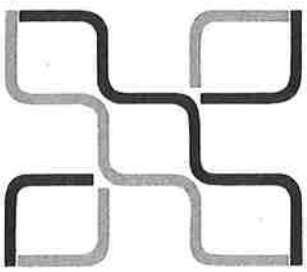
POLYBASIC MANUAL



polycorp

POLYBASIC MANUAL

Version 2.3

December 1984

POLYCORP

New Zealand Limited

Progeni House, 14/18 Pretoria Street
Telephone (04) 693-302, P.O. Box 30-243, Lower Hutt
Telex PO LHT NZ3740 Attn Progeni

CONTENTS

1. THE POLY SYSTEM

POLYBASIC incorporates the standard BASIC constructs (with many enhancements) and extensions to control the special features of the POLY System (such as multiple screens, colour and graphics). This manual is a reference manual which describes the POLYBASIC programming language.

A POLY operates in either System or Standalone mode.

1.1. SYSTEM MODE

In System mode, each of the POLY units is connected via a communications line to a central disk unit on which programs and data may be stored. All commands and statements are available in System mode. Within System mode, the POLY may be in BASIC mode, TEXT mode or DOS mode. This manual describes BASIC mode. TEXT and DOS modes are described in the POLYSYS Utilities Manual. Within BASIC mode the POLY may also operate in courseware mode when menus are displayed and programs executed depending on menu selection. Menu operation is described in the POLY System Operating Manual.

1.2. STANDALONE MODE

In Standalone mode, the POLY is not connected to other units or any disk unit. Only restricted POLYBASIC is available as the disk extensions cannot normally be loaded. However, if the optional cassette interface is installed in your POLY, it is possible to load the normally disk-based extensions via cassette. See the description of the CLOAD statement.

1.3. THE POLY KEYBOARD

The POLY keyboard has been specifically designed to handle the special requirements of educational computing. It contains a standard "QWERTY" keyboard with special keys added on the right hand side. See Appendix 4.7.

Special keys include:-

1. Cursor control keys. These arrow keys may be used to move the cursor around the screen and are used extensively when editing. As well, they may be programmed to the user's requirements.

2. The character insert and delete keys may also be used to edit source and are programmable.

3. The line insert key may be used to move lines down the screen enabling a new line to be entered. The line delete key may be used to remove lines from the screen (they are not removed from the file). Both keys are user-programmable.

4. If the POLY is waiting for input, pressing the <CALC> key allows calculations to be performed on the bottom line of the screen.

   For example:

   2*2 <ENTER> gives 4

   To exit calc mode, press the <CALC> key again.

5. The <HELP> key may be used following the display of a error message to display a more detailed explanation. It may also be programmed.

6. The <PAUSE> key is used to temporarily stop the operation of a POLYBASIC program or a listing when using either the LIST or +LIST command. Operation is restarted by pressing any key. If the <PAUSE> or the <SPACE> bar is pressed, the program lines are executed or listed one by one.

7. The <EXIT> key halts the current program. Execution may be restarted from where the program left off by entering the command CONT. In most teaching modules the <EXIT> key is trapped and causes the MENU program to be executed. Pressing the <EXIT> key when paused during a listing (with LIST but not +LIST) will cause the listing to be terminated.

8. The <NEXT>, <BACK> and <REPEAT> keys are used extensively in the teaching modules and may be programmed as required.

9. The numeric keypad returns the digits 0 to 9 but may also be programmed for special functions as well as being able to be set to return any ASCII value.

1.4. THE POLY SCREENS

The POLY system has the unique feature of multiple screens. These may be compared to a series of transparencies one behind the other. There are two text screens, two graphics screens and a half intensity background screen. The individual screens may be switched ON or OFF as required, and have a fixed display priority of 1, 2, 3, 4, Background. Screens 1 and 3 are the text screens and 2 and 4 are the graphics screens.

The text screens contain 24 rows of 40 characters, the rows being numbered 0 to 23 from the top of the screen, and the columns being numbered 0 to 39 from the left hand side.

