametri nelle procedure in **ERRE-VIC20 1.3** le chiamate alle routine sono le seguenti:

  HGRIN, HGROUT → *<nome_procedura>*
  PLOT → *X=<valore_colonna>:Y=<valore_riga> PLOT*

- La unit va inserita nel programma nel punto desiderato tramite il comando **MERGE** di EDITOR13.

Unit

Scopo:
Controllo dello schermo in modo testo (sostituisce ed integra la CURSOR.LIB).

Procedure disponibili:

```
!
! CRT.LIB: procedure per la ge-
! stione dello schermo modo testo.
!
procedure home
procedure cls
procedure up(arg%)
procedure down(arg%)
procedure left(arg%)
procedure right(arg%)
! obsoleta - sostituire con @(row%,col%)
procedure at(row%,col%)
procedure crsrsave(->cx%,cy%)
procedure color(ch%,bo%,bg%)
```

CURSOR.LIB

Unit

Scopo:
Controllo del cursore in modo testo (mantenuta per compatibilità con ERRE-C64 2.3).

Procedure disponibili:

```
!
! Questa libreria fornisce al linguag-
! gio ERRE delle procedure per la  ge-
! stione del cursore.
!
procedure home
procedure cls
procedure up(arg%)
procedure down(arg%)
procedure left(arg%)
procedure right(arg%)
procedure at(row%,col%)
```

Nota:

- In ERRE-C64 2.3 le procedure sopraelencate sono parte integrante del linguaggio.

Unit

Scopo:
Controllo dello schermo in modo grafico (320x200).

Procedure disponibili:

```
! Pulisce la pagina ad alta risoluzione
! e setta i colori di sfondo e pixel
!
procedure hgrclr(c1%,c2%)
!
! Inizializza la pagina grafica
!
procedure hgrin
!
! Fa uscire dalla pagina grafica
!
procedure hgrout
!
! Cambia colore alla pagina grafica
!
procedure hgrcol(c1%,c2%)
!
! c1% è il colore dello sfondo
! c2% è il colore del pixel
!
!
! Plotta un punto alle coordinate x1%,y1%
!
procedure plot(x1%,y1%,mode%)
!
! Plotta il segmento di estremi xa%,ya%
! e xb%,yb%
!
procedure line(xa%,ya%,xb%,yb%,mode%)
!
! Plotta il cerchio di centro xa%,ya%
! e raggio rr%
!
procedure circle(xa%,ya%,rr%,mode%)
!
! Visualizza un tasto in alta risoluzione
! usando i caratteri ASCII tra 32 e 95
!
procedure text(xx%,yy%,aa$)
```

Unit

Scopo:
Implementa la funzione MATCH (mutuata dal linguaggio CBASIC per CP/M). E' una estensione dell'istruzione **INSTR** e consente la ricerca di sottostringhe usando anche metacaratteri.

Procedure disponibili:

```
! MATCH.LIB: funzione match del CBASIC
!
! i%=match(a$,b$,ps%)
!
! trova a$ *in* b$ a partire dalla posizione ps% (case sensitive)
!
! metacaratteri usabili:
! ----------------------
! # qualsiasi numero
! ! qualsiasi lettera maiuscola o minuscola
! ? qualsiasi carattere
! £ annulla effetto metacarattere

procedure match(a$,b$,ps%->match)
```

Unit

Scopo:
Gestione di finestre video in modo testo.

Procedure disponibili:

```
! Definizione di una finestra
!
! numwin% --> n. finestra richiesta
! vx%     --> ascissa vertice alto sin.
! vy%     --> ordinata vertice alto sin.
! lx%     --> lunghezza finestra (asse x)
! ly%     --> altezza finestra (asse y)
!
procedure wdef(numwin%,vx%,vy%,lx%,ly%)
!
! Definisce il colore di una finestra
!
procedure wcolor(numwin%,colr%)
!
! Stampa un testo entro una finestra
!
procedure wprint(numwin%,string$,mode%)
!
! Pulisce una finestra
!
procedure wclear(numwin%)
```

Nota:

- Con la versione 3.2, la gestione delle finestre in modo testo non è più preinstallata ma è disponibile solo caricando il file '*ADD32X.LM*'.

Unit

Scopo:
Libreria principale in **ERRE-PC 2.6**. Richiamata tramite la direttiva **!$INCLUDE=**"PC.LIB".

