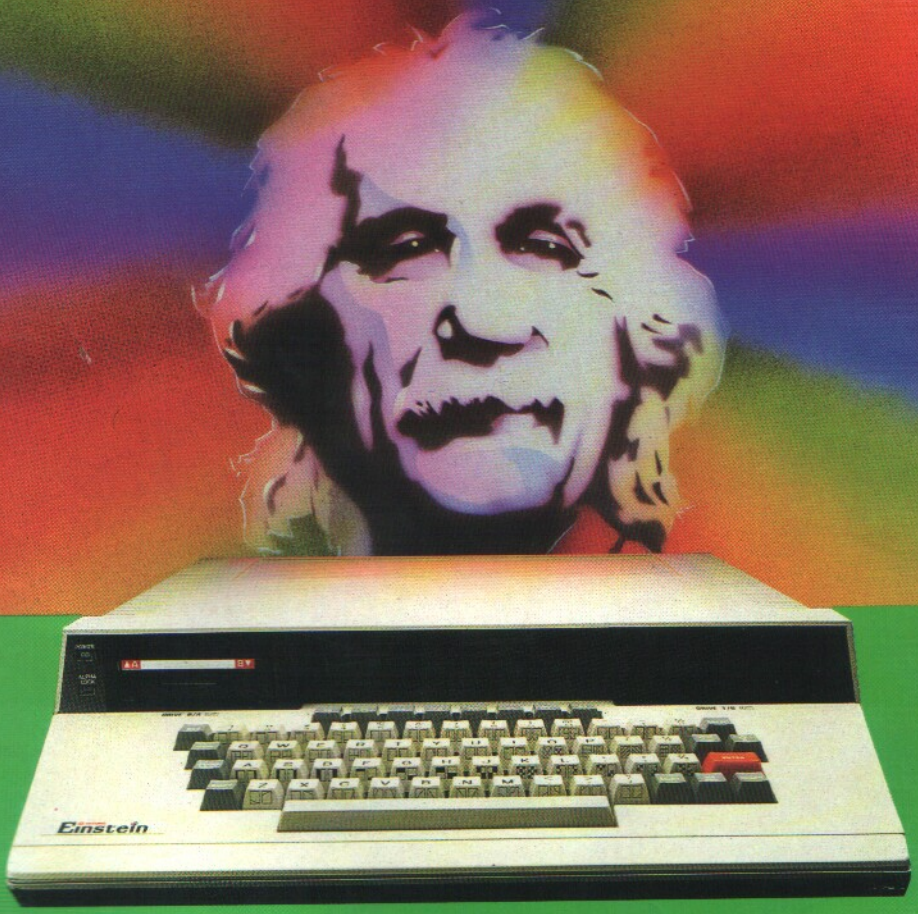
BASIC Reference Manual



TATUNG

Einstein

COLOUR MICRO COMPUTER

Acknowledgement

Written By: Alan Stancliffe
Edited By: R.M. Clarke

Our grateful thanks to Crystal Research Limited of Torquay, for their kind permission to reproduce extracts from their manuals, and for their help in checking this publication; and to the General Instrument Corporation for their kind permission to reproduce information relating to the Programmable Sound Generator.

ISBN:1-85086-003-3. 1st Edition 1984
ISBN:
ISBN:

Tatung (UK) Ltd.,
Computer Division,
BRIDGNORTH,
Shropshire WV15 6BQ

First Printed in 1984 by:
Spellman Walker Ltd.,
Bradford, West Yorkshire
ENGLAND.

CONTENTS

PAGE

APPENDICES

INTRODUCTION

TATUNG/Xtal BASIC 4 is based upon Xtal BASIC 3.0 which has been extensively enhanced to provide extra facilities for the user.

One of the outstanding features of this BASIC is the ability to create **user defined** reserved words and **user defined** error messages with the aid of machine code programming. The interpreter can be expanded by writing appropriate sub-routines and by inserting your own defined words in an auxiliary reserved word table to give the type of BASIC most suited to your particular requirements.

The Editor provides a wide range of facilities which are not complex to use, and are designed to make the editing process much easier than with most other systems. There are also extensive screen formatting facilities available to the user.

TATUNG/Xtal BASIC 4 offers many capabilities which are normally regarded by other BASICS as enhancements to be purchased separately by the user. For example, the advanced graphics which make use of sprites, the fully programmable sound generator, and the file handling commands.

This manual describes the facilities offered by TATUNG/Xtal BASIC 4 and includes a list, in alphabetical order, describing in detail the functions of all the reserved words.

1

MODES OF OPERATION

There are two modes of operation in BASIC:

i) DIRECT MODE
ii) DEFERRED MODE.

DIRECT MODE

In direct mode, BASIC commands and statements are:-

a) NOT preceded by line numbers.
b) Executed immediately they are entered.

This mode is useful for debugging and for using BASIC as a "calculator" involving computations with arithmetic and logical operators.

In other words there is an **immediate** response to the instructions typed into the computer.

DEFERRED MODE

This is used for entering programs. Program lines are preceded by a number and stored in memory. Execution of a program is invoked by use of the RUN, CHAIN and GOTO commands.

Line numbers may range between 1 and 65535 and are selected arbitrarily by the user. It is recommended that reasonable gaps be left between line numbers (e.g. 10) to facilitate the insertion of extra lines if they become necessary later.

Programs can begin with any line number but the first line to be interpreted will always be the lowest line number entered.

2

CHARACTER SET

TATUNG/Xtal BASIC 4 character set which consists of:-

i) alphabetic characters
ii) numeric characters
iii) graphics characters
iv) selection of punctuation and other symbols normally found on a typewriter keyboard.

The normal keyboard characters are listed below with their corresponding ASCII code. Graphics characters will be found in the full table given in Appendix D of the Introduction Manual.

ASCII CODE	CHAR.	HEX	ASCII CODE	CHAR.	HEX	ASCII CODE	CHAR.	HEX
32	SP	20	52	4	34	72	H	48
33	!	21	53	5	35	73	I	49
34	"	22	54	6	36	74	J	4A
35	#	23	55	7	37	75	K	4B
36	$	24	56	8	38	76	L	4C
37	%	25	57	9	39	77	M	4D
38	&	26	58	:	3A	78	N	4E
39	'	27	59	;	3B	79	O	4F
40	(	28	60	<	3C	80	P	50
41	)	29	61	=	3D	81	Q	51
42	*	2A	62	>	3E	82	R	52
43	+	2B	63	?	3F	83	S	53
44	,	2C	64	@	40	84	T	54
45	-	2D	65	A	41	85	U	55
46	.	2E	66	B	42	86	V	56
47	/	2F	67	C	43	87	W	57
48	0	30	68	D	44	88	X	58
49	1	31	69	E	45	89	Y	59
50	2	32	70	F	46	90	Z	5A
51	3	33	71	G	47	91	←	5B

ASCII CODE	CHAR.	HEX	ASCII CODE	CHAR.	HEX	ASCII CODE	CHAR.	HEX
92	½	5C	104	h	68	116	t	74
93	→	5D	105	i	69	117	u	75
94	↑	5E	106	j	6A	118	v	76
95	←	5F	107	k	6B	119	w	77
96	£	60	108	l	6C	120	x	78
97	a	61	109	m	6D	121	y	79
98	b	62	110	n	6E	122	z	7A
99	c	63	111	o	6F	123	¼	7B
100	d	64	112	p	70	124	‖	7C
101	e	65	113	q	71	125	¾	7D
102	f	66	114	r	72	126	÷	7E
103	g	67	115	s	73			

3

NUMBERS AND STRINGS

There are two types of data used in TATUNG/Xtal BASIC 4:-

i) NUMERIC DATA
ii) STRING DATA

NUMERIC DATA

These can be whole numbers (integers), or floating point numbers (reals).

Integers
Integers are whole numbers without fractions or decimal points.

The number can be positive or negative and must fall within the range -32768 to +32767. (BASIC supports 16 bit integers).

EXAMPLES: 44, 6, -17, 32500, -29076

Numbers in the ranges -65535 to -32769, and 32768 to 65535, may be accepted by integer variables. In these cases the values are internally converted to fall in the range 1 to 32767 and -32768 to -1 respectively (otherwise 17 bits would be needed to store each number). Integers, in BASIC, are shown by the % sign after the variable name

Floating-Point Numbers

As the name suggests, "floating point" numbers can be **whole** numbers, or **fractional** numbers.

They can be **positive** or **negative** and if no sign is given the number is assumed to be positive.

The following are examples of floating point numbers which are also equivalent to integers.

6 -14 2650 -21

The following examples include decimal points.

7530.23 0.7 7.4 -0.0008 -27.029

Commas must NOT be used in numbers otherwise a SYNTAX ERROR MESSAGE may be given.

For storage purposes, floating point numbers are internally converted to SCIENTIFIC NOTATION.

Scientific Notation

A number in scientific notation is expressed as a BASE NUMBER (MANTISSA) multiplied by 10 raised to a particular POWER of 10 (EXPONENT)

NUMBER = MANTISSA x $10^{EXPONENT}$ (Power of 10)

EXAMPLE: 3113 = 3.113 x 10^{3}

In BASIC this "scientific notation" is as follows:-

3.265 x 10^{4} is entered as 3.265E4 in BASIC

For a NEGATIVE power:-

3.265 x 10^{-4} is entered as 3.265E-4 in BASIC

By using Scientific Notation very large and very small numbers are easily handled by the computer.

The exponent range is given below:-

MAXIMUM - 1.701411E38
MINIMUM - 0.940396E-38

Any computations yielding a result above or below these values will display a quantity error.

Four bytes are used to store numbers internally, one of which represent the "signed Exponent" and the other three the "signed mantissa".

The "exponent" can range from -38 to +38 whilst the signed mantissa can be up to **seven digits,** (any value outside this range will cause a quantity error to be displayed).

The full seven digits of a mantissa are used internally for calculations but are then rounded off to six significant figures for output. (Always use a seventh figure if known for accuracy even though only six are displayed).

Leading and trailing zeros are always suppressed on output. This avoids long trails of zeros either preceding or following a number.

In practice, very large numbers or very small numbers can be entered in "decimal" or "scientific notation" as required, but may be automatically output in scientific notation only.

Hexadecimal (HEX) Numbers

Hexadecimal numbers are to base 16, and are expressed using a combination of numbers in the range 0 to 9 and letters in the range A to F. The following table gives the Hexadecimal characters with decimal number equivalents.

HEXADECIMAL	DECIMAL	HEXADECIMAL	DECIMAL
0	0	8	8
1	1	9	9
2	2	A	10
3	3	B	11
4	4	C	12
5	5	D	13
6	6	E	14
7	7	F	15

EXAMPLE: HEX A7 is equivalent to 167 decimal.

Evaluation

$$7 \times \text{units} = 7$$

$$A \times 16^1 = \underline{160} \text{ (ie } 10 \times 16\text{)}$$

$$\text{Total } \underline{167}$$

Using HEX numbers:

1) An ampersand symbol (&) is used as a prefix to indicate Hexadecimal numbers.
   e.g: &1298 &A7 &1F34

2) When hexadecimal numbers are used in numeric expressions they are internally converted to a decimal number. The hexadecimal number must not exceed four digits. If more than four digits are entered only the LAST FOUR will be used.

EXAMPLE:
&1F34 equivalent to 7988 decimal
&71F34 still equivalent to 7988 decimal because only the LAST FOUR digits are used. (i.e. the 7 is ignored).

STRING DATA

Strings are combinations of ASCII characters representing letters, numbers, and symbols.

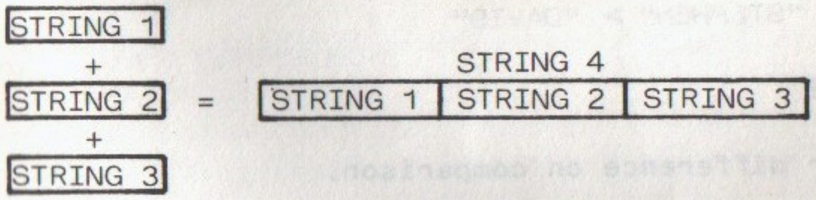
They are useful for storing names, titles, and text, but can also be used to hold numeric values.

A string can be any combination of up to 255 characters, usually shown in quotes.

EXAMPLE:
```
- "HELPLESS"
  "*!ZK"
  "1379.76" etc.
```

Concatenation of Strings

Strings can be CONCATENATED (i.e. strung together consecutively) using the "+" sign.



EXAMPLE:
```
10 A$  = "MOUSE"
20 B$  = "TRAP"
30 C$  = A$ + B$
40 PRINT C$
RUN
```

This will give a display of MOUSETRAP.

Strings and Relational Operators

Strings can be compared using relational operators:-

=,<>,<=,>=,>,<

EXAMPLES:

```
A$ > B$
"SMIT" < "SMITE"
"MOUSETRAP" = "MOUSETRAP"
```

The comparison is done character by character until a position is found in which the two differ. The "greater" string is the one whose character has the greater ASCII code. If no differences are found, but one string is longer than the other, the longer string is considered to be the greater.

EXAMPLE 1

Character difference on comparison.

```
"STEPHEN"     "DAVID"
    ↑             ↑
ASCII 83      ASCII 68
```

∴ "STEPHEN" is greater than "DAVID"

"STEPHEN" > "DAVID"

EXAMPLE 2

Character difference on comparison.

```
"ANDY"       "ANDREW"
   ↑             ↑
ASCII 89    ASCII 82
```

∴ "ANDY" is greater than "ANDREW"

"ANDY" > "ANDREW"

EXAMPLE 3

No character difference detected but one string longer.

"MOUSE" "MOUSETRAP"

∴ "MOUSETRAP" is greater than "MOUSE"

MOUSETRAP > MOUSE

4

VARIABLES

Variables are "names" used to represent values. The values are either assigned by the programmer, or as a result of calculations/operations within the program. Prior to being assigned a value, a variable is assumed to be zero.

A variable "name" can be a combination of letters or letters and numbers but the **first** character must **always** be a letter.

EXAMPLES:

VALID NAMES	INVALID NAMES
ZFBC	6BLF
SAP	*YXL
L93A7	!79B

Variables may be of the following types:-

NUMERIC - FLOATING POINT - holds numbers
NUMERIC - INTEGER - holds whole numbers.
STRING - holds strings.

Variable "types" are indicated by characters which suffix the variable "name".

i) Floating point variables are the default type, having no suffix after the variable name.

EXAMPLE: ATC

ii) Integer variables are indicated by the presence of a % sign after the variable name.

EXAMPLE: BX7%

iii) A string variable is indicated by the presence of a $ sign immediately following the variable name.

EXAMPLE: ZUP$

TATUNG/Xtal BASIC 4 uses the first **five** characters only of a variable name. More can be entered (within the limits of the maximum line length) however, characters after the 5th will be ingored by BASIC.

EXAMPLES:

AB123$
SLOPY$
SLOPYXZ$

All three are valid variable names but the BASIC would not distinguish between the 2nd and 3rd example because the first five characters of each are identical.

Care must be taken to ensure that variable names do not contain reserved words!

EXAMPLES:

TONE,LETTERS,PINCH,TERROR

All the above examples contain reserved words, as indicated below, and could cause problems!

TOne,LETter,pINCH,tERROR

It is advisable to keep variable names to two characters to avoid this problem, and commonly only one letter is used. (the only two character reserved words are IF,OR,LN,TO,ON,FN,PI)

5

ARRAYS

An ARRAY is in effect a list or table full of variables. There are NUMERIC arrays and STRING arrays.

In the case of a list each element of the list is numbered in order of appearance. The numbers are then used to refer to individual elements by inserting them immediately after the variable name.

EXAMPLE: A(2)

This refers to the 3rd element of the array variable A. (Array subscripts start at 0)

This is known as SUBSCRIPTING the array, the numbers, or variables, within the brackets being the subscripts.

Lists have a single subscript and are known as one dimension arrays.

In the case of tables there would be two subscripts referencing individual elements in a similar manner to map or graph co-ordinates. These are known as two dimensional arrays.

EXAMPLE: B(2,3)

The numbers indicate vertical column and horizontal row numbers relative to the element required.

TATUNG/Xtal BASIC 4 also caters for 3 dimensional arrays thereby giving 3 subscripts after the array name.

EXAMPLE: B(2,0,1)

Arrays in higher dimensions can be supported by TATUNG/Xtal BASIC 4, the limit being determined by the amount of memory available at any one time.

NOTE:

i) The array subscripts always number from zero.

ii) Arrays containing more than 10 elements must be dimensioned with a DIM statement to inform the BASIC how much space to allocate for it. DIM statements are explained in detail in a later section of this manual. (Page 61).

iii) If you are unfamiliar with arrays refer to the details given in the INTRODUCTORY MANUAL.

6

EXPRESSIONS AND OPERATORS

EXPRESSIONS

Expressions consist of:-

i) Numeric Variables
ii) Numeric Data
iii) String Variables
iv) String Data

These can be combined using arithmetic, logical and relational operators.

ARITHMETIC OPERATORS

The arithmetic operators in order of precedence are as follows:-

OPERATOR	OPERATION	EXAMPLE	MATHEMATICAL EXPRESSION
( )	Parenthesis	3*(2+6)	3(2+6)
↑	Exponentiation (raise to power)	5↑2	5^2
*	Multiply	3*4	3x4
/	Divide	4/2	4÷2
MOD	Remainder	8MOD3	8÷3=2 rem.2
+	Addition	3+2	3+2
-	Subtraction	2-1	2-1

EXAMPLE:

BASIC	MATHS
6*(2+6)/2↑3-5	$\frac{6(2+6)}{2^3}-5$

The order of operation in this expression is as follows:-

i) Brackets evaluation
ii) Powers evaluation
iii) Multiplication and Division evaluation
iv) Subtraction evaluation

The operators are fairly self explanatory as they relate directly to normal arithmetic functions but the MOD operator requires further explanation.

MOD accounts for the remainder after Division -

EXAMPLE:

5MOD3 would return a value of 2 (i.e. 5÷3=1 remainder 2)
7MOD2 would return a value of 1 (ie. 7÷2=3 remainder 1)
8MOD3 would return a value of 2 (ie. 8÷3=2 remainder 2)

The actual process undertaken to obtain these values is a follows:-

```
FOR ANY TWO VALUES 'X' and 'Y'
THEN XMODY = X-Y*INT(X/Y)
```

This section returns the largest integer (whole number) **less** than **or equal** to the result of the division computation.

Examples of MOD:

1) For a value of X=5 and Y=3

```
5MOD3 = 5-3*INT(5/3)
      = 5-3*INT(1.6666)
      = 5-3*1
      .·.  5MOD3 = 2
```

2) For a value of X=7 and Y=2

```
7MOD2 = 7-2*INT(7/2)
      = 7-2*INT(3.5)
      = 7-2*3
      .·.  7MOD2 = 1
```

If the values of 'X' are negative then because of the INT function the results are sometimes unexpected. Using the same examples as above but with X=-5, and -7 respectively we would see the following:-

1)
```
-5MOD3 = -5-3*INT(-5/3)
       = -5-3*INT(-1.6666)
       = -5-3*(-2)
       = -5-(-6)
       .·.  -5MOD3 = 1
```

2)
```
-7MOD2 = -7-2*INT(-7/2)
       = -7-2*INT(-3.5)
       = -7-2*(-4)
       = -7-(-8)
       .·.  -7MOD2 = 1
```

Thus we can see the following comparison:-

```
5MOD3 = 2 BUT -5MOD3 = 1
7MOD2 = 1 AND -7MOD2 = 1
```


RELATIONAL OPERATORS

Relational operators are commonly used with IF statements (Page 102) for comparisons and the evaluation of conditions. TATUNG/Xtal BASIC 4 uses the following:-

> greater than **>=** greater than or equal to
< less than **<=** less than or equal to
= equal to **<>** not equal to

LOGICAL OPERATORS

Again logical operators are commonly used with IF statements (Page 102) in conjunction with relational operators. TATUNG/Xtal BASIC 4 contains the following (listed in descending order of precedence).

NOT, AND, OR, XOR (Exclusive-OR)

EXAMPLE: 10 IF(X+Y-Z)**>**3 **AND** Y**<=**20 THEN 100

This performs a bit-by-bit logical AND of each numeric expression. If both bits are set to a '1' then the result bit is set to a '1'. If either bit is a '0' the result bit is set to '0'.

NOTE: Although expressions involving relational and logical operators are normally used within IF statements, they can also be used within arithmetic expressions (a relational expression returns a value of -1 if it is TRUE, and 0 if FALSE).

In some cases, quite a lot of space can be saved:-

EXAMPLE: IF X **>**15 THEN A=0:ELSE A=1

This could be replaced by:- A = -(X**>**15)

7

THE EDITOR

SCREEN CONTROL CODES

The control key (CTRL), is used in conjunction with various other character keys to provide the following facilities which assist output to the screen.

1) **CTRL-J** Line Feed (LF)

This will create a "line feed" (move to the next line). In other words the cursor moves down one line. The repeat facility on the function is activated if the keys are held down.

When the bottom of the screen is reached, the display will "scroll up" one line at a time. This duplicates the "cursor down" key.

2) **CTRL-L** Cursor Home and Clear Screen

This will "clear" the screen and move the cursor to the "Home" position (top left hand corner of the screen).

3) **CTRL-M** Carriage Return

This will operate a "carriage return" **with** a "line feed" (i.e. move the cursor to the beginning of the next line). This duplicates the ENTER key.

4) **CTRL-A** Screen Dump to Printer

Will cause a transfer of the screen display contents to a printer (i.e. make a hard copy on paper).

5) **CTRL-H** Backspace (BS)

This will simply move the cursor to the LEFT one character space at a time. The function will "repeat" if the keys are held down. This duplicates the "cursor left" key.

6) **CTRL-D** Horizontal Tabulations (HT)

This will move the cursor to the RIGHT one character space at a time. The function will "repeat" if the keys are held down. This duplicates the "cursor right" key.

7) **CTRL-I** TAB

This will move the cursor and display to the RIGHT by TEN "spaces" each time. (10 being the Print Zone value for tabbing). Again, the function will "repeat" if the keys are held down.

8) **CTRL-K** Vertical Tabulation (VT)

This will move the cursor UP one line at a time and there is a "repeat" function activated if the keys are held down. This duplicates the "cursor up" key.

9) **CTRL-T** Cursor OFF

This will turn the cursor off, should this occasionally be required.

10) **CTRL-Q** Cursor ON

This will turn the cursor back on again as a reversal of the CTRL-T function.

11) **CTRL-R** Printer Screen Echo ON

This causes anything output to the screen to be echoed to the printer.

12) **CTRL-S** Printer Screen Echo OFF

This turns the printer screen echo off reversing the effect of CTRL-R.

13) **CTRL-↑** Cursor Home

This returns the cursor to the "home" position (top left hand corner of the screen).

BASIC EDITOR

This facility is available from the moment that you enter BASIC and the following general points relating to it should be noted.

It has been specifically designed to make debugging and editing more versatile than with most other BASIC EDITORS.

Input lines can be up to 126 characters long and may occupy one or more rows on the screen. BASIC keeps a note, at all times, of the start and end of each line.

If there are several lines in a listing, the cursor may be moved to any particular line on the screen in order to make modifications, regardless of the number of rows the line occupies.

If a line is extended so that it apparently will run into the next one, the lines below simply move down one row to accommodate it.

A "modified" line will need to be entered into a program by pressing the ENTER key while the cursor is located on one of the rows of the screen containing that line. The cursor then moves to the beginning of the next line (NOT row).

The functions available within the Basic Editor are the INS/DEL key, the CURSOR CONTROL KEYS, the SCREEN CONTROL CODES (previously described), and the following additional control codes:-

1) **DELETE CHARACTER** - at the Cursor

CTRL-F

or

CTRL-DEL

Either of these combinations will delete a character at the cursor, moving the remainder of the line one character space to the left.

2) **ERASE WHOLE LINE**

CTRL-X

This returns the cursor to the beginning of a line and then erases the entire line

3) ERASE TO END OF LINE

CTRL-U

This will erase to the end of a line from the current cursor position (regardless of the number of rows the line occupies).

4) ERASE TO END OF SCREEN

CTRL-V

This will erase to the end of the screen from the current cursor position.

5) DELETE CHARACTER - to the left of Cursor

CTRL-Y

or

DEL

This deletes the character to the left of the cursor, moving the remainder of the line one character space to the left. When the cursor is at the end of a line, deletion acts like a "rub out" to the **left** of the cursor.

LINE EDITOR

The "line edit" mode is available primarily as an alternative for use with programs where the "screen editor" might not be quite so convenient.

In "line edit" mode the only editing functions which operate are as follows:-

a) The "cursor left" function on the cursor control key acts as a "back space" and deletes characters as it moves. All other functions of cursor control keys are non-operational.

b) CTRL-A transfers the screen contents to a printer.

c) Re-typing of a program line so as to overwrite the original line.

All other control functions and cursor movements as described in "SCREEN CONTROL CODES" and "BASIC EDITOR" are non-operational under LINE EDIT mode.
To use the LINE EDITOR carry out the following procedure:-

1) Type IOM 2,0

This activates an internal "switch" which then allows the option of selecting either LINE EDIT or SCREEN EDIT.

2) Type IOM 0,0

This now sets the "switch" to LINE EDIT mode (i.e. goes into line edit)

3) To return to SCREEN EDIT type IOM 0,1.

When line edit is no longer required IOM 2,1 is used to de-activate the internal "switch".

NOTE:

i) During "line edit" the prompt in front of the cursor becomes a horizontal arrow (→) in direct mode.

ii) The arrow changes to a question mark (?) if an INPUT statement is used without a specified "prompt string".

8

SYSTEM "COMMAND KEYS"

CTRL-BREAK

Simultaneously pressing CONTROL and BREAK keys will cause a transfer to the disc operating system from BASIC.

SHIFT-BREAK

Holding down the SHIFT key and then pressing BREAK will halt program execution, preserving all variables. The CONT command can then be used to allow continuation of program execution if required. The message "Break in line..." is displayed on the screen when SHIFT-BREAK is used.

BREAK

The BREAK key will halt program execution whilst it is held down, but execution continues when the key is released. (Variables are preserved)

ESC

Pressing the ESCAPE key causes listings and tabulations to be aborted.

9

DEVICES AND I/O ASSIGNMENT

I/O is an abbreviation for INPUT/OUTPUT

Special forms of the INPUT and PRINT statements allow the user to assign different I/O devices to the system. (Details of these statements are given on Page 113)

Examples of I/O devices are:-

PRINTERS
SERIAL OR PARALLEL DEVICES
DISCS

Each device is assigned a DEVICE NUMBER in the range 0 to 254. Thus there can be 255 "output devices" and 255 "input devices".

Input and output can also be handled to and from files which are stored on Disc. All file I/O is handled through device 255 which is assigned internally and more information is given on this in a later section.

DEVICE ASSIGNMENT

Three devices are currently assigned under the BASIC as supplied and are as follows:-

DEVICE #	OUTPUT	INPUT
0	Screen	Keyboard
1	Printer	N/A
2	Serial (RS232)	Serial (RS232)

NOTE: Device 0 is the only one which utilises the "Screen Editor".

10

BASIC RESERVED WORDS

This chapter contains all the reserved words used in TATUNG/Xtal BASIC 4. They are listed in alphabetic order for ease of reference.

Each reserved word is given at the top of a page and then further explained under the following headings.

Syntax:

Indicates the "grammatical structure" of the particular word when used in BASIC.

The following notations have been adopted in this list which is not intended to be exhaustive but covers most of the BASIC syntax requirements. Other symbols have been used where these are more meaningful or where the parameters used are restricted to particular values; these exceptions are explained more fully in the individual commands to which they refer.

1) **< >** encloses the syntax requirements where the information required does not easily fall into one of the categories listed below. The chevrons are not included in the text entered.

2) **J** indicates an expression which must evaluate to a number in the range of 0 to 255. If the result of the expression is not an integer, then it is rounded down to the nearest integer value.

3) **I** as above (J) but the range is increased to 0 to 65535, or -32768 to 32767.

4) **N** a numeric expression which evaluates to a real number (i.e. not necessarily an integer).

5) **V** a variable name which can be either a numeric or string variable.

6) file a valid file name including (optional) drive number, filename, and (optional) file descriptor. For full description of valid file names see page 264.

Purpose: Details the function of the word within BASIC and offers any necessary explanation to aid further understanding of how the word is used.

Examples:

Where appropriate examples are given.

Related Reserved Words:

Lists other reserved words used in conjunction with the given word, or of a similar type.

ABS (Absolute value)

Syntax: ABS (N)

Where N is any numeric expression.

Purpose: The ABS function returns the **absolute** value of the numeric expression, i.e. ignores signs, always returning positive values.

Example

```
X=ABS (-3.14159)
```

This will return a value of 3.14159 in X.

Related Keyword : SGN

ADC

ADC (Analogue Digital Converter)

Syntax: ADC (J)

Where J is a "channel number" given as 0,1,2, or 3.

Purpose: This is a function which reads the value at the Analogue port for the channel specified in J.

Channels 0 and 1 relate to Analogue 1 socket.
Channels 2 and 3 relate to Analogue 2 socket.

The value returned from each channel is in the range 0 to 255.

The most common application of this function is to incorporate joy-stick control into a program. The values given to channels 0 and 2 determine horizontal movement, and channels 1 and 3 vertical movement for joy-sticks connected to the respective ports (Analogue 1, Analogue 2).

EXAMPLES:

1)
```
A = ADC(0)
PRINT A
```

This will read the current value at the analogue 1 port for channel 0, and place it in A. The value can then be output to the screen by the PRINT statement.

2) The following program can be used with a joy-stick control connected to the Analogue 1 port.

```
10 X = ADC(0)
20 Y = ADC(1)*3/4
30 DRAW TO X,Y
40 GOTO 10
```

When RUN, this program will draw lines on the screen corresponding to the movement of the joystick.

Related Keywords: BTN

AND

AND

Syntax: <statement> **AND** <statement>

Purpose: This is a LOGICAL OPERATOR used in the evaluation/comparison of statements and/or numeric expressions.

EXAMPLES:

```
10 IF (x + y)<>3  AND y = 20 THEN 100
```

This will transfer program execution to line 100 if (x + y) is not equal to 3 and at the same time y = 20

```
20 A = 15 AND 7
```

This returns a value A = 7.

```
30 K = INCH AND &DF
```

This returns upper case ASCII whether upper or lower case entered.

Related Keywords: ELSE IF NOT OR THEN XOR

APPEND

APPEND

Syntax: APPEND <file>, SV

<file> must be a legal "file name" as described in the section on File-Handling (Page 264).

SV is a string variable name (but **not** a string array element), and is the "file descriptor".

Purpose: This is a File-Handling Command which is used to write extra information at the end of a "sequential file", when to OPEN the file and read up to the end would be very inefficient.

It is similar to OPEN (Page 163), the difference being that the internal file pointer moves to the end of a file instead of the beginning, and no record length is specified.

If the file specified by <file> does not exist on the disc then a NO FILE ERROR will be given.

Example:

```
APPEND "O:SILLY.DAT",FD$
```

This will perform the following:-

a) opens the file SILLY.DAT on the disc currently in drive 0, and moves the pointer to the end of the file so data may be added.

b) assigns FD$ as the file descriptor.

Related Keywords : CREATE CLOSE OPEN

ASC

ASC (American Standard Code For Information Interchange - ASCII)

Syntax: ASC (<string expression>)

Purpose: This is a Standard String Function which returns the ASCII value (in decimal) of the **first** character of the string given in the function.

To display the values given by this function use the PRINT command as a prefix.

EXAMPLE:

```
X=ASC ("ABC")
```

This will return a value of 65 in X. X contains the ASCII value of A (i.e. the first character of the string).

65 is the decimal code for A.

Related Keyword : CHR$ STR$

ATN

ATN (Arctangent)

Syntax: ATN (N)

Where N is a number or a numeric expression.

