DOCUMENTO STONEMAN

STONE MAN

by POMMODORE INC.

LUGLIO 1984

DOCUMENTO STONEMAN

1.0 INTRODUZIONE

Questo documento presenta una proposta per formalizzare la sintassi del linguaggio denominato ERRE.
A questo proposito si userà una notazione non strettamente formale, comunque molto efficace: precisamente si converrà che

[item] significa che 'item' è opzionale;

item # significa che 'item' è ripetuto zero o più volte.

Nel seguito si assumerà che il lettore abbia familiarità con concetti quali passaggio di parametri; programma strutturato, ecc ...

2.0 DEFINIZIONE DI ERRE

ERRE è un linguaggio strutturato con possibilità di definire tipi utente, variabili, procedure interne ed esterne (dette LIBRERIE).
Il passaggio dei parametri tra i vari moduli del programma avviene, per esigenze tecniche, realizzato ESCLUSIVAMENTE PER INDIRIZZO potendosi usare solo variabili semplici o array.

3.0 SINTASSI DI ERRE

Le parole scritte in maiuscolo indicheranno parole chiave del linguaggio.

Un programma è : PROGRAM identifier

```
[ include declaration ]
[ data declaration ]
[ type declaration ]
[ var declaration ]
# subprogram declaration#
BEGIN
  statement #:statement#
END.
```

3.1 Parte dichiarativa

Una include declaration è :

```
INCLUDE(identifier,#identifier#)
```

ove per identifier si intende il nome di una libreria esterna esistente su disco e organizzata a procedure. Tali librerie possono essere realizzate pure dagli utenti e messe a disposizione. (vedi per maggiori dettagli il capitolo seguente)

Una subprogram declaration è una delle due seguenti:

```
procedure declaration
function declaration
```

Una procedure declaration è:

```
PROCEDURE identifier
  [(identifier:type # ;identifier:type # ←
  identifier:type # ;identifier;type #)]
VAR identifier#;identifier#:predefined-type
    #identifier#,identifier#:predefined-type#
       statement #:statement #
ENDPROCEDURE
```

In questo contesto predefined-type si riferisce specificatamente ai tipi standard definiti più sotto (v. record-type) con in più l'array-type.

Una function declaration è:

```
FUNCTION identifier
  identifier(variable-identifier)=expression
ENDFUNCTION
```

Una data declaration è:

DATA item #,item#

dove " item " è una stringa di caratteri ASCII attribuibili come valori a variabili.

Una type declaration è: TYPE identifier=type
identifier=type

Una var declaration è : VAR identifier #,identifier#:type
#identifier #,identifier#:type#

Un type è uno dei seguenti :

- INTEGER
- REAL
- BOOLEAN
- STRING
- array-type
- record-type
- type-identifier

Un array-type è :

ARRAY[integer .. integer#,integer .. integer#] OF type

In questo caso type intende tutti i precedenti eccetto array-type.

Un record-type è :

RECORD
identifier=predefined-type
#identifier=predefined-type#
ENDRECORD

Per predefined-type si intende uno dei seguenti tipi: INTEGER, REAL, BOOLEAN o STRING.

3.2 Statement

Uno statement è uno dei seguenti :

3.2.1 Statement di tipo generale

PRINT[([file-identifier,] [output-item # output-item#])]@

@ può essere <CR> o ';' o ',' .
Invece ' output-item ' può essere o una stringa di caratteri o una lista di variabili separate da "," o da ";" o una delle funzioni SPC(x) o TAB(x) o una alternanza delle precedenti.
Serve per scrivere sul file file-identifier aperto con una clausola OPEN.

INPUT([file-identifier,][character string,] variable-identifier #,variable-1dentifier#

Serve per leggere dal file file-identifier aperto con una clausola OPEN. La presenza contemporanea di file-identifier e di character string è illegale.

GET([file-identifier)] variable-identifier)

Questo statement si usa per leggere dal file file-identifier aperto con una clausola OPEN un carattere che viene assegnato a variable-identifier.

assignement : variable=expression

procedure-calling: procedure-identifier [expression#;expression#
← expression#;expression#]

Ci dev'essere corrispondenza biunivoca tra i parametri attuali dati e quelli formali ricevuti: è escluso in ogni caso il passaggio di un intero record; resta però possibile il passaggio di un campo di questo.