Procedure disponibili:

```
SINH(X)                        ! Funzioni iperboliche
COSH(X)
TANH(X)
ATANH(X)
SPACES$(X)                     ! Stringa composta da X spazi
CLS                            ! Pulisce lo schermo (modo testo)
HOME                           ! Cursore in alto a sinistra (home)
DOWN(ZA%)                      ! Cursore in giù di ZA% posizioni
UP(ZA%)                        ! Cursore in sù di ZA% posizioni
LEFT(ZA%)                      ! Cursore a sinistra di ZA% posizioni
RIGHT(ZA%)                     ! Cursore a destra di ZA% posizioni
LOCATE(ZR%,ZC%)                ! Posiziona il cursore
AT(ZR%,ZC%)                    ! LOCATE (AT) in stile C-64
CURSOR(ZD%,ZI%,ZF%)            ! Definisce lo stato del cursore
                               ! ZD%=0 invisibile,1=visibile
                               ! ZI% e ZF% la riga iniziale e
                               ! la finale
CURSOR_SAVE(->CURX%,CURY%)     ! Salva lo stato del cursore
                               ! CURX% e CURY% sono var. globali
WIDTH(ZD$,ZL%)                 ! Stabilisce il num. colonne del
                               ! device: ZD$ può essere SCRN:,
                               ! COMx:,LPTx:,#file
                               ! ZL%=40/80 per schermo
VIEW_PRINT(ZI%,ZF%)            ! imposta una finestra di schermo
                               ! tra la riga ZI% e la riga ZF%
FKEY_DISABLE                   ! disabilita i tasti funzione
SETVIDEOSEG(->VIDSEG#)         ! setta il corretto segmento video
CHANGE(ZX$,ZX%,ZY$->ZX$)       ! simula la MID$(.)= del Basic
SUBST(ZN%,ZX$,ZY$,ZZ$->ZX$)    ! Cambia in ZX$ ZZ$ al posto di
                               ! ZY$ a partire da ZN%
UPPER(ZX$->ZX$)                ! da minuscolo a maiuscolo
LOWER(ZX$->ZX$)                ! da maiuscolo a minuscolo
LTRIM(ZX$->ZX$)                ! toglie spazi bianchi a sinistra
RTRIM(ZX$->ZX$)                ! toglie spazi bianchi a destra
BLOAD(PFILE$,PP)               ! Carica segmenti di memoria
BSAVE(PFILE$,PP,LL)            ! Salva segmenti di memoria
SCREEN(ZN%)                    ! Attiva il modo testo (0)
                               ! 320*3200 (1), 640*200 (2)
                               ! EGA(7,8,9,10)
GCLS                           ! Pulisce lo schermo (in modo
                               ! grafico)
VSCREEN(ZR%,ZC%,ZT%->ZV%)      ! Fornisce il codice ASCII del
                               ! carattere sullo schermo in
                               ! ZR%,ZC% se ZT%=0, se ZT%=1
                               ! ne fornisce l'attributo
COLOR(ZF%,ZS%)                 ! Seleziona il colore
```

```
PSET(ZX,ZY,ZC%)                ! Disegna un punto in ZX% e ZY%
                               ! con colore ZC% (assoluto)
REL_PSET(ZX,ZY,ZC%)            ! Disegna un punto in ZX% e ZY%
                               ! con colore ZC% (relativo)
POINT(ZX,ZY->ZC%)              ! Restituisce il colore del punto
                               ! di coordinate ZX,ZY
PALETTE(ZA%,ZC%)               ! Assegna il colore ZC% all'attributo
                               ! ZA% (solo con EGA): 0¾ZA%¾15 e
                               ! 0¾ZC%¾63. Se =-1 colori di default.
PALETTE_USING(ZP%[])           ! Palette per tutti i colori
LINE(ZX,ZY,ZK,ZZ,ZC%,BB%)      ! Disegna una linea (BB%=FALSE) o un
                               ! rettangolo (BB%=TRUE)
PAINT(ZX,ZY,ZC%)               ! Colora un'area con colore ZC%
REL_LINE(ZK,ZZ,ZC%,BB%)        ! Disegna una linea o un rettangolo
                               ! in coord. relative
CIRCLE(ZX,ZY,ZR,ZC%)           ! Disegna un cerchio con centro
                               ! ZX%,ZY% raggio ZR% e colore ZC%
GR_WINDOW(ZX,ZY,ZK,ZZ)         ! Da qui in poi le coordinate delle
                               ! istruzioni grafiche non sono
                               ! calcolate secondo le dimensioni
                               ! dello schermo
SOUND(ZF,ZD)                   ! Suona una nota (frequenza ZF e
                               ! durata ZD)
PLAY(ZA$)                      ! Suona un motivo
BEEP                           ! Suona un BEEP
OS_DIR(ZPATH$)                 ! Comando DIR
OS_DELETE(ZFILE$)              ! Cancella file
OS_DELETE2(ZFILE$)             ! Cancella un file con richiesta
                               ! se cancellazione totale
OS_RENAME(ZFILE1$,ZFILE2$)     ! Rinomina un file
OS_CHDIR(ZPATH$)               ! Cambia directory
OS_MKDIR(ZPATH$)               ! Crea directory
OS_RMDIR(ZPATH$)               ! Cancella directory (solo se è
                               ! vuota!)
OS_ENVIRON(ZPATH$)             ! Inserisce una voce nella tabella di
                               ! ambiente
OS_READ_ENVIRON(ZPATH$->ZVAR$)! legge una voce della tabella di
                               ! ambiente
```