Purpose: This is a Standard Function which returns the arctangent of N, in radians, within the range - PI/2 to +PI/2

EXAMPLE:

```
X=ATN (1)
```

Returns a value of 0.785398 radians in X. (i.e. PI/4 radians or 45°)

Other Transcendental Functions:

```
ASN(X)=ATN(X/SQR(1-X*X))    arcsin(x)
ACS(X)=(PI/2)-ASN(X)        arccos(x)
HCS(X)=(EXP(X)+EXP(-x))/2   cosh(x)
HSN(X)=(EXP(X)-EXP(-X))/2   sinh(x)
HTN(X)=1-2/(1+ESP(X*2))     tanh(x)
```

Related Keywords : COS SIN TAN

AUTO

AUTO (Automatic Line Numbering)

Syntax: AUTO L1,L2

L1 is the line number from which automatic numbering is to commence.

L2 is an increment value for the numbers to be used in the automatic numbering sequence.

Both L1 and L2 will default to a value of 10 if they are not stated.

Purpose: This is a System Command which gives automatic line numbering while entering a program.

EXAMPLES:

AUTO 100,5	Starts at line 100 and continues 105,110,115,etc.
AUTO 100	Starts at line 100 and continues 110,120,130,etc.
AUTO,20	Starts at line 10 and continues 30,50,70,etc.
AUTO	Starts at line 10 and continues 20,30,40,50,etc.

When AUTO has been invoked, the next line number automatically appears for the user to continue after the ENTER key has been pressed.

Each number is displayed just as if it had been typed from the keyboard.

The automatic line numbering may be abandoned by deleting the current line number which AUTO has produced.

The "editing mode" is not affected by the use of AUTO.

Related Keywords:

BCOL

BCOL (Backdrop colour)

Syntax: BCOL X

X can be a value from 0 to 15, each number representing a particular colour as listed below.

Purpose: This is a Display Command which sets the backdrop colour according to the value of X.

X	Colour	X	Colour
0	Transparent	8	Medium Red
1	Black	9	Light Red
2	Medium Green	10	Dark Yellow
3	Light Green	11	Light Yellow
4	Dark Blue	12	Dark Green
5	Light Blue	13	Magenta
6	Dark Red	14	Grey
7	Cyan	15	White

EXAMPLE:

BCOL 6

This would set the backdrop colour on the screen to Dark Red.

When BASIC is loaded the backdrop colour is set to 4 (Dark Blue)

Related Keywords : GCOL TCOL

BEEP

BEEP

Syntax: BEEP J

Purpose: This is a Sound Command which causes an 880Hz tone to be sounded for a length of time indicated by the value of J which must be in the range 1 to 255.

J-VALUE	TIME
1	100ms
2	200ms
3	300ms
4	400ms
5	500ms
.	.
.	.
.	.
255	25,500ms

EXAMPLE:

BEEP 20

This will cause the tone to be sounded for a length of time equivalent to 2000ms (2 seconds).

Related Keywords : MUSIC PSG

BIN$

BIN$ (Binary String)

Syntax: BIN$(I,J)

Purpose: This is a Machine Code related command which returns the Binary number which corresponds to the decimal number given by **I**.

J indicates the number of binary digits to be returned in the result and must evaluate to an integer which is less than, or equal to, 16. If **J** is omitted then 16 binary digits are returned (the number being "padded" with leading zeros if necessary).

If the value of **J** is too small for the binary number to be returned, then only the **J** least significant digits will be returned.

EXAMPLE:

X$=BIN$(86)

This will return a result of 0000000001010110 in X$

X$=BIN$(86,8)

This will return a result of 01010110 in X$

Related Keywords : HEX$

BTN

BTN (Press Button)

Syntax: BTN (J)

Where J is given as 0 or 1.

Purpose: This function returns a value of 0 or 1 for a press button connected to the Analogue 1 and Analogue 2 ports.

For the Analogue 1 port J=0
For the Analogue 2 port J=1

When the button is pressed a value of 0 is returned.
When the button is not pressed a value of 1 is returned.

A common application of this function relates to the firing button incorporated with joy-stick controls.

EXAMPLES:

1) B = BTN(0)

   PRINT B

   This will return a value (either 0 or 1) in B according to the status of the Analogue 1 port. The value can be output using the PRINT statement.

2) The following program can be used with a joy-stick control, which incorporates a firing button, connected to the Analogue 1 port

```
10 X = ADC(0)
20 Y = ADC(1)*3/4
30 IF BTN(0) = 0 THEN GCOL RND(16)
40 DRAW TO X,Y
50 GOTO 10
```

When RUN, this program will draw lines on the screen corresponding to the movement of the joy-stick. Pressing the firing button will change the colour of the graphics on a random basis as given by the statement in line 30 of the program. The statement could be changed to give different results to the condition as shown below.

```
30 IF BTN(0) = 0 THEN CLS
```

This will clear the screen each time the firing button is pressed.

Related Keywords: ADC

CALL

CALL

Syntax: CALL I

Purpose: This is a Machine Code related command which calls a machine code subroutine which starts from the address given by I. The related machine code subroutine must be terminated with a &C9 (return) code. This will automatically return control to BASIC.

EXAMPLE: CALL 3840

This causes execution of machine code from location &OF00 (i.e. HEX equivalent of 3840 decimal).

NOTE: The pointer to the current position in the program text will be available at the top of stack, if required.

Syntax: CALL (E)

Purpose: E is passed to the floating point accumulator within the BASIC interpreter. Machine code is then executed from the address set by means of the PTR 9,I command. This defines the location for the machine code routine. Again a &C9 (return) code returns control to BASIC, and the contents of the floating point accumulator form the argument of the CALL function.

EXAMPLE: A = CALL(B)

The value of B is loaded into the floating point accumulator. On return from the machine code routine, the contents of the floating point accumulator are passed into variable A.

Related Keywords:

CHAIN

CHAIN

Syntax: CHAIN L

Where L, if given, is a line number.

Purpose: In this case CHAIN is similar to the RUN command except that all variables are preserved and can be passed from one program to another.

EXAMPLES:

CHAIN - Executes the program currently in memory.

CHAIN 50 - Begins execution at line number 50.

Syntax: CHAIN <file>

Where **<file>** must be a legal "file name" as described in the section on File Handling (Page 264)

Purpose: In this case the file specified by **<file>** is loaded from the disc and executed. This is similar to RUN **<file>** except that all variables are preserved.

EXAMPLE:

CHAIN "PROG" - This loades and executes a program called "PROG" preserving any variables.

NOTE: Programs can be combined using a combination of CHAIN, HOLD, and MGE commands. this could be useful in large applications which may be divided into smaller programs, all using the same variables. Further explanation of this technique can be found on page 261.

Related Keyword : RUN

CHR$

CHR$ (Character String)

Syntax: CHR$(J)

Purpose: This is a Standard String Function which returns the single character string whose ASCII value is given by J.

EXAMPLE:

```
X$=CHR$(75)
```

The decimal ASCII value 75 is that of the character K. Therefore the string K will be stored in X$.

Related Keywords : ASC STR$

CLEAR

CLEAR

Syntax: CLEAR I1 , I2

I1 when specified, sets up the topmost location of memory available to BASIC. I2, when specified, allocates the size of the "stack".

Purpose: This Command clears all variables, arrays and strings from the system. The top of memory would be set in order to leave space for object (.OBJ) files (i.e. machine code routines/data). Normally, no space is reserved, and the stack size is left unchanged if I2 is omitted.

If I1 is set above the top of RAM, or set too low, or too large a stack size (I2) is set, then a MEM FULL ERROR will occur.

The stack is normally 256 bytes and cannot be smaller. Normally it would not be necessary to increase this size unless large numbers of nested FOR loops, subroutines, and expressions are used. (If a STACK FULL ERROR is encountered it is usually because subroutines are being ENTERED but not RETURNED from!).

EXAMPLES:

CLEAR,500	-	sets 500 bytes of stack space.
CLEAR&7FFF	-	sets the top of RAM for BASIC programs and variables to $7FFF_H$. Thus machine code programs can be placed in the area from 8000_H up.
CLEAR &AFFF,300	-	sets 300 bytes of stack space, and the top location to &AFFF.

Related Keyword : PTR

CLOSE

CLOSE

Syntax: CLOSE SV1,SV2,..,SVn

SV1,SV2, etc. must be string variable names (but not string array elements).

Purpose: This is a File-Handling command which performs the following:-

Closes any open files given by the file descriptors SV1 to SVn, these files having previously been opened using OPEN, CREATE, or APPEND. If **no** file descriptors are specified then **all** files currently open will be closed. (no error is given if there are no files open).

If any of the string variables SV1 to SVn specified is not the file descriptor for an open file, then a FILE ERROR will be given. (Note that the file descriptors are internally marked so that the BASIC can distinguish them from normal strings).

On closing a file, the remaining contents of the appropriate buffer is written to the file if the last operation performed on it was a write. The file descriptors are then set to null strings, which makes the space available for use by variables or other files, and the directory updated.

NOTE: In addition to preforming the above processes CLOSE induces an automatic PRINT #0: INPUT #0 (see page 183, 113). This will cause all output and input to go through the console and the CLOSE command can be used at any time when these **two** statements are required (it is shorter).

Related keywords : CREATE OPEN

CLS

CLS (Clear Screen)

Syntax: CLS N

Where N has a value of 32 or 40

Purpose: To clear the display screen or send a form feed character to any other selected output device.

If current output is to the screen then:-

If N = 40 (i.e.CLS40) this will clear the screen and set up the 40 column Display.

If N = 32 (i.e.CLS32) this will clear the screen and set up the 32 column Display.

If N is omitted, the screen will clear, leaving the current Display unchanged.

For any other output device:-

A "form feed" character is sent to the selected device.

Related Keywords:

CONT

CONT (Continue)

Syntax: CONT

Purpose: Causes an interrupted program to resume without clearing the variables.

May be used after a program has been terminated with a STOP command. During the "stopped" period, the user may look at or alter variables without causing any problems, but any attempt to alter the program itself will result in a CONT ERROR.

CONT may be used to re-start a program which has been halted by SHIFT-BREAK. This is quite a useful aid when debugging a program.

Related Keywords: END RUN STOP

COS

COS (Cosine)

Syntax: COS (N)

Where N is an angle, or a numerical expression returning an angle, given in radians.

Purpose: This is a Standard Function which returns the COSINE value of N.

EXAMPLES:

```
X = COS (1.0472)
```

This gives a value of 0.499998 in X. (1.0472 radians is 60°)

```
PRINT COS (0.5236)
```

The value 0.866025 will appear on the screen. (0.5236 radians is 30°)

EXAMPLE:

This example shows an entry made in degrees using the RAD function to convert the angle within the COS function.

```
PRINT COS(RAD(30))
```

The value 0.866025 will appear on the screen.

Related Keywords: ATN DEG RAD SIN TAN

CREATE

CREATE

Syntax: CREATE <file>,SV,I

The file must be a "legal file name" as described in the section on File-Handling (Page 264).

SV is a string variable name (but not a string array element) and is the file descriptor.

I is the random record size (length) and is given as a value in the range 0 to 65535, indicating the number of characters involved.

Purpose: This is a File-Handling Command which creates and opens a new serial data file as follows:-

a) deletes any existing file with the same name as given in the command.

b) creates and opens a new empty serial data file having the name given in the command, and identified by the string variable SV, to be structured into data records of length I bytes. I is only specified for "random access", if "sequential access" is to be applied then I is omitted (in fact a random record length of 0 indicates that sequential access is to be performed).

EXAMPLE:

```
CREATE "0:SILLY.DAT",FD$,15
```

This will perform the following:-

a) creates and opens the file SILLY.DAT on the disc currently in drive 0 (if any file of the same name already exists on the disc it will be deleted prior to the new empty file being created).

b) assigns FD$ as the file descriptor.

c) sets up for "random access" using a 15 character length record size.

Related Keywords: APPEND CLOSE OPEN

DATA

DATA

Syntax: DATA data1 , data2 , ... , datan

The items of data (data1, data2, etc) may be any of the following types.

a) numeric
b) strings in quotes.
c) strings without quotes, providing there are no leading spaces or separators.

Purpose: This statement holds items of data required within a program. It is used in conjunction with the READ statement.

Any number of DATA statements can be used within a program, each containing as many or as few items as are convenient

DATA statements may appear at any position in a program but will be read as though they were all in one block.

They are ignored when encountered during the running of a program in the same manner as REM statements. (see Page 194).

The SEPARATOR between items of data is normally a comma (,) but this may be modified by use of the SEP command (see Page 206) (This will also affect INPUT and READ statements within the same program).

EXAMPLE:

```
DATA 6, "NO", YEB
```

Related Keywords: READ RESTORE

In line 100, 6 is passed to the defined expression X+3 as the dummy variable X and hence Y takes the value 9.

If a function call is made before the appropriate DEF FN statement has been made a FN DEFN ERROR will occur.

Related Keywords: FN

DEG

DEG (Degrees)

Syntax: DEG (N)

Where N is given in radians.

Purpose: This is a Standard Function which converts the number expressed in radians given by N, to degrees.

EXAMPLE:

```
DEG (0.523604) - returns a value of 30 degrees.
```

EXAMPLE:

```
PRINT DEG (0.523604)
```

This will cause the value 30 to appear on the screen.

Related Keywords: ATN COS RAD SIN TAN

DEL

DEL (Delete)

Syntax: DEL L1,L2

Where L1 and L2 are given as line numbers of a program.

L1 will default to 0 if not specified.

If L1 is LARGER than L2, or if L1 is LARGER than the largest line number of the program, a RANGE ERROR will occur.

Purpose: This is a System Command which deletes all lines from a program in the range of L1 to L2.

EXAMPLES:

DEL 100,199

This will delete all program lines with numbers from 100 to 199 inclusive.

DEL,155

This will delete all lines up to 155 inclusive (value of L1 omitted therefore defaulting to 0).

Related Keywords:

DIM

DIM (Dimension)

Syntax: DIM <array name> (I1,I2,I3..,In)

The array name can be either a numeric or string variable.

I1,I2,I3 are numeric expressions, known as subscripts, in the range 0-65535 and represent the number of elements in an array.

If an array is not dimensioned it is assumed to have a maximum value of 10 for each subscript (dimension) which is referenced. Therefore if subscripts are less than 10 the DIM statement may be omitted.

An array must only be dimensioned once in a program. If dimensioned more than once a DIMENSION ERROR will occur.

Purpose: This is used to reserve storage space for numeric or string arrays.

An array with only one subscript is known as one dimensional.

A (I)

An array with two subscripts is known as two dimensional.

A (I1,I2)

An array with three subscripts is known as three dimensional.

A (I1,I2,I3)

Each subscript represents the maximum number of elements of each dimension in the array. TATUNG/Xtal BASIC 4 will support multi dimensional arrays, the limit being determined by the amount of memory available at any one time.

Several arrays may be dimensioned in one DIM statement using a comma as a separator.

DIM A(I1,I2), B(I), C(I1,I2,I3) etc.

EXAMPLES:

DIM A(50)

Defines array A as having one dimension with storage for 51 elements (0 to 50).

DIM B(60,20)

Defines array B as having two dimensions with storage for 61 x 21 elements (i.e. 1281 elements in total).

DIM A(50), B(20), C(40)

Defines three arrays A,B,C, with one dimension each, containing storage for 51, 21, and 41 elements respectively.

Related Keywords:

DIR

DIR (Directory)

Syntax: DIR <file types>

Purpose: This is a Disc Command which displays the DIRECTORY of a disc, showing the files specified by file types (<file types> being the required file name Types - see Page 264 for file name conventions).

If <file types> is omitted or given as "*.*", **all** the files in the directory of the disc in the current **default drive** will be listed.

Locked files are indicated by a * symbol in front of their names in a directory listing.

EXAMPLE:

```
DIR
```

This will give a display of all the directory for the disc in the default drive, similar in format to the one given below (the disc contains five files in this example).

```
:*XBAS.COM        : XYZ.XBS
: XYZ.ASC         : ROUTINES.OBJ
:*INVADERS.ASC
```

This represents a list of the five files contained on the disc. The file name is given, followed by the type name for each file (see Page 264 for **file types**). The colons (:) are displayed to indicate the beginning of each entry.

DOS

DOS (Disc Operating System)

Syntax: DOS

Purpose: This is a System Command and is used to transfer control to the "Disc Operating System".

Usually used in "Direct Mode".

Related Keywords: MOS

DRAW

DRAW

Syntax: DRAW x1,y1,z1 TO x2,y2,z2 TO ... TO xn,yn,zn

x and y can have values in the range -32768 to +32767.

z is a qualifier which defines the type of line to be drawn in accordance with the table given below.

If z is omitted a value of 0 is assumed.

Purpose: This is a Graphics Command which will draw a line, in the current foreground colour, from the point given by the co-ordinates x1,y1 to the point given by co-ordinates x2,y2 etc.

If the first pair of co-ordinates is omitted, then drawing will take place from the **last plotted** point.

VALUE - z	TYPE OF LINE
0	Continuous line
1	Continuous unplot (i.e. line drawn in background colour)
2	Dotted line, 2 dots ON and 2 OFF
3	Dashed line, 4 dots ON and 4 OFF
4	Dotted-Dashed line, 10 dots ON, 2 OFF, 2 ON, 2 OFF
5	Dashed-Dotted line, 2 ON, 2 OFF, 2 ON 10 dots OFF.

EXAMPLE:

```
DRAW(40,70) TO (90,80),3 TO (100,90) TO (40,70)
```

time of line

This will draw a continuous line from the point (40,70) to the point (90,80) and a dotted line to the point (100,90). Then draw a continuous line back to the point (40,70). In other words a triangle has been drawn with two sides as solid lines and one side dotted.

Related Keywords: ELLIPSE ORIGIN PLOT POLY UNPLOT

DRIVE

DRIVE (Disc Drive)

Syntax: DRIVE J

J is specified as a number from 0 to 3 depending on the number of drive units available within individual systems.

If the drive specified in J is not available on the system a DRIVE SELECT ERROR will be given.

Purpose: This command sets up the default disc drive as specified by drive name for any subsequent access to a disc.

EXAMPLE:

```
DRIVE 1
```

This selects drive 1 as the default drive.

Related Keywords: DIR ERA LOAD REN SAVE

ELLIPSE

ELLIPSE

Syntax: ELLIPSE x,y,R,T,z

Purpose: This command draws an ellipse or a circle depending on the values of the parameters given.

x,y are the co-ordinates of the centre for the ellipse and can have values in the range -32768 to +32767.

R is the value of half the horizontal axis of the required ellipse.

T is a qualifier and is given by the ratio of the two axes as follows:-

$$T = \frac{\text{VERTICAL AXIS}}{\text{HORIZONTAL AXIS}}$$

If T is omitted it defaults to 4/3 and a circle of radius R is drawn (owing to the aspect ratio of the VDU screen being 4:3).

The value of z is a number in the range 0 to 5 which indicates the type of line in accordance with the table given below. (If omitted z will default to 0).

0 - Continuous line.
1 - Continuous unplot (i.e. line drawn in background colour).
2 - Dotted line, 2 dots on 2 dots off.
3 - Dashed line, 4 dots on, 4 dots off.
4 - Dotted-dashed line, 10 dots on, 2 dots off, 2 dots on, 2 dots off
5 - Dashed-dotted line, 2 dots on, 2 dots off, 2 dots on, 10 dots off.

EXAMPLE:

```
ELLIPSE 100,100,50
```

A circle of radius 50 is drawn with its centre at co-ordinates 100,100 (T defaults to 4/3 and z to a continuous line)

Related Keywords: DRAW ORIGIN PLOT POLY UNPLOT

ELSE

ELSE

Syntax: IF <condition> **THEN** <statement> **ELSE** <statement>

Purpose: This command is used in conjunction with the IF - THEN statement to provide an alternative course of action.

EXAMPLE:

```
IF X = 10 THEN 100:ELSE 50
```

If x equals 10 then execution is transferred to line 100. If x does not equal 10 then the program branches to line 50.

Refer to the IF statement (page 102) for further information on the use of ELSE.

Related Keywords: GOTO IF THEN

END

END

Syntax: END

Purpose: This command is used in deferred mode i.e. as a line of a program.

This command terminates the execution of a program.

It is not strictly necessary when the end of a program coincides with the highest line number and in such cases can be omitted.

Related Keywords: CONT STOP

EOF

EOF (End of File)

Syntax: <statement> EOF <statement>

Purpose: This command is used in relation to File Handling and invokes a specific action following detection of the end of a file during processing.

EOF is used within the following statements.

ON EOF GOTO "Line No"
ON EOF GOSUB "Line No"

Either of these two statements could be used in a program involving file handling. If an "end-of-file" is detected then a GOTO/GOSUB is made to a particular routine which would carry out a predetermined course of action. The last statement of this routine should direct execution back into the main program. Execution would then continue from the statement immediately following the one which detected the "end of file".

When either of the two statements are used, an internal flag is set in order to activate the above procedure.

OFF EOF is used to turn off the ON EOF mode. Any subsequent end-of-file encountered will then cause an END OF TEXT ERROR to be displayed. However, when a program "ends" in the normal way, the ON EOF is automatically turned off.

EXAMPLE:

```
10 - - - - - -
20 - - - - - -
30 - - - - - -
40 ON EOF GOTO 120
50 - - - - - -
etc.
```

In this example any "end-of-file" detected after line 40 would cause a branch to a routine starting at line 120. This routine would then carry out a predetermined course of action in respect of the "end-of-file" before returning execution to the main program.

Related Keywords: GOTO GOSUB OFF ON ERR

ERA

ERA (Erase)

Syntax: ERA **<** file **>**

Purpose: This is a Disc Command which will erase the file, given by file from a disc in the current **default drive** (see Page 264 for file name conventions).

If **<**file**>** does not exist then a NO FILE error will be given.

If **<**file**>** is a "locked" file then a FILE LOCKED error will be given.

If the write-protect of the disc is in operation a DISC LOCKED error will be given.

The **default drive** may be changed or re-selected by use of the DRIVE command (see Page 69).

Related Keywords: DIR DRIVE

ERL

ERL (Error line number)

Syntax: ERL

Purpose: This function is used in error handling routines and returns the line number at which the last error occurred. See Page 248 for further information on error handling.

EXAMPLE:

```
10 PRTNT "DOS"
```

The spelling mistake in line 10 would generate a SYNTAX ERROR. ERL would now contain 10 and PRINT ERL would then display 10

Related Keywords: ERR ERR$ ON

ERR

ERR (Error)

Syntax: ERR

Purpose: This command returns the value of the last error generated.

EXAMPLE:

```
PRINT ERR
```

This will display the value of the last error generated.

ERR can also be used in conjunction with the ON command as follows:-

```
ON ERR GOTO L
ON ERR GOSUB L
```

Where L is a line number.

For further information relating to ON ERR refer to ERROR HANDLING section on page 248.

Related Keywords: ERL ERR$ GOTO GOSUB OFF ON

ERR$

ERR$ (Error string)

Syntax: ERR$

Purposes: This function is used in error handling routines and returns the error string message, without the word "ERROR", corresponding to the last error which occurred. See Page 248 for further information on error handling.

EXAMPLE:

```
10 PRJNT "ONE"
```

A SYNTAX ERROR is generated in line 10 due to the incorrect spelling of PRINT.

PRINT ERR$ will now display 'syntax'

Related Keywords: ERL ERR ON ERR

EVAL

EVAL (Evaluate)

Syntax: EVAL(<string expression>)

Purpose: This is a Standard Function where the string expression is calculated as if it were a numeric expression and returned as a numeric value.

The string expressions must be syntactically correct as a numeric expression otherwise a SYNTAX ERROR will occur.

EXAMPLE:

```
A = EVAL ("10*3+4")
```

10*3+4 is a string expression but is treated as if it were a numeric expression and the value 34 is returned into A.

Related Keywords: VAL

EXP

EXP (Exponent)

Syntax: EXP (N)

Purpose: This is a Standard Function which raises the exponential, e, to the power given by N.

If N exceeds 87 an OVFL ERROR (overflow error) will occur (since the result would be greater than 1E39).

e has the value of 2.71828.

EXAMPLE:

```
X=EXP (12)
```

This will raise e to the power 12 (i.e. e^{12}) and store the result 162755 in X

Related Keywords: ATN LN LOG

FILL

FILL

Syntax: FILL x,y,J

x and y are the co-ordinates of a required point and can be in the range -32767 to +32767.

J can be a value in the range 0 to 15, each number representing a colour as listed below.

0 Transparent	6 Dark Red	11 Light Yellow
1 Black	7 Cyan	12 Dark Green
2 Medium Green	8 Medium Red	13 Magenta
3 Light Green	9 Light Red	14 Grey
4 Dark Blue	10 Dark Yellow	15 White
5 Light Blue		

Purpose: This is a Graphics Command which will fill in colour J an area of the screen which has its perimeter drawn in a foreground colour and which encloses the point x,y.

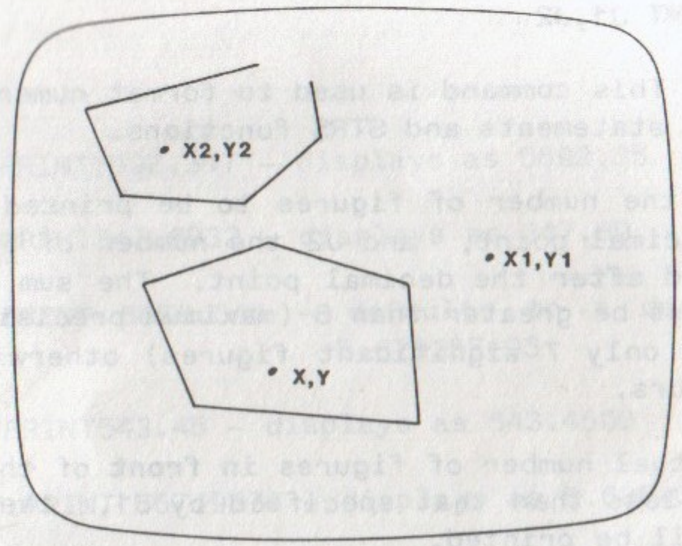
This command will fill in foreground if point x,y is background or in background if the point is foreground. If J is omitted the colour of the fill will be the current graphics colour for the fill type (i.e. background or foreground) unless J is declared.

EXAMPLE:

```
10 CLS:GCOL7,0:ORIGIN128,96
20 ELLIPSE 0,0,60:FILL 0,0
30 GCOL,8
40 POLY 6,0,0,45,,1:FILL 0,0
50 END
```

This example draws a cyan-coloured disc, and then draws a red hexagon inside it.

EXAMPLES:



In the examples illustrated above:-

If point X,Y is specified then the command will fill the given polygon to its boundary.

If point X1,Y1 is specified then the command will fill the screen outside the polygon.

If X2,Y2 is specified then the command will fill the associated polygon and then spill out, because the shape is not fully enclosed, and fill the remainder of the screen.

NOTE: A 'STACK FULL' error may occur when FILLing around text. This is caused by overflow of the 'FILL' stack.

Related Keywords: DRAW ELLIPSE GCOL ORIGIN PLOT POLY TCOL UNPLOT

FMT

FMT (Numeric Output Format)

Syntax: FMT J1,J2

Purpose: This command is used to format numeric output for PRINT statements and STR$ functions.

J1 gives the number of figures to be printed **in front** of the decimal point, and J2 the number of figures to be printed **after** the decimal point. The sum of J1 and J2 must not be greater than 8 (maximum precision of the system is only 7 significant figures) otherwise a QTY ERROR occurs.

If the actual number of figures **in front** of the decimal point is less than that specified by J1, then leading spaces will be printed.

If the number of figures **in front** of the decimal point is greater than that specified by J1, then the output defaults to "scientific notation". Scientific notation may be forced by setting J1 to 15.

If the number of figures **after** the decimal point is less than that specified by J2, then trailing zeros are printed up to the required number of figures.

If the numbers of figures after the decimal point is greater than that specified by J2, the last figure which complies with J2 will be rounded up or down accordingly (see examples)

Normal output format is to 6 significant figures and scientific notation is invoked if the magnitude of a number is greater than 1E6 or less than 1E-2. Trailing zeros are always suppressed in normal output.

On conclusion of processing involved with a particular FMT command, **normal** output can be restored by use of FMT0,0.

EXAMPLES:

```
FMT4,2:PRINT5692.347 - displays as 5692.35

FMT4,2:PRINT347.6932 - displays as 347.69

FMT3,2:PRINT 5678.346 - defaults to a display of
                        5.67835E+03

FMT3,4:PRINT543.45 - displays as 543.4500

FMT15,2:PRINT 567.9876 - displays as 5.68E+02
```

If the sign of a number is given, it is not counted with the number of figures but appears in the leading space at the start of the entire number.

EXAMPLES:

```
FMT3,3:PRINT-1.73205 - displays as -  1.732

FMT3,3:PRINT-42.763  - displays as - 42.763
```

Related Keywords:

FN

FN (Function)

Syntax: FN V1(V2)=E

Purpose: This command is used with DEF to provide the facility for creating functions not normally contained within the BASIC. For further information see DEF page 57.

Related Keywords: DEF

FOR

FOR

Syntax: FOR V = N1 TO N2 STEP N3

Purpose: This statement sets up a **loop** within a program to repeat a sequence of operations. It is used in conjunction with the TO, STEP and NEXT statements.

V is the CONTROL VARIABLE which must be a numeric variable.

The values, or numeric expression string values, for N1,N2, and N3 have the following functions.

N1 is the INITIAL VALUE from which **V** starts the loop.

N2 is the LIMIT VALUE of **V**, which, when passed, ends the loop.

N3 is an optional STEP VALUE which is the amount by which **V** is incremented on each cycle of the loop. If "STEP N3" is omitted then a step value of +1 is assumed.

The FOR statement indicates the beginning of the loop. The NEXT statement indicates the end of the loop. The NEXT statement is followed by the control variable to link it with the relevant FOR statement.

The program loops between these FOR and NEXT statements. On reaching the NEXT statement V is incremented to the next value and program execution jumps back to the FOR statement. The loop is repeated until V has incremented past the value of N2

Any program lines between the FOR and NEXT statements are executed on each cycle of the loop.

EXAMPLE:

```
10 FOR I = 1 TO 10 STEP 3
20 NEXT I
```

The control variable, I, starts at 1 (initial value) for the first loop and then increments by 3 (step value) for each repetition of the loop until the limit value of 10 is exceeded.

```
10 FOR I = 1 TO 10
20 NEXT I
```

In this case the step value has been omitted and therefore a value of 1 will be assumed. Thus I will start at 1 and increment by 1 until it exceeds 10.

The necessary operation statements of the loop immediately follow the FOR statement.

EXAMPLE:

```
10 FOR I = 1 TO 10
20 A = I + 3
30 PRINT A
40 NEXT I
```

In this instance the expression A=I+3 is evaluated and displayed for each value of I from 1 to 10 (I increments by 1 after each evaluation).

If the control variable is missing off the NEXT statement, then it is assumed to be linked with the previous FOR statement.

EXAMPLE:

```
10 FOR I = 1 TO 10
20 PRINT "RED"
30 NEXT
```

I is not needed in line 30 to complete this.

NESTED LOOPS:
FOR-TO-NEXT statements can be **nested** (i.e. one loop contained within another).

In "nested" loops the NEXT statement takes on the following format.

```
NEXT V1,V2,..,Vn
```

This is the equivalent of:-

```
NEXT V1:NEXT V2;..;NEXT Vn
```

EXAMPLE:

```
10 FOR I = 0 TO 7
20 FOR J = 5 TO 19 STEP 2
30 K = I+J
40 PRINT K
50 NEXT J,I
```

This example will print out all the values of I+J using values of I from 0 to 7 and values of J from 5 to 19. The processing will be executed in the following manner.

FOR FIRST VALUE OF I=O

```
I +  J =  K
0 +  5 =  5
0 +  7 =  7
0 +  9 =  9
0 + 11 = 11
0 + 13 = 13
0 + 15 = 15
0 + 17 + 17
0 + 19 = 19
```

FOR SECOND VALUE OF I=1

```
I +  J =  K
1 +  5 =  6
1 +  7 =  8
1 +  9 = 10
1 + 11 = 12
1 + 13 = 14
1 + 15 = 16
1 + 17 = 18
1 + 19 = 20
```

This process is repeated for each value of I until all the required results have been printed.