READ (variable-identifier #,variable-identifier#)

Serve per leggere dati interni al programma dichiarati sotto TYPE tramite la clausola DATA. (v. dichiarazione relativa ai tipi)

SWAP(variable-identifier,variable-identifier)

Serve per scambiare i contenuti delle due variable-identifier.

3.2.2 Strutture di controllo

Sono le seguenti :

```
IF expression
 THEN
   statement#:statement#
 [ ELSE
   statement#:statement#]
ENDIF
```

```
       CASE expression OF
        expression←
          statement#:statement#
        END←
          # expression←
          statement #:statement#
        END←#
        [ OTHERWISE
          statement #:statement#]
       ENDCASE

       WHILE expression
          statement#:statement#
       ENDWHILE

       FOR variable-identifier=expression TO expression [STEP
                                                          expression]
         [statement#:statement]
       NEXT

       REPEAT
          statement#:statement#
       UNTIL expression
```

A una qualsiasi expression è attribuibile un valore logico.

3.2.3 Statement per il trattamento dei file

Per il trattamento dei file abbiamo a disposizione cinque primitive : OPEN,CLOSE,INPUT,PRINT,GET; la sintassi delle ultime tre è già stata esaminata. Per le altre due vale :

```
       OPEN(file-identifier ON device-name,command string)

       CLOSE(file-identifier)
```

dove file-identifier può essere :

1) variable-identifier
2) character string

device-name può essere scelto tra :

```
       KEYBOARD
       CRT
       DISK
       TAPE
       PRINTER
       PLOTTER
       RS-232
```

```
       #n:
```

dove n è un integer indicante il numero del device.

command string corrisponde alle modalità di uso delle varie periferiche e si può pure omettere. Ad esempio

```
       OPEN("stampascarto" ON PRINTER, 7)
```

apre il file "stampascarto" sulla stampante permettendo la scrittura in minuscolo.

3.2.4 Interfaccia con il linguaggio macchina

Per permettere l'unione tra programmi scritti in ERRE con routine scritte in Assembler 6510 si mettono a disposizione due statement e due funzioni. I due statement sono

```
           POKE(long-expression,expression)

           SYS(long-expression)
```

dove long-expression è una espressione integer di valore compreso tra 0 e 65535, mentre expression è un integer tra 0 e 255.
Per le due funzioni vedi il paragrafo 3.3.1 (Funzioni numeriche).

In generale gli statement possono essere separati tra loro anche con <CR> oltreché da '' : '' .

3.3 Funzioni predefinite di ERRE

Esaminiamo brevemente le funzioni predefinite di ERRE:

3.3.1 Funzioni numeriche

```
     SIN(x)
     COS(x)
     TAN(x)
     ATN(x)
     LOG(x)
     EXP(x)
     ABS(x)
     SGN(x)
     RND(x)
     SQR(x)
     INT(x)
```

```
     PEEK(x)
     USR(x)
```

Data come x una espressione intera o reale forniscono in uscita un valore reale, eccetto le ultime due che servono rispettivamente per leggere locazioni di memoria (0<=x<=65535) e per passare ad una routine in L.M. un parametro x.

3.3.2 Funzioni di stringa

```
     ASC(x)
     CHR(x)
     LEFT(x,i)
     LEN(x)
     MID(x,i,p)
     RIGHT(x,i)
     STR(x)
     VAL(x)
```

x è in questo caso una variabile di tipo stringa, mentre p ed i sono integer; il significato delle funzioni è identico a quello del BASIC.

3.3.3 Altre funzioni

```
     SPC(x)
     TAB(x)
```

si usano in congiunzione con lo statement PRINT

```
     CUP(x)
     CDOWN(x)
     CLEFT(x)
     CRIGHT(x)
     AT(x,y)
```

Queste funzioni servono per controllare il movimento del cursore sullo schermo: in ogni caso x è una expression intera compresa tra 0 e 39, mentre y è compresa tra 0 e 24.

```
     PAUSE(x)
```

ferma l'esecuzione del programma per n secondi (n integer).