Note:

- Comprende la maggior parte delle funzioni e delle procedure organizzate poi in **ERRE-PC 3.0** come "unit" separate.

Unit

Scopo:
Gestione schermo in modo testo.

Procedure disponibili:

```
PROCEDURE CLS                          ! Pulisce lo schermo (modo testo)
PROCEDURE HOME                         ! Cursore in alto a sinistra (home)
PROCEDURE DOWN(ZA%)                    ! Cursore in gi— di ZA% posizioni
PROCEDURE UP(ZA%)                      ! Cursore in s— di ZA% posizioni
PROCEDURE LEFT(ZA%)                    ! Cursore a sinistra di ZA% posizioni
PROCEDURE RIGHT(ZA%)                   ! Cursore a destra di ZA% posiziono
PROCEDURE LOCATE(ZR%,ZC%)              ! Posiziona il cursore
PROCEDURE AT(ZR%,ZC%)                  ! LOCATE (AT) in stile C-64
PROCEDURE CURSOR(ZD%,ZI%,ZF%)          ! Definisce lo stato del cursore
                                       ! ZD%=0 invisibile,1=visibile
                                       ! ZI% e ZF% la riga iniziale e la finale
                                       ! di scansione
PROCEDURE CURSOR_SAVE(->ZX%,ZY%)       ! Salva la posizione del cursore
PROCEDURE VIDEO_SEGMENT(ZZ%->ZB#)      ! Trova l'indirizzo base dello schermo
                                       ! e setta (se richiesto - ZZ%=TRUE) il
                                       ! corretto segmento video
PROCEDURE WIDTH(ZD$,ZL%)               ! Stabilisce il num. colonne del device
                                       ! ZD$ può essere SCRN:,COMx:,LPTx:,#file
                                       ! ZL%=40/80 per larghezza schermo
PROCEDURE VIEW_PRINT(ZI%,ZF%)          ! imposta una finestra di schermo tra
                                       ! la riga ZI% e la riga ZF%
PROCEDURE COLOR(ZF%,ZS%)               ! Seleziona il colore car,sfondo
                                       ! uno dei parametri manca se = -1
PROCEDURE VSCREEN(ZR%,ZC%,ZT%->ZV%)    ! Fornisce il codice ASCII del
                                       ! carattere sullo schermo in
                                       ! ZR%,ZC% se ZT%=0, se ZT%=1
                                       ! ne fornisce l'attributo
PROCEDURE LCOPY                        ! Hardcopy dello schermo
PROCEDURE FLUSHKB                      ! Svuota il buffer di tastiera
```

Unit

Scopo:
Estende le funzionalità della funzione “VAL” permettendo di valutare espressioni complete, in accordo con la sintassi di ERRE, assegnate come input.

Procedure disponibili:

```
PROCEDURE EVAL(ZZ$->ZZ#)            ! Valuta espressioni – estende funzione VAL
```

Unit

Scopo:
Gestione dei tasti funzione.

Procedure disponibili:

```
PROCEDURE FKEY_DISABLE              ! non visualizza i tasti funzione 25°riga
PROCEDURE FKEY_ENABLE               ! visualizza i tasti funzione 25°riga
PROCEDURE FKEY_LIST                 ! elenca i tasti funzione
PROCEDURE FKEY_SET(ZZ%,ZZ$)         ! assegna stringa a tasto funzione
PROCEDURE FKEY_CLEAR                ! cancella in blocco tasti funzione
```

Unit

Scopo:
Funzioni aggiuntive di sistema.