EXAMPLE:

The following example illustrates how documentation often presents the listing in a format which indicates the nested loops.


```
10 FOR Z = 1 TO 10
20    FOR X = 0 TO 11
30       FOR Y = 1 TO X+13
40          PRINT CHR$(170)+MUL$(CHR$(203),16)
               +CHR$(170)
50       NEXT Y
60       PRINT MUL$(CHR$(32),18)
70       FOR Y = X+15 TO 40
80          PRINT CHR$(170)+MUL$(CHR$(203),16)
               +CHR$(170)
90       NEXT Y
100   NEXT X
110 NEXT Z
120 END
```

CROSSING

Although loops can be nested they must not be allowed to cross over each other.

```
10 FOR X
20 ----
30 ----
40 FOR Y
50 ----
60 NEXT X
70 ----
80 NEXT Y
```

The format shown above would not work and is an example of bad logic in setting up the loops.

NOTE: If the value of the control variable V in the NEXT statement does not correspond to an active FOR loop, then a NEXT ERROR will be given.

If the control variable V is **omitted** from the NEXT statement, then the **last** FOR statement is assumed to be the one required.

There is no limit to the amount of nesting allowed with loops other than the capacity of the memory to deal with the necessary program operation.

Related Keywords: TO STEP NEXT

GCOL

GCOL (Graphics Colour)

Syntax: GCOL J1,J2

Purpose: This is a Display Command which selects the colour of graphics displayed on the screen according to the values of J1 and J2.

J1 represents the **Foreground** colour (i.e. the colour of the pixels forming the line or character shape) and J2 the **Background** colour (the colour of the surrounding pixels). J1 and J2 can be any value from 0 to 15, each number representing a particular colour as listed below. If either J1 or J2 is omitted, and no previous GCOL command has been given, then J1 will default to White (15) and J2 will default to Dark Blue (4).

0 Transparent	8 Medium Red
1 Black	9 Light Red
2 Medium Green	10 Dark Yellow
3 Light Green	11 Light Yellow
4 Dark Blue	12 Dark Green
5 Light Yellow	13 Magenta
6 Dark Red	14 Grey
7 Cyan	15 White

EXAMPLE: GCOL 10,6

Any graphics produced following this command will appear in Dark Yellow (foreground) on a Dark Red background.

NOTE: This command only directly affects the graphics pixels as they are printed on the screen and does not change the overall backdrop colour of the screen.

Related Keywords: BCOL TCOL

GOSUB

GOSUB (Go to Subroutine)

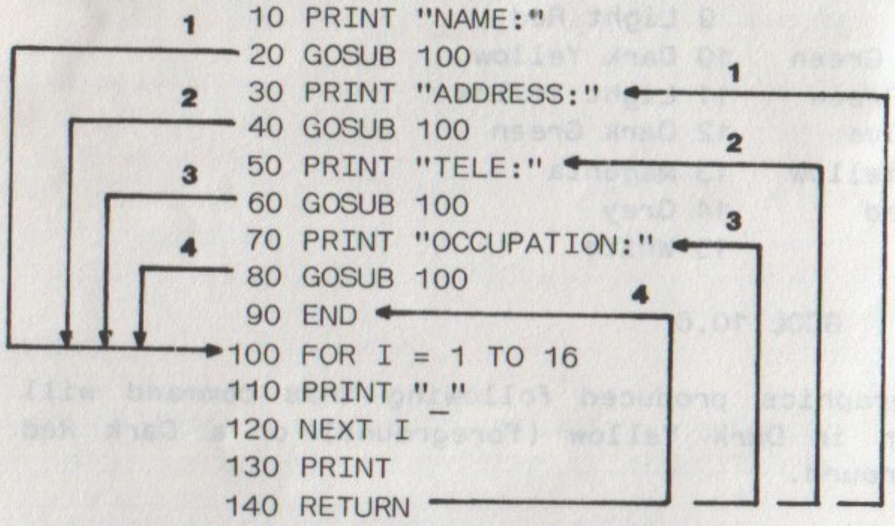
Syntax: GOSUB line number

Purpose: This transfers execution of the program to a subroutine which starts at the line number specified

If line number is not specified, or does not exist, a BRANCH ERROR will occur.

Execution continues from the specified line number onwards until a RETURN statement (Page 199) is encountered, whereupon execution is returned to the line immediately following the original GOSUB statement.

EXAMPLE:



This program prints out the following format:-

NAME:

ADDRESS:

TELE:

OCCUPATION:

Lines 20, 40, 60, and 80 cause a branch to the subroutine contained within lines 100 to 140. This subroutine carries out the repetitive process of printing the broken lines between each of the titles. Line 140 returns program execution each time to the lines immediately following the last executed GOSUB.

Related keywords: GOTO POP RETURN

GOTO

GOTO

Syntax: GOTO <line number>

Purpose: This transfers program execution directly to a specified line. (i.e. creates a BRANCH).

If the line number is not specified or, does not exist, then a BRANCH ERROR will occur.

EXAMPLE:

```
10 I = 0
20 PRINT I
30 I = I+1
40 GOTO 20
50 END
```

This program prints the value of I in line 20 and adds one to it in line 30. Line 40 causes program execution to return to line 20. Try running the program. It should print out 0,1,2,3,4 --- etc. until the program is terminated by pressing SHIFT-BREAK.

Related Keywords: GOSUB IF ON THEN

HEX$

HEX$ (Hexadecimal String)

Syntax: HEX$(I,J)

Purpose: This command returns a Hexadecimal string which corresponds to the number given by I (in decimal).

J dictates the number of characters to be returned in the Hexadecimal string and must be in the range 1 to 4. If J is omitted a value of 4 is assumed.

The Hexadecimal number will be "padded" with leading zeros if necessary.

If the value of J is too small for the HEX number to be returned, then only the J least significant digits will be returned.

EXAMPLES:

```
X$=HEX$ (1234)     - returns the string "04D2"
X$=HEX$ (100,2)    - returns the string "64"
X$=HEX$ (4708,2)   - also returns the string "64"
                     (only the two least significant characters
                     are displayed as dictated by the value of J
                     being 2).
```

Related Keywords: BIN$

HOLD

HOLD

Syntax: HOLD L1,L2

L1 and L2 indicate the first and last line numbers of a range to be held in view. If omitted, L1 will default to 0 and L2 to 65535.

Purpose: This is a System Command which "holds" a range of lines from a program "in view" for manipulation or execution in isolation from the remainder of the program.

The "non held" part of the program seems to disappear; in fact it is still present in memory but cannot be accessed directly.

EXAMPLES:

HOLD 100,199 - Holds the range of lines from 100-199 inclusive.

HOLD 100 - Holds all lines from 100 upwards.

HOLD,199 - Holds all lines up to, and including,199.

This provides two facilites:-

i) The range of lines held can be renumbered and thereby moved to another area of the program.

ii) Another program can be appended to the range of lines held "in view".

EXAMPLES:

1. To RENUMBER a section within a single program.

Using the following program, the area outlined is to be renumbered so as to move it to the end of the program.

```
10 FOR I = 1 TO 5
20 PRINT "*";
30 NEXT I
40 FOR B = 6 TO 10
50 PRINT B
60 NEXT B
70 FOR I = 1 TO 5
80 PRINT "#"
90 NEXT I
```

The section required is selected from the program using HOLD 40,60.

The held section is renumbered using RENUM 100,10 (Page 196) to change the line numbers to 100,110, and 120.

The MGE command (Page 144) is used to replace the section in the program and bring the "non-held" part back "in view".

The final result when listed then appears as below with the original held section now in its new position.

```
10 FOR I = 1 TO 5
20 PRINT "*";
30 NEXT I
70 FOR I = 1 TO 5
80 PRINT "#"
90 NEXT I
100 FOR B = 6 TO 10
110 PRINT B
120 NEXT B
```

NOTE: It is important that the section renumbered is not renumbered to lines that exist within the non-held sections. If in the above example the held section was renumbered as RENUM 70,10 then two lines could be created for each line number 70, 80 and 90

2. To append another program.

   Starting with the following initial program.

```
10 FOR I = 1 TO 5
20 PRINT I
30 NEXT I
```

   The whole program is held using HOLD,30.

   The second program can then be loaded without affecting the held program.

```
10 FOR I = 6 TO 10
20 PRINT I
30 NEXT I
```

   The second program is then renumbered using RENUM (Page 196) such that its line numbers are greater than the last line number of the held program.

```
40 FOR I = 6 TO 10
50 PRINT I
60 NEXT I
```


   The MGE command (Page 144) is then used to bring the two programs together giving the following result.

```
10 FOR I = 1 TO 5
20 PRINT I
30 NEXT I
40 FOR I = 6 TO 10
50 PRINT I
60 NEXT I
```

CAUTIONARY NOTES:

When renumbering, care must be taken in the selection of the new line numbers. Any duplication of line numbers will result in BOTH lines being listed in the final program.

The line numbers of the "non-held" part of the program are not affected by the renumbering process, but any line number references following GOTO, GOSUB, RUN, THEN, ELSE, RESTORE, contained within the non-held lines will be altered accordingly. (This is obviously a most necessary and useful facility when moving sections about within one program but care should be exercised in respect of this when adding a second program).

Hold is also used in conjunction with CHAINING and SEMI-CHAINING of programs. This is dealt with in more details on Page 261 of this Manual. Note also that RUN and CHAIN will restore a HELD program.

Related Keywords: CHAIN MGE RENUM

IF

IF

Syntax: IF ‹condition› **THEN** statement› **ELSE** ‹statement›

Purpose: This allows the evaluation of **conditions** so that a choice of execution can be made, depending on whether a **condition** is TRUE or FALSE (i.e. it is a Conditional Branch instruction). The IF command is used in conjunction with the THEN and ELSE commands.

THEN ‹statement› is the statement executed if the ‹condition› is TRUE. The ELSE options being ignored in this case.

ELSE ‹statement› is the statement of execution if the ‹condition› is FALSE. In this case the THEN options are ignored. ELSE is optional and quite often becomes a redundant part of the statement, in which case it is omitted.

EXAMPLE:

```
IF A=3 THEN PRINT "YES": ELSE PRINT "NO"
```

If A does equal 3 the PRINT "YES" becomes operative (i.e. the TRUE statement).

If A does **not** equal 3 the PRINT "NO" becomes operative (i.e. the FALSE statement).

ELSE's must not be NESTED but the following does however work.

IF...THEN...ELSE IF...THEN...ELSE...

VARIATIONS:

IF ‹condition› THEN L1 ELSE L2

L1 and L2 are line numbers to which execution will transfer according to TRUE or FALSE CONDITIONS. The following format will also produce the same result.

IF ‹condition› GOTO L1 ELSE L2

EXAMPLES:

```
IF B=7 THEN 70 ELSE 120

IF B=9 GOTO 90 ELSE 50
```

ELSE is optional in both the above cases and if omitted execution will transfer to the next line of the program if the condition is FALSE.

NOTE: The formats may be mixed, replacing either of the lines, L1 and L2, with a statement but the following points must be observed.

i) If L1 is replaced by statements there **must** be a separator (:) between the last statement and the ELSE.

ii) A **line number** must always follow the GOTO if that format is used.

EXAMPLES:

```
IF C=5 THEN PRINT "YES":ELSE 90
IF D=9 GOTO 120 ELSE PRINT "NO"
IF A=20 THEN 70 ELSE PRINT "WHITE"
```

Related Keywords: ELSE GOSUB GOTO THEN

INCH

INCH (Input Character)

Syntax: INCH

Purpose: This is a Standard Function which waits for the next input character and then returns the ASCII value of that character.

This function is quite useful for pauses in instructions as for example at the end of a page or section.

EXAMPLE:

```
PRINT INCH
```

This will cause the machine to await the next input character and then display its ASCII value on the screen.

Related Keywords: INCH$ INPUT KBD KBD$

INCH$

INCH$ (Input Character String)

Syntax: INCH$

Purpose: This is a Standard String Function which waits for an input character, and then returns it as a one-character string. This function is useful for single-character responses such as Y/N (YES/NO), in respect of "prompts" on the screen, and performs in a similar manner to the INPUT statement (see Page 111).

EXAMPLE:

```
10 PRINT "DO YOU DRIVE?"
20 PRINT
30 PRINT "TYPE Y FOR YES, N FOR NO";:A$=INCH$
40 PRINT
50 PRINT "YOU HAVE TYPED:";A$
```

This program would produce the following display:-

```
DO YOU DRIVE?

TYPE Y FOR YES, N FOR NO
```

The machine would then await the input character (Y/N) before continuing to display the following (assuming Y was the character typed in).

```
YOU HAVE TYPED:Y
```

NOTE: This function does not "echo" the key character back to the screen. If this **is** required then either PRINT the "string" as soon as it is input, or use the alternative format **INCH$(J).**

Syntax: INCH$(J)

Purpose: This variation differs from INCH$ in that it waits for an input string with a number of characters as given by J. Unlike INCH$, each character **will** be echoed on input (i.e. displayed on the screen), unless the IOM command (see Page 117) has been activated to suspend echoing of characters.

No special characters are recognised, and the **exact** number of characters specified by J must be input.

This function is especially useful for file input since it does not react to selected characters (unlike INPUT), and therefore can be used to read program or machine-code files.

Related Keywords: INCH INPUT INPUT# KBD KBD$ PRINT PRINT#

INP (Input)

Syntax: INP(J)

Where J represents the address of an INPUT/OUTPUT port in Hexadecimal.

Purpose: This is a command relating to direct input from any I/O device.

The user INPUT/OUTPUT port for example, is at I/O address &32 so this will be the value of J each time this command is used to access the port.

The value returned by this command will be a number in the range 0 to 255

EXAMPLE:

```
A% = INP(&32)
```

This will read the current value at the user INPUT/OUTPUT port and place it in A%.

PORT ASSIGNMENTS:

Programmable Sound Generator.

ADDRESS (HEX)	MODE	FUNCTION
03	READ	Inactive
	WRITE	Write to PSG
02	READ	Read from PSG
	WRITE	Latch Address
00	READ/WRITE	Software Reset for PSG
01		(+ FDC)

Video Display Processor (TMS9129).

ADDRESS (HEX)	MODE	FUNCTION
08	READ/WRITE	VRAM Data
09	WRITE ONLY	Register Data

Programmable Communication Interface (8251).

ADDRESS (HEX)	MODE	FUNCTION
10	READ/WRITE	Data Register
11	READ/WRITE	Control/Status register

Floppy-Disc Controller (FD1770)

ADDRESS (HEX)	MODE	FUNCTION
18	READ/WRITE	Status/Command Register
19	READ/WRITE	Track Register
1A	READ/WRITE	Sector Register
1B	READ/WRITE	Data Register

ROM Select Port

ADDRESS (HEX)	MODE	FUNCTION
24	READ/WRITE	Toggles within each access (ROM Selected on RESET)

'FIRE' Button interrupt mask

ADDRESS (HEX)	MODE	FUNCTION
25	WRITE/ONLY	b0 = 1 to mask b1-b7 not used

Auxiliary Command/Status Register

ADDRESS (HEX)	MODE	FUNCTION
20	READ	b0 'FIRE' Button 1 b1 'FIRE' Button 2 b2 Printer "BUSY" b3 Printer "PAPER EMPTY" b4 Printer "ERROR" b5 GRAPH/ALPHA KEY b6 CONTROL KEY b7 SHIFT KEY
	WRITE	b0 KEYBOARD INTERRUPT MASK = 1 to mask. b1-b7 not used

Analogue - Digitial Convertor Mask

ADDRESS (HEX)	MODE	FUNCTION
21	WRITE ONLY	b0 = 1 to mask. b1-b7 not used
22	READ/WRITE	Alpha-lock LED toggles with each port access (Reset lights LED)

Disc-Drive Select Port

ADDRESS (HEX)	MODE	FUNCTION
23	WRITE ONLY	b0 Drive 1 b1 2 b2 3 b3 4 =1 to select b4 Side Select

Counter - Time Circuit (Z80ACTC)

ADDRESS (HEX)	MODE	FUNCTION
28	READ/WRITE	Channel 0 Control/Data register
29	READ/WRITE	Channel 1 Control/Data register
2A	READ/WRITE	Channel 2 Control/Data register
2B	READ/WRITE	Channel 3 Control/Data register

Parallel Input/Output (Z80API0)

ADDRESS (HEX)	MODE	FUNCTION
30	READ/WRITE	Port A Data register (Printer)
31	WRITE ONLY	Port A Control register (Printer)
32	READ/WRITE	Port B Data Register
33	WRITE ONLY	Port B Control register

Analogue - Digitial Converter (uPD7002)

ADDRESS (HEX)	MODE	FUNCTION
38	WRITE READ	Control Register Data Register

Related Keywords: OUT WAIT

INPUT

INPUT

Syntax: INPUT <Prompt>;V1,V2,..,Vn

The Prompt is optional, but if used must be a string in quotes followed by a semi-colon(;).

V1,V2, etc are numeric or string variable names, strings, or strings within quotes. When more than one variable is used they are normally separated by a comma, but this can be changed by previous use of the SEP command (Page 206).

Purpose: This statement is used to input data from the keyboard during the execution of a program.

If the "Prompt" is omitted then BASIC will introduce its own prompt in the form of a question mark (?), unless the system is in "screen edit" mode, in which case no question mark will appear. This facilitates the input of a line without any unwanted characters appearing before it. Alternatively, if the prompt is declared as an empty string the "?" will not appear on the screen.

If the number of entries typed in **exceeds** the number of variables listed in the INPUT statement, then only the first values entered will be used and the message EXTRA IGNORED is displayed.

If the number of entries typed in is **less** than the number of variables listed in the INPUT statement, then a further prompt (?) will appear.

If a **string** is entered when **numeric** data is expected, then the non-numeric data will be ignored; 0 will be assumed if the first character of the string is non-numeric.

EXAMPLE:

```
10 INPUT "NAME,STREET,HOUSE NUMBER ";NAME$,STREET$,N
```

The INPUT statement causes execution of a BASIC program to be interrupted and then it waits until the required data is input from the keyboard.

EXAMPLE:

```
10 REM PROGRAM TO CALCULATE COST OF ITEMS.
20 INPUT "COST OF ONE ITEM IN PENCE ";I
30 INPUT "NUMBER OF ITEMS ";N
40 PRINT "TOTAL COST IS ",I*N
50 GOTO 20
```

This simple program calculates the cost of a quantity of items, knowing the cost per item.

It illustrates a use of INPUT. The program does not commence execution until variables I (cost of one item) and N (number of items) are typed in from the keyboard; a result then appears on the screen.

Related Keywords: INCH INCH$ INPUT# KBD KBD$ PRINT PRINT#

INPUT#

INPUT# (Input Device Number)

Syntax: INPUT#J

Where J gives a "device number" assigned to a device and in the range 0 to 254.

Purpose: This statement assigns a new input device (eg. serial port) indicated by the value of J.

All input statements, such as INPUT and INCH$, will be received from the new device selected by J until another INPUT# statement is encountered to change the device selection.

When a program ends or aborts, either through an error or as directed from the keyboard, the input device **reverts** back to the keyboard (i.e. device 0).

If INPUT# is used in direct mode the corresponding input statement must appear in the same line.

The following two input devices are currently assigned within the BASIC language provided.

DEVICE	DEVICE NUMBER
KEYBOARD	0
SERIAL PORT (RS232)	2

EXAMPLE:

```
INPUT#2
```

All input following this statement in a program will be received from the serial port.

INPUT# can also be used in the same manner as a normal INPUT statement but the "device number" must be followed by a semi-colon (;) so as to distinguish the remainder of the statement. INPUT# may NOT contain prompts.

EXAMPLE:

```
INPUT#2;X
```

EXAMPLE:

```
10 PRINT#1
20 INPUT#2;X
30 IF X= 0 THEN END
40 FORI = 1 TO X
50 INPUT A$: PRINT A$
60 NEXT I
70 INPUT#0
80 PRINT#0;"DO YOU WISH TO CONTINUE ";:Y$=INCH$
90 IF Y$="Y" THEN 10 ELSE END
```

This program illustrates the use of INPUT# and also PRINT# (see Page 183). Data must be provided externally to the RS232 serial port in order to run this program successfully.

X represents the number of items of data to be read. First X is read, then data is accepted from input device 2. This data is printed, as it is read, on the printer (output device 1).

After X items have been read both input and output revert back to device 0 (keyboard and VDU screen) for further instructions from the user.

USE WITH FILES

Syntax: INPUT# SV,J; <variable list>

Purpose: This is a File-Handling Command which takes input from the file specified by the file descriptor SV, starting at the first character of the record given by J.

For information relating to the application of this command refer to the section on FILE HANDLING, Page 264 of this manual.

Related Keywords: INCH INCH$ PRINT PRINT#

INT

INT (Integer)

Syntax: INT (N)

Purpose: This is a Function which returns the largest INTEGER value **less** than or **equal** to the value given by N.

EXAMPLE:

X=INT(4.62132) - returns a value of 4

X=INT(-4.62132) - returns a value of -5

(Note the effect on negative numbers).

EXAMPLE:

```
PRINT INT(9.72513)
```

The value 9 will appear on the screen.

Related Keywords: MOD

IOM

IOM (Input Output Mode)

Syntax: IOM J1,J2

Where J1 is in the range 0 to 15 and J2 can take the value 0 or 1.

Purpose: This command is used to set up various input and output conditions. J1 declares which condition (i.e. which bit of the 2 IOM bytes) is to be set to the value declared in J2.

12 of the 16 IOM bits have been allocated to particular input and output options. The remaining 4 bits are left available for future expansion.

Until this command is used all the IOM BITS are set to 1. Thus all the conditions are operative as specified for when the BITS are set to ON.

IOM BIT ASSIGNMENT:

BIT 0 (Edit Mode):

When = 1, this gives SCREEN EDIT mode.

When = 0, this gives LINE EDIT mode.

EXAMPLE: IOM 0,0

This invokes LINE EDIT mode only.

BIT 1 (Echo Mode):

When = 1, all input characters are echoed to the output device (eg. input from keyboard is echoed to appear on output screen)

When = 0, there is no echo of characters and LINE EDIT mode is automatically selected regardless of the setting of BIT 0 (otherwise the whole system locks up).

EXAMPLE: IOM 1,0

This would prevent echo of characters eg. from keyboard to screen.

BIT 2 (Switch Mode):

When = 1, the system automatically reverts to LINE EDIT mode when a program runs, and back again to SCREEN EDIT mode when the program ends or is abandoned.

When = 0, the automatic change of EDIT mode is prevented (i.e. the current EDIT mode will be maintained before, during, and after a program is run).

BIT 3 (Breaks Mode):

When = 1, this allows the use of the SHIFT-BREAK keys to interrupt a program, and EOF to indicate an "end-of-file".

When = 0, program interruption is not available from the SHIFT-BREAK keys, and an "end-of-file" will only be indicated by the last block in a file being detected. (This is useful for reading files that may contain ANY characters as part of the data eg. "program files".

BIT 4 (Trailing Space Mode):

When = 1, trailing spaces will be printed after any NUMERIC output.

When = 0, no trailing spaces are printed and numbers will run into one another (in the same way that strings do normally).

BIT 5 (Leading Space Mode):

When = 1, a leading space is printed for numeric output of positive numbers, or alternatively a negative sign for the output of negative numbers.

When = 0, no leading spaces are printed. The examples below show a comparision of this effect.

```
IOM 5,1                 IOM 5,0
10 A=456                10 A=456
20 B=765                20 B=765
30 PRINT A;B            30 PRINT A;B

This displays           This displays
456 765                 456765
```

BIT 6 (Automatic LINE FEED Mode);

When = 1, this outputs a **line feed** character whenever a new line is output. In other words the BASIC thinks it is sending a CARRIAGE RETURN/LINE FEED at the end of each line. (The Print Head moves to the beginning of the next line down).

When = 0, this outputs a carriage return only (i.e. Print Head returns to the beginning of the same line).

The setting of this BIT has NO affect on device 0 (the display screen), thus normal PRINT statements to the screen are unchanged.

It is useful when set OFF for reducing the size of a file on output, since the Line Feed code is ignored in input of a file using PRINT#. However, some files may need the Line Feed code, in which case this BIT should be set ON.

BIT 7 (Expand TABs):

When = 1, the TAB character is expanded to the required number of spaces (eg. the comma (,) is expanded to produce a 10 space zone in PRINT statements).

When = 0, the actual TAB character itself is output as an ASCII code, and is then transmitted to the output device in use.

EXAMPLE: IOM 7,0

This TAB character is output as an ASCII code and transmitted to the output device for interpretation (eg. printers which have no special TAB settings). Thus the format of the output on a device can be controlled from the software of the computer.

BIT 8 (Printer Port definition)

When = 1, the printer **port** is defined as the CENTRONICS PORT.

When = 0 the printer **port** is defined as the RS232-C PORT.

BIT 9 (Combined output selection)

When = 1, this causes output to the **selected** output device only.

When = 0 this causes output to device 0 (VDU) as well as the **selected** device.

BIT 10 (Printer Echo)

When = 1, output is to the current device only.

When = 0 this causes the PRINTER to "echo" the current output device.

This command is mutually exclusive with IOM bit 11.

BIT 11

When = 1 this inhibits the modification of "ESC" and "FF" to the VDU when the VDU "echos" the current output device.

When = 0, there is no inhibition of the functions.

This command is mutually exclusive with IOM bit 10.

AS A FUNCTION:

IOM may be used as a function to return the current setting of any of the BITS.

EXAMPLE: PRINT IOM(3)

This will display the current setting of BIT 3 (i.e. either 0 or 1) on the screen.

Related Keywords:

KBD

KBD (Keyboard)

Syntax: KBD

Purpose: This is a Function similar to **INCH** (Page 104) but only scans the keyboard for input. It does **not wait** for an input character.

This function returns a value of 0 if no character is input, or the ASCII value if one has been input.

A useful application of this function (and also KBD$) can be seen in "space invader" type games where horizontal and vertical movement of an object is controlled by allocating certain keys from left/right and up/down movement. The object appears to wait for a response from the particular keys but without halting program execution (i.e. other objects on the screen continue to move).

See also KBD$.

EXAMPLE:

```
x = KBD
```

This will return the ASCII value of any character, input from the keyboard, in x.

Related Keywords: INCH INCH$ INPUT KBD$

KBD$

KBD$ (Keyboard String)

Syntax: KBD$

Purpose: This is a String Function which performs in similar manner to **INCH$** (see Page 105), except that it does not wait for an input character, but will return a null string if no character is available.

This function is only operative with the computer keyboard, irrespective of what alternative input device is in use at the time, whereas **INCH$** responds to all input devices.

This function can be used in the same format as **INCH$** (see Page 105) bearing in mind the points of difference indicated above.

Related Keywords: INCH INCH$ INPUT KBD

KEY

KEY

Syntax: KEY N, <string expression>

Where N is the function key number and the data is the particular function to be allocated to that key

Purpose: The KEY command allows the user to program the 8 special "function keys" labelled F0 to F7 on the keyboard, according to individual requirements.

The key numbers are 0 to 7 in "unshifted" mode (as labelled) and 8 to 15 in shifted mode (i.e. each key can be used for two separate functions)

BASIC reserved words are stored as tokens for efficiency. ASCII characters can also be stored.

The total storage capacity for all 8 keys is 128 bytes. This can be allocated to one key or shared between the 16 functions of all 8 keys.

To display the contents of the function keys use the KEY LIST command.

Notes on Use:
Each time a programmed function key is used, the BASIC statement stored in the string expression is displayed on the screen and ENTER must again be keyed to execute the statement. Alternatively, when programming the function key, GRAPH-ENTER may be keyed after the BASIC statement: this inserts a carriage return at the end of the statement and therefore when the function key is pressed execution is automatic.

The function keys may be programmed in direct mode or indirect mode (i.e. from within a program for use with that program).

The function key number may be expressed as a variable.

EXAMPLE:

KEY 0,"LIST c/r" - this programs function key 0 to list the current program when pressed.

KEY 1,"PRINT CHR$(A) c/r" - this programs function key 1 to print CHR$(A) when pressed.

KEY 3,"BCOL5:TCOL15 c/r" - this programs function key 3 to set backdrop and text colours when pressed.

KEY 5,"B=58:PRINT CHR$(B) c/r" - this programs function key 5 to set variable B to 58 and then print it.

KEY LIST - this will display the contents of all the function keys, on the screen, as shown in the example display given below.

```
F0: LIST c/r
F1: PRINT CHR$(A) c/r
F2:
F3: BCOL5:TCOL15 c/r
F4:
F5: B = 58:PRINT CHR$(B) c/r
F6:
F7:
```

sF0:
sF1:
sF2:
sF3:
sF4:
sF5:
sF6:
sF7:

Note that shifted functions are indicated by sF0 to sF7.

Related Keywords:

LEFT$

LEFT$ (Left String)

Syntax: LEFT$ (<string expression>**,J)**

Purpose: This is a String Function which returns the left most J characters from the string expression.

EXAMPLES:

a) PRINT LEFT$ ("ABCDEF",3)

The 3 left most characters of the string will appear on the screen (i.e. "ABC")

b) A$ = LEFT$("XYZLBJ",4)

This will return the 4 left most characters in A$. Thus A$ will be "XYZL"

Related Keywords: LEN$ MID$ MUL$ RIGHT$ SCRN$

LET NAME$="JOHN"	Assigns the string "JOHN" to the string variable NAME$
C(4)=6	Assigns the value 6 to variable C(4) which represents the fifth element in the array C.
D$="NO"	Assigns the string "NO" to the string variable D$.
NAME$="FRED"	Assigns the string "FRED" to the string variable NAME$.
A%=PI	This would give the same result as A=INT(PI)

Therefore the value assigned to A% would be 3. (PI=3.14159)

Related Keywords:

LIST

LIST

Syntax: LIST L1,L2,L3

L1 is a line number from which the listing will commence and defaults to 0 if omitted.

L2 is the number of lines to be listed at one time and defaults to 15 until changed by a value in the command.

L3 is the last line number to be listed.

L1,L2,L3, can also be in the form of EXPRESSIONS.

Purpose: This is a System Command which lists the current program to the current output device.

After L2 lines have been listed there is a pause. The listing will continue when the user presses the space-bar or any key, and another L2 lines will be listed. BASIC always remembers the last value used for L2 and will continue to use that value until LIST is used with a different L2 value.

Any or all of the expressions L1,L2,L3, may be omitted but if L2 or L3 are specified, either individually or together, then the appropriate "commas" must be inserted.

Listings may be abandoned at any time, whether paused or not, by use of the ESC key.

When a listing has paused, the cursor movement keys may be used for editing purposes and this simultaneously abandons the listing without the use of ESC. (This is only applicable in SCREEN EDIT MODE).

LIST may be used as a normal statement **within** a program. This is useful for displaying segments of a program listing during execution, and a delay could be incorporated so as to hold the display on the screen for a short period of time. REM statements containing titles, and statements containing data, for example, could be programmed to appear for a short time on the screen at predetermined stages of execution.

EXAMPLES:

LIST 300,5,999 - This will list 5 lines at a time starting at line 300 and ending at line 999.

LIST - Lists the whole program.

LIST,5 - Lists the whole program, 5 lines at a time.

LIST 100,7 - Lists 7 lines at a time, starting from line 100

LIST 200 - Lists from line 200 using the last used value for L2 for the number of lines to be listed at one time

LIST 100,,199 - Lists from 100 to 199 using the last used value for L2 for the number of lines to be listed at one time.

LIST,4,299 - Lists 4 lines at a time from the start of the program to line 299.

LIST,,199 - Lists from the start of the program to line 199 using the last used value for L2 for the number of lines to be listed at one time.

X=100:Y=50:LISTX,Y,X+99 - This will list from 100 to 199, 50 lines at a time.

Related Keywords: LISTP

LISTP

LISTP (List to Printer)

Syntax: LISTP L1,L2,L3

Purpose: This is a System Command which will list the current program to the printer from line number L1 to line number L3.

If L1, L2, and L3 are omitted the whole program will be listed.

It is the same as doing PRINT#1;LIST (see Page 183).

During the listing, L2 the 'number of lines at a time' feature incorporated in the LIST command is ignored and the list is printed continuously up to line L3.

After listing, the output reverts back to device 0 (the monitor)

If the command is called from within a program, device 1 (the printer) will remain selected.

Related Keywords: LIST PRINT# PRINT

LN

LN (Logarithm - Natural)

Syntax: LN(N)

Purpose: This is a Function which returns the NATURAL LOGARITHM of the numeric expression N (Natural logs are to the base e).

If the value given by N is less than, or equal to, zero a QTY ERROR occurs.