The Teletext character generator is used to display on the text screens and the Teletext control character conventions are therefore followed.

Control characters are printed onto the screen to achieve colour, flashing, double height, and chunky graphics. On the text screens, the chunky graphics have six blocks per character position and may be set by using special control characters followed by the printing of one of the characters which has a chunky graphics equivalent. Individual chunks may be set using SET and RESET.

The text screens may be SPLIT at any row so that the upper and lower sections may be used independently, each with its own scrolling. Alternatively, the scrolling may be turned off. Specific sections of the screen may be printed using PRINT@.

The graphics screens each contain 204 rows of 240 pixels, the rows being numbered 0 to 203 from the top of the screen, and the columns 0 to 239 from the left of the screen (to be consistent with text screen numbering). Coordinates for all screens are specified in the order row, column.

Each graphics screen is capable of displaying four colours at a time. These are either:

	Red	Blue	Green	White
or	Red	Magenta	Yellow	White
or	Magenta	Blue	Cyan	White
or	Yellow	Cyan	Green	White

These colours are selected using the COLOUR and MIX commands.

Finer graphics may be achieved by using screen 5. This combines graphics screens 2 and 4 and gives 204 rows of 480 pixels. Any of the 4 colour sets given above are available. The 480 graphics screen has the same priority as screen 2.

As well as using SET and RESET for displaying particular points on the screen, LINE, DRAW, DRAW@, FILL and FILL@ provide simple means of displaying more complicated graphics.

1.5. SWITCHING ON

The POLY System Operating Manual describes the plugging in and switching on procedures. When the system is connected up, each POLY unit is switched on using the switch on the back of the unit. The magenta start up screen immediately appears.

If the POLY is in System mode (i.e. on the network) at start up, then after typing any key, the POLY operating system, BASIC, and a BASIC program called LOGON.BAC will be loaded into the POLY from the disk in drive 0 of the network controller. LOGON.BAC will then be executed.

If the POLY is in System mode at start up and STANDALONE BASIC appears after typing any key, then enter LOGOFF and wait 5 seconds before attempting to start up again.

If the POLY is in Standalone mode at start up, then after typing any key, the POLY is placed in restricted BASIC programming mode.

1.6. DESCRIPTION CONVENTIONS

In the syntactic descriptions of POLYBASIC

(i)	language entities are underlined
(ii)	reserved words are capitalised
(iii)	optional components are enclosed in square brackets
(iv)	... denotes that the last optional component may be repeated an arbitary number of times.

2. PROGRAMMING IN BASIC

POLYBASIC commands and programs may be entered and run whenever the yellow prompt Ready or the cursor appears. POLYBASIC commands and statements may be entered in either Immediate or Program mode.

2.1. IMMEDIATE MODE

BASIC commands and statements when entered from the keyboard without a line number, are executed immediately the <ENTER> key is pressed.

For example:

```
PRINT 24 * 6
```

prints the answer 144 on the screen.

This statement has been executed in Immediate mode.

2.2. PROGRAM MODE

When a statement is preceded by a line number it becomes part of a program as soon as the <ENTER> key is pressed. This program is stored in the memory of the POLY unit and is not executed until a RUN command is entered.

For example:

```
10 PRINT 24 * 6
```

does not print out the answer until RUN is entered.

A program is executed in Program Mode.

2.3. CONVENTIONS

a. A program line may contain several statements separated by colons (:).

b. The maximum length of any line, including the line number, is 255 characters.

c. Line numbers must be integers between 1 and 65535, and are delimited by blanks.

d. Execution of a POLYBASIC program starts at the lowest line number (unless a specific line number is indicated).

e. When writing programs, line numbers should be allocated in steps of 10 or more, to allow later insertion of new lines. Spacing within lines makes them easier to read.

2.4. THE POLY EDITOR

The POLY system provides a full screen editor which can be used to edit either BASIC or TEXT files.