Procedure disponibili:

```
VARIANT FUNCTION SINH?(X?)          ! Funzioni iperboliche
VARIANT FUNCTION COSH?(X?)
VARIANT FUNCTION TANH?(X?)
VARIANT FUNCTION ATANH?(X?)
VARIANT FUNCTION LOG10?(X?)         ! Logaritmo base 10
VARIANT FUNCTION CBRT?(X?)          ! Radice cubica
FUNCTION SPACES$(X)                 ! Stringa composta da X spazi
FUNCTION DEEK(X)                    ! PEEK a 16 bit
FUNCTION IS_LETTER(X)               ! ASCII X è una lettera ?
FUNCTION IS_NUMBER(X)               ! ASCII X è un numero ?
FUNCTION UI(X)                      ! da integer ad unsigned integer
FUNCTION IU(X)                      ! da unsigned integer a integer
```

Unit

Scopo:
Gestione schermo in modo grafico.

Procedure disponibili:

```
PROCEDURE SCREEN(ZN%)                ! Attiva il modo testo (0)
                                     ! 320*3200 (1), 640*200 (2)
                                     ! EGA(7,8,9,10)
PROCEDURE XSCREEN(ZN%,ZA%,ZV%)       ! Seleziona la pagina attiva
                                     ! e la pagina visualizzata.
                                     ! (1,2,4 o 8 pagine in funzione
                                     ! dell'adattatore video utilizzato).
PROCEDURE PCOPY(ZS%,ZD%)             ! Copia la pagina video ZS% sulla ZD%
                                     ! (se consentito dall'adattatore video)
PROCEDURE XCLS(ZQ%)                  ! CLS esteso. Se ZQ%=2 cancella solo
                                     ! la viewport definita con VIEW_PRINT
PROCEDURE XCOLOR(ZF%,ZS%)            ! Seleziona il colore car,sfondo
                                     ! se uno dei parametri =-1 è mancante
PROCEDURE PLOT(ZX,ZY,ZC%)            ! Disegna un punto in ZX e ZY con
                                     ! colore ZC% (assoluto)
PROCEDURE PLOT_REL(ZX,ZY,ZC%)        ! Disegna un punto in ZX e ZY con
                                     ! colore ZC% (relativo)
PROCEDURE POINT(ZX,ZY->ZC%)          ! Restituisce il colore del punto
                                     ! di coordinate ZX,ZY
PROCEDURE PALETTE(ZA%,ZC%)           ! Assegna il colore ZC% all'attributo
                                     ! ZA% (solo con EGA): 0<=ZA%<=15 e
                                     ! 0<=ZC%<=63. Se =-1 colori di default.
PROCEDURE LINE(ZX,ZY,ZK,ZZ,ZC%,BB%)  ! Disegna una linea (BB%=FALSE) o un
                                     ! rettangolo (BB%=TRUE) con colore ZC%
                                     ! se ZC%=-1 usa colore corrente
PROCEDURE LINE_TO(ZK,ZZ,ZC%,BB%)     ! Disegna una linea o un rettangolo
                                     ! dall'ultimo punto noto. Se ZC%=-1
                                     ! ignora colore
PROCEDURE RECTANGLE_FILL(ZX,ZY,ZK,ZZ,ZC%,ZS) ! Disegna una rettangolo e lo
                                     ! riempie con colore ZC% e stile ZS
                                     ! se ZC%=-1 usa colore corrente
                                     ! se ZS=-1 nessuno stile
PROCEDURE RECTANGLE_FILL_TO(ZK,ZZ,ZC%) ! Disegna una rettangolo e lo riempie
                                     ! con colore ZC% partendo dall'ultimo
                                     ! punto noto. Se ZC%=-1 usa colore
                                     ! corrente
PROCEDURE PAINT(ZX,ZY,ZC%,ZB%)       ! Colora un'area con colore ZC% e bordo
                                     ! ZC%:se area è rettang. vedi RECTANGLE
                                     ! _FILL e RECTANGLE_FILL_TO
PROCEDURE PAINT_TILE(ZX,ZY,ZC$,ZB%)  ! Colora un'area con bordo ZB% col trat-
                                     ! teggio ZC$ (vedi anche PAINT)
PROCEDURE CIRCLE(ZX,ZY,ZR,ZC%,ZA)    ! Disegna un cerchio con centro assoluto
                                     ! ZX,ZY raggio ZR%, colore ZC% e aspetto
                                     ! ZA (se =-1 aspetto standard)
PROCEDURE CIRCLE_REL(ZX,ZY,ZR,ZC%,ZA) ! Disegna un cerchio con centro relati-
                                     ! vo ZX,ZY raggio ZR%,colore ZC% e aspet-
                                     ! to ZA (se =-1 aspetto standard)
```

```
PROCEDURE ARC(ZX,ZY,ZR,ZC%,ZS,ZE,ZA)  ! Disegna un arco di cerchio tra ZS e
                                       ! ZE con centro assoluto ZX,ZY raggio ZR%,
                                       ! colore ZC% e aspetto ZA (se =-1 aspetto
                                       ! standard)
PROCEDURE ARC_REL(ZX,ZY,ZR,ZC%,ZA,ZE,ZA) ! Disegna un arco di cerchio tra ZS
                                       ! e ZE, centro relativo ZX,ZY, raggio ZR%,
                                       ! colore ZC% e aspetto ZA (se =-1 aspetto
                                       ! standard)
PROCEDURE WINDOW(ZX,ZY,ZK,ZZ,ZS%) ! Abilita le coordinate grafiche utente
                                       ! nelle istruzioni grafiche e disabilita
                                       ! le coordinate grafiche schermo. Con ZS%
                                       ! TRUE viene invertita la direzione
                                       ! dell'asse y.
PROCEDURE WINDOW_OFF                   ! Disabilita le coordinate grafiche utente
PROCEDURE DRAW(ZA$)                    ! Disegna una figura stile Logo: usa
                                       ! il G.M.L. di GWBASIC (vedi manuale)
```