EXAMPLE:

```
PRINT LN(2)
```

The value 0.693147 will appear on the screen.

Related Keywords: EXP LOG

LOAD

LOAD

Syntax: LOAD <file>

(See file naming conventions Page 264)

Purpose: This is a System Command which loads files/programs from the disc into the computor's memory. If a file is not present on the target disc then a NO FILE ERROR will be given.

If the drive name is omitted from within the **<file>** the current default drive will be assumed. The default drive is initially set up as 0 but can be changed using the DRIVE command (see Page 69).

If the file type is not declared within **<file>** then XBS will be assumed.

LOADING BASIC PROGRAM FILES

When loading BASIC program files any existing BASIC program in memory is deleted but variables are **not** destroyed.

EXAMPLE: LOAD "1:PROG.XBS"

This will load the BASIC program file whose name is PROG from the disc in drive 1 (assuming, of course, that drive 1 is fitted into EINSTEIN).

EXAMPLE: LOAD "PROG"

This will load the BASIC program file PROG.XBS from the default drive 0 (unless changed by the DRIVE command) into the computer's memory.

LOADING ASCII PROGRAM FILES

When loading ASCII program files, any existing program is **not** deleted and variables are **not** destroyed. Thus extra routines may be added to existing programs and the extra lines will appear at their correct positions in relation to the existing ones (care being taken not to duplicate line numbers otherwise the original will be over written by the new line).

If a "new" program is to be loaded as an ASCII file then the NEW command must be executed prior to loading (see Page 155). In this mode the user may observe the program file loading line by line on the screen.

EXAMPLE: LOAD "1:TEST.ASC"

This will load the ASCII program file TEST.ASC from drive 1 (regardless of current default drive) into the computer's memory, the lines of the program appearing on the screen as it is loading.

LOADING OBJECT PROGRAM FILES (MACHINE-CODE FILES)

When loading OBJECT files, an area in memory has to be reserved prior to loading using the CLEAR command (see Page 48).

The start address will be assumed to be the first location above the cleared area. (i.e. if CLEAR &9FFF has been used then loading will commence from &A000).

Machine Code Subroutines can be loaded during execution of a BASIC program using this facility and then accessed by use of the CALL command as required. The subroutine should terminate with a &C9 code in order to return execution to the BASIC program at the line immediately following that containing the CALL command.

If the size of a file is larger than the area of memory available then a MEM FULL ERROR will be given.

EXAMPLE: LOAD "0:ROUTINES.OBJ"

This will load the machine-code routines, or data from the file ROUTINES.OBJ, from the disc currently in drive 0, into the area of memory previously reserved by a CLEAR command, i.e. &A000 in the example above.

GENERAL NOTE: If a file is not present on a particular disc then a NO FILE ERROR will be given.

Related Keywords: CLEAR DRIVE SAVE

LOCK

LOCK

Syntax: LOCK <file>

Purpose: This is a Disc Command which **locks** the file given by <file> for the disc in the current default drive. (see Page 264 for file name conventions).

Files PROTECTED in this manner may not be re-written over, erased, or renamed (i.e. cannot be corrupted). They are identified in the directory display by an * symbol before the file .

If the file does not exist a NO FILE error occurs.

Related Keywords: UNLOCK

LOG

LOG (Logarithm)

Syntax: LOG (N)

Purpose: This is a Function which returns the LOGARITHM to the base 10, of the value given by N.

If N is less than or equal to zero a QTY ERROR occurs.

EXAMPLE:

```
PRINT LOG (2)
```

The value .30103 will appear on the screen.

Related Keywords: LN EXP

MAG

MAG (Magnify)

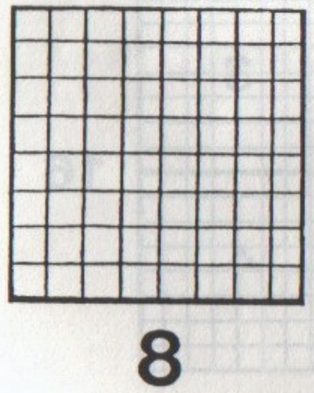
Syntax: MAG J

Where J can have a value from 0 to 3, each number representing a particular sprite magnification. All 32 sprites are affected equally.

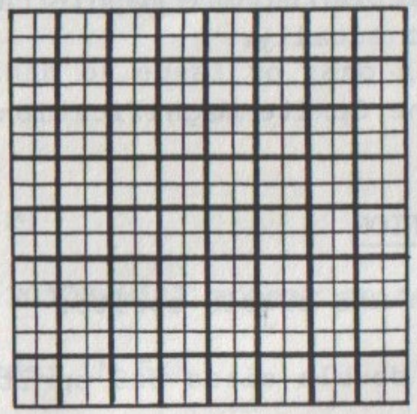
Purpose: This is the Sprite Magnification command (see SPRITE Page 216).

J		MAGNIFICATION
0	-	defines an 8x8 pixels sprite.
1	-	defines a double sized 8x8 sprite
2	-	defines a 16x16 pixels sprite
3	-	defines a double sized 16x16 sprite

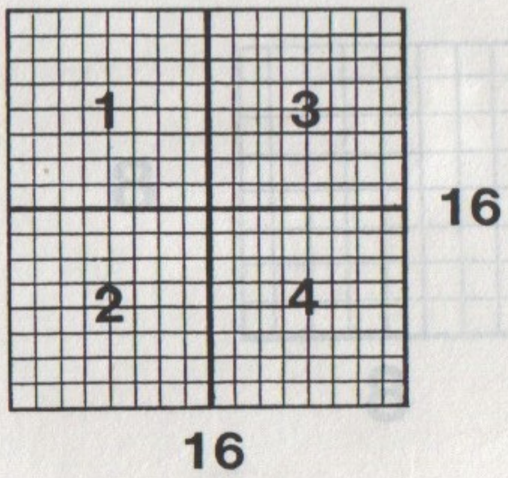
MAG 0 and 1 apply to a shape which has been defined on a single 8x8 pixel grid. In MAG 0 the shape remains as a single 8x8 pixel character as shown below.



In MAG 1 the single 8x8 shape is doubled in size (magnified) to occupy an area equivalent to a 16x16 pixel grid. Each original pixel is in fact made 4 times larger, resulting in a grid of 8x8 larger pixels as shown below.



MAG 2 and 3 apply to a shape which is built up from four 8x8 pixel grid shapes to form a single shape on a 16x16 grid. In MAG 2 the four shapes are printed as a single shape as a 16x16 pixel grid as shown below.



When "defining" the complete shape the data for each 8x8 grid should be used with the SHAPE command in the order shown above. In MAG 3 the shape on this 16x16 grid is doubled in size to occuply an area equivalent to 32x32 pixels.

EXAMPLE:

```
MAG 1
```

Following this command all 32 sprites would be double sized 8x8 until another MAG command was introduced to change the selections.

Related Keywords: SHAPE SPRITE

MGE

MGE (Merge)

Syntax: MGE

Purpose: This is a System Command used to "merge" sections of a program which have been "held" or, add a second program to one already in memory. (see HOLD Page 98).

MGE takes no account of two or more lines having the same number and therefore both lines would appear in the final result.

EXAMPLE:

```
10 REM LINES KEPT IN VIEW
20 PRINT
30 PRINT "*"
40 PRINT
50 PRINT "#"
60 REM LINES HIDDEN
70 PRINT "A"
80 PRINT
90 PRINT "B"
100 PRINT
```

A HOLD 10,50 will keep lines 10 to 50 of the program while lines 60 to 100 are apparently lost. Thus a LIST after this use of HOLD would give the following.

```
10 REM LINES KEPT IN VIEW
20 PRINT
30 PRINT "*"
40 PRINT
50 PRINT "#"
```


The execution of MGE brings back lines 60 to 100 and if LIST is used after MGE the original listing will reappear (i.e. lines 10 to 100).

See also the example mailing list in File Handling (Page 264) which illustrates the use of MGE.

Related Keywords: CHAIN HOLD RENUM

MID$

MID$ (Middle of String)

Syntax: MID$ (<string expression>, J1,J2)

Purpose: This is a Standard String Function which returns a number of characters, given by the value of J2, starting from the character position given by J1, in respect of the string specified in the function.

J2 may be omitted, in which case the remainder of the string, starting from the character position given by J1, is returned.

EXAMPLES:

```
PRINT MID$("HELLO",3,2)
```

The result will appear on the screen as "LL".

```
PRINT MID$("HELLO",3)
```

The result will appear on the screen as "LLO".

Related Keywords: LEFT$ RIGHT$

MOD

MOD (Modulus)

Syntax: I1 MOD I2

Purpose: This is an ARTHMETIC OPERATOR which forms an expression equal to the integer value of the **remainder** resulting when I1 is divided by I2.

EXAMPLE:

```
5 MOD 3  this returns a value of 2
```

Related Keywords:

MOS

MOS (Machine Operating System)

Syntax: MOS

Purpose: This is a System Command and is used to transfer control to the "Machine Code Monitor" section of the "Machine Operating System".

The X and Y commands in MOS can be used for cold and warm starts when re entering the BASIC (see MOS/DOS manual).

Related Keywords: DOS

MUL$

MUL$ (Multiple String)

Syntax: MUL$(<string expression> ,J)

Purpose: This is a String Function which will cause a string to be repeated the number of times, given by the value of J (i.e. String Multiplication).

The resultant string must not exceed 255 characters.

EXAMPLE: PRINT MUL$("AB",10)

This will give the following display.

ABABABABABABABABABAB

It is a useful function for displaying repeated patterns.

EXAMPLES:

i) PRINT MUL$("*",15)

This will give the following.

****

ii) PRINT MUL$("+=",6)

This will give the following.

+=+=+=+=+=+=

Related Keywords: STR$

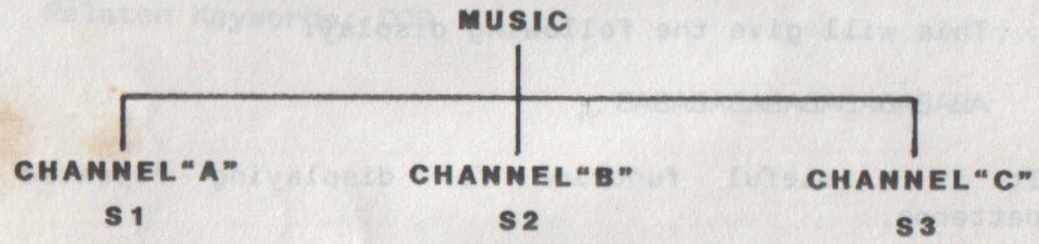
MUSIC

MUSIC

Syntax: MUSIC S1;S2;S3

Purpose: This command can be used to create various sounds and play tunes composed by the user.

String expressions S1,S2 and S3 each represent musical expressions. These expressions are each allocated to a separate channel, the 3 channels are then played simultaneously.



It is not necessary to use all the three channels every time and any expressions which are left "empty" will cause a previous string expression specified for that channel to be replayed.It is therfore good practice to specify a rest (R) for channels not required to sound. The MUSIC statement does not end until all three channels have completed their appropriate measures. This makes it important to ensure that all three channels have the same execution time within a given measure.

The contents of the expressions S1,S2, and S3 indicate the actual structure of the music to be played. Individual notes are indicated by the letters CDEFGAB, these representing the chromatic scale.

The following can be included in a music string:-

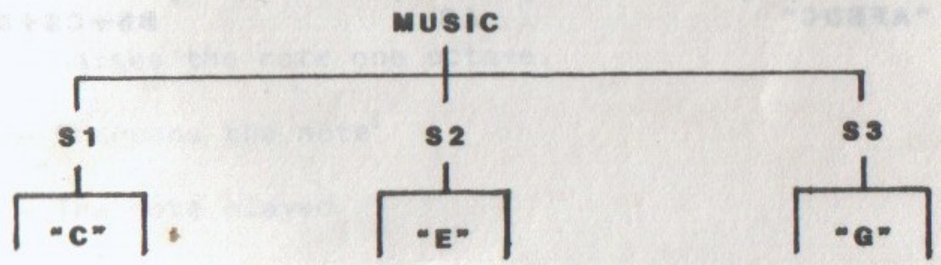
1. R represents a 'rest'.
2. X represents a 'beat', and turns on the noise generator of the PSG rather than a tone.
3. Vn, where n is numeric data in the range 0 to 6 which selects a voice defined using the VOICE commands.

The duration of a note (or rest) is specified by a number, in the range 0 to 9, immediately following the note (or rest), giving the note values as listed below.

0 = 32nd note (demi-semi-quaver)
1 = 16th note (semi-quaver)
2 = "dotted" 16th note
3 = 8th note (quaver)
4 = "dotted" 8th note
5 = Quarter note (crotchet)
6 = "dotted" quarter note
7 = Half note (minim)
8 = "dotted" half note
9 = Whole note (semi-breve)

If the duration specification is missing off a note, the duration defaults to the last specified duration on that channel. On power up the duration is 5.

In a simple form each expression would contain a single note.



When written into the music command the following format would be given.

```
MUSIC "CR","ER","GR"
```

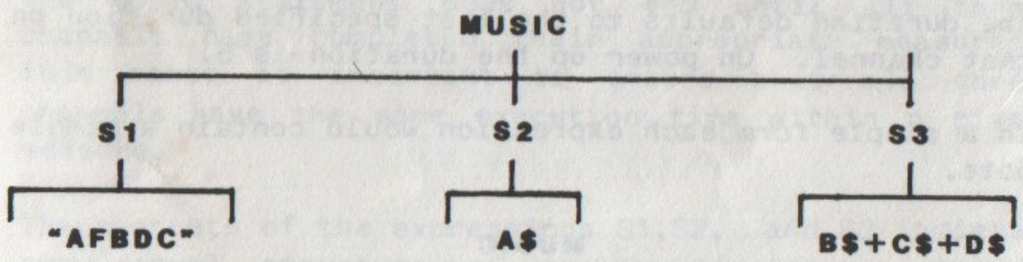
All three notes would be played simultaneously thereby producing a chord. It is of course possible to play one single channel by omitting the other two from the command as shown below.

```
MUSIC "CEGR","R","R"
```

This would play the three single notes in turn, on channel A, one after the other, followed by a rest in order to silence the last note.

The simple format, shown above, can be extended such that the contents of each channel may be represented in any of the following formats.

a) a single String of notes.
b) a single String Variable (previously allocated to a set of notes).
c) a combination of several String Variables or Strings.



EXAMPLE:

```
10 A$ = "AFGCDE"
20 B$ = "EFCDG"
30 C$ = "CDEGBR"
40 MUSIC "GEBCD",A$,B$+C$
```

To increase the octave range each note can be preceded by a symbol to produce the effect given in the list which follows:-

+ - raises the note one octave above middle c.
' - raises the note two octaves above middle c.
- - lowers the note one octave below middle c.
= - lowers the note two octaves below middle c.
. - lowers the note three octaves below middle c.

"Sharps" and "flats" are indicated as follows.

- precedes the note and indicates a "sharp".
b - precedes the note and indicates a "flat".

EXAMPLE:

```
10 MUSIC "VO +#A7 =bE5 R","R","R"
```

Music is played on channel A as follows:-

VO - VOICE 0 is selected (VOICE 0 having been defined in a voice statement).

+ - Raises the note one octave.

- Sharpens the note[3]

A - The note played

7 - Duration setting of a half note (minim)

NOTE: Spaces included in the string are ignored and can be used to increase legibility.

The second note played is E flat played two octaves down for a duration of a quarter note (crochet).

Related Keywords: BEEP PSG TEMP VOICE

NEW

NEW

Syntax: NEW

Purpose: This is a System Command which deletes all program lines and variables currently in the memory.

It is used to delete any program currently in memory in preparation for the input of another program and different variables.

Used in "Direct Mode".

Related Keywords:

NEXT

NEXT

Syntax: NEXT V1,V2,..,Vn

V1,V2, etc. are the **control variables** used in the associated FOR statements, and must be numeric.

Purpose: This statement is used to indicate the end of a FOR statement loop.

NEXT is always used in conjunction with the FOR statement to produce what is commonly referred to as a FOR-NEXT LOOP.

For a more detailed explanation of the combined use of the FOR and NEXT statements see Page 88.

EXAMPLE:

```
10 CLS
20 FOR I=1 TO  10
30 PRINT I
40 NEXT I
50 END
```

Related Keywords: FOR STEP TO

NOT

NOT

Syntax: <statement> **NOT** <statement>

Purpose: This is a LOGICAL OPERATOR used in the evaluation/comparision of statements.

EXAMPLE:

```
IF NOT X THEN END
```

This shows the use of NOT in an "IF" statement. In other words IF the condition is NOT true THEN act accordingly.

Related Keywords: AND OR XOR

NULL

NULL

Syntax: NULL J

The value given by J is the number of nulls to be printed and can be from 0 to 255. The default value is 0.

Purpose: This is a special command relating to output. It sets the number of nulls to be printed after every CARRIAGE RETURN (i.e. acts as a delay).

This command is designed to be used when operating slow serial devices, where the CARRIAGE RETURN code may take a little over the time allowed for one character to print. (Thus causing errors).

The correct setting for a particular device will be found by experimenting with various values but normally 1 would be sufficient for most devices.

NULL can be used as a function to return the current setting.

EXAMPLE: NULL 5

PRINT NULL

This will display the value of the current setting for NULL on the screen (i.e. 5 in this instance).

Related Keywords: SPEED

OFF

OFF

Syntax: OFF <statement>

Purpose: This command is used to disable a particular action or operation setting. Often used with ERR and EOF commands. Refer to Page 249 for further information.

EXAMPLES:

OFF ERR - Disables the ON ERROR trap
OFF EOF - Disables the ON EOF (end-of-file) trap

Related Keywords: EOF ERR

ON

ON

Syntax: ON J GOTO L1,L2,..,Ln

J is an expression and L1,L2 etc are line numbers. If J is negative then a QTY ERROR occurs.

Purpose: This alters the order in which BASIC executes a program by jumping to one of a selection of lines depending on the value of expression J.

The expression J is evaluated to give a number. If the number is 1 then execution transfers to the 1st line number after the GOTO. If the number is 2 then execution transfers to the 2nd line number after GOTO. and so on.

FXAMPLE:

```
 10 INPUT "TYPE IN A NUMBER FROM (1-5)";A
 20 PRINT "YOU HAVE SELECTED";
 30 ON A GOTO 40, 60, 80, 100, 120
 40 PRINT "GLASSES";
 50 GOTO 130
 60 PRINT "CUPS;
 70 GOTO 130
 80 PRINT "TANKARDS";
 90 GOTO 130
100 PRINT "DISHES";
110 GOTO 130
120 PRINT "MUGS";
130 END
```

When this program is RUN the following appears on the screen to begin with.

```
TYPE IN A NUMBER FROM (1-5)
```


When a number is typed in then the appropriate line number is selected from the ON-GOTO statement.

```
NUMBER 1 selects line number 40
NUMBER 2 selects line number 60
NUMBER 3 selects line numner 80
NUMBER 4 selects line number 100
NUMBER 5 selects line number 120
```

Execution is then transferred to the selected line and the appropriate action is taken; in this case a message is output to the screen. Thus transfer of execution has been controlled by the variable A.

Syntax: ON J GOSUB L1,L2,..Ln

Purpose: The same principle can be applied to accessing subroutines using the GOSUB format.

Thus several subroutines can be accessed from a single line of program depending on the evaluation of a variable.

EXAMPLE:

```
 10 REM PROCESSING TO EVALUATE A VARIABLE B
 20 - - - - - - - - - -
 30 - - - - - - - - - -
 40 - - - - - - - - - -
 50 - - - - - - - - - -
 60 ON B GOSUB 100,140,180.
 70 - - - - - - - - - -
 80 - - - - - - - - - -
 90 END
100 SUBROUTINE 1
110 - - - - - - - - - -
120 - - - - - - - - - -
130 RETURN
```

```
140 SUBROUTINE 2
150 - - - - - - - - - - -
160 - - - - - - - - - - -
170 RETURN
180 SUBROUTINE 3
190 - - - - - - - - - - -
200 - - - - - - - - - - -
210 RETURN
```

Let us assume that there are three subroutines in the program which will be accessed by the ON-GOTO statement in line 60

The variable B is evaluated by the processing involved in lines 20 to 50 and will produce a number from (1-3).

The Subroutines are then accessed via the ON-GOSUB statement, selection depending on the value of B.

If B = 1 then line 100 (subroutine 1) would be accessed.
If B = 2 then line 140 (subroutine 2) would be accessed.
If B = 3 then line 180 (subroutine 3) would be accessed.

The subroutines would RETURN to line 70 as normal to continue program execution. (see Page 199).

Syntax: ON ERR GOTO
ON ERR GOSUB
ON EOF

Purpose: See ERR and EOF commands and Error Handling section.

Related Keywords: EOF ERR GOSUB GOTO OFF

OPEN

OPEN

Syntax: OPEN <file>,SV,R

Purpose: This is a File-Handling Command which opens an existing file, assigns internal file information and buffer space to the file descriptor, and indicates the record size to be used.

SV is a string variable name (but not a string array element) and is the file descriptor.

R is the random record size (length) and is given as a value in the range 0 to 65535, indicating the number of characters involved. R is only specified for "random access"; if "sequential access" is to be applied then R is omitted (in fact a random record length of 0 indicates that sequential access is to be performed).

If a file is not present on the specified drive then a NO FILE ERROR will be given.

EXAMPLE:

```
OPEN "O:SILLY.DAT",FD$,15
```

This will perform the following:-

a) opens the file SILLY.DAT on the disc currently in drive 0.
b) assigns FD$ as the file descriptor.
c) sets up for random access using 15-character length records.

Related Keywords: APPEND CLOSE CREATE

OR

OR

Syntax: <statement> **OR** <statement>

Purpose: This is a LOGICAL OPERATOR used in the evaluation/comparison of statements (i.e. defines an alternative).

EXAMPLE:

```
10 IF (X AND Y)=3 OR Y=0 THEN 100
```

This will cause execution to transfer to line 100 if the result of either of the statements (X and Y)=3, or Y=0 is TRUE

Related Keywords: AND NOT XOR

ORIGIN

ORIGIN

Syntax: ORIGIN x,y

Both x and y can have values in the range -32768 to +32767 and are the co-ordinates of a point on the screen.

Purpose: This statement defines the origin of the imaginary screen grid in respect of PLOT, UNPLOT, DRAW, ELLIPSE, and POLY commands, x and y being the co-ordinates of the new origin. On entry to BASIC the default co-ordinates are 0,0 and this represents the bottom left hand corner of the screen.

EXAMPLES:

a) ORIGIN 20, 24

This would establish the new position of the origin on the screen grid and all subsequent PLOT, UNPLOT, DRAW, ELLIPSE, and POLY commands would be executed relative to the new origin.

b) PLOT 128,96

this plots a point at the cente of the screen, assuming that the origin has not been redefined since entry to BASIC.

whereas:-

ORIGIN 128,96:PLOT 128,96

this plots a point at the top right hand corner of the screen.

Related Keywords: DRAW ELLIPSE PLOT POINT POLY UNPLOT

OUT

OUT

Syntax: OUT J1,J2

J2 is the value which is to be output, and J1 is the "address" of the Z80 port.

Purpose: This command provides direct output to the ports of the computer.

EXAMPLE:

The 8 bit user port is at I/O address &32 so this will be the value of J1 each time this command is used to access the port.

```
OUT &32,5
```

This sends the value 5 to the 8 bit user port (&32).

Further information relating to the functioning of the 8 bit user port will be found in the INTRODUCTORY MANUAL.

Related Keywords: INP WAIT

PEEK

PEEK

Syntax: PEEK (I)

I must evaluate to a number in the range 0 to +65535 and can be given in decimal or hexadecimal.

Purpose: This is a Machine Code related command which returns an integer in the range 0 to 255 which represents the contents of the memory location given by I.

EXAMPLE:

```
PRINT PEEK (&4100)
```

This will display the contents of memory location & 4100 as an integer.

Related Keywords: CALL DEEK DOKE POKE VDEEK VDOKE VPEEK VPOKE

PI

PI

Syntax: PI

Purpose: This is a Function which returns the value of pi as 3.14159 for use in expressions.

It is much faster than using a variable to hold the value of pi.

EXAMPLE:

```
PRINT PI
```

The value 3.14159 will appear on the screen.

Related Keywords:

PLOT

PLOT

Syntax: PLOT x,y

x and y are the co-ordinates of any point on the screen. x being horizontal and ranging from -32768 to +32767, y being vertical and ranging from -32768 to +32767.

Purpose: This is a Command used to illuminate (turn on) a single pixel point on the display screen.

The screen is divided up into a GRID of 256 pixels horizontal and 192 pixels vertical.

If the ORIGIN is defined as 0,0 then only pixels in the range x = 0 to 255 and y = 0 to 191 can be illuminated on the screen. However, all other values in the range specified above are allowed, but will be off the screen.

If the ORIGIN was defined -16383, -16383, then the display area would effectively be x = -16383 to - 16128 and y = -16383 to -16191.

EXAMPLE:

```
PLOT 120,90
```

This will illuminate the pixel at 120,90 in the current foreground graphics colour (as set by a previous GCOL command), provided the ORIGIN is defined as 0,0

Related Keywords: DRAW ELLIPSE ORIGIN POINT POLY UNPLOT

POINT

POINT

Syntax: POINT (x,y)

x and y are the co-ordinates of a graphics point on the imaginary screen grid. Values for x and y can be in the range -32768 to +32767.

Purpose: This is a function which returns a value corresponding to the state of the pixel at x,y. 0 = pixel off and 1 = pixel illuminated.

EXAMPLE:

```
X = POINT (70,65)
PRINT X
```

This will display a value of 0 or 1 according to the condition of the point whose co-ordinates are 70,65. (i.e. whether lit or not).

Related Keywords: DRAW ELLIPSE ORIGIN PLOT POLY UNPLOT

POKE

POKE

Syntax: POKE I,J1,J2,..,Jn

Purpose: This is a Machine Code related command which places the values of the expressions J1,J2 etc., into memory, **starting** at the location given by I.

EXAMPLES:

```
POKE 16384,132
```

This will place 132 (i.e.&84) into location 16384 (i.e.&4000)

```
POKE &5100,&77,&34,&61
```

This will put &77 into location &5100, &34 into location &5101, and &61 into location &5102.

Related Keywords: DEEK DOKE PEEK VDEEK VDOKE VPEEK VPOKE

POLY

POLY (Polygon)

Syntax: POLY N,x,y,R,T,z

Purpose: This graphics command will draw a polygon according to the values given in the parameters.

N is the number of sides of the polygon. x,y are the co-ordinates of the centre of the polygon and can have values in the range -32768 to +32767. R is the distance from point (x,y) to the vertices of the polygon i.e. R is the horizontal radius of an ellipse which would contain the polygon and T is the ellipse qualifier given by the following:-

$$T = \frac{\text{VERTICAL AXIS (of ellipse)}}{\text{HORIZONTAL AXIS (of ellipse)}}$$

If T is omitted it will default to 4/3 thereby having the same effect as in the ELLIPSE command, resulting in a REGULAR POLYGON (owing to the aspect ratio of the screen being 4:3).

POLY N,x,y,R

z is a number in the range 0 to 5 indicating the type of line to be drawn (if omitted z will default to 0).

0 - Continuous Line
1 - Continuous Unplot
2 - Dotted line 2 dots on, 2 dot off.
3 - Dashed line 4 dots on, 2 dots off.
4 - Dotted-Dashed line 10 dots on, 2 dots off, 2 dots on, 2 dots off.
5 - Dashed-dotted line 10 dots off, 2 dots on, 2 dots off, 2 dots on, 10 dots off.

Related Keywords: DRAW ELLIPSE ORIGIN PLOT UNPLOT

POP

POP

Syntax: POP

Purpose: This statement is used in association with subroutines.

It allows a **nested** subroutine to return to the statement immediately following the GOSUB statement preceding the GOSUB relating to the particular routine which is being executed.

POP is only used from within a **nested** subroutine and this is best explained by example:-

```
 10 GOSUB 50
 20 - - - -
 30 - - - -
 40 END
 50 REM.SUBROUTINE
 60 - - - -
 70 - - - -
 80 GOSUB 120
 90 - - - -
100 - - - -
110 RETURN
120 REM-NESTED SUBROUTINE
130 - - - -
140 - - - -
150 - - TEST CONDITION
160 IF "TEST CONDITION" THEN POP
170 RETURN
```

OUTER SUBROUTINE CALLED BY LINE 10

INNER SUBROUTINE (NESTED) CALLED BY LINE 80

LINE 10 calls up a subroutine which starts at line 50.

LINE 80 calls up a second subroutine from within the existing subroutine (i.e. the nested subroutine), which starts at line 120

This second subroutine would normally return to line 90 and continue with the remainder of the original subroutine. However, as a result of some kind of "test condition" we might wish it to "return" to line 20 (i.e. to the line following the **original** GOSUB call). Thus the processing contained within lines 90 and 100 of the original subroutine would be omitted. It is this facility which POP provides.

EXAMPLE:

In the example given below, the result from execution of lines 230 and 240 will direct subsequent execution to **either** invoke POP or transfer to line 250.

The resulting action from line 250 is indicated by the arrowed lines and will determine whether or not the # symbol will be printed (as contained in the processing of lines 160, 170, and 180).

If the POP statement is executed, i.e. A is not 2, the RETURN statement will direct execution to line 50 rather than returning to line 150.

EXAMPLE:

```
 10 PRINT "NUMBERS"
 20 GOSUB 70
 30 PRINT "LETTERS"
 40 GOSUB 110
 50 PRINT "END OF THIS SEQUENCE"
 60 END

 70 FOR I = 1 TO 5
 80 PRINT I
 90 NEXT I
100 RETURN
```

FIRST SUBROUTINE CALLED BY LINE 20

```
110 FOR I = 1 TO 5
120 PRINT "ABCDE"
130 NEXT I
140 GOSUB 200
150 PRINT "END OF SYMBOLS"
160 FOR I = 1 TO 5
170 PRINT "#"
180 NEXT I
190 RETURN
```

SECOND SUBROUTINE CALLED BY LINE 40

```
200 FOR I = 1 TO 5
210 PRINT "*"
220 NEXT I
230 INPUT "IF NUMBERS TYPE 1 IF SYMBOLS TYPE 2";A
240 IF A <> 2 THEN POP
250 RETURN
```

NESTED SUBROUTINES CALLED BY LINE 140

Related Keywords: GOSUB

POS

POS (Position)

Syntax: POS (J)

J can be 0,1, or 2, each number associating a particular function.

Purpose: This Function is used to obtain the current output column or row position, depending on J as below.

POS (0) -
This gives the PRINT COLUMN count of the current output device. It is independant of screen size and is zeroed when a CARRIAGE RETURN, HOME or CLEAR SCREEN/FORM FEED code is output, or if the column count exceeds 255. If output is not directed to any other output device then POS(0) reflects the cursor column position on the screen.

POS (1) -
This gives the current column position of the cursor on the screen.

POS (2) -
This gives the current row position of the cursor on the screen.

POS(1) and POS(2) are designed to be used in association with the PRINT @ facility.

EXAMPLE:

```
PRINT POS(0)
```

This will display the current value of the "column count" of the current output device.

Related Keywords: PRINT PRINT@

PRINT

PRINT

Syntax: PRINT E

Where E can be a single expression or a list of expressions, which may be numeric or string type.

Purpose: This command is used to send data to the current output device (screen, printer, etc.).

PRINT may be abbreviated to ? when typed in as a line of program text. When the program is listed this will appear as PRINT, but will not affect the query (?) character where it appears elsewhere in program text.

A PRINT statement on its own will generate a carriage return and line feed (i.e. leaves a line blank and moves immediately to the beginning of the next line down).

If several expressions are used they are separated by one of a selection of SEPARATORS. These SEPARATORS control the presentation of the final output.

A CRLF (carriage-return, line-feed) is generated at the end of PRINT statements except when a comma (,) or semi-colon (;) separator appears earlier at the end of a print statement.

NOTE: CRLF is still generated if the print output extends into the last (right-hand) column.

SEPARATORS:

SEMI-COLON ;

This leaves the cursor where it is so that the next expression will print from the end of the previous one.