BASIC mode is available whenever POLYBASIC is loaded. The prompt Ready will appear printed in yellow. The default extension for files is .BAS. In BASIC mode line numbers are part of the file. In BASIC mode, whenever a program is edited, all variable values are reinitialised and any files left open are closed.

TEXT mode is entered from BASIC or DOS via the TEXT command. The prompt Ready is always printed in cyan. The default extension for files is .TXT. In TEXT mode line numbers are not part of the file, they are added to the lines when loading (starting at 10, with intervals of 10) and deleted when saving. In TEXT mode line numbers are used to reference lines for listing, deleting and inserting.

2.4.1. Entering new lines

All new lines are entered with a line number at the start which indicates the position in the file into which the line is to be inserted. If the line number is omitted the line is treated as an immediate command. Entering a line is the act of typing the line and pressing the <ENTER> key.

The cursor may be moved back to an incorrect line and the line corrected. The line is re-inserted into the file on pressing the <ENTER> key. If <ENTER> is not pressed, the line is only stored on the screen and is not updated in memory.

2.4.2. Using the AUTO command

The AUTO command is used to save time when entering new lines, it automatically sets up the line numbers.

Syntax:- AUTO [start-line] [,increment]

The start-line is the first line number at which the automatic numbering will start. If not specified, 10 is used.

The increment is the amount added to each line number to get the next number. If not specified, 10 is assumed.

For example:

```
AUTO
```

starts automatic line numbering at 10 with an increment of 10, i.e. 10 20 30 40 ...

```
AUTO 100, 200
```

starts automatic line numbering at 100 with increments of 200, i.e. 100 300 500 ...

In BASIC mode, the next line number is displayed, as soon as <ENTER> has been pressed for the previous line.

In TEXT mode, the line numbers are not displayed on the screen but are incremented in memory each time <ENTER> is pressed.

To exit from AUTO mode either enter a null line (i.e. just press <ENTER> at the start of a new line) or press <EXIT>.

AUTO will not allow the entering of lines with line numbers the same as those already entered.

2.4.3. Looking at lines already entered

The LIST command displays text already entered, on the screen.

Syntax:- LIST [startline] [-] [endline]

Startline and endline refer to the line numbers as entered. If startline is not specified, then the listing starts at the beginning of the file. If endline is not specified the listing will stop at the end of the file. The <PAUSE> is used to halt the listing at any time. To restart the listings, press any key. If the <SPACEBAR> is pressed following <PAUSE>, then the lines are listed one at a time. If the <EXIT> key is pressed, then the listing is terminated.

If only the startline number is specified then only that line is displayed.

For example:

```
LIST
```

displays the whole file.

```
LIST 100
```

displays only line 100.

```
LIST 100-
```

displays all lines from 100 to the end.

```
LIST -100
```

displays all lines up to 100.

```
LIST 100-200
```

displays lines 100 to 200 inclusive.

2.4.4. Altering lines

To alter a line, list it on the screen using LIST, move the cursor up to the line using the arrow keys, make the alterations necessary, and press <ENTER>.

While changing a line, the <CHAR INS> and the <CHAR DEL> keys may be used for insertion and deletion of characters on that line.

<ENTER> may be pressed when cursor is anywhere on the line, it does not necessarily need to be at the end of the line.

The <LINE INS> and <LINE DEL> keys enable lines to be inserted and deleted on the screen but do not cause changes to the file.

2.4.5. Deletion of lines

A line may be deleted by either:

(i) entering the line number with no data following it, or

(ii) by use of the DEL command.

The DEL command may be used to either delete individual lines or a group of lines from memory.

Syntax:- DEL startline [-endline]

The startline must be given. If the -endline is missing, only the startline is deleted.

For example:

```
DEL 280
```

deletes line 280.

```
DEL 280 - 1000
```

deletes lines 280 to 1000 inclusive.