Unit

Scopo:
Gestione aree di memoria e porte I/O.

Procedure disponibili:

```
PROCEDURE BLOAD(ZFILE$,ZP)          ! Carica segmenti di memoria
PROCEDURE BSAVE(ZFILE$,ZP,ZL)       ! Salva segmenti di memoria
PROCEDURE DOKE(ZA,ZV)               ! POKE a 16 bit
PROCEDURE PORT_INP(ZP->ZA%)         ! Legge la porta di I/O ZP
                                    ! (0<=ZP<=65535)
PROCEDURE PORT_OUT(ZP,ZA%)          ! Scrive ZA% sulla porta di I/O ZP
                                    ! (0<=ZP<=65535)
```

Unit

Scopo:
Gestione del mouse.

Procedure disponibili:

```
PROCEDURE INITASM                    ! Load assembly language subroutine
PROCEDURE EXECUTE_ASM(INTERRUPT%)
PROCEDURE MOUSE_RESETANDSTATUS(->STATUS,BUTTONS)
PROCEDURE MOUSE_SHOWCURSOR
PROCEDURE MOUSE_HIDECURSOR
PROCEDURE MOUSE_GETCURSORPOSITION(->X,Y,LEFT%,RIGHT%,BOTH%,MIDDLE%)
PROCEDURE MOUSE_SETCURSORPOSITION(X,Y)
PROCEDURE MOUSE_GETBUTTONPRESSINFO(->STATUS,BUTTONS,XPOS,YPOS)
PROCEDURE MOUSE_GETBUTTONRELEASEINFO(->STATUS,BUTTONS,XPOS,YPOS)
PROCEDURE MOUSE_SETCURSORLIMITS(TOPLIMIT,BOTTOMLIMIT,LEFTLIMIT,RIGHTLIMIT)
PROCEDURE MOUSE_SETEXTCURSOR
PROCEDURE MOUSE_READMOTIONCOUNTERS(->XMICKEY,YMICKEY)
PROCEDURE MOUSE_SETSENSITIVITY(XSENS,YSENS,THRESHOLD)
PROCEDURE MOUSE_GETINFO(->VERSION,MOUSE_TYPE,MOUSE_IRQ)
```

Nota:

- La procedura EXECUTE_ASM si usa, più genericamente, per accedere agli interrupt del PC.
- Presente anche in **ERRE-PC 2.6**.