3.3.4 Operatori

```
     + - * / ^      operatori algebrici:  '-' serve pure per segnare un
                    numero.
     AND,OR,NOT     operatori logici
     = <> > >= < <= operatori relazionali
```

Le espressioni contenenti operatori vengono valutate seguendo le

consuete regole relative alla priorità algebrica.

3.3.5 Variabili riservate

STATUS
TIME

che forniscono rispettivamente un controllo sulla esattezza delle operazioni con le periferiche e un orologio aggiornato.

3.4 Alfabeto di ERRE

Vediamo ora l'alfabeto usato da ERRE e come si combinano i suoi elementi per formare identifier, variable-identifier ecc.
L'alfabeto usato è il codice ASCII a 8 bit, perciò si definisce :

CHARACTER : un elemento del codice ASCII

LETTER : una delle seguenti

a b c d e f g h i j k l m n o p q r s t u v w x y u z

DIGIT : uno dei seguenti

0 1 2 3 4 5 6 7 8 9

SPECIAL CHARACTER è uno dei seguenti:

+ - * / ^ < > = [ ] : ; , ' # ! @ % & ( ) . ← ? b (sta per blank)

UNSIGNED INTEGER è uno o più digit senza spazi bianchi intermedi.

UNSIGNED REAL assume due forme: una consiste di due o più digit con un punto decimale (almeno una cifra - anche 0 - deve precedere il punto decimale e una deve seguirlo); l'altra consiste in una mantissa seguita da un esponente (sempre senza spazi bianchi intermedi) separati da una '' E ''.

La mantissa è un unsigned real della prima forma mentre l'esponente è un integer (eventualmente con segno).

mantissaEesponente

Chiaramente un SIGNED INTEGER (o REAL) sarà preceduto da + (facoltativo) o da -

CHARACTER STRING è un insieme di caratteri delimitati da "

IDENTIFIER (VARIABLE-IDENTIFIER) è una lettera minuscola seguita da

lettere minuscole o numeri fino ad un totale di 256 caratteri. Inoltre un identifier non può essere uguale a nessuna delle parole chiave di ERRE (vedi Appendice UNO) né può contenere blanks (sostituibili con '-' o con "_").

COMMENT è una stringa di caratteri preceduta dal carattere speciale '!' e delimitata in fondo da <CR>.

EXPRESSION è una delle seguenti:

    unsigned integer
    unsigned real
    character string
    + expression
    - expression
    expression <OPERATOR> expression

    ove <OPERATOR> è uno dei seguenti :

    + - * / ^ AND OR NOT = < > >= <= <>

    (expression)
    funzioni (predefinite o di utente)
    variable identifier

4.0 PROBLEMI E DETTAGLI IMPLEMENTATIVI RELATIVI AL COMPILATORE

In questa parte si tratterà dei dettagli relativi all'implementazione del compilatore ERRE, dando solo delle specifiche di massima.

4.1. Traduzione del programma.

Il textfile in ERRE tramite varie passate (v. più avanti) viene tradotto in linguaggio BASIC e eventualmente compilato in codice-macchina.
Distinguiamo le seguenti fasi:

1) Eventuale aggancio in coda di librerie dichiarate tramite la clausola INCLUDE.

2) Creazione della tabella dei tipi e delle variabili.

3) Ristrutturazione di detta tabella ribattezzando le variabili con nomi adattti al BASIC.

4) Traduzione vera e propria della parte esecutiva, con diagnostica appropriata degli errori.

Il textfile ERRE può essere memorizzato in due modi:
- tramite un array di stringhe.
- in un area di memoria riservata secondo un bytestream.

La sintassi prima esposta fa riferimento ad una struttura a bytestream, in quanto non esistono vincoli su come debba essere disposto il textfile ERRE; in ogni caso il tipo di Editor condizionerà il tipo di questa struttura.
La traduzione in BASIC verrà anch'essa memorizzata sotto forma di bytestream (sarà compito del TKB rendere eseguibile la compilazione).
Esaminiamo ora le passate.

4.1.1 Eventuale aggancio di librerie

Avviene facendo un merge con il programma principale: verrà posta la clausola LIBRARY dopo END. e END OF LIBRARY al termine di tutto.
A questo punto il programma è completo e pronto per la fase due.