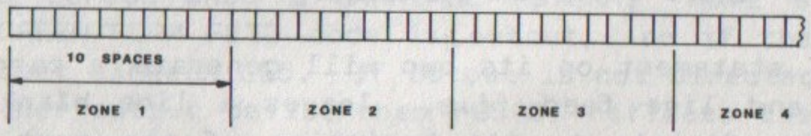
EXAMPLE: PRINT "JABBER";"WOGGY"

This will be output as follows:-

JABBERWOGGY

COMMA

This moves the cursor to the start of the next PRINT ZONE. PRINT ZONES are simply specified positions in a line of text and are situated at intervals of 10 spaces (character positions), thus creating a series of ZONES.



EXAMPLE: PRINT "JABBER","WOGGY","WOGGY"

This will be output as follows:-

JABBER WOGGY WOGGY

Each line on the screen display contains 40 character positions (in 40 column display) and therefore there are 4 PRINT ZONES.

The ZONE LIMIT indicates the character position at which the final ZONE starts in a line of text. When the existing printing has gone beyond this point the next expression will be printed at the beginning of the next line (i.e. a carriage-return, line-feed is generated).

The ZONE WIDTH and ZONE LIMIT, may be modified by use of the ZONE command (see Page 247).

The @ symbol.

This allows printing of expressions to commence from a specified point on the screen by use of co-ordinates. (See **PRINT@**).

PRINTING NUMBERS:

All numbers are printed with a **leading** and **trailing** space.

EXAMPLE: PRINT "TO YOU";987,71;321

This will give the following output:-

TO YOU 987 71 321

The **leading** space is reserved for the sign of the number (+ or -) which is only shown when the number is negative. Both spaces may be removed, if desired, by use of the IOM command (see Page 117).

Numeric printout may be specially formatted by use of the FMT command (see Page 84).

Related Keywords: FMT IOM PRINT @ PRINT# SPC TAB WIDTH ZONE

PRINT@

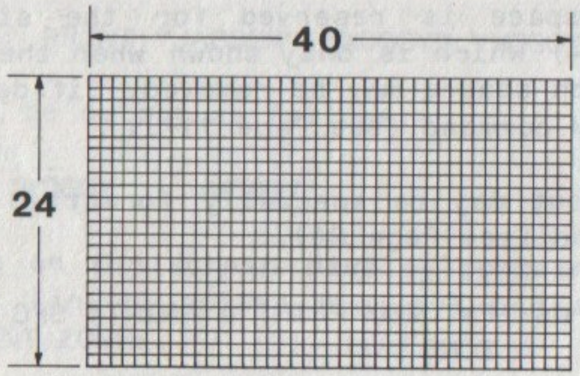
PRINT@

Syntax: PRINT @ x,y,E

Where x and y are the co-ordinates of the first character position to be used and E is the expression to be output. x and y can have values in the range 0 to 255.

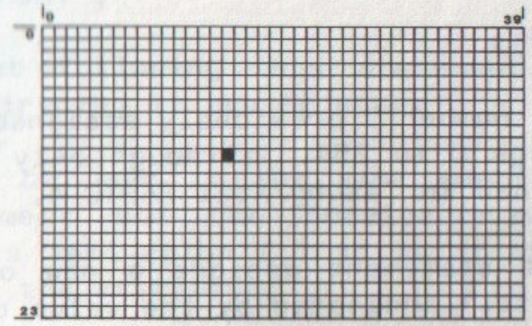
Purpose: This command allows print of expressions to commence from a specified point on the screen by use of the co-ordinates x and y. The co-ordinates must each be separated by a comma and there must be either a comma or semi-colon between the expression E and the co-ordinates (in this instance the commas and semi-colons are **not** executed as PRINT SEPARATORS).

The screen is divided into a grid of 40 columns across by 24 rows down (in 40 column display).



Each "box" of the grid represents a character position and can be referenced by the number of the column (0 to 39) and the number of the row (0 to 23) in which it is situated.

EXAMPLE:



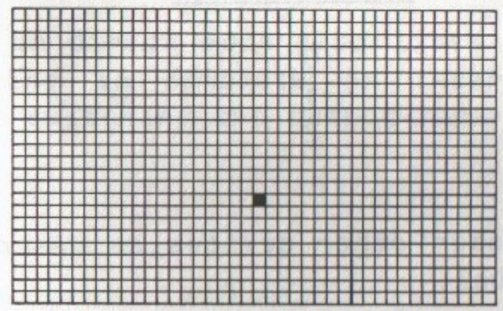
CHARACTER SPACE 15,10

This character position is defined as being at co-ordinates 15,10.

If either of the co-ordinates are greater than the maximum number of columns or rows then a "wrap around" will occur. Thus in PRINT @ 50,38 the cursor will move to 10,14.

EXAMPLE: PRINT @ 20,15;"START POSITION"

The cursor will move to the character position indicated by the co-ordinates 20,15 and the expression will then be printed from that point as shown below.



PRINT@20,15

Related Keywords: FMT IOM PRINT PRINT# SPC TAB WIDTH ZONE

PRINT#

PRINT#

Syntax: PRINT# J

J is a "device number " previously assigned to a device and in the range 0 to 254, although only devices 0, 1 and 2 are defined in Tatung/Xtal BASIC 4.

Purpose: This statement assigns a new output device (eg. Printer etc.) indicated by the value of J.

All output from statements such as PRINT and LIST will be directed to the new device selected by N until another PRINT# statement is encountered to change the device selection, the program ends or the program is aborted either by an error, or from the keyboard.

In DIRECT mode each line acts like a small program therefore if PRINT# is used, the corresponding output statement must appear in the same line. After execution in direct mode the keyboard and display (input device 0 and output device 0) are automatically selected. The following three devices are currently assigned within Tatung/Xtal BASIC 4.

DEVICE		DEVICE NUMBER
VDU (SCREEN)	-	0
PRINTER	-	1
SERIAL PORT (RS232)	-	2

The user may assign other devices as described on Page 113 of this manual.

When a program ends or aborts, either through an error or as directed from the keyboard, the output device reverts back to the display unit screen (i.e.device 0).

EXAMPLE: PRINT # 1

All output following this statement in a program will be directed to the Printer.

PRINT # can be used in the same manner as a normal PRINT statement but the "device number" must be followed by a semi-colon (;) so as to distinguish the remainder of the statement.

EXAMPLE: PRINT#1; "ANSWER THE FOLLOWING QUESTIONS"

EXAMPLE:
```
10 PRINT#1
20 LIST
```

This will list a whole program to the printer device. When the listing of the program is complete "ready" is printed on the display.

EXAMPLE: PRINT#2

All output following this statement in a program will be directed to the serial port (RS232).

EXAMPLE: PRINT#1:LIST

When used in direct mode also lists to the printer. When listing is complete an internal PRINT#0 is performed, switching the output back to the screen.

USE WITH FILES:

Syntax: PRINT #SV,I; string expressions

Purpose: This is a File-Handling command which outputs the string expressions to the file specified by the file descriptor SV, from the start of record number I.

For the application of this command refer to the section on FILE-HANDLING, Page 264 of this manual.

Related Keywords: LIST LISTP PRINT

PSG

PSG (Programmable Sound Generator)

Syntax: PSG R,J

Purpose: This is a Sound Generator command which allows direct access to the sound generator "registers".

R is the register number and has a value from 0 to 15.

J is the register value in the range 0 to 255.

This command can be used to create particular "sound effects" as described in a later section which is devoted to the details of the Programmable Sound Generator (Page 288).

EXAMPLE:

```
PSG 12,120
```

This will store 120 in register 12 of the sound generator.

PSG can be used as a function to obtain the current value of a specified register.

EXAMPLE:

```
X = PSG(8)
```

This will return the value of the channel A amplitude register in X.

Related Keywords: MUSIC TEMPO VOICE

PSW

PSW (Password)

Syntax: PSW <password>

Where password is an 8 character name enclosed in quotes, selected by the user and may contain any characters other than control characters.

Purpose: This command sets up the password protection facility which can be used for security purposes to limit the access to any given file to authorised personnel only.

Once a password has been invoked any files saved can then only be loaded back under the same password. Any files which exist on the disc either without a password, or under a different password, cannot be loaded whilst the current password is in operation.

To change the password use PSW again with a different password. To turn off the password (or make sure that no password is in force!) use PSW by itself (i.e. PSW and no password).

NOTES:

1. An unprotected file must (i.e. a file saved with no password invoked) be read back without a password being in force.

2. The password itself is not stored anywhere, on the disc, therefore the user must know it or record it elsewhere.

3. There is no indication given in the directory that a file has been protected; the file can apparently be read, but appears to be complete rubbish.

4. The directory itself is unaffected by the password so that it is perfectly acceptable to mix unprotected files and files saved under various passwords stored on the same drive (as long as you know which are which!).

Example:

```
PSW"IXZ247Y5"
SAVE"MAIL.XBS"
```

Having saved the file MAIL.XBS under the password IXZ347Y5 it can only be read or loaded back if that password is in force. Likewise other files not saved under this password cannot be read or loaded while it is in force.

Related Keywords:

PTR

PTR (Pointer)

Syntax: PTR J,I

J is a number in the range 0 to 24.

Purpose: This allows the user to set selected scratch pad locations without using PEEK or POKE, but using the number J to select the locations, and I to be the new value. J is in the range 0 to 24.

Location numbers, J, are selected from the list given below.

0	HTEXT	Default or 'hard' pointer to start of BASIC program
1	TEXT	Pointer to start of BASIC program (modified by HOLD)
2	SCMD	Pointer to standard reserved word table.
3	AUXCMD	Pointer to auxiliary (user) reserved word table.
4	ERRTAB	Pointer to normal error message table.
5	AUXERR	Pointer to auxiliary error message table.
6	SADR	Pointer to standard address table.
7	SFNADR	Pointer to standard function table.
8	AUXADR	Pointer to auxiliary address table.
9	USRLOC	Pointer to user machine-code routine (CALL as function).
10	DEVPTR	Pointer to list of available I/O devices.
11	DEFLST	No of lines to 'LIST' at a time.
12	BUFPTR	Pointer to start of input buffer.
13	BUFLEN	Length of input buffer.

14	TXTTOP	Pointer to end of BASIC program.
15	VARTOP	Pointer to end of simple variable space.
16	ARRTOP	Pointer to end of array space.
17	STRBOT	Pointer to bottom of string space.
18	STKBOT	Pointer to bottom of stack area.
19	IVDU	Pointer to bottom of 'internal VDU' area.
20	LIMIT	Pointer to top of RAM used by Tatung/Xtal BASIC.
21	TOPRAM	Pointer to top byte of RAM available to user.
22	LNNO	Current line number being executed.
23	DATLN	Line number of current DATA statement (undefined before a READ statement has been done).
24	DATPTR	Pointer to current position in DATA statement (If using READ statements). Can be moved to specified line by RESTORE L statement

Any value for J outside the range listed above will give a RANGE ERROR.

The current value of any PTR location may be accessed by using PTR as a function with the argument representing the required location as follows.

EXAMPLES:

```
A= PTR (12)
```

This puts the start address of the current input buffer area into the variable A.

```
PRINT HEX$(PTR(12))
```

This will display the current address value for the input buffer area (in hexadecimal format).

NOTE: Caution should be adopted when using this command as there are no facilites for checking that alterations are not affecting other locations which might already be in those areas of memory. For example, the CLEAR command (Page 48) should be used to set up the LIMIT and STKBOT locations, NOT PTR 20,E and PTR18,E.

Related Keywords: DEEK DOKE PEEK POKE

RAD

RAD (Radians)

Syntax: RAD (N)

Purpose: This is a Function which converts the angle given by N (in degrees) to radians.

EXAMPLES:

```
X = RAD (30)      - Returns a value of .523599 radians
                    in X.

PRINT RAD(30)     - This will display the value .523599
                    on the screen.

X = SIN(RAD(30))  - Returns the sine of 30° in X
                    (i.e. 0.5)
```

Related Keywords: ATN COS DEG SIN TAN

READ

READ

Syntax: READ V1,V2,..,Vn

V1,V2,etc. are variables which are linked up to corresponding values in the same order as listed in a DATA statement (see Page 55).

Purpose: This statement is used to access data, stored in DATA statements, from within a program as opposed to input from the keyboard. BASIC positions a "pointer" at the last item of data read so that **subsequent** READ statements will continue from that point.

If there is insufficient data available for the READ statement a DATA ERROR will occur.

EXAMPLE:

```
10 READ A,B,C
20 - - - - - -
30 - - - - - -   Processing lines.
40 - - - - - -
50 DATA 9,20,30
```

This will READ a value of 9 for A, 20 for B, and 30 for C.

EXAMPLE:

```
10 READ A,B,C
20 - - - - - -
30 - - - - - -
40 READ D,E,F
50 - - - - - -
60 - - - - - -
70 DATA 30,6,19,20,64,71
```


The first READ statement in line 0 will read a value of 30 for A, 6 for B, and 19 for C. The pointer is then positioned at 19 in the DATA statement so that the second READ statement, in line 40, will start from that point.

```
30,6,19,20,64,71
      ↑
   Pointer
```

Thus the second READ statement will read a value of 20, for D, 64 for E, and 71 for F.

Related Keywords: DATA RESTORE

REM

REM (Remark)

Syntax: REM

Purpose: This causes the remainder of the line to be ignored by the Interpreter (i.e. it is not processed).

It is used for entering notes anywhere in a program to clarify the purpose of particular sections and their functions.

EXAMPLE:

```
10 REM PROGRAM TEST FOR COLOURS
```

This line is not processed and is merely a comment line as an aid to the programmer.

NOTE: No further BASIC statements can be entered on the same line after a REM statement.

```
eg. 100 REM PROGRAM ENDING:STOP
```

Here the stop will not be executed.

Related Keywords:

REN

REN (Rename)

Syntax: **REN** <old file> **TO** <new file>

Purpose: This is a Disc Command which **renames** the file given by <old file> to the name given by <new file> for the disc in the current default drive. (note the order of appearance in the statement and see Page 264 for file name conventions).

If <new file> is already in use on the disc a FILE EXISTS error will occur.

If <old file> does not exist a NO FILE error will occur.

If <old file> is a locked file a FILE LOCKED error will occur.

The **default drive** may be changed or re-selected by use of the DRIVE command (see Page 69).

EXAMPLE:

```
REN "ROUTINES" TO "PROCESS"
```

This will change the name ROUTINES to PROCESS for that particular BASIC file.

```
REN "PARTY.ASC" TO "GROUP.ASC"
```

This will rename the .ASC file PARTY to become GROUP.ASC.

Related Keywords: DRIVE DIR

RENUM

RENUM (Renumber)

Syntax: RENUM L1,L2

L1 is the new starting line and L2 is the increment to be used. If omitted, both L1 and L2 will default to 10.

Purpose: This is a System Command used to renumber a whole program or a "held" part of a program.

All line number references following GOTO, GOSUB, RUN, THEN, ELSE, AND RESTORE commands are modified accordingly during the renumbering process.

EXAMPLES:

RENUM 1000,5	Will renumber, making the first line 1000 and then increment by 5 (1000,1005,1010,1015 etc).
RENUM	Will renumber making the first line 10 and then increment by 10 (10,20,30,40 etc).
RENUM 500	Will make the first line 500, and then increment by 10 (500,510,520,530 etc.).
RENUM ,20	Will make the first line 10 and then increment by 20 (10,30,50,70,etc.).

Related Keywords: LIST

RESTORE

RESTORE

Syntax: RESTORE L

Where L is given as a line number.

Purpose: This statement positions (restores) the internal pointer, used by BASIC in DATA statements, to the beginning of the first DATA statement following the line number L, regardless of where the pointer had been left by previous READ statements.

This facility allows DATA statements to be re-read several times within the same program thus avoiding having to store the "data items" in variables throughout the whole execution of a program.

L (line number) is optional and if omitted, the pointer is restored to the beginning of the very first DATA statement in the program.

EXAMPLE:

```
 10 READ A,B,C
 20 - - - - - -
 30 - - - - - -
 40 DATA 10,20,40,70,90,110,15,17,150
 50 READ X,Y,Z
 60 - - - - - -
 70 - - - - - -
 80 RESTORE 30
 90 READ D,E,F
100 - - - - - -
110 - - - - - -
```

The sequence of operations would be as follows:-

i) The READ statement in line 10 will give A a value of 10, B a value of 20, and C a value of 40.

ii) The internal pointer is then positioned at 40 in the DATA statement.

10,20,40,70,90,110,15,17,150
↑
Pointer

iii) The READ statement in line 40 will continue from the pointer and give X a value of 70, Y a value of 90, and Z a value of 110.

iv) The internal pointer is then repositoned to 110.

10,20,40,70,90 110,15,17,150
↑
Pointer

v) The RESTORE statement in line 60 causes the pointer to move to the beginning of the DATA statement in line 30.

10,20,40,70,90,110,15,17,150
↑
Pointer

vi) The read statement in line 70 therefore gives D a value of 10, E a value of 20, F a value of 40.

vii) The pointer is then positioned once again at 40.

10,20,40,70,90,110,15,17,150.
↑
Pointer

Related Keywords: READ DATA

RETURN

RETURN

Syntax: RETURN

The last line of any subroutine should always be RETURN.

Purpose: This terminates a subroutine accessed by a GOSUB statement.

Execution is transferred back to the line immediately following the original GOSUB statement.

If a RETURN is encountered without having been preceded by a GOSUB then a RETURN ERROR will occur.

Related Keywords: GOSUB

RIGHT$

RIGHT$

Syntax: RIGHT$ (<string expression>**,J)**

Purpose: This is a String Function which will return the rightmost number of characters, given by the value of J, of the string specified in the function.

EXAMPLE:

```
PRINT RIGHT$("HELLO",2)
```

This will display the following result on the screen:-

```
LO
```

Related Keywords: LEN MID$ LEFT$

RND

RND (Random)

Syntax: RND (I)

Purpose: This is a Function which returns a random number.

When I=1 (i.e. RND(1)) the function returns a random number in the range 0 to 1, as a floating point number.

When I is in the range 2 to 65535 the function returns an integer random number ranging from 0 to (I-1).

When I=0 (i.e.RND (0)) the functions returns the last random number produced, whether integer or real.

EXAMPLE:

```
RND (9) - returns a number in the range 0 to 8
RND (307) - returns a number in the range 0 to 306.
```

EXAMPLE:

```
PRINT RND(11)
```

The random number selected from the range 0 to 10 will be displayed on the screen.

Related Keywords:

RUN

RUN

Syntax: RUN L

Where L is a line number

Purpose: RUN is used to begin the execution of a program currently in memory, starting at the line number given by L, and clearing all variables.

If L is omitted execution will begin from the lowest line number of the program.

EXAMPLE:

```
RUN 45
```

This will cause a program to begin execution at line 45.

Syntax: RUN file

Purpose: In this alternative form RUN will load the program file declared in file and then commence its execution from the lowest line number.

EXAMPLE:

```
RUN "TESTPROG"
```

This will load the program whose name is TESTPROG.XBS from disc and then execute it.

Related Keywords: CHAIN LOAD

SAVE

SAVE

SAVING BASIC PROGRAM FILES

Syntax: SAVE <file>

(See FILE NAMING CONVENTIONS on Page 264)

Purpose: This Command saves files/programs, currently in the computer's memory, onto disc.

If the drive is omitted from within the **<file>** the current "default drive" will be assumed. The default drive is initially set up as 0 but can be changed using the DRIVE command (see Page 69).

If the file type is not declared within **<file>** then XBS will be assumed.

EXAMPLE: SAVE "1:PROG.XBS"

This will save the BASIC Program file currently in memory onto the disc in drive 1, giving it the name PROG. (assuming drive 1 is fitted to EINSTEIN).

EXAMPLE: SAVE"PROG"

This will save the program onto the disc in the current default drive, as a XBS type file.

SAVING ASCII PROGRAM FILES

Syntax: SAVE <file>,L1,L2,L3

Purpose: In this form this command saves all or part of the program memory as an ASCII file.

The L1,L2,L3 format is as for LIST (Page 132), L1 and L3 being the start and end of the section required in ASCII form. As with list L1,L2 and L3 may be omitted.

In this instance L2 has no effect.

The file type part of **<file>** must be ASC.

EXAMPLE: SAVE"0:PROG.ASC"

This will save the whole of the program currently in memory onto the disc in drive 0, as an ASCII file PROG.ASC.

EXAMPLE: SAVE"1:PROG.ASC",90,10,150.

This will save the section of the program currently in memory from line 90 to line 150 onto the disc in drive 1, as an ASCII file PROG.ASC.

SAVING OBJECT PROGRAM FILES (MACHINE-CODE FILES)

Syntax: SAVE <file>,I1,I2

Purpose: This command saves the area of memory from address I1 to address I2 inclusive as an object file. I2 must be greater than I1. Both I1 and I2 must be declared. The file type within **<file>** must be .OBJ.

EXAMPLE: SAVE"1:TEST.OBJ",&8A3D,&9000

This will save the area of memory from location &8A3D to location &9000 onto the disc in drive 1 as the object file TEST.OBJ.

Related Keywords: DRIVE LOAD

SCRN

SCRN$ (Screen String)

Syntax: SCRN$(J)

J must be a value in the range 0 to 23 (i.e. number of rows available on the screen).

Purpose: This is a String Function which will return the full string of characters (40) from a row on the screen, indicated by the value of J.

EXAMPLES:

X$ = SCRN$(11)

This will return the string of characters contained in row 11 of the display screen in X$.

PRINT SCRN$(11)

This will output the contents of screen row 11.

Related Keywords: LEFT$ MID$ RIGHT$

SEP

SEP (Separator)

Syntax: SEP J

Where J is given as an ASCII value.

Purpose: This command is used to re-define the separator character used in DATA and INPUT statements, J being the ASCII value of the required character.

Under normal operation the separator is a comma (,) but this can be changed by use of the SEP command.

One common application is to use SEP 0. This is used when only one item is required which is to include commas as part of the input data. It allows the user to put any string of characters (including the comma) into an INPUT or DATA statement as a single item.

On conclusion of the processing involving the redefined separator character, normal operation can be restored by use of the SEP 44 command (44 being the ASCII value for the comma).

SEP can be used as a function to return the value of the current separator.

NOTE:

1) Some characters will not work well as separators with numeric data, the full stop (.) for example. Obvious confusion could occur here with decimal points.

2) Remember that SEP will affect DATA statements as well as INPUT statments.

EXAMPLES: SEP 43

This would change the separator to a + symbol as given by the ASCII value 43.

```
PRINT SEP
```

This will output the ASCII value of the current separator on the screen.

```
A = SEP
```

Thus A will contain the ASCII value of the current separator character.

The following example illustrates another use of SEP.

```
10 SEP 47:REM '/' IS SEPARATOR
20 INPUT "TYPE IN THE DATE AS DD/MM/YY:";.DAY,MNTH,YEAR
30 PRINT "DAY IS  ",DAY,"MONTH IS  ";MNTH;"YEAR IS  ";
   YEAR
40 END
```

This program will display the following.

```
TYPE IN THE DATE AS DD/MM/YY:
```

The date is then typed as - 14/12/84

As a result of the SEP command the slash symbols (/) are accepted in place of the comma separators given in the corresponding part of the INPUT statement. Program execution then continues and displays the following.

```
DAY IS  14 MONTH IS  12 YEAR IS  84
```

Related Keywords: INPUT DATA

SGN

SGN (Sign)

Syntax: SGN (N)

Where N can be given either as a number or a numeric expression

Purpose: This is a Function which returns the sign of N (i.e. indicates whether N is a positive or negative number).

The following values are returned according to the given conditions.

N less than 0 a value of -1 is returned.
N equal to 0 a value of 0 is returned
N greater than 0 a value of +1 is returned.

EXAMPLES:

PRINT SGN (-5.721) - a value of -1 appears on screen
(i.e. a negative number).

PRINT SGN (7.6219) - a value of 1 appears on screen
(i.e. a positive number).

PRINT SGN (0) - a value of 0 appears on screen
(i.e. a zero value).

Related Keywords: ABS INT

SHAPE

SHAPE

Syntax: SHAPE N,<string expression>

N is the ASCII code nominated for a character.

<string expression> is the required data and must consist of 2 digit hexadecimal numbers contained within quote marks ("").

Purpose: This Command allows the user to define a character shape or re-define an existing character.

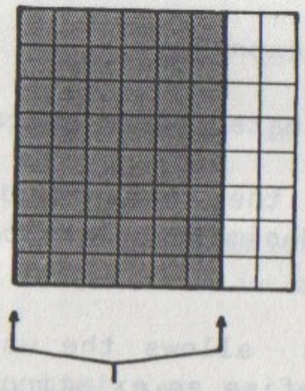
ASCII codes 32 to 127 and 161 to 255 are used for the keyboard text and graphic characters. Unless there is a need to redefine any of the characters avoid using the codes in these ranges.

The following character codes do not have any shape programmed at power-up.

0 to 31, 130 to 134, 142 to 154, 156 to 160

A shape consists of 8 bytes, each byte representing one row of the character cell. The **most** significant bit of each byte is the leftmost pixel of each row.

If the shape programmed is to be displayed in 40 column display, only the 6 most significant bits will be displayed on the screen. When in 32 column display or when defining a SPRITE shape all 8 bits are displayed.

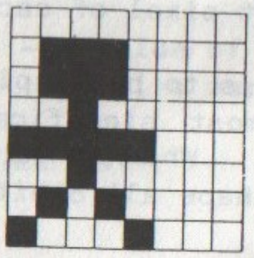


6 Most Significant BITS of each BYTE

displayed in 40 column display

Two or more shapes can be defined from within a single command by adding 8 bytes for each new shape. Each block of eight bytes will be asigned to the next ASCII character code in sequence.

EXAMPLE: The following "little man" shape is defined within one character cell.



The HEX values for each byte are placed into the shape command (in order from the top downwards), and an ASCII code is selected (eg. 130). The command would then appear as follows:-

```
SHAPE 130, "00 70 70 20 F8 20 50 88"
```

The shape can be called onto the screen as follows:-

```
PRINT CHR$(130)
```

The shape can also be used with the SPRITE command (Page 216.

If ASCII code 160 is used then the shape can be accessed from the keyboard by pressing the GRAPH key and the SPACE bar simultaneously.

EXAMPLE:

```
SHAPE 160,"00 70 70 20 F8 20 50 88 00 88 50 20 F8 20 70
70"
```

The first 8 bytes define character code 160
The second 8 bytes define character code 161

Related Keywords: SPRITE MAG

SIN

SIN (Sine)

Syntax: SIN (N)

Where N is an angle given in radians.

Purpose: This is a Function which returns the SINE of N.

EXAMPLES:

A = SIN (0.523599) - For an angle of 30°

This returns a value of 0.5 in A

The values are returned for use in expressions but can be output by using the PRINT command.

PRINT SIN (1.0472) - For an angle of 60°

The value 0.866027 will appear on the screen.

Related Keywords: ATN COS DEG TAN

SIZE

SIZE

Syntax: SIZE

Purpose: This is a Function which returns the size of memory available for the PROGRAM, VARIABLES, POINTERS, and STRINGS, as a positive value in the range 0 to maximum size of the system.

NOTE: Also used to clear space prior to a HOLD and MERGE.

EXAMPLE: X = SIZE

This will return a value in X indicating the size of memory available.

Related Keywords:

SPC

SPC (Space)

Syntax: SPC (J)

Purpose: This is a Function which specifies a number of spaces, as given by J, which are to be printed.

This function is only valid within a PRINT statement.

EXAMPLE:

```
PRINT SPC(10)
```

This will cause 10 spaces to be printed.

Related Keywords: PRINT TAB

SPEED

SPEED

Syntax: SPEED J

J can be any integer value from 0 to 255.

Purpose: This is a special command relating to output. It sets a delay on character output, to the current output device, according to the value of J.

0 gives the longest delay i.e. slowest speed. 255 is the fastest speed. (BASIC defaults to 255)

SPEED can also be used as a function to return the current set speed and the value given can be stored as a variable.

EXAMPLES: SPEED 100

All output following this statement in a program will be slower than normal (normal being 255).

```
PRINT SPEED
```

This will display the current value (as a number from 0-255) of speed set, onto the screen.

```
SPEED = 200
   A  =  SPEED
```

Thus A will contain 200.

Related Keywords: NULL

SPC

SPC (Space)

Syntax: SPC (J)

Purpose: This is a Function which specifies a number of spaces, as given by J, which are to be printed.

This function is only valid within a PRINT statement.

EXAMPLE:

```
PRINT SPC(10)
```

This will cause 10 spaces to be printed.

Related Keywords: PRINT TAB

SPEED

SPEED

Syntax: SPEED J

J can be any integer value from 0 to 255.

Purpose: This is a special command relating to output. It sets a delay on character output, to the current output device, according to the value of J.

0 gives the longest delay i.e. slowest speed. 255 is the fastest speed. (BASIC defaults to 255)

SPEED can also be used as a function to return the current set speed and the value given can be stored as a variable.

EXAMPLES: SPEED 100

All output following this statement in a program will be slower than normal (normal being 255).

```
PRINT SPEED
```

This will display the current value (as a number from 0-255) of speed set, onto the screen.

```
SPEED = 200
   A  =  SPEED
```

Thus A will contain 200.

Related Keywords: NULL

SPRITE

SPRITE

Syntax: SPRITE S,x,y,C,N

Purpose: This is a Graphics Command which sets up a particular SPRITE.

S is the "sprite number" and can be in the range 0 to 31. This allocates each sprite a priority (0 being the highest priority), this determines which shape is masked when two sprites have the same screen position.

x and y are the co-ordinates which position the top left hand corner of the sprite on the screen. The range of values for x and y is -32768 to +32767.

C is a value in the range 0 to 15 indicating the foreground colour shown in the code table below (C is optional and if omitted it will default to the last previously specified value of foreground colour). The background colour is always zero, i.e. transparent.

0 Transparent	8 Medium Red
1 Black	9 Light Red
2 Medium Green	10 Dark Yellow
3 Light Green	11 Light Yellow
4 Dark Blue	12 Dark Green
5 Light Blue	13 Magenta
6 Dark Red	14 Grey
7 Cyan	15 White

N is the ASCII code number of a previously defined shape which is to be used as a sprite.

Related Keywords: SHAPE MAG SPRITE OFF

SPRITE OFF

SPRITE OFF

Syntax: SPRITE OFF S

Where S is given as a sprite number in the range 0 to 31.

Purpose: This is a graphics Command which will "turn off" the sprite given by S (sprite number)

If S is omitted then all sprites will be "turned off".

Related Keywords: SPRITE SHAPE MAG

SQR

SQR (Square Root)

Syntax: SQR (N)

Purpose: This is a Function which returns the SQUARE ROOT value of N for use in expressions.

If N is given as being less than 0 a QTY ERROR will occur.

EXAMPLES: X = SQR (22)

This returns a value of 4.69042 in X

PRINT SQR (25)

The result 5 will appear on the screen.

Related Keywords:

STEP

STEP

Syntax: FOR V = N1 TO N2 STEP N3

Purpose: This command is used in the FOR-TO statement to specify a particular increment within the statement. Refer to Page 88 for further information.

EXAMPLE:

```
FOR I = 1 TO 10 STEP 2
```

Related Keywords: FOR TO NEXT

STOP

STOP

Syntax: STOP

Purpose: This is similar to END but is used to terminate programs at various points from which they may be restarted again.

The message BREAK IN L is displayed, where L is the line number at which execution has stopped.

Program execution can be restarted using the CONT command, provided that no alterations have been made to the program during the break (variables may, however, have their values altered).

This command is useful when debugging BASIC programs as it allows sections of the program to be executed and intermediate results inspected.

Related Keywords: CONT END

STR$

STR$ (String string)

Syntax: STR$ (N)

Where N can be given as a numeric variable or numeric expression.

Purpose: This is a String Function which returns a **string** representation of the value given by N.

The format in which the number is given by this function can be manipulated using the FMT command (see Page 84) and also the IOM 5 command (see Page 117).

Leading spaces are maintained by this function but NOT trailing spaces, in respect of the numerical format output as a string.

EXAMPLE:

```
A$ = STR$ (1.234)  -  gives the string " 1.234" in
A$
```

EXAMPLE:

```
PRINT STR$ (1.234)
```

The string " 1.234" will be displayed on the screen.