Unit

Scopo:
Conversione di basi numeriche

Procedure disponibili:

```
PROCEDURE HEX2DEC(ZA$->ZA#)          ! conversioni di basi numeriche
PROCEDURE OCT2DEC(ZA$->ZA#)
PROCEDURE BIN2DEC(ZA$->ZA#)
PROCEDURE DEC2HEX(ZA#->ZX$)
PROCEDURE DEC2OCT(ZA#->ZX$)
PROCEDURE DEC2BIN(ZA#->ZX$)
```

Unit

Scopo:
Interfaccia con il Sistema Operativo.

Procedure disponibili:

```
PROCEDURE OS_DIR(ZPATH$)              ! Comando DIR
PROCEDURE OS_DELETE(ZFILE$)           ! Cancella file
PROCEDURE OS_DELETE2(ZFILE$)          ! Cancella file con richiesta
                                      ! se cancellazione totale
PROCEDURE OS_RENAME(ZFILE1$,ZFILE2$) ! Rinomina un file
PROCEDURE OS_CHDIR(ZPATH$)            ! Cambia directory
PROCEDURE OS_MKDIR(ZPATH$)            ! Crea directory
PROCEDURE OS_RMDIR(ZPATH$)            ! Cancella directory (solo se è vuota!)
PROCEDURE OS_ENVIRON(ZPATH$)          ! Inserisce una voce nella tabella di
                                      ! ambiente
PROCEDURE OS_READ_ENVIRON(ZPATH$->ZVAR$) ! Legge una voce della tabella di
                                      ! ambiente
```

Unit

Scopo:
Gestione dello speaker del PC.

Procedure disponibili:

```
PROCEDURE SOUND(ZF,ZD)              ! Suona una nota (frequenza ZF e durata
                                    ! ZD)
PROCEDURE PLAY(ZA$)                 ! Suona un motivo: usa lo stesso M.M.L.
                                    ! di GWBASIC (vedi manuale)
PROCEDURE BEEP                      ! Suona un BEEP
```

Unit

Scopo:
Permette la sintesi vocale, utilizzando il Sistema Operativo ospite. *E' attivo solo per sistemi Windows dalla versione XP in poi.*

Procedure disponibili:

```
PROCEDURE SAY(SPEAK$)
! solo per Windows 32 bit
```

Unit

Scopo:
Gestione stringhe estesa

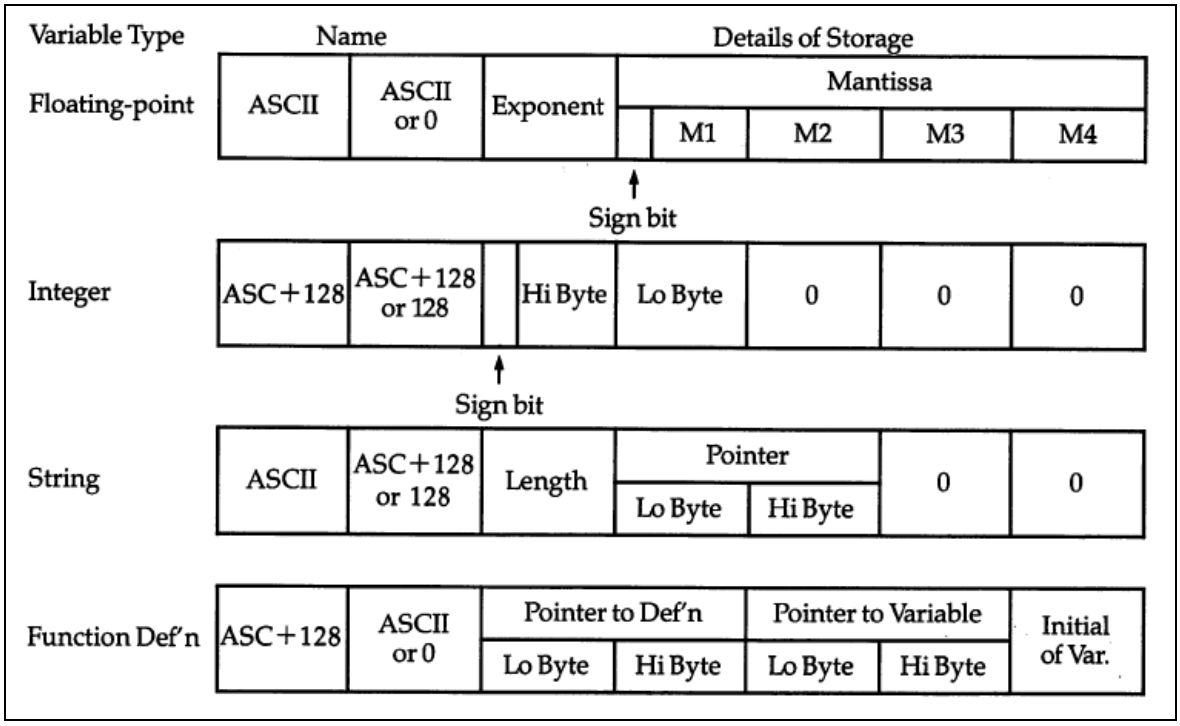
Procedure disponibili:

```
PROCEDURE SUBST(ZN%,ZX$,ZY$,ZZ$->ZX$) ! Cambia in ZX$ ZZ$ al posto di ZY$
                                       ! da ZN%
PROCEDURE UCASE(ZX$->ZX$)              ! stringa da minuscolo a maiuscolo
PROCEDURE LCASE(ZX$->ZX$)              ! stringa da maiuscolo a minuscolo
PROCEDURE LTRIM(ZX$->ZX$)              ! toglie spazi bianchi a sinistra
PROCEDURE RTRIM(ZX$->ZX$)              ! toglie spazi bianchi a destra
PROCEDURE STR_REVERSE(ZX$->ZX$)        ! Inverte una stringa
PROCEDURE STR_REPEAT(ZN%,ZY$->ZX$)     ! Estensione funzione STRING$
PROCEDURE MATCH(ZA$,ZB$,ZP%->ZM)
! funzione MATCH del CBASIC
!
! i%=MATCH(a$,b$,ps%)
!
! trova a$ *in* b$ a partire dalla posizione ps% (case sensitive)
!
! metacaratteri usabili:
! ----------------------
! # qualsiasi numero
! ! qualsiasi lettera maiuscola o minuscola
! ? qualsiasi carattere
! \ annulla effetto metacarattere
```