4.1.2 Creazione della tabella dei tipi e delle variabili

Sono costituite da due campi: nel primo caso il primo contiene il nome del tipo dichiarato dall'utente, il secondo un puntatore all'eventuale tipo standard. Il puntatore sarà 0 se si è raggiunto un tipo standard.
Nel secondo caso il primo campo contiene il nome della variabile mentre

il secondo un puntatore al tipo corrispondente della tabella dei tipi.
Eventualmente si dovrà creare una TABELLA FILE che contiene il nome dei file dichiarati assieme al device usato.
Verrà distinto se la variabile dichiarata è interna ad una procedura: ciò permetterà di usare due variabili con lo stesso nome (una globale ed una locale) ma l'eventuale globale non potrà essere usata nella procedura in cui figura la sua omonima.

4.1.3 Ridenominazione delle variabili

Le variabili vengono ridenominate da A0 ad AZ fino a Z0....ZZ accodando l'eventuale identificatore a secondo dei tipi [% $ o ( ].
Verrà fatto un dimensionamento delle variabili array e record secondo il seguente schema:

i) array
   Dato che è possibile dichiarare gli estremi dell'array, si dovrà dato ad esempio un
   ```
   pippo=array[8..15] of real;
   ```
   effettuare un dimensionamento del genere(se a pippo corrisponde a0):
   ```
   dim a0(7)
   ```
   Più in generale se a,b sono gli estremi dell'array farò un DIM su b-a.

ii) record
   Non c'è bisogno di dimensionamenti: basterà, una volta dichiarata una variabile semplice di tipo record, riscriverla tante volte quanti sono i campi del record.

iii) array di record
   Bisognerà riscrivere le variabili tante volte quanti sono i campi del record e ridenominarle dimensionandole con i valori della matrice di partenza tenendo conto del genere del singolo campo.

Ad esempio se avessi

```
type
    pippo=record
            c1=string
            c2=integer
            c3=real
          endrecord

var
   pluto:array[1..10]of pippo
```

si scriverà pluto così:

```
   pluto.c1
   pluto.c2
   pluto.c3
```

effettuando tre dimensionamenti del tipo

```
   dim a0$(9),a1%(9),a2(9)
```

se a0,a1,a2 sono i nomi delle variabili ridenominate.

4.1.4 Traduzione e diagnostica

La traduzione del textfile ERRE avverrà in almeno due passate :

i) trascrizione delle variabili usando i nuovi nomi; nel far ciò si toglieranno tutte quelle parti non indispensabili per la traduzione in BASIC quali commenti, parti dichiarative ecc..

ii) traduzione delle strutture di controllo, array, procedure,function usando i soliti schemi già applicati nella versione 2.2

Per facilitare la comprensione di quanto è stato esposto, nella Appendice due verrà "tradotto" un programma ERRE.

APPENDICE UNO

Elenco delle parole chiave di ERRE

AND	FOR	PROGRAM
ARRAY	FUNCTION	READ
BEGIN	GET	REAL
BOOLEAN	IF	RECORD
CASE	INCLUDE	REPEAT
CLOSE	INPUT	STEP
DATA	INTEGER	STRING
ELSE	NEXT	SYS
ENDCASE	NOT	SWAP
ENDFUNCTION	OF	THEN
ENDIF	ON	TO
ENDPROCEDURE	OPEN	TRUE
ENDRECORD	OR	TYPE
ENDWHILE	OTHERWISE	UNTIL
END.	POKE	VAR
END←	PRINT	WHILE
FALSE	PROCEDURE	←

Elenco degli identificatori standard di ERRE

ABS	INT	SPC
ASC	LEFT	SQR
AT	LEN	STATUS
ATN	LOG	STR
CDOWN	MID	TAB
CHR	PAUSE	TAN
CLEFT	PEEK	TIME
CRIGHT	RIGHT	USR
CUP	RND	VAL
COS	SGN	
EXP	SIN	