EXAMPLE:

```
FMT 2,3:A$ = STR$ (37.7325)
```

This places the string " 37.733" into A$ as a result of the combination of FMT and the STR$ function.

Thus A$ = "37.733".

Related Keywords: ASC LEFT$ LEN MID$
MUL$ RIGHT$ SCRN$ VAL

SWAP

SWAP

Syntax: SWAP V1,V2

V1 and V2 may be numeric or string variables, or array elements. They must be of a similar type in any one statement otherwise a TYPE ERROR will occur.

Purpose: This statement "swaps" the contents of the variables V1 and V2 with each other.

The command is very useful in "sorting algorithms"

EXAMPLE:

SWAP A,C - the contents of A become the contents of C and vice-versa.

SWAP D$,E$ - the contents of D$ become the contents of E$ and vice-versa.

SWAP A(I),B(I) - the array element of A(I) becomes the array element of B(I) and vice-versa.

Related Keywords:

TAB

TAB

Syntax: TAB (J1,J2)

Purpose: This Function is designed to be used within a PRINT statement only.

Characters will be printed to the output device until the cursor reaches the column given by the value of J1.If the print column is **past** or **at** the column given by J1, **no** TAB will occur.

J2 represents the ASCII value of the character which the function calls on to be printed. If J2 is omitted EITHER the character specified in the last previous TAB function will be used ,OR, if **no** previous TAB function has been specified, a space character will be printed.

Any valid ASCII character code may be used with this function.

Uses include formatting headings etc. for output.

EXAMPLE:
```
10 PRINT "NAME";TAB (20,46); "TATUNG"
20 PRINT "ADDRESS";TAB(20); "BRIDGNORTH"
```

This will produce the following printout:-

```
NAME.....................TATUNG
ADDRESS..................BRIDGNORTH
```

Note that the second TAB function defaults to the specified character of the previous TAB function because J2 is omitted.

Related Keywords: PRINT SPC

TAN

TAN (Tangent)

Syntax: TAN (N)

Where N is an angle given in radians.

Purpose: This is a Function which returns the TANGENT value of N.

EXAMPLES:

```
X = TAN (1.0472)  -  For an angle of 60°

This returns a value of 1.73206 in X

PRINT TAN (1.39626)  -  For an angle of 80°

The value 5.67117 will appear on the screen.
```

Related Keywords: RAD SIN

TCOL

TCOL (Text Colour)

Syntax: TCOL N1,N2

Where N1 and N2 are given as numbers in the range 0 to 15.

Purpose: This is a Display Command which selects the colour of the TEXT displayed on the screen according to the values of N1 and N2.

N1 represents the **Foreground** colour (i.e. the characters) and N2 the **Background** colour, for each individual character cell. They can be any value from 0 to 15, each number representing a particular colour as listed below.

If either N1 or N2 is omitted, it will default to the previous TCOL parameters.

0 Transparent	8 Medium Red
1 Black	9 Light red
2 Medium Green	10 Dark Yellow
3 Light Green	11 Light Yellow
4 Dark Blue	12 Dark Green
5 Light Blue	13 Magenta
6 Dark Red	14 Grey
7 Cyan	15 White

When BASIC is first loaded, N1 = 15 (White) and N2 = 4 (Dark Blue).

This command only directly affects individual character cells as they are printed on the screen and does not change the overall backdrop colour of the screen.

EXAMPLE:

```
TCOL 9,12
```

Any TEXT produced following this command will appear as Light Red (foreground) characters on a Dark Green background (in respect of individual cells).

NOTE: When a background colour other than 0 (transparent) is specified, a CLS will fill the display area with the new background colour.

Related Keywords: BCOL GCOL

TEMPO

TEMPO

Syntax: TEMPO N

Where N is given as a number in the range 0 to 7.

Purpose: This command sets the tempo (speed) of the music output according to the value given by N. (If TEMPO is not specified a value of 4, see table, is assumed).

Each value of N represents a tempo as listed below:-

0 - 50 beats per minute
1 - 100 beats per minute
2 - 150 beats per minute
3 - 200 beats per minute
4 - 250 beats per minute
5 - 300 beats per minute
6 - 350 beats per minute
7 - 400 beats per minute

EXAMPLE: TEMPO 3

This will set a tempo of 200 beats per minute for the music output.

Related Keywords: BEEP MUSIC PSG VOICE

THEN

THEN

Syntax: IF <condition> **THEN** <statement>

Purpose: This is used in the IF statement to direct the resulting sequence of operation.

For further information see Page 102

EXAMPLE:

```
IF x > 10 THEN 120
```

Related Keywords: IF ELSE GOTO GOSUB

TI$

TI$ (Time String)

Syntax: TI$="HHMMSS"

Purpose: This command is used to set the real-time clock contained within the system to a particular value given by HH as hours, MM as minutes, and SS as seconds.

On power up the clock is set to "00000" but once set by the TI$ command it will continue to keep the time until the machine is switched off or reset.

The time may only be set to an even number of seconds, although both odd and even numbered seconds are displayed.

The current time value can be returned using TI$ as a function.

EXAMPLES: TI$ = "174032"

This will set the clock to a time of 17 hours, 40 minutes, 32 seconds, i.e. 40 minutes, and 32 seconds past 5 o'clock in the afternoon.

```
PRINT TI$
```

This will display the current time setting in the following format:-

```
HHMMSS
```

Related Keywords:

TO

TO

Syntax: FOR V=N1 TO N2

Purpose: This is used within the FOR statement in order to specify the upper limit for a required loop.

Refer to Page 88 for further information.

EXAMPLE: FOR x = 3 TO 19

Related Keywords: FOR NEXT STEP

UNLOCK

UNLOCK

Syntax: UNLOCK <file>

(see Page 264 for file name conventions).

Purpose: This is a Disc Command which unlocks a previously locked file (given by **<file>**) on the disc in the current default drive.

Files unlocked using this command may then be written to, erased, or renamed as required.

If the file does not exist a NO FILE error occurs.

The **default drive** may be changed or re-selected by use of the DRIVE command (see Page 69).

Related Keywords: DRIVE DIR LOCK

UNPLOT

UNPLOT

Syntax: UNPLOT x,y

Where x and y are the co-ordinates of a point on the screen and can have values in the range of -32768 to +32767(the screen grid is 256 pixels horizontal by 192 pixels vertical).

Purpose: This is a Graphics Command which turns OFF a pixel which is illuminated i.e. the pixel at the co-ordinates specified is changed from foreground to background. If the pixel is already in background, then the pixel will remain unchanged.

EXAMPLE: UNPLOT 120,90

This will turn off the pixel at co-ordinates 120,90.

Related Keywords: DRAW ELLIPSE PLOT POINT POLY

VAL

VAL (Value)

Syntax: VAL (<string expression>)

Purpose: This is a String Function which returns the numerical value of the specified string up to the first **non-numeric** character.

("+","-",".", and "E" are regarded as numeric).

The "&" character is taken to indicate that a "hex number " will follow.

EXAMPLES: VAL ("1.234ABC")

This returns the value 1.234

PRINT VAL ("1.7993XYZ")

The result will appear on the screen as 1.7993.

VAL ("&" + "ABCD")

This gives a value of 43981 (i.e. decimal equivalent of &ABCD)

Related Keywords: ASC EVAL STR$

VDEEK

VDEEK (Video Deek)

Syntax: VDEEK (I)

Where I represents a memory location.

Purpose: This is a Machine Code related command which operates in a similar manner to DEEK (see Page 56) with the following differences.

1) The memory operations take place on the VIDEO RAM.

2) The video memory location given in I must be in the range 0 to 16383 (0 to &3FFF)

Related Keywords: DEEK DOKE PEEK POKE VDOKE VPEEK VPOKE

VDOKE

VDOKE (Video Doke)

Syntax: VDOKE I,I1, I2,..,In

Purpose: This is a Machine Code related command which operates in a similar manner to DOKE (see Page 65) with the following differences.

1) The memory operations take place on the VIDEO RAM.

2) The memory location (I) must have values in the range 0 to 16383 (0 to &3FFF).

Related Keywords: DEEK DOKE PEEK POKE VDEEK VPEEK VPOKE

VERIFY

VERIFY

Syntax: VERIFY <file>

Purpose: This is a System Command which checks a file on disc.

It is commonly used following execution of a SAVE command in order to check that a program has been transferred to disc correctly.

The command works in a similar manner to LOAD, except that a program file is not loaded into memory, but is compared with the memory contents.

Any error in the comparison is reported as BAD DATA ERROR and a NO FILE ERROR indicates that a program file is non-existent.

EXAMPLE: VERIFY "1:HICK.XBS"

This will check the disc in drive 1 for the BASIC program HICK and then verify the contents of the file with the memory contents.

Related Keywords: DRIVE SAVE LOAD

VOICE

VOICE

Syntax: VOICE N,N1,N2,N3,N4,N5

Purpose: This is a Sound Command which sets up a voice for use by the MUSIC statement.

N is the "voice number" and can have values in the range 0 to 7.

Each voice then has give parameters as follows:

N1 is the "noise period", in the range 0 to 31. 0 represents the highest frequency noise.

N2 is the "maximum amplitude" of notes, in the range 0 to 15, where 0 is the quietest and 15 the loudest sound.

N3,N4,N5 allow construction of the sound envelope where N3 represents the "attack", N4 represents the "sustain", and N5 the "decay" times of each note played in a given channel. These timings are independent of the "note length", so that it is possible to start a new note before the last one has completed, or to complete a note before its time has expired, thus allowing "legato" and "staccato" playing. The values for all three timings fall within the range 0 to 255 where 0 represents the shortest time.

Up to eight voices may be defined in this way, which can be invoked by means of the letter V within a MUSIC statement. e.g. V0 selects voice 0.

Related Keywords: BEEP MUSIC PSG TEMPO

VPEEK

VPEEK (Video Peek)

Syntax: VPEEK (I)

Purpose: This is a Machine Code related command which operates in a similar manner to PEEK (see Page 167) with the following differences.

1) The memory operations take place on the Video RAM.

2) The memory locations specified by I must be in the range 0 to 16383 (0 to &3FFF)

Related Keywords: DEEK DOKE PEEK POKE VDEEK VDOKE VPOKE

VPOKE

VPOKE (Video Poke)

Syntax: VPOKE I,J1,J2,..,Jn

Purpose: This is a Machine Code related command which operates in a similar manner to POKE (see Page 171) with the following differences.

1) The memory operations take place on the video RAM

2) The memory locations specified by I must be in the range 0 to 16383 (0 to &3FFF)

Related Keywords: DEEK DOKE PEEK POKE VDEEK VDOKE VPEEK

WAIT

WAIT

Syntax: WAIT J1,J2,J3

J1 is a port number and J2 and J3 are data items treated as 8 bit binary numbers.

Purpose: This command monitors directly the input and output from a particular I/O port.

The command suspends execution of a program whilst it monitors the port given by the value of J1 (eg. for 8 bit user port J1=&32). Execution will continue when a required condition, controlled by the selection of the data for J2 and J3, exists at the port.

The data which appears at the port is combined with the two items of data given in J2 and J3 in the following sequence.

1) The port data and J3 are combined using an EXCLUSIVE OR operation which is performed bit-by-bit on the two numbers.

   J3 is optional and may be omitted. If J3 is not used it is assumed to be 0. Any XOR comparison with 0 simply produces a result identical to the port data.

2) The result of the first operation is combined with J2 using an AND operation, again on a bit-by-bit basis. The result of this becomes the FINAL RESULT.

3) The two steps above are repeated until the FINAL RESULT is NON-ZERO. At this point program execution will then continue.

The comparisons are made using the two tables given below.

"XOR" (Exclusive OR) TABLE

BIT SETTING COMBINATIONS		RESULT
A	B	R
0	0	0
0	1	1
1	0	1
1	1	0

"AND" TABLE

BIT SETTING COMBINATIONS		RESULT
A	B	R
0	0	0
0	1	0
1	0	0
1	1	1

a) These tables give all the possible combinations of settings for two BITS.

b) Column A represents the setting (i.e. 0 or 1) of one BIT, and column B the setting of the second BIT.

c) For each combination of setting a result is given in column R

Thus if two BITS are set to 0 and 1 respectively, and we are doing an "AND" comparison, that particular combination is located in the "AND" table and the value of the result given in column R is read.

EXAMPLE: WAIT &32,&FF,&OF

i) Execution of the program is suspended whilst the 8 bit user port (&32) is monitored and the bit setting first combined with the &OF data. Let us assume that the BIT settings at the port are as follows:-

00000111

ii) This is now compared with the &OF data (J3) in accordance with the XOR table.

&OF data	-	00001111
Port data	-	00000111
Results from XOR table	-	00000000

iii) The RESULT from the XOR comparison is now combined with the &FF data in accordance with the AND table.

&FF data	- 11111111
Result from XOR comparison	- 00000000
Final RESULT of AND comparison	- 00000000

Thus a FINAL RESULT of 0 is obtained.

iv) Execution of the program remains suspended and the sequence is repeated until the BIT settings at the user port produced a "NON-ZERO" FINAL RESULT, at which point execution then continues.

In this example, for a "NON-ZERO" FINAL RESULT to be obtained, one of the following conditions must exist.

EITHER - any of the 4 **most** significant BITS of the user port must be "set" (i.e. 1)

OR - any of the 4 **least** significant BITS of the user port must be "reset" (i.e. 0)

Thus execution would be suspended until one of these conditions exist. The following example illustrates this.

EXAMPLE: WAIT &32,&FF,&OF

i) 8 bit port set to 01101111 (i.e. 2 of the 4 **most** significant bits are "set").

ii) XOR comparison

&OF	- 00001111
Port data	- 01101111
Result	- 01100000

iii) AND comparison

&FF	- 11111111
Result of XOR	- 01100000
Final result	- 01100000

Thus a "NON-ZERO FINAL RESULT is obtained and program execution will then continue.

EXAMPLE: WAIT &32,&40

i) Let us assume a 00001010 setting at the 8 bit user port.

ii) 'AND' comparison

```
&40         - 01000000
Port data - 00001010
Result      - 00000000
```

iii) A ZERO result is given therefore the sequence repeats until a NON-ZERO result is obtained.

EXAMPLE: WAIT &32,&40

i) 8 bit port setting - 00001010

ii) 'AND' comparison

```
&40         - 01000000
Port data - 01001100
Result      - 01000000
```

iii) A "NON-ZERO" result is given and therefore program execution will continue.

It can be seen that in this example execution is suspended until BIT 6 of the port is set (1) (i.e. the condition required to produce a NON-ZERO result)

Related Keywords: INP OUT

WIDTH

WIDTH

Syntax: WIDTH J

Purpose: This is a special command relating to output. It sets the width of the current output device, so that an automatic CARRIAGE RETURN/LINE FEED is generated as soon as the column count reaches the value given in J.

This is useful in certain printers such as Teletypes and Teleprinters where overprinting might occur when the print head reaches the end of a line.

Under normal operation the WIDTH is set to 0, when no automatic CARRIAGE RETURN/LINE FEED is produced.

WIDTH can also be used as a function to return the current width setting.

EXAMPLE: WIDTH 20

This would invoke a CARRIAGE RETURN/LINE FEED at columnn 20 of the output.

EXAMPLE: PRINT WIDTH

This would display the value of the current width setting on the screen.

Related Keywords: ZONE

XOR

XOR (Exclusive OR)

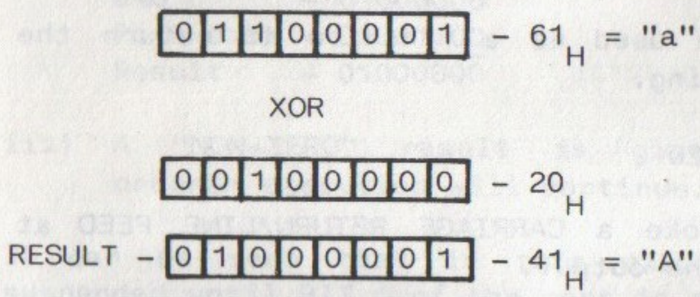
Syntax: I XOR I

Purpose: This is a LOGICAL OPERATOR used in the evaluation/comparison of statements.

EXAMPLE:

```
PRINT CHR$(&61 XOR &20)
```

This example shows on application of XOR which results in the lower case 'a' character (ASCII &61) being printed as upper case 'A'. The logical process is illustrated below:-



Related Keywords: OR AND NOT

ZONE

ZONE

Syntax: ZONE J1,J2

J1 gives the value of the largest column number and is known as the ZONE LIMIT. J2 gives the value of the ZONE width in number of columns.

Purpose: This is a special command relating to the format of output. It sets the ZONE width (see Page 178), and the largest column number for which printing to the next zone will stay on the same line.

The command is used to change the settings of the TAB functions contained in PRINT statements, according to particular requirements of output.

If omitted, J1 will default to 28 and J2 to 10.

ZONE can be used as a function to return the current values of J1 and J2.

EXAMPLES:

```
ZONE 32,16
```

This sets a zone width of 16 columns with column 32 indicating the ZONE LIMIT.

```
PRINT ZONE (0) - this displays the current ZONE
                 LIMIT(J1)

PRINT ZONE (1) - this displays the current ZONE
                 WIDTH(J2)
```

Related Keywords: WIDTH

11

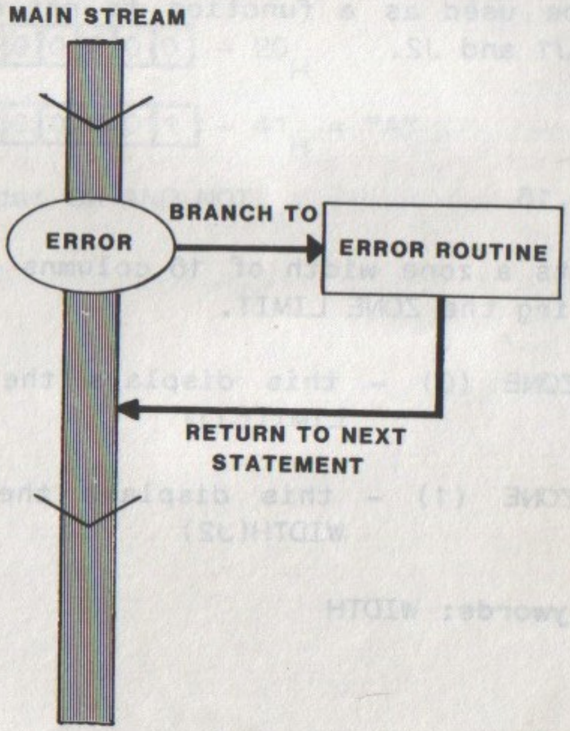
ERROR HANDLING WITHIN BASIC

The facility is available to handle error routines from within a BASIC program rather than abandoning of execution. The errors are simply allowed to occur and are then dealt with by subroutines.

The following commands/statements are involved:

ON ERR GOTO "Line No."
ON ERR GOSUB "Line No."

Either of these two commands may be contained within a program listing. If an error occurs AFTER one of these commands then a GOTO/GOSUB is made to the particular line number given where a routine may be stored which deals with a particular error as programmed by the user.



If ON ERROR GOSUB "Line NO." is used the last statement of an error handling routine should be a RETURN so as to send execution back into the main program. The execution re-enters the main program at the statement immediately following the one which caused the error.

When either of the two commands are used, an internal flag is set in order to activate the above procedure.

This flag reverts to "normal" after the first error and will therefore need setting again by another "ON ERR" statement positioned either at the end of the error routine, or soon after re-entering the main program.

OFF ERR
This command is used to restore the ON ERR flag to "normal" from within a program. However when a program "ends" normally, the flag reverts automatically. When OFF ERR has been used, any subsequent errors will be displayed as normal.

ERR,ERL,ERR$
These three statements are useful when included in the error handling routines.

ERR - returns the code number of the last error thereby indicating the nature of the error.

ERL - returns the line number at which the last error occurred.

ERR$ - returns the error STRING, without the word "ERROR", corresponding to the last error that occurred. This is useful when one particular error is expected and avoids having to flag every possible kind of error.

ON EOF GOTO "Line No."
ON EOF GOSUB "Line No."
This is a similar operation to ON ERR but relates specifically to routines which deal with encountering an "end-of-file" when "reading".

The difference is that the ON EOF flag is **not** reset by the execution of an ON EOF routine and therefore it remains in force.

OFF EOF
This is used to turn off the ON EOF mode. Any subsequent end-of-file encountered will then cause an END OF TEXT ERROR to be displayed. Again when the program "ends" in the normal way, the ON EOF is automatically turned off.

REMEMBER:-
ERRORS are only dealt with if they occur AFTER any of the statements. The programmer learns from experience where to situate the statements within a program in order to be of any advantage to the program execution.

12

ERROR MESSAGES WITHIN BASIC

When an error occurs, either in direct mode or from within a program, execution halts and a message will be output. Unless ERROR trap statements are used (as described in the previous section) the message will appear in the following format:-

For direct mode - "Description of Error" Error
For deferred mode - "Description of Error" Error in "Line No."

The "Description of Error" is a statement which indicates the type of error which has been made. The "Line No." in deferred mode is the program line number in which the error occurred.

EXAMPLES:
Direct Mode - Branch Error
Deferred Mode - Branch Error in 75

Details of the various error messages which might be output are given below.

Bad Data
A checksum error has been detected while loading or verifying a program/data file from disc. i.e. disc has been corrupted or memory contents do not match file (in the case of verify).

ACTION:- Retry with backup disc!

Branch
Reference has been made to a non-existent line number i.e. an attempt to use a line which is not included in a program.

ACTION:- Check program listing and make the necessary corrections in relation to the particular line/line number.

Cmd (Command)

An attempt has been made to reference a reserved word which does not exist. This often happens when programs are adapted from other systems.

ACTION:- Trace the offending word and convert or re-structure to comply with current system.

Cont. (Continue)

An attempt has been made to continue a program, (using the CONT command after a specified interrupt) when either:-

a) an error occurred or
b) alterations have been made within the program.

ACTION:- Check for errors which may have been introducted in any modifications and rectify as necessary.

Data

A READ statement has been used but insufficient data has been presented in the corresponding DATA statement.

ACTION:- Check the corresponding DATA statement and rectify as necessary.

Dimension

An attempt has been made to redimension an array. Arrays may only be dimensioned once within a program, including those arrays of under 10 elements which have not been formally dimensioned.

ACTION:- Check the appropriate statements, processes, and corresponding logic sequences. Rectify as necessary.

Division

An attempt has been made to divide a number by zero.

ACTION:- Check the appropriate expression and rectify as necessary.

Drive Select

A disc drive has been selected which is not available on the system.

ACTION:-

i) Check the appropriate select statement and rectify as necessary.
ii) Check that the particular drive is included in the system.

End of Text

Either:-

a) An end-of-file marker has been encountered in a data file or
b) The last block of a file has been read.

This error may be controlled within the system by use of the ON EOF command. (see Page 249)

File

Either:-

a) An attempt has been made to open a file which is already open or
b) An attempt has been made to read from, or write to, a file which is not open.

ACTION:- Check the logical sequence of the operations involved and rectify as necesaary.

File Type

A particular file type has been specified when in fact another type was expected.

ACTION:- Check the appropriate file types and rectify as necessary for them to correspond as required.

Fn Defn (Function Definition)

Either:-

a) A user-defined function has been used without first defining it or

b) CALL has been used as a function without first setting up the USRLOC.

ACTION:-

i) Check definition of appropriate user defined functions or

ii) Check that USRLOC has been correctly set up. Rectify as necessary.

Mem Full

An attempt has been made to use a command which would need more memory than is available.

ACTION:- Either re-structure or eliminate the command so as to comply with current memory space available.

Next

A NEXT has been used which does not correspond to a FOR statement.

ACTION:- Check the loop structure and either:-

a) eliminate the NEXT or

b) insert the required FOR statement

Operand

Operand is missing after an operator.

EXAMPLE: PRINT 6.2*7+

ACTION: Check the expression and rectify as necessary.

Ovfl (Overflow)

Numeric overflow from a calculation, i.e. number is outside the normal range for numeric variables.

ACTION:- Check the appropriate expressions and rectify as necessary.

Qty (Quantity)

A particular parameter in an array, command, or function, falls outside the declared range.

ACTION:-

i) Check parameter values with the individual commands and functions concerned to determine the maximum and minimum values allowed.

ii) Rectify values or re-structure as necessary.

Range

An attempt has been made to access an element of an array which does not fall within the limits of the delcared dimensions.

ACTION:- Check the dimensions of the array and rectify to accommodate the particular element as required.

Return

A RETURN has been used without a corresponding GOSUB.

ACTION:- Check the logical sequence of the processes involved and either:-

i) re-structure the sequence or
ii) insert a GOSUB.

depending on the particular requirements.

Stack Full

This will make reference to one or more of the following situations:-

a) FOR loops
b) GOSUBs
c) Parentheses in Expressions
d) FILL

The error message will be displayed if any of the above conditions have been NESTED too deeply, causing a "stack overflow".

ACTION:- Re-structure so as to reduce the nesting to an acceptable level and thereby rectify the situation. In the case of FILL check that the area in question is fully enclosed.

Str Ovfl (String Overflow)

A string has been included which exceeds the maximum number of characters allowed (255).

ACTION:- Check the offending string and either:-

i) re-structure the string, reducing the number of characters or
ii) transpose the single string into two or more separate strings with less then 255 characters per string.

Str Complex (String Complex)

A string expression has been used which is too long or complex.

ACTION:- Break the expression into smaller sections.

Syntax

This indicates that either a typing error has been made or a particular statement has been constructed incorrectly.

ACTIONS:- Check the appropriate sections of data and carry out the following:-

i) correct typing errors.
ii) correct statement construction errors.

Type

An incorrect data type has been used. i.e. a "numeric" quantity has been used when a "string" type was expected, or vice versa.

Dir Full (Directory Full)

This indicates the directory section of a disc is full.

ACTION:- If further information is to be placed in the directory then it must be at the expense of some of the existing data (either by deletion or overwriting).

Disc Full

This indicates that there is no more space available on a particular disc.

ACTION:- Further information can only be placed on the disc at the expense of existing data (either by deletion or overwriting).

Disc Locked

An attempt has been made to write to a disc which does not match the map of the disc held in the computer memory.

EXAMPLE: This error usually occurs when a dis is changed and an attempt is made to save a file on this disc.

ACTION:- Execute a DRIVE n before using a SAVE command, where n is the drive number containing the new disc.

Disc Seek

An attempt has been made to access a particular sector which is not on the disc. This quite often happens with "random-access" files when the record required is off the disc.

ACTION:- Check the appropriate data and rectify as necessary.

File Exists

An attempt has been made to use a name for a file which is already in existance (usually with the REN statement).

ACTION:- Check file names and rectify as necessary.

File Locked

An attempt has been made to erase, or write to, a file which has been "locked".

ACTION:- Check that the correct file has been referenced and rectify as necessary.

No File

A particular file cannot be found in a directory.

ACTION:-

i) Check that the file name has been constructed correctly and rectify as necessary.

ii) Check documentation to determine whether or not the file has been deleted previously.

Shape Defn (Shape Definition)

The number of (hexadecimal) characters entered in the shape definition string are not a multiple of 2. Two characters must be entered for each row of the shape being formed.

ACTION:- Edit SHAPE string by adding zero's or removing any extra characters to give 2 characters for each row of the shape.

Key Defn (Programmable Function Key Definition)

A function key string has been declared beyond its legal length.

ACTION:- Redefine the key concerned with a shorter string.

Write Protect

An attempt has been made to write to a disc which is "write protected" (on a "read only disc").

ACTION:- Check that the correct disc has been used and rectify as necessary, or de-activate the write protect tabs on the disc cassette.

NOTE: Error messages are also identified by a code number and this is often used in routines to reference individual error messages.

A summary of the list of error messages, indicating their allotted code numbers appears in appendix A.

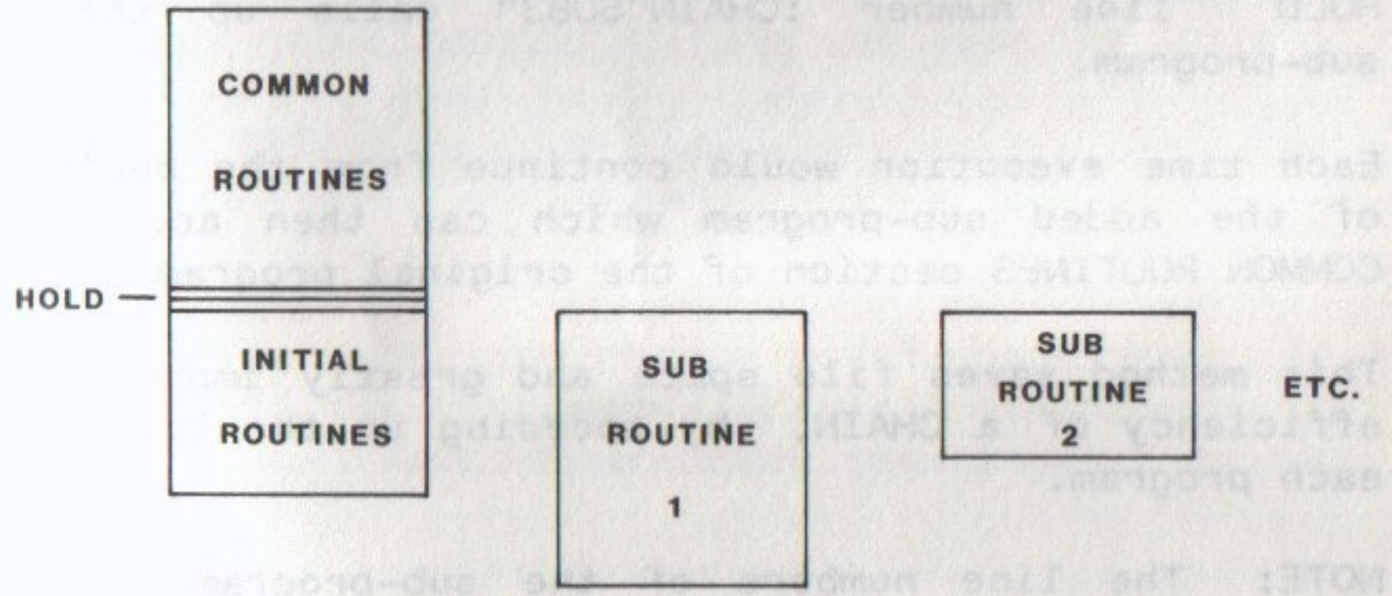
13

CHAINING AND SEMI-CHAINING PROGRAMS

In addition to the normal use of the RUN and CHAIN commands in direct or deferred mode, TATUNG/Xtal BASIC 4 provides the facility to SEMI-CHAIN programs.

This allows several programs to access a "common pool of routines" without having to keep the same set of routines within each **sub-program.**

To do this the HOLD command (Page 98) is used immediately prior to executing a RUN or CHAIN. Both RUN and CHAIN will restore a "held" program by re-setting the TEXT pointer as soon as the program is loaded. However, execution of the resulting program will commence from the start of the **added section** NOT from the beginning of the original program.



The diagram illustrates an original program consisting of a common routines section and an initial routines section. The initial routines would only be needed once to set up arrays, variables, memory space as in DIM and CLEAR statements, etc.

The HOLD/CHAIN combination separates the two sections such that the other sub-programs illustrated can be called up in place of the INITIAL ROUTINES section each time. (The flow of the original program would be structured such that the initial routines are executed before the line containing the HOLD/CHAIN commands is encountered).

Thus:-

HOLD line number :CHAIN"SUB1" calls up the 1st sub-program.
HOLD line number :CHAIN"SUB2" calls up the 2nd sub-program.
HOLD line number :CHAIN"SUB3" calls up the 3rd sub-program.

Each time execution would continue from the beginning of the added sub-program which can then access the COMMON ROUTINES section of the original program.

This method saves file space and greatly improves the efficiency of a CHAIN, by speeding up the loading of each program.

NOTE: The line numbers of the **sub-program** must be selected so as to be greater than those of the COMMON ROUTINES section.

The example Mailing List program given in the section on FILE HANDLING (Page 278) illustrates the use of this method of SEMI-CHAINING programs.

The **original program** consists of the COMMON ROUTINES section in lines 10 to 890 and the INITIAL ROUTINES section in lines 1000 to 1110.