APPENDICE

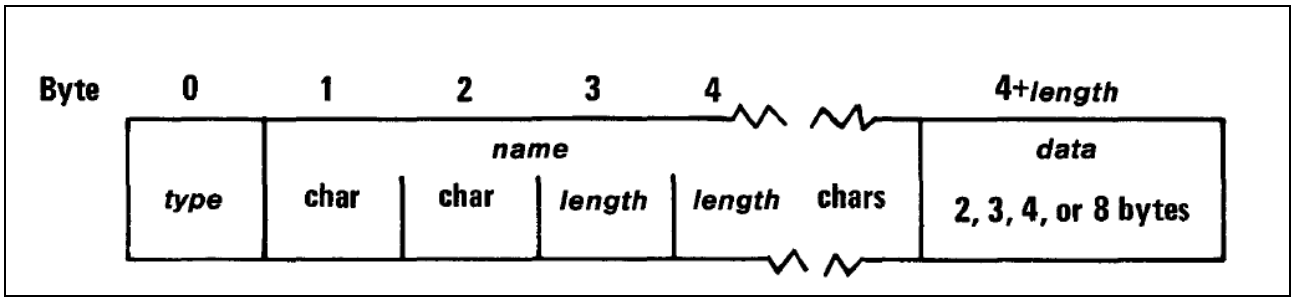
Come l'interprete di R-Code memorizza le variabili scalari ("Microsoft Binary Format" - MBF)

ERRE-VIC20 1.3 ERRE-C64 2.3 ERRE-C64 3.2



- Ogni variabile occupa 7 byte: lo "string descriptor" delle variabili stringa punta all'area di memoria che contiene il valore effettivo.

ERRE-PC 2.6 ERRE-PC 3.0



dove:

- byte 0 ("type") rappresenta il tipo di variabile: 2=integer, 3=string, 4=real, 8=double
- byte 1-2 ("char") rappresenta i primi due caratteri dell'identificatore di variabile
- byte 3 ("length") il numero degli altri caratteri dell'identificatore di variabile
- byte 4 a seguire gli altri caratteri dell'identificatore di variabile (se esistono)
- byte 4+length ("data") il valore della variabile; nel caso di stringa è il cosiddetto "string descriptor" (lunghezza stringa + suo indirizzo - LSB-MSB). VARPTR e & puntano a questo indirizzo.

ERRE – MANUALE DI RIFERIMENTO

v 1.3 per VIC-20, v 3.2A per C-64 e v 3.0B per PC