APPENDICE DUE

Esempio di traduzione di un programma ERRE

In questa appendice vedremo un esempio di traduzione, usando i concetti esposti nel paragrafo 4, di un programma relativamente semplice.
Il programma è il seguente:

```
program numbers
!
! legge, ordina per numero telefonico e stampa l'elenco
!
data
  5
  JOHNSTON R.L.,53 JONSTON CRES.,491-6405
  KEAST P.,77 KREDLE HAVEN DR.,439-7216
  LIPSON J.D.,15 WEEDWOOD ROAD,787-8515
  MATHON R.A.,666 REGINA AVE.,962-8885
  CRAWFORD C.R.,39 THEATERSON AVE.,922-7999
type
  customertype=
  record
    name:string
    address:string
    phonenumber=string
  endrecord
var
  workspace:customertype
  customer:array[1..25]of customertype
  i,j,numberofrecords:integer
procedure sort(← numberofrecords:integer)
!
! ordina i record con i numeri telefonici
!
var
  i,j:integer
 for i=1 to numberofrecords-1
   for j=1 to numbersofrecords-i
     if customer[j].phonenumber > customer[j+1].phonenumber
       then
         swap(customer[j].name,customer[j+1].name)
         swap(customer[j].address,customer[j+1].address)
         swap(customer[j].phonenumber,customer[j+1].phonenumber)
     endif
   next
 next
end procedure
begin
!
```

```
! inizia il programma principale

read(numberofrecords)
for i=1 to numberofrecords
  read(workspace.name)
  read(workspace.address)
  read(workspace.phonenumber)
  customer[i].name=workspace.name
  customer[i].address=workspace.address
  customer[i].phonenumber=workspace.phonenumber
next
sort(← numberofrecords)
!
! stampa l'array ordinato
!
  for i=1 to numberofrecords
    print(customer[i].phonenumber);
    print(customer[i].name,"   ");
    print(customer[i].address)
  next
end.
```

Come già visto il primo passo consiste nella creazione delle tabelle dei tipi e delle variabili: queste si ottengono isolando le parti dichiarative relative al programma principale e alle eventuali procedure.
Otterremo nel nostro caso

TABELLA DEI TIPI

```
    1       INTEGER       0 |
    2       BOOLEAN       0 | presenti per
    3       STRING        0 | default
    4       REAL          0 |
    5       CUSTOMERTYPE  6
    6       RECORD        7
    7       .NAME         3
    8       .ADDRESS      3
    9       .PHONENUMBER  3
   10       ENDRECORD     3
   11       ARRAY[1..25]  5

    ^          ^          ^
 indice       nome tipo puntatore
```

e per ciò che riguarda la tabella della variabili:

TABELLA DELLE VARIABILI

```
        WORKSPACE       5       |  puntano
        CUSTOMER        11      |  al campo
        I               1       |  indice
        J               1       |  della
        NUMBEROFRECORDS 1       |  tabella
    5   NUMBEROFRECORDS 1       |  dei tipi.
        I               1       |
        J               1       |
```

In questo caso 5 si riferisce all'inizio delle variabili della procedura SORT nella traduzione si terrà conto di questa fatto.
A questo punto bisogna contare le variabili effettive contenute nel programma, tenendo conto dei puntatori si ricava:

```
     WORKSPACE.NAME                 a0$
     WORKSPACE.ADDRESS              a1$
     WORKSPACE.PHONENUMBER          a2$
     CUSTOMER[.NAME          1..25  a3$
     CUSTOMER[.ADDRESS       1..25  a4$
     CUSTOMER[.PHONENUMBER   1..25  a5$
     I                              a6%
     J                              a7%
     NUMBEROFRECORDS                a8%
  10 NUMBEROFRECORDS I              a9%
     I                              aa%
     J                              ab%
```

Fatto ciò si ridenominano le variabili (già fatto nella tabella superiore) e si iniziano a togliere dal programma ERRE tutte le parti inutili) cosicché resta :

```
program number
data ..............
     ..............
     ..............
     ..............
     ..............
     ..............
dim a3$[24],a4$[24,a5$[24]
procedure sort(←a9%)
for aa%=1 to a9%-1
  for ab%=1 to a9%-aa%

   ...................
endprocedure
begin
for a6%=1 to a8%
  ...................
next
sort(←a8%)
```

```
  ..................
end.
```

Da qui in poi si procederà al controllo della semantica del programma per poi passare alla traduzione delle strutture di controllo e deg1i statement generali.
Un errore nella fase di esame dei tipi bloccherà il resto della compilazione, si provvederà più avanti a dare delle limitazioni al numero massimo dei campi di un record, al massimo numero di label di un case e al massimo numero di nesting permessi per le strutture di controllo.

INDICE