The HOLD/CHAIN combination is found in line 900 and the flow of the **original program** is structured such that the INITIAL ROUTINES (lines 1000 to 1110) are executed before line 900 is encountered.

There are three **sub-programs** involved with titles "MSUB1", "MSUB2", and "MSUB3". These sub-programs are accessed according to the input in line 870 which places a value in N$ (given by the user response to the question "which?")

14

FILE HANDLING

FILE NAMING CONVENTIONS

The file naming conventions used in TATUNG/Xtal BASIC 4 requires the following 3 items to be specified when naming a file:-

a) an optional, one character DRIVE NAME.
b) a FILE NAME of up to 8 characters in length.
c) a FILE TYPE of up to 3 characters in length, known as the file extension.

Drive Name

The **drive name**, when used, is specified as a single number from 0 to 3. If not given then the default drive is assumed to be 0. The default drive may be changed at any time by use of the DRIVE command (Page 69).

File Name

This is the name assigned to a particular file by the user. It may consist of any combination of ASCII characters (i.e. from the character set), up to 8 characters in length, with the exception of the following:-

a) the characters .,"**<**;:=?*__**>**
b) characters with ASCII codes greater than 127.
c) characters with ASCII codes less than 32

Example: SILLY

File Type

This may consist of any combination of ASCII characters (i.e. from the character set), up to 3 characters in length, with the following exceptions:-

a) the characters .,"**>**;:?*=**<**
b) characters with ASCII codes greater than 127.
c) characters with ASCII codes less than 32

There are 4 file types recognised by TATUNG/Xtal BASIC 4 as follows:-

1) XBS - This is a BASIC **source** file (i.e. a normal program file). If the "file type" is not specified then XBS is assumed.

2) ASC - This is an ASCII program file. These files are uncompressed **source** files. (XBS files contain "tokens" for **reserved words** whereas ASC files contain the words in full as they appear in a LIST).

3) OBJ - This is an OBJECT file, or machine-code subroutine/data. A special area can be set up within the memory map (Page 314) for the storage of machine-code routines by use of the CLEAR command (Page 48). Anything stored in this area can be SAVEd (page 203) as a .OBJ file, and LOADed into the area.

4) DATA FILES - Any other combination of characters specified for "file type" will be treated as a DATA FILE. Data files consist of a series of ASCII characters divided into one or more records, which can be serially accessed by a BASIC program (in its broadest sense this catagory also encompases the three special file types XBS, OBJ, and ASC).

The user may select combinations of characters for data file extensions according to individual reqirements. The following are given as suggestions.

.BAK for backup copies
.DAT for data storage
.DOC for document files

EXAMPLES:

0:CATA.ASC - this refers to an ASCII file named CATA. for a disc in drive 0.

1:SILK.XBS - this refers to a BASIC source file named SILK, for a disc in drive 1

0:MEMO.OBJ - this refers to an OBJECT file named MEMO, for a disc in drive 0.

1:SILLYDAT - this refers to a DATA file named SILLYDAT, for a disc in drive 1.

NOTE: When working in BASIC, file names are contained within quote marks (") as shown in the example below:-

"1:CATEL.XBS"

MATCHING FILE NAMES

In some disc applications it is necessary to specify more than one file, for example when specifying the files for a DIRECTORY list. If one of the characters in a file name is replaced by a ?, then it will match any character in that position in the file name found.

EXAMPLE:

X?Z.ASC - Matches XYZ.ASC,XAZ.ASC,X9Z.ASC,etc.
?X.D?T - Matches AX.DAT,BX.D7T,4X.DZT,etc.

If an * is inserted in place of a character then it will match all the characters at and after that particular position in the file name or file type in which it appears.

EXAMPLE:

*.XBS - matches any XBS file.

* - also matches any XBS file (XBS being the default)

. - matches any file, and is the same as ????????.???

PROG*.ASC matches PROG1.ASC,PROG10.ASC,PROGABC.ASC,etc.

FILE DESCRIPTOR (FDESC)

This is a new concept in the approach to file handling.

Most BASICS have some method of assigning a storage area for use by a file during the time it is open. This area usually contains the following:-

a) a buffer
b) information describing the file
c) information relating to the location of the file on disc.

The problem that arises with this system is that the area has to be fixed and set aside **before** running the program. Thus this space may not be used for anything else, even after the file is closed. There is also a constraint upon the number of files which may be open at one time.

TATUNG/Xtal BASIC 4 overcomes the problems outlined above by adopting a different method as follows:-

A special "string variable" is assigned to a file when it is opened for access. This variable is then dropped when the file is closed.

The string variable is known as the FILE DESCRIPTOR for the file (usually abbreviated to FDESC).

The FILE DESCRIPTOR must be a simple string variable (but not part of a string array) and always contains the following:-

i) 168 characters (length of the descriptor)
ii) a 128 byte buffer
iii) 40 bytes of special file information

The layout for a file descriptor string is shown below:

FDESC:

DRIVE	1 byte	Disc drive name 01,02 etc.
FILNAM	8 bytes	File name.
FILTYP	3 bytes	File type.
INFO	21 Bytes	Internal allocation information.
RECORD	2 bytes	Record number, in the range 0 to 65535, + 1 overflow byte.
FILPTR	1 byte	Pointer to current byte in buffer for I/O.
RWFLAG	1 byte	Read/Write Flag 0=INPUT,1=OUTPUT.
RECLEN	2 Bytes	Random Record Length (Random access only).
FILBUF	128 bytes	128 byte file buffer.

NOTE: All parts of the descriptor string are accessible by normal string functions eg. the buffer contents could be inspected by doing a RIGHT$ of the last 128 bytes. (Page 200)

A FDESC may not be modified by LET statements etc. If this is attempted a FILE ERROR will occur the next time it is used in a PRINT# OR INPUT# statement.

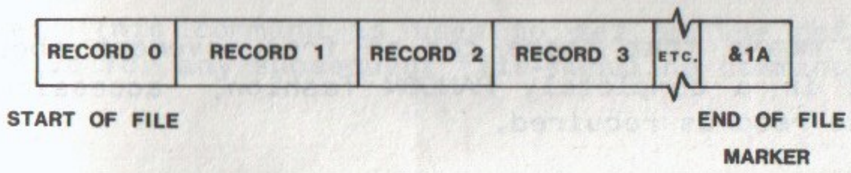
ACCESSING FILES

There are two commonly used methods for accessing files and these are known as:-

a) SEQUENTIAL ACCESS.
b) RANDOM ACCESS.

Sequential Access

Sequential Access is most often used for the manipulation of text or index files, where records may be of variable length, and need to be scanned (examined) sequentially i.e. one after the other, starting from the beginning of the file until the desired record location is found.



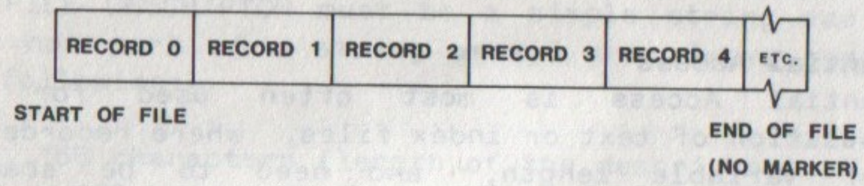
Each record should normally have a terminator, such as a carriage return code. There is a special code to mark the end-of-file (EOF).

Under TATUNG/Xtal BASIC 4 an EOF code is supplied when closing a sequential file, if the last operation was a WRITE, and will normally detect the end-of-file marker on a READ.

An end-of-file condition will occur if an attempt is made to read beyond the last allocated sector of the file.

Random Access

Records stored in random-access files are normally of a fixed length, **OR** of a variable length but contained within fixed-length blocks.



When a file is opened the "record length" is specified. Whenever a file input or output is required a "record number" is specified.

This means that there can be free movement about the file in a completely RANDOM fashion, accessing only those records required.

Having accessed a particular record it is also possible to then read or write sequentially to the file from that point onwards (even though a random record length has been specified).

It is also possible, if required, to write a file with one record length, and then read or write to the same file with a different record length.

An EOF marker is NOT supplied when closing a random-access file. However, the end-of-file condition will occur when an attempt is made to read a sector of the disc that does not exist.

Unfortunately this will not always be the case with a non-existent record since the disc space may have already been created for it as a side effect of writing another record which uses the same physical disc sector. In that case it will be read as an empty record.

FILE-HANDLING COMMANDS

The commands which provide the facilities within file-handling are listed below:-

DRIVE	OPEN	CREATE
CLOSE	APPEND	PRINT#
INPUT#	INCH#	INCH#(N)

DRIVE

Syntax: DRIVE N

Where N is a single number from 0 to 3.

Purpose: This command is used to set up the default disc drive for any subsequent file-handling commands.

EXAMPLE:

```
DRIVE 1              - Selects disc drive 1
OPEN "DATA.TXT",F$ - Opens the file DATA.TXT on drive 1
```

If the drive specified is not available on the user's system a DRIVE SELECT ERROR will be given.

NOTE: Some software may refer to drive 0 as A and drive 1 as B, 2 as C, and 3 as D. In such cases use the appropriate drive number.

OPEN

Syntax: OPEN <file>,SV,R

<file> must be a legal file name as described earlier.

SV is a string variable name (but **not** a string array element) and is the file descriptor.

R is the random record size (length) and is given as a value in the range 0 to 65535, indicating the number of characters involved. R is only specified for random access, it is omitted for sequential access (random record length 0 indicates that sequential access is to be performed).

Purpose: This command is used to open an existing file and assigns internal file information and buffer space to the file descriptor. It also indicates the record size to be used.

EXAMPLE: OPEN "0:SILLY.DAT",FD$15

This will perform the following:-

a) opens the file SILLY.DAT on the disc currently in drive 0.
b) assigns FD$ as the file descriptor.
c) sets up for random access using 15-character length records.

NOTE: If a file is not present on the specified drive then a NO FILE ERROR will be given.

CREATE

Syntax: CREATE <file>,SV,R

Purpose: This command is used to set up a new file.

The operation of this command is exactly the same as OPEN except that any existing file with the same name as specified in the command is first deleted, and a new empty file is opened.

EXAMPLE: CREATE "0:SILLY.DAT",FD$,15

This will perform the following:-

a) creates and opens the file SILLY.DAT on the disc currently in drive 0 (if any file of the same name already exists on the disc it will be deleted prior to the new empty file being created).
b) Assigns FD$ as the file descriptor.
c) sets up for random access using a 15-character length record size.

CLOSE

Syntax: CLOSE SV1,SV2,..,SVn

SV1,SV2,etc. must be string variable names (but **not** string array elements) and are the file descriptor .

Purpose: This command performs the following:-

a) writes the remaining contents of the appropriate buffers to their files.
b) stores directory information.
c) closes the files given by the file descriptors in SV1 to SVn.

A buffer will only be written out if the **last** operation performed on it was a WRITE. The file descriptors are then set to null strings which makes the space available for use by variables or other files.

If any of the file descriptors specified is not active (or is an ordinary string) then a FILE ERROR will be given. (File descriptors are internally marked so BASIC can distinguish them from normal strings).

If **no** file descriptors are specified then **all** files currently open will be closed. (No error is given if there are no files open).

NOTE: In addition to the above processing, CLOSE induces an automatic PRINT#0:INPUT#0. This will cause all output and input to go through the screen and keyboard and the CLOSE command can be used at any time when these **two** statements are required (it is shorter).

APPEND

Syntax: APPEND <file>,SV

Purpose: This command is used to write extra information at the end of a sequential file, when to OPEN the file and read to the end would be most inefficient.

The operation is similar to OPEN with the following differences:-

a) no record length is supplied.
b) the internal file pointer moves to the end of the file not the start of the file.

EXAMPLE: APPEND "0:SILLY.DAT",FD$

This will perform the following:-

a) opens the file SILLY.DAT on the disc currently in drive 0, and moves the pointer to the end of the file so data may be added.

b) assigns FD$ as the file descriptor.

NOTE: If the file specified by file does not exist on the disc then a NO FILE ERROR will be given.

PRINT#

Syntax: PRINT# SV,R;E

Purpose: This command is used to output the expression list given by E, to the file given by the file-descriptor SV, from the start of the record number given by R in the file.

The location relative to the start of the file is calculated as R multiplied by the record length given when the file was opened. (This only applies to random access and is not allowed in sequential access). For "sequential access", omit the (,R) but keep the (;) giving the following format.

PRINT# SV;E

Output will then start from the current place in that file since the BASIC keeps account of its place in a particular file even when several files at once may be open for output. (In fact the only purpose of specifying the record number R is to define the point within the file at which input or output is to begin, therefore it will be assumed that it "starts from where it left off" if no record number is given).

With disc files opened for sequential access, the internal file pointer can be set to the start of the file by specifiying a record number (any number will do, since it will be multiplied by the ZERO record length).

The expression list given by E is as for a normal PRINT statement and the data output will be EXACTLY as for that. Hence a carriage-return-line-feed is output at the end of the statement **unless terminated** by the **semi-colon.**

All subsequent statements supplying output, following this command, will now go to a file until another PRINT# or CLOSE statement is encountered.

PRINT#SV,E (no semi-colon) and PRINT#SV can be used to set up the specified file for output. All subsequent normal output statements will then direct data to the file (eg. PRINT,LIST etc.)

If PRINT statements are then terminated with a semi-colon (;) there will be no carriage-return-line-feeds, and a stream of data may be output to a file.

The automatic tab expansion (where CHR$(9) is expanded to spaces) may need suspending by use of the IOM7,0 command (see Page 117.)

This now facilitates the output of strings which contain machine-code.

INPUT#

Syntax: PRINT#SV,R;V

Purpose: This command takes input from the file given by the file-descriptor SV, starting at the first character of the record number given by R in the file.

The variable list given by V is as for the normal INPUT command (Page 111), and items are assigned to the variable names given in the same way.

If R is omitted the file will be read from the last point reached, or from the beginning if it has just been opened (same application as in PRINT#). The format of the command is then

INPUT#SV;

Following this command all subsequent statements relating to input will attempt to access the file specified by SV (i.e. INPUT, INCH, INCH$, INCH$(N)) until another INPUT# OR CLOSE is encountered.

As before, the E can be omitted giving the following format(s).

INPUT#SV,R
or
INPUT#SV

Both these versions will set up the file specified by SV for input, and all subsequent normal INPUT statements will access that particular file.

INCH$ AND INCH$(N)

Syntax: **INCH$**
INCH$(N)

Purpose: This command is used when files containing control characters are required to be input (eg. machine-code files).

Normal INPUT statements ignore most control characters and usually terminate on a carriage-return or null character whereas INCH$ does not.

INCH$(N) is even more effective since it creates a string of length N and is usually much faster than INCH$ on its own.

An EOF condition on INCH$ (N) causes truncation of the string to the length reached at the time when the EOF occurred. Therefore all information will still be passed into an expression **before** the "EOF error" is actually flagged. The **next** input from that file will then flag the EOF condition in the usual way (i.e. END OF TEXT ERROR or ON ERR/ON EOF routine).

FILE HANDLING EXAMPLE

The following examples are given to illustrate the facilities outlined in this section.

a) A text file display program.

This program allows the display of data or ASC files on the screen. It performs virtually the same function as the TYPE command in CP/M, and it works at approximately the same speed.


```
10 REM TEXT FILE DISPLAY PROGRAM
20 N=128: REM No. of characters read at a time.
30 INPUT "File to display?"; NAME$
35 IF NAME$="" THEN DIR: GOTO30
40 ON EOF GOTO80
50 OPEN NAME$,FD$
60 INPUT# FD$
70 PRINT INCH$(N);: GOTO 70
80 CLOSE FD$
90 END
```

Try replacing line 70 with the following, noting how much slower it is:

70 PRINT INCH$;: GOTO 70 or try smaller values of N in line 20.

b) A simple Mailing List (Sequential Access).

The program below is a simple mailing list program showing as it does the use of sequential access for reading and writing files. In this case, the data file is read into a large string array M$ at the start of the program, and rewritten to the file SMAIL.DAT at the end.

This means that access to particular customers is very quick, but at the expense of keeping the entire file in memory at once. Moreover, the maximum number of customers that the system can handle is limited by memory size, and the size of M$ as dimensioned in line 9000.

The information under each customer consists of his/her name, telephone no. and address, the address being stored in to lines, or fields. The array CUST$ holds these items temporarily when being accessed by one of the program options.

The options supported by the program are to add a customer to the list, to access a customer from the list for modification, and to list all customers to the screen or printer.

```
 5 DRIVE0
 10 REM *** SIMPLE MAILING LIST PROGRAM (SEQUENTIAL
    ACCESS) ***
 20 REM
 30 GOTO 9000
 98 REM
 99 REM ***COMMON ROUTINES***
198 REM
199 REM      ***OPEN DATA FILE***
200 PRINT:PRINT "Do you have a file to load
    (Y/N)?";:Y$=INCH$
210 PRINT Y$: IF Y$="N" THEN RETURN
220 CLS: PRINT@4,10;"Reading in data file..."
230 OPEN FILE$,FD$
240 INPUT# FD$; NCUST: REM Get No. of customers on
    file.
250 IF NCUST=0 THEN 290
260 FOR I=0 TO NCUST-1
270 FOR J=0 TO 3: INPUT M$(I,J)
280 NEXT J,I
290 CLOSE
295 RETURN
298 REM
299 REM ***HEADING DISPLAY***
300 CLS: PRINT@8,0;HEAD$
310 PRINT@3,2;"Number of customers on file: ";NCUST:
    PRINT
320 RETURN
398 REM
399 REM ***WRITE NEW DATA FILE***
400 PRINT: PRINT "Do you wish to save the file
    (Y/N)?";: Y$=INCH$
```



```
410 PRINT Y$: IF Y$="N" THEN RETURN
420 CLS: PRINT@4,10;"Writing New Data file..."
430 CREATE FILE$,FD$
440 PRINT# FD$; NCUST
450 IF NCUST=0 THEN 490
460 FOR I=0 TO NCUST-1
470 FOR J=0 TO 3: PRINT M$(I,J)
480 NEXT J,I
490 CLOSE
499 RETURN
798 REM
799 REM ***MENU DISPLAY***
800 CLS: PRINT@8,0;"SIMPLE MAIL LIST PROGRAM"
810 PRINT@4,3;"Options:"
820 PRINT@4,5;"0. Exit Program"
830 PRINT@4,7;"1. Enter Customers"
840 PRINT@4,9;"2. Modify Customers"
850 PRINT @4,11;"3.List Customers"
870 PRINT@4,13;"Which? ";: N$=INCH$(1): PRINT
880 N=VAL(N$): IF N<0 OR N>3 THEN PRINT BEL$: GOTO 870
890 IF N=0 THEN GOSUB 400: CLS:PRINT@8,0;"GOODBYE!";
    BEL$: END
898 REM
899 REM ***SELECT OPTIONS***
900 ON N GOTO 1000,2000,3000
910 STOP: REM SHOULD NEVER GET HERE!
998 REM
999 REM ***END OF COMMON ROUTINES***
1000 REM ***MSUB1 -- Enter Customers***
1010 HEAD$="ENTER CUSTOMERS"
1020 GOSUB 300
1030 PRINT"Any more customers to add (Y/N)?";:Y$=INCH$:
     PRINT Y$:  PRINT
1040 IF Y$<>"Y" THEN 800
1050 FOR I=0 TO 3
1060 PRINT PRMPT$(I);: INPUT CUST$(I)
1070 NEXT
```

```
1080 FOR I=0 TO 3: M$(NCUST,I)=CUST$(I):NEXT:NCUST=
     NCUST+1
1090 GOTO 1020
1999 REM
2000 REM ***MSUB2 -- Modify Customers***
2010 HEAD$="MODIFY CUSTOMERS"
2030 GOSUB 300
2040 INPUT "Customer No.?(If mod's finished press F)
     .. ";CN$: IF CN$="F" THEN 800
2050 CN=VAL(CN$): IF CN=0 OR CN>NCUST THEN 2040
2060 CN=CN-1
2070 FOR I=0 TO 3: CUST$(I)=M$(CN,I): NEXT
2080 PRINT@3,8;"Customer No. :",CN+1
2090 FOR I=0 TO 3
2100 PRINT I+1;PRMPT$(I),CUST$(I)
2110 NEXT: PRINT
2120 PRINT "Any changes for this item
     (Y/N)?";:Y$=INCH$:PRINT Y$
2130 IF Y$<>"Y" THEN 2180
2140 PRINT "Which line (2-4)?";:Y$=INCH$: PRINT Y$:
     PRINT
2150 I=VAL(Y$)-1
2160 IF I=0 OR I>4 THEN 2030 ELSE PRINT PRMPT$(I);:
     INPUT CUST$(I): CLS
2170 GOTO 2080
2180 FOR I=0 TO 3: M$(CN,I)=CUST$(I): NEXT
2190 GOTO 2030
2999 REM
3000 REM ***MSUB3 -- List Customers***
3010 HEAD$="LIST CUSTOMERS"
3020 GOSUB 300: IF NCUST=0 THEN 800
3030 PRINT "To Screen or Printer (S/P)?";: PF$=INCH$:
     PRINT PF$:PRINT
3040 IF PF$="P" THEN PRINT#1
3050 FOR CN=0 TO NCUST-1
3060 PRINT "Customer No. :",CN+1
```



```
3070 FOR I=0 TO 3:  PRINT PRMPT$(I),M$(CN,I):  NEXT:
     PRINT
3080 IF PF$<>"P" THEN INPUT "PRESS ENTER to go on:";Y$:
     PRINT
3090 NEXT CN
3100 PRINT# 0: GOTO 800

8998 REM
8999 REM ** INITIALISATION **
9000 SEP 44: REM Use separator for DATA below
9010 BEL$=CHR$(7):REM Beep
9020 CMAX=100: REM Max. No. of customers allowed
9030 DIM M$(CMAX-1,3),PRMPT$(3),CUST$(3)
9040 FOR I=0 TO 3: READ PRMPT$(I): NEXT
9050 FILE$="SMAIL.DAT": REM file name
9060 SEP 0: REM Allow commas in input text
9070 ZONE 28,20: REM Set up zone width
9080 GOSUB 200: REM Read in data file
9090 GOTO 800: REM Go and do your stuff!
9098 REM
9099 REM *** DATA FOR FIELD PROMPTS***
9100 DATA "Customer Name:","Telephone No.:"
9110 DATA "Addr. Line 1","Addr. Line 2 :"
```

c) A simple Mailing List (Random Access)

The program suite below is given to illustrate both the use of random-access files and the 'semi-CHAIN' facility. It does the same job as the single program at example b), but with much less memory, and shows how the random-access method improves the file-handling capability. The limit on the number of customers is now dictated only by the free disc space available, and the array M$ of example b., is dispensed with.

The suite consists of four programs, the common and setting-up routines, and the three sub-programs which deal with the three options currently supported (see example b. above).

A record length of 75 characters is used, this limits the amount of information that may be held on each customer, checks being needed to ensure that the total lengths of the fields entered (NB, including CR and LF codes!) do not exceed this length. Such checking may be found at lines 1090-1100 in MSUB1, and 1170-1180 in MSUB2 below. This kind of check is not necessary with a sequential file.

The first record contains the total number of records on file (NCUST), and provides a useful way of preventing access above the limit available.

Finally, note the use of the ON ERR routine at 100, which makes special checks for CHAINing to a non-existent sub-program, and allows the user to create a new data file if one is not present.

```
10 REM **SIMPLE MAILING LIST PROGRAM (RANDOM ACCESS)***
20 REM *** COMMON ROUTINES ***
30 ON ERR GOTO 100
40 GOTO 1000
98 REM
99 REM *** ERROR ROUTINE ***
100 IF ERL=900 THEN PRINT "CANNOT INVOKE DESIRED
    OPTION";BEL$: GOTO 800
110 IF ERR<>25 THEN PRINT ERR$;" Error in line ";ERL:
    END
120 PRINT "No data file -- Create (Y/N)?";: Y$=INCH$
130 IF Y$="Y" THEN CREATE FILE$,FD$: PRINT# FD$;"0":
    CLOSE
```



```
140 GOTO 800
198 REM
199 REM *** OPEN DATA FILE ***
200 OPEN FILE$,FD$,RL
210 INPUT# FD$,0;NCUST: INPUT# 0: REM Get No. of
    customers on file
220 RETURN
298 REM
299 REM *** HEADING DISPLAY ***
300 CLS: PRINT@8,0;HEAD$
310 PRINT@3,2;"Number of customers on file: ";NCUST:
    PRINT
320 RETURN
798 REM
799 REM *** MENU DISPLAY ***
800 CLOSE: CLS: PRINT@8,0;"SIMPLE MAIL LIST PROGRAM"
810 PRINT@4,3;"Options:"
820 PRINT@4,5;"0. Exit Program"
830 PRINT@4,7;"1. Enter Customers"
840 PRINT@4,9;"2. Modify Customers"
850 PRINT@4,11;"3. List Customers"
870 PRINT@4,13;"Which? ";: N$=INCH$: PRINT N$
880 N=VAL(N$): IF N<0 OR N>3 THEN PRINT BEL$: GOTO 870
890 IF N=0 THEN CLS: PRINT@8,0;"GOODBYE!";BEL$: END
898 REM
899 REM *** CHAIN TO OTHER SUB-PROGRAMS ***
900 HOLD 1000: CHAIN "MSUB"+N$
910 STOP: REM SHOULD NEVER GET HERE!
998 REM
999 REM *** END OF COMMON ROUTINES ***
1000 REM ** INITIALISATION **
1010 SEP 44: REM Use separator for DATA below
1020 BEL$=CHR$(7):REM Beep
1030 DIM CUST$(3),PRMPT$(3)
1040 FOR I=0 TO 3: READ PRMPT$(I): NEXT
1050 FILE$="RMAIL.DAT": RL=75: REM File name & record
     size
1060 SEP 0: REM Allow commas in input text
```

```
1070 ZONE 28,20: REM Set up zone width
1080 GOTO 800:
1098 REM
1099 REM *** DATA FOR FIELD PROMPTS ***
1100 DATA "Customer Name:","Telephone No.:"
1110 DATA "Addr. Line 1 :","Addr. Line 2 :"

1000 REM *** MSUB1 -- Enter Customers ***
1010 HEAD$="ENTER CUSTOMERS"
1020 GOSUB 200
1030 GOSUB 300
1040 PRINT"Any more customers to add (Y/N)?";:Y$=INCH$:
     PRINT Y$: PRINT
1050 IF Y$<>"Y" THEN 800
1060 FOR I=0 TO 3
1070 PRINT PRMPT$(I);: INPUT CUST$(I)
1080 NEXT
1090 L=0: FOR I=0 TO 3: L=L+LEN(CUST$(I))+2: NEXT
1100 IF L>RL THEN PRINT "RECORD TOO LONG";BEL$:  GOTO
     1030
1110 PRINT# FD$,NCUST+1
1120 FOR I=0 TO 3: PRINT CUST$(I): NEXT: NCUST=NCUST+1
1130 PRINT# FD$,0;  NCUST:  PRINT# 0: REM Update No. of
     customers
1140 GOTO 1030

1000 REM *** MSUB2 -- Modify Customers ***
1010 HEAD$="MODIFY CUSTOMERS"
1020 GOSUB 200
1030 GOSUB 300
1040 INPUT "Customer No.?(If mod's finished press F)
     .. ";CN$: IF CN$="F" THEN 800
1050 CN=VAL(CN$): IF CN=0 OR CN NCUST THEN 1040
1060 INPUT# FD$,CN
1070 FOR I=0 TO 3: INPUT CUST$(I): NEXT: INPUT# 0
1080 PRINT@3,8;"Customer No. :",CN
```



```
1090 FOR I=0 TO 3
1100 PRINT I+1;PRMPT$(I),CUST$(I)
1110 NEXT: PRINT
1120 PRINT "Any changes for this item (Y/N)?";:Y$=
     INCH$: PRINT Y$
1130 IF Y$<>"Y" THEN 1170
1140 PRINT "Which Line (2-4)?";:  Y$=INCH$:  PRINT Y$:
     PRINT
1150 I=VAL(Y$)-1: PRINT PRMPT$(I);: INPUT CUST$(I): CLS
1160 GOTO 1080
1170 L=0: FOR I=0 TO 3: L=L+LEN(CUST$(I))+2:NEXT
1180 IF L>RL THEN PRINT "RECORD TOO LONG";BEL$:GOTO
     1080
1190 PRINT# FD$,CN: FOR I=0 TO 3: PRINT CUST$(I): NEXT:
     PRINT# 0
1200 GOTO 1030

1000 REM *** MSUB3 -- List Customers ***
1010 HEAD$="LIST CUSTOMERS"
1020 GOSUB 200
1040 GOSUB 300
1050 PRINT "To Screen or Printer (S/P)?";:  PF$=INCH$:
     PRINT PF$: PRINT
1060 IF PF$="P" THEN PRINT#1
1070 FOR CN=1 TO NCUST
1080 INPUT# FD$,CN: REM Read Customer record from file
1090 FOR I=0 TO 3: INPUT CUST$(I): NEXT: INPUT# 0
1100 PRINT "Customer No. :",CN
1110 FOR I=0 TO 3:  PRINT PRMPT$(I),CUST$(I):  NEXT:
     PRINT
1120 IF PF$<>"P" THEN INPUT "PRESS ENTER to go on:";Y$:
     PRINT
1130 NEXT CN
1140 GOTO 800
```

15

PROGRAMMABLE SOUND GENERATOR

All the functions of the Programmable Sound Generator (usually abbreviated to PSG) are controlled by the Z80 processor by means of a series of REGISTER loads. Each register of the PSG relates to a specific function involved with the creation of a particular sound or sound effect. The following table indicates the respective functions and value ranges of the PSG registers.

Register	Function	Range
0	Channel A - lower 8 bits Pitch	0 to 255
1	Channel A - upper 4 bits	0 to 15
2	Channel B - lower 8 bits Pitch	0 to 255
3	Channel B - upper 4 bits	0 to 15
4	Channel C - lower 8 bits Ptich	0 to 255
5	Channel C - upper 4 bits	0 to 15
6	Noise period	0 to 31
7	Enable	0 to 255
8	Channel A - Amplitude	0 to 31
9	Channel B - Amplitude	0 to 31
10	Channel C - Amplitude	0 to 31
11	Envelope period lower 8 bits	0 to 255
12	Envelope period upper 8 bits	0 to 255
13	Envelope shape/cycle	0 to 15
14	Port A	0 to 255
15	Port B	0 to 255

REGISTER 0 to 5

The values stored in these registers determine the frequency, or pitch, of the outputs of the respective channels A, B and C. Registers 0 and 1 control the pitch of Channel A, register 2 and 3 the pitch of Channel B, and registers 4 and 5 the pitch of Channel C.

To Determine the Pitch

The pitch generated by each of the 3 channels is determined by two registers for each channel. The values stored in the two registers represent a 12-bit number (0 to 4095 in decimal).

The equivalent 12-bit number, in decimal, can be found from:-

$$TP = 256 \times RU + RL$$

Where:-

TP is the decimal equivalent of the 12-bit tone period number.
RL is the lower 8 bits (Register 0 for Channel A)
RU is the upper 4 bits (Register 1 for Channel A)

RL can be thought of as a "fine tune" register and can have any value in the range 0 to 255.
RU can be thought of as a "coarse tune" register and can have any value in the range 0 to 15.

To find the pitch:-

$$\text{Pitch} = \frac{2 \times 10^6}{16 \times TP} \quad \text{Hertz}$$

Thus the higher the register values, the lower the pitch.

To find the Register Values

Given the desired pitch, it is possible to find the values for the two registers for each channel.

First find the Tone Period (TP) from:-

$$TP = \frac{2 \times 10^6}{16 \times \text{pitch (in Hertz)}}$$

Having determined the tone period, the register values can be obtained as follows:-

$$RU + \frac{RL}{256} = \frac{TP}{256}$$

Thus RU is given by the integer of (TP ÷ 256) and RL is given by multiplying the remainder from $\frac{TP}{256}$ by 256

Example:
Required frequency is 100Hz - what are the register values for Channel A?

For channel A RL is register 0 and RU is register 1

Calculating the Tone Period (TP):-

$$TP = \frac{2 \times 10^6}{16 \times 100}$$

$$= \underline{\underline{1250}}$$

To find the value in register 1 (R1)

$$R1 = \frac{1250}{256} \text{ (ie integer of } 1250 \div 256$$

$$= \underline{\underline{4}} \quad (1250 \div 256 = 4.88281)$$

Remainder from division is 0.88281

∴ register 0 (R0) = 0.88281 x 256

$$= \underline{\underline{226}}$$

Although the range of frequencies which can be produced by the sound generator is from 30.5Hz to 125kHz, this is, in practice, limited by the capabilities of the audio amplifier/speaker combination, which sets an upper frequency limit of about 15kHz. (In practice, the human ear also sets a limit of about 10 to 17kHz, depending upon the age of the listener).

REGISTER 6

This register controls the frequency of the noise generated by the PSG. It is similar to the pitch registers, previously described, except that there is only one register, and the range of noise frequencies produced is from 4kHz to 125kHz.

To determine the Noise Frequency

$$\text{Noise frequency} = \frac{2 \times 10^6}{16 \times R6} \text{ Hertz}$$

R6 represents the contents of register 6 (in decimal), and can have values from 0 to 31.

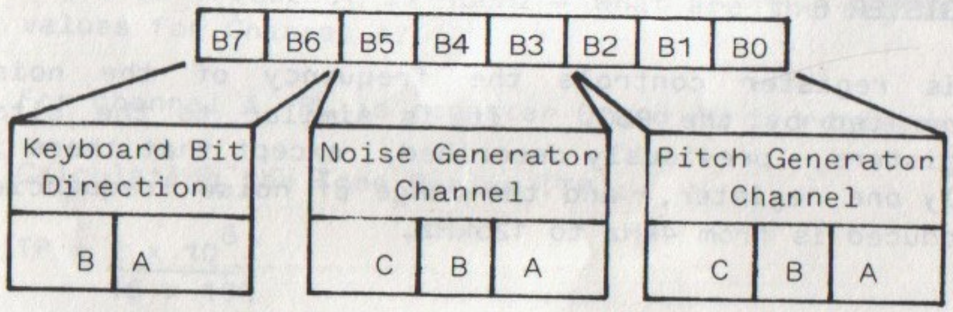
Similarly, given the noise frequency:-

$$\text{Then } R6 = \frac{2 \times 10^6}{16 \times \text{Noise Frequency (Hertz)}}$$

REGISTER 7

This is an 8-bit control register which is used to enable noise and pitch on channels A, B and C, and to control the direction of the keyboard scan ports, A and B.

REGISTER 7



Bits 0 to 2 - Enable pitch generator on channels A, B and C

Bits 3 to 5 - Enable noise generator on channels A, B and C

A logic '0' in bits 0 to 5 will enable the respective channel (ie 0=on, 1=off)

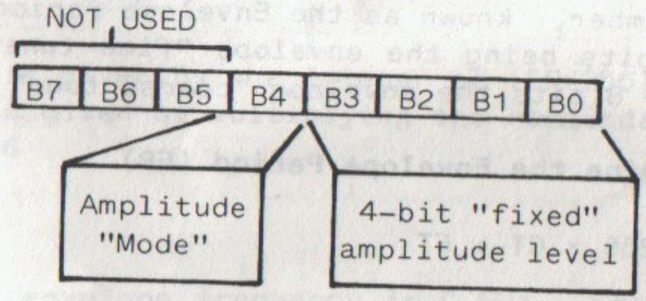
Bits 6 and 7 - A '0' in bits 6 and 7 will configure the keyboard ports as inputs, whilst a '1' will configure them as outputs. The machine software normally configures port A as output and port B as input (i.e. B6 = '1', B7 = '0')

Notes: Care should be excercised when setting up this register, as writing to bits 6 and 7 could result in the keyboard being disabled.

The PSG command in BASIC expects entry in decimal or hexadecimal. It would be wise to select values for this register which do not change the setting of bits 6 and 7 (as configured by the machine software).

REGISTERS 8 to 10

These registers control the amplitude of the signals generated by each of the channels A, B and C, and also selects the "amplitude mode". Register 8 controls channel A, register 9 controls channel B, and register 10 controls channel C.



Bits 0 to 3 - Define the "fixed" level amplitude of a channel according to the value loaded for each register (values from 0 to 15)

Bit 4 - A '0' in bit 4 selects "fixed level amplitude" mode, whilst a '1' will select "variable level amplitude" and hence enable the Envelope Generator. It follows therefore, that bits 0 to 3, defining the value of a "fixed" level amplitude, are only active when bit 4 = '0' (i.e. they are ignored when bit 4 = 1 and amplitude control passes to the Envelope Generator)

Note: To turn a channel off the all zeros code is used in bits 0 to 3 (i.e. 0000)

REGISTERS 11 and 12

These registers are used to control and vary the frequency of the envelope generated (i.e. the Envelope Period Control).

The values stored in these two registers represents a 16 bit number, known as the Envelope Period (EP), the lower 8 bits being the envelope "fine tune" (R11) and the upper 8 bits the envelope "coarse tune" (R12).

To Determine the Envelope Period (EP)

$$EP = 256 \times CT + FT$$

Where:-

EP is the decimal equivalent of the 16 bit Envelope period number.

FT is the decimal equivalent of the "fine tune" register bits (i.e. lower 8 bit number).
CT is the decimal equivalent of the "coarse tune" register bits (i.e. upper 8 bit number).

To Determine the Envelope Frequency

$$\text{Envelope Frequency} = \frac{2 \times 10^6}{256 \times EP}$$

To find the register values
Given the envelope frequencies it is possible to find the values of the two registers.

First find the Envelope Period (EP) from:-

$$EP = \frac{2 \times 10^6}{256 \times \text{Frequency}}$$

Having determined the envelope period, the register values can be obtained as follows:-

$$CT + \frac{FT}{256} = \frac{EP}{256}$$

Thus CT is given by the integer of (EP÷256)
and FT is given by multiplying the remainder from $\frac{EP}{256}$ by 256

Example:
Required envelope frequency is 0.5Hz - what are the register values?

Calculating the Envelope Period (EP):-

$$EP = \frac{2 \times 10^6}{256 \times 0.5} = \underline{\underline{15,625}}$$

To find the value of register 12 (coarse tune register CT)

Register 12 (CT) = $\left|\frac{15,625}{256}\right|$ (i.e. integer of 15,625 ÷ 256)

= 61 (15,256 ÷ 256 = 61.035156)

Remainder from division is 0.035156

∴ register 11 (FT) = 0.035156 x 256

= 9

REGISTER 13

The lower 4 bits of register 13 control the envelope shape and cycle. The upper 4 bits of the register are not used.

Each of the lower 4 bits controls a function in the envelope generator as follows:-

Bit 0 - HOLD
Bit 1 - ALTERNATE
Bit 2 - ATTACK
Bit 3 - CONTINUE

HOLD - When set to logic '1' limts the envelope to one cycle, holding the last count of the envelope counter (0000 or 1111, depending whether the envelope counter was in a count-down or count-up mode, respectively).

ALTERNATE - When set to logic '1', the envelope counter reverses count direction (up-down) after each cycle.

ATTACK - When set to logic '1', then envelope counter will count up (attack) from 0000 to 1111; when set to logic '0', the envelope counter will count down (decay) from 1111 to 0000.

CONTINUE - When set to logic '1', the cycle pattern will be as defined by the HOLD bit; when set to logic '0', the envelope generator will reset to 0000 after one cycle and hold at that count.

Note: When both the HOLD bit and ALTERNATE bit are set to '1', the envelope counter is reset to its initial count before holding.

Fig.15.1 illustrates the various options available for envelope generator output.

REGISTER 14 and 15

These registers function as intermediate data storage registers between the PSG/CPU data bus and the two I/O ports available on the PSG. Using these registers for the transfer of I/O data has no effect at all on sound generation.

To output data from CPU to a peripheral on I/O Port A

1. Latch address R7 (select $\overline{\text{Enable}}$ register)
2. Write data to PSG (setting bit 6 of R7 to '1')
3. Latch address R14 (select IOA register)
4. Write data to PSG (data to be output on I/O Port A)

ENVELOPE SHAPE/CYCLE CONTROL

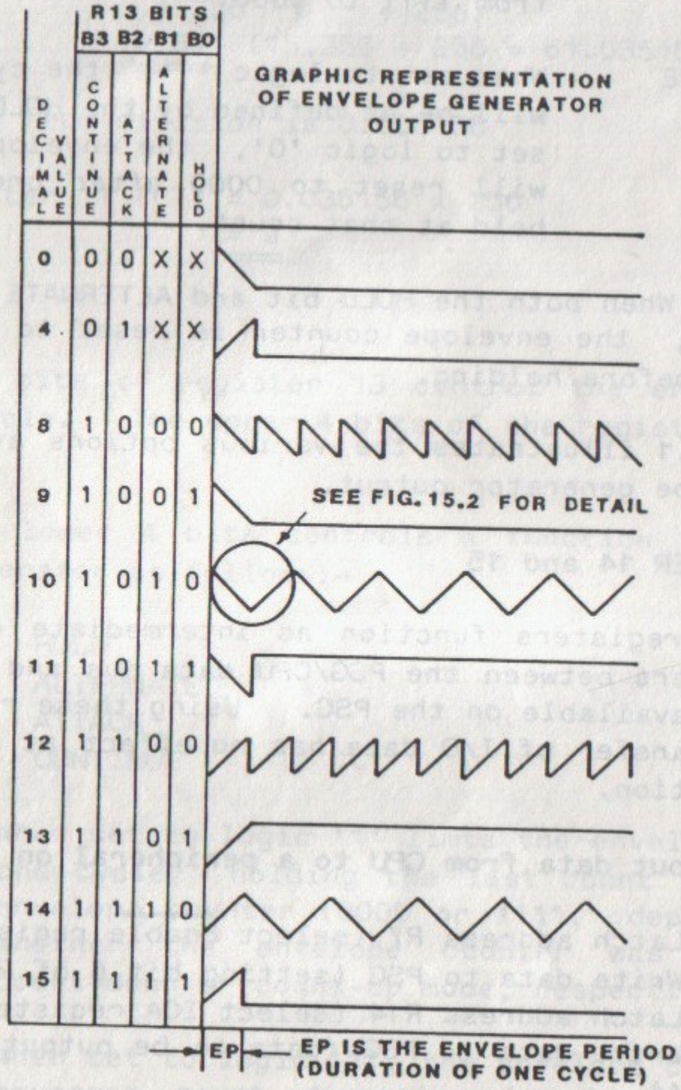


Fig.15.1

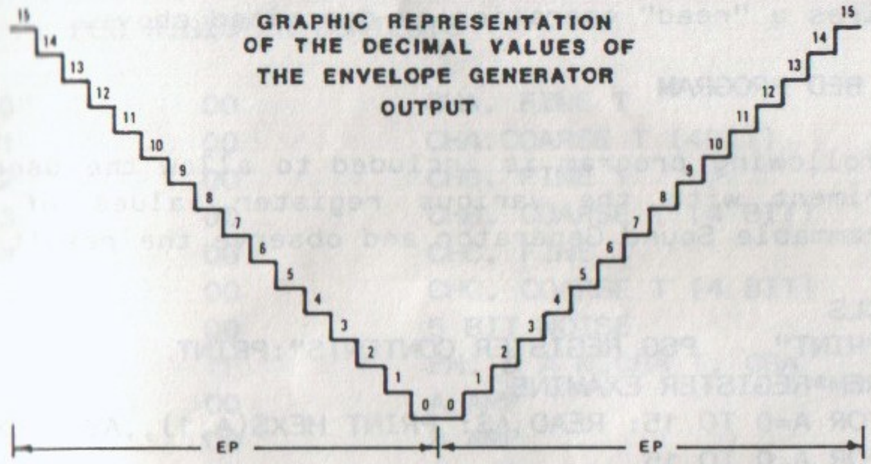


Fig.15.2

To input data from I/O Port A to CPU

1. Latch address R7 (select Enable register)
2. Write data to PSG (setting bit 6 of R7 to '0')
3. Latch address R14 (select IOA register)
4. Read data from PSG (data from I/O Port A)

Notes: Once loaded with data in the output mode, the data will remain on the I/O port(s) until changed either by loading different data, by applying a reset or by switching to the input mode.

When in the input mode, the contents of registers 14 and/or 15 will follow the signals applied to the I/O port(s). However, transfer of this data to the CPU bus requires a "read" operation as described above.

TEST BED PROGRAM

The following program is included to allow the user to experiment with the various register values of the Programmable Sound Generator and observe the results.

```
 10 CLS
 20 PRINT"    PSG REGISTER CONTENTS":PRINT
 30 REM*REGISTER EXAMINE
 40 FOR A=0 TO 15: READ A$: PRINT HEX$(A,1),,A$: NEXT A
 50 FOR A=0 TO 15
 60 PRINT@10,A+2;HEX$(PSG(A),2)
 70 NEXT A
 80 REM *REGISTER CHANGE
 90 PRINT"REGISTER(0 TO F)  DATA(HEX)"
100 A$=INCH$: IF A$=CHR$(13) THEN END
110 IF A$ < "0" OR A$>"F"  THEN 100
120 PRINT A$,
130 B$=INCH$(2): D=VAL("&"+B$)
140 IF D < 0 OR D > 255 THEN 130
150 PSG VAL("&"+A$),D
160 GOTO 50
170 DATA"CHA.FINE T","CHA.COARSE T(4 BIT)"
180 DATA"CHB.FINE T","CHB.COARSE T(4 BIT)"
190 DATA"CHC.FINE T","CHC.COARSE T(4 BIT)"
200 DATA"5 BIT NOISE","EN.B A N.CBA T.CBA","A AMP","B
    AMP","C AMP","ENV.FINE","ENV.COARSE","ENV.SHAPE (4
    BIT)"
210 DATA"I/O PORT A","I/O PORT B"
```

Type in the program in BASIC if you wish to experiment with the PSG.

The following display appears on the screen when the program is RUN

```
        PSG REGISTER CONTENTS

0           00          CHA. FINE T
1           00          CHA.COARSE T (4BIT)
2           00          CHB. FINE T
3           00          CHB. COARSE T (4 BIT)
4           00          CHC. FINE T
5           00          CHC. COARSE T (4 BIT)
6           00          5 BIT NOISE
7           7F          EN. B A N.CBA T. CBA
8           00          A AMP
9           00          B AMP
A           00          C AMP
B           00          ENV. FINE
C           00          ENV. COARSE
D           00          ENV. SHAPE (4 BIT)
E           00          I/O PORT A
F           FF          I/O PORT B
REGISTER    (0 to F)    DATA (HEX)
```

The registers are numbered down the left hand side 0 to F (in HEX). The centre column displays the register contents (in HEX). The corresponding functions are listed down the right hand side (relate these to the previous sections of this chapter).

To use the program

1. Select a register number from 0 to F and key it in. The number will appear below the column of register numbers at the initial cursor position. The cursor will then transfer to a position below the centre column of register values.

2. Key in the required value (in HEX) for that register. The value will automatically appear in the table on the screen and the cursor will then return to the left hand column position ready for the next entry.

The following values are given as an example to begin with. These values will set up a single tone from all three channels (A, B, C).

Register 0 - 20	Tone Generator Control (Fine and Coarse Tune)
Register 1 - 01	
Register 2 - 20	
Register 3 - 01	
Register 4 - 20	
Register 5 - 01	
Register 6 - 00	Noise Generator
Register 7 - 78	Noise/Tone Select and Enable
Register 8 - 18	Amplitude of Channel A
Register 9 - 18	Amplitude of Channel B
Register A - 18	Amplitude of Channel C
Register B - 00	Envelope Period Control (Fine and Coarse Tune)
Register C - 0D	
Register D - 08	Envelope Shape (See Fig.15.1)
Register E - 00	I/O Port A
Register F - FF	I/O Port B

The values in registers 0 to 5 set the Coarse and Fine tune of the tone generator.

Register 6 is not given a value because the example is using a tone rather than a noise.

The value given in Register 7 selects the Tone Generator, as appose to the Noise Generator, and enables each of the channels A, B, C.

Register B (decimal 11) is the envelope period fine tune and is not used in this example whereas the envelope period coarse tune is set by the value in register C (decimal 12).

The value in register D (decimal 13) selects an envelope shape which gives an intermittent effect (see table in Fig.15.1)

The values in registers E (decimal 14) and F (decimal 15) should not be changed since the I/O ports are not being used. (The I/O ports are used to scan the keyboard).

Having set up the given example, change some of the register values and observe the effect on the sound given. When selecting new values for the register the following points should be noted.

a) Values for register 7 should lie within the range &40 to &7F so as to avoid the possibility of disabling the keyboard.

b) A value of &47 in register 7 will select the Noise Generator, rather than tone generator, and enable channels A, B, C. In this case a value can also be given to register 6

SOUND VARIATION

Relative Channel Volume

The independently programmable amplitude control for each channel allows up to 16 levels if using the processor controlled amplitude mode (bit 4 of registers 8, 9, or 10 = 0). In the case of a decaying or steady note, when a note is played or "fired", a frequency may be set up in the coarse and fine tune registers and then an amplitude value placed in the respective register 10, 11, or 12. The value which is placed to play the tune can be an independent variable, allowing channels to play their respective melody lines with varying force.

Decay

One difference between sounds is the speed with which the note gains and loses volume. This is known as attack and decay. If all of the notes can be decayed at a uniform rate, the automatic envelope generator can be set to produce a decaying waveform. Each of the three channels can have the same decay constant but differing playing times to simulate the same instrument with differing note strike-times.

Other Effects

The addition of variable noise to any or all of the channels can produce effects such as "breathing" with a wind instrument. Or noise can be used alone to produce a drum rhythm. The fact that the noise dominant frequencies are variable allows "synthesizer" type effects with simple processor interaction.

Other pleasing effects include vibrato and tremolo, the cyclical variation of the frequency and volume. Because an intelligent microprocessor is controlling the effect, they can be all keyed to the tune itself or to other external stimuli.

SPECIAL SOUND EFFECTS

One of the main uses of the PSG is to produce non-musical sound effects to accompany visual action or as a feature in itself. The following sections outline techniques and provide actual examples of some popular effects.

Tone Only Effects

Many effects are possible using only the tone generation capability of the PSG without adding noise and without using the PSG's envelope generation capability. Examples of this type of effect would include telephone tone frequencies (two distinct frequencies produced simultaneously) or the European Siren effect listed below (two distinct frequencies sequentially produced).

EUROPEAN SIREN SOUND EFFECT

The following BASIC listing will produce the European Siren effect.

```
 10 REM SIREN EFFECT
 20 FOR I=0 TO 9
 30 PSG0,254:PSG1,0
 40 PSG7,126
 50 PSG8,15
 60 FOR J=1 TO 300:NEXT
 70 PSG0,86:PSG1,1
 80 FOR J=1 TO 300:NEXT
 90 NEXT I
100 PSG8,0
110 END
```

Noise Only Effects

Some of the more commonly required sounds require only the use of noise and the envelope generator (or processor control of channel envelope if other channels are using the envelope generator).

Examples of this, listed below, are gunshot and explosion. In both cases pure noise is used with a decaying envelope.

In the examples shown the only changes are in the length of the envelope as modified by the coarse tune register and in the noise period.

GUNSHOT SOUND EFFECT

The following BASIC listing will produce the effect.

```
10 REM GUNSHOT
20 FOR J=1 TO 4
30 PSG6,15:PSG7,71
40 PSG8,16:PSG9,16:PSG10,16
50 PSG12,16
60 PSG3,0
70 T=RND(2000):FOR I=1 TO T:NEXTI
80 NEXT J
90 END
```


EXPLOSION SOUND EFFECT

The following BASIC listing will produce the Exposion effect

```
10 REM EXPLOSION
20 PSG6,31:PSG7,71
30 PSG 8,16:PSG9,16:PSG10,16
40 PSG12,100
50 PSG13,0
60 END
```

Frequency Sweep Effect

The Laser, Whistling Bomb, Wolf Whistle, and Race Car sounds, listed below, all utilize frequency sweeping effects. In all cases they involve the increasing or decreasing of the values in the tone period registers with variable start, end, and time between frequency changes. For example, the sweep speed of the Laser is much more rapid than the high gear accelerate in the race car, yet both use the same computer routine with differing parameters.

Other easily achievable results include "doppler" and noise sweep effects. The sweeping of the noise clocking register (R6) produces a "doppler" effect which seems well suited for "space war" type games.

LASER SOUND EFFECT

The following BASIC listing will produce the Laser Sound effect.

```
 10 REM LASER
 20 PSG7,126
 30 PSG8,15
 40 FOR I=18 TO 255 STEP 40
 50 PSG0,I
 60 NEXT
 70 FOR I=255 TO 18 STEP-10
 80 PSG0,I
 90 NEXT
100 END
```

WHISTLING BOMB EFFECT

The following BASIC listing will produce the Whistling Bomb effect.

```
 10 REM WHISTLING BOMB
 20 PSG7,126
 30 PSG8,15
 40 FOR I=1 TO 255
 50 PSG0,I
 60 NEXT
 70 PSG6,31:PSG7,71
 80 PSG8,16:PSG9,16:PSG10,16
 90 PSG12,100
100 PSG13,0
110 END
```


Multi-Channel Effect

Because of the independent architecture of the PSG, many rather complex effects are possible without burdening the processor. For example, the Wolf Whistle effect below shows two channels in use to add constant breath hissing noise to the three concentrated frequency sweeps of the whistle. Once the noise is put on the channel, the processor only need be concerned with the frequency sweep operation.

WOLF WHISTLE SOUND EFFECT

The following BASIC listing will produce the Wolf Whistle effect.

```
 10 REM WOLF WHISTLE
 20 PSG6,1:PSG7,110:PSG9,9
 30 PSG1,0:PSG8,15
 40 FOR I=64 TO 32 STEP-.35:PSG0,I:NEXT
 50 FOR I=0 TO 150:NEXT
 60 FOR I=64 TO 48 STEP-.17:PSG0,I:NEXT
 70 FOR I=48 TO 104 STEP.5:PSG0,I:NEXT
 80 PSG8,0:PSG9,0
 90 END
```

RACE CAR SOUND EFFECT

The following BASIC listing will produce the Race Car effect, including gear changes.

```
 10 REM RACING CAR
 20 PSG3,15
 30 PSG7,124
 40 PSG8,15:PSG9,10
 50 S=11:F=4:GOSUB110
 60 S=9:F=3:GOSUB110
 70 S=6:F=1:GOSUB110
 80 PSG8,0:PSG9,0
```

```
 90 END
100 REM SWEEP ROUTINE
110 FOR I=S TO F STEP-1:PSG1,I
120 PSG0,255
130 FOR J=255 TO 0 STEP-1:PSG0,J
140 NEXT J,I
150 RETURN
```


APPENDIX A

LIST OF RESERVED WORDS

WORD	WORD	WORD	WORD	WORD
ABS	ELSE	LISTP	PSG	THEN
ADC	END	LN	PSW	TI$
AND	EOF	LOAD	PTR	TO
APPEND	ERA	LOCK	RAD	UNLOCK
ASC	ERL	LOG	READ	UNPLOT
ATN	ERR	MAG	REM	VAL
AUTO	EVAL	MGE	REN	VDEEK
BCOL	EXP	MID$	RENUM	VDOKE
BEEP	FILL	MOD	RESTORE	VERIFY
BIN$	FMT	MOS	RETURN	VOICE
BTN	FN	MUL$	RIGHT$	VPEEK
CALL	FOR	MUSIC	RND	VPOKE
CHAIN	GCOL	NEW	RUN	WAIT
CHR$	GOSUB	NEXT	SAVE	WIDTH
CLEAR	GOTO	NOT	SCRN$	XOR
CLOSE	HEX$	NULL	SEP	ZONE
CLS	HOLD	OFF	SGN	
CONT	IF	ON	SHAPE	
COS	INCH	OPEN	SIN	
CREATE	INCH$	OR	SIZE	
DATA	INP	ORIGIN	SPC	
DEEK	INPUT	OUT	SPEED	
DEF	INPUT#	PEEK	SPRITE	
DEG	INT	PI	SQR	
DEL	IOM	PLOT	STEP	
DIM	KBD	POINT	STOP	
DIR	KBD$	POKE	STR$	
DOKE	KEY	POLY	SWAP	
DOS	LEFT$	POP	TAB	
DRAW	LEN	POS	TAN	
DRIVE	LET	PRINT	TCOL	
ELLIPSE	LIST	PRINT#	TEMPO	

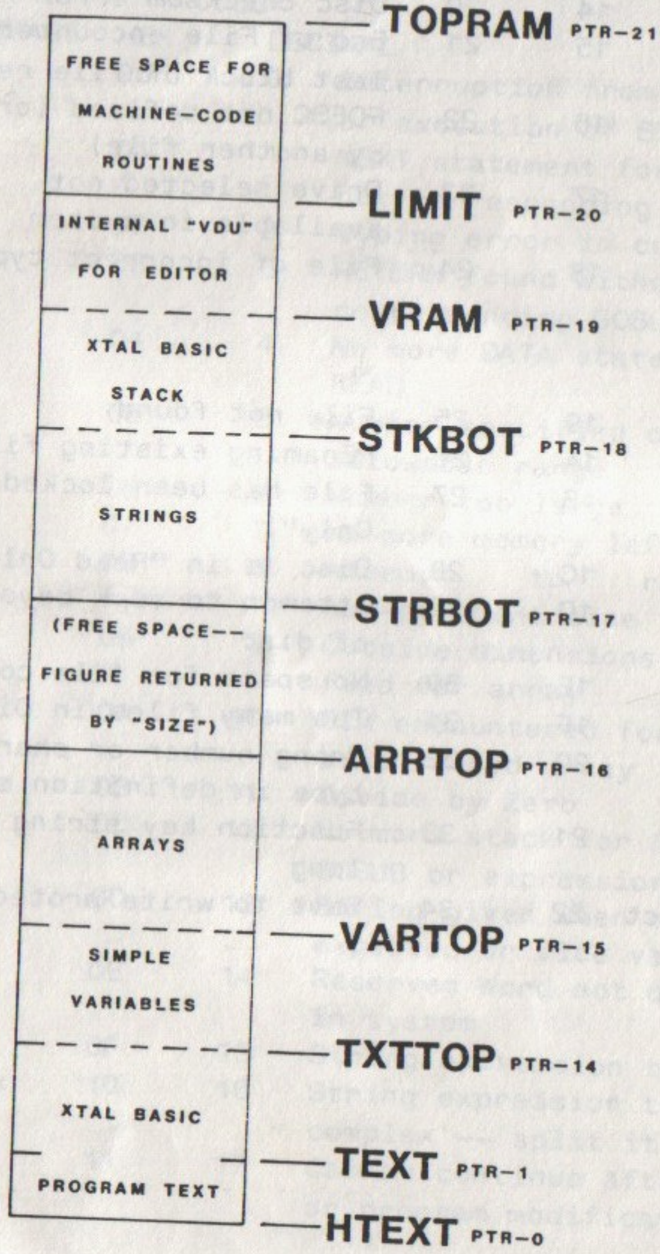
APPENDIX B

INDEX TO ERROR MESSAGES

ERROR MESSAGE	CODE HEX	DECIMAL	
Break	00	0	Interruption from keyboard or execution of STOP
Next	01	1	NEXT statement found without corresponding FOR
Syntax	02	2	Typing error in current line
Return	03	3	RETURN found without corresponding GOSUB
Data	04	4	No more DATA statements for READ
Qty	05	5	Number specified outside allowable range
Ovfl	06	6	Number too large
Mem Full	07	7	No more memory left
Branch	08	8	Attempt to refer to non-existent line
Range	09	9	Outside dimensions specified for array
Dimension	0A	10	DIM encountered for already dimensioned array
Division	0B	11	Divide by Zero
Stack Full	0C	12	No more stack for FILL FOR, GOSUB or expressions
Type	0D	13	String given when number expected or vice versa
Cmd	0E	14	Reserved Word not defined in system
Str Ovfl	0F	15	String expression too long
Str Complex	10	16	String expression too complex -- split it up!
Cont	11	17	Cannot continue after error or program modification
Fn Defn	12	18	FN user function not defined by a previous DEF
Operand	13	19	Operand expected in expresssion
Bad Data	14	20	Disc checksum error
End of Text	15	21	End of File encountered or last block of file read.
File	16	22	FDESC not defined (or used by another file)
Drive Select	17	23	Drive selected not available in system
File Type	18	24	File of incorrect type
DISC ERRORS			
No File	19	25	File not found
File Exists	1A	26	REnaming existing file
File Locked	1B	27	File has been locked ("Read Only")
Disc Locked	1C	28	Disc is in "Read Only" mode
Disc Seek	1D	29	Attempt to seek beyond end of disc
Disc Full	1E	30	No space for file contents
Dir Full	1F	31	Too many files in Directory
Shape Defn	20	32	Wrong number or character type in definition string
Key Defn	21	33	Function key string is too long
Write Protect	22	34	Save to write protected disc

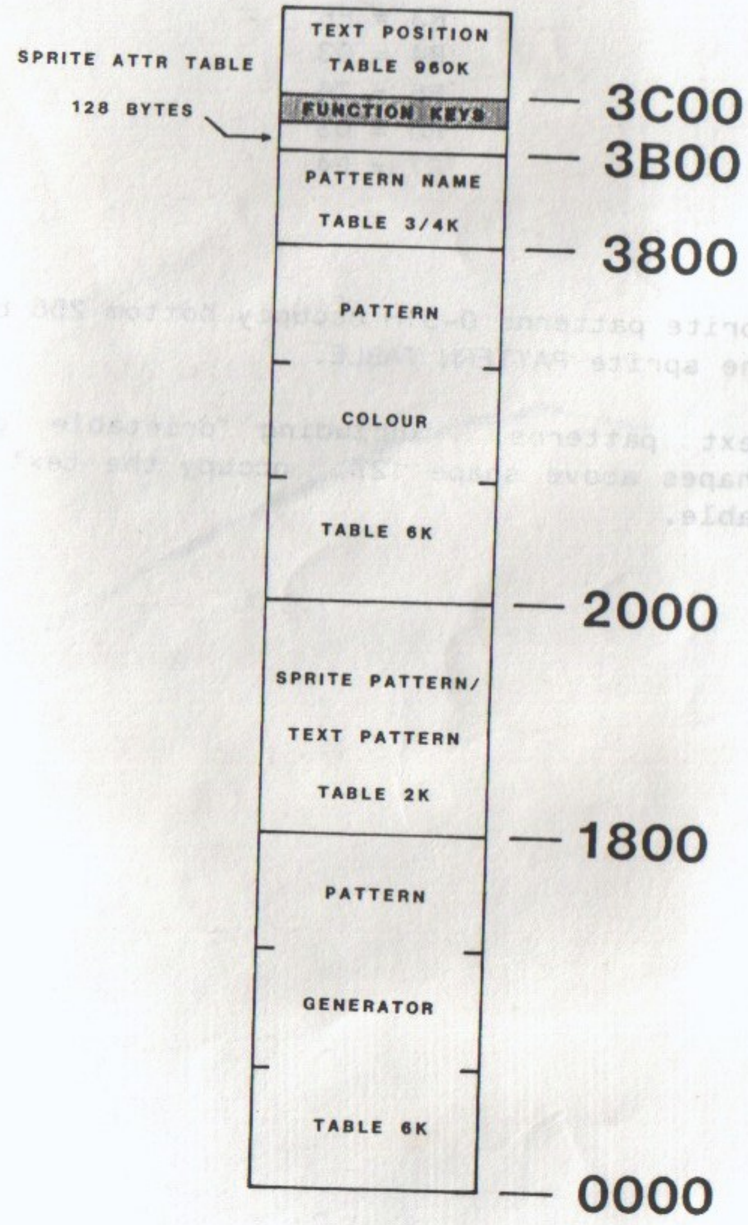
APPENDIX C

MEMORY MAP FOR TATUNG/Xtal BASIC 4



APPENDIX D

VDP MEMORY MAP



VDP REGISTERS DEFAULT SETTINGS

R0 = 02
R1 = C0
R2 = 0E
R3 = FF
R4 = 03
R5 = 76
R6 = 03
R7 = F4

NOTES

1. Sprite patterns 0-31H occuply bottom 256 bytes of the sprite PATTERN TABLE.

2. Text patterns, including printable graphics shapes above shape 127, occupy the text pattern table.