NOTE that the following forms are NOT allowed:

```
DEL 280-
```

or

```
DEL -1000
```

2.4.6. Renumbering of lines

At times, all available line numbers in a particular sequence may have been used. Alternatively, due to a large number of insertions and deletions the line numbers may be badly distributed. In both these cases, it is advisable to use the RENUM command to renumber the file.

Syntax:- RENUM [startline] [,increment]

Renumbering a BASIC file not only changes the line numbers but also changes all references to them in GOTO, GOSUB and other statements. RENUM may also be used to renumber part of a file (see the description of the RENUM command).

Renumbering a TEXT file only changes the line numbers.

The startline is the first line number allocated. If not given, 10 is used.

The increment is the amount added to each succeeding line number. If not given, 10 is used.

For example:

```
RENUM
```

renumbers the file from line 10, in increments of 10, i.e. the new line numbers are 10, 20, 30, 40 ...

```
RENUM 100
```

renumbers the file from 100 in increments of 10, i.e. the new line numbers are 100, 110, 120, 130 ...

```
RENUM ,100
```

renumbers the file from 10 in increments of 100, i.e. the new line numbers are 10, 110, 210, 310 ...

```
RENUM 1000,100
```

renumbers the file from 1000 in increments of 100, i.e. the new line numbers are 1000, 1100, 1200 ...

2.4.7. Saving the edited file on disk

At any stage during editing, the file may be saved using the SAVE command. A BASIC file is saved with line numbers, a TEXT file is saved without line numbers.

Syntax:- SAVE "filename"
SAVE

The filename may specify the extension and the drive number.

For example:

```
SAVE "0.MYFILE.TXT"
```

If the drive number is not given then the file is written to the current drive for that POLY.

If the extension is not given then a BASIC file is given the extension .BAS and a TEXT file the extension .TXT.

Following a SAVE, the file is still in the POLY memory and further editing may be performed.

For example:

```
SAVE "MYFILE"
```

If the POLY is in TEXT mode and the current drive is 0, then the file will be saved on drive 0 as MYFILE.TXT.

SAVE may be used without a file name if the file has been previously LOADed from disk. In this case the user will be prompted with

```
Save filename (Y/N) ?
```

where filename is the name of the file that was LOADed.

2.4.8. Loading files from disk

A file stored on disk is loaded into POLY memory using the LOAD command. This clears any program or file currently in POLY memory, and loads the file from disk.

Syntax:- LOAD "filename"

The filename may specify the drive number and the extension.

For example:

```
LOAD "1.MYFILE.BAS"
```

will load MYFILE.BAS from the disk in drive 1.

If the drive number is not given, then the file is loaded from the current drive for that POLY.

If the extension is not specified then .BAS is used in BASIC mode and .TXT in TEXT mode.

When a TEXT file is loaded, line numbers are added, starting at 10 and incrementing in steps of 10.

For example:

```
LOAD "MYFILE"
```

If entered on a POLY with the current drive as 1 and in TEXT mode, then the file 1.MYFILE.TXT will be loaded into POLY memory, starting at line 10 and incrementing in steps of 10.

2.4.9. Merging files from disk

The MERGE command merges a file from disk into the file currently being edited. BASIC files are merged on line number such that where the same line exists in both files, the new line replaces the old line.

In TEXT mode, the disk file is appended onto the end of the file being edited and line numbers above those currently in use are allocated.

Syntax:- MERGE "filename"

The filename may specify the drive number and the extension.

If the drive number is not given, then the file is loaded from the disk on the current drive for that POLY.

If the extension is not specified then, for BASIC .BAS is assumed, and for TEXT, .TXT is assumed.

For example:

If a POLY (in BASIC mode) contains the following file:

```
10 CLS
20 FOR row = 0 TO 10
30 PRINT @(row,0) "11Q"
40 NEXT row
```

and the file MYFILE.BAS on disk contains:

```
30 PRINT @(row,0) "  R";
50 REM DRAW A CAR
60 REM etc...
```

then when the command:

```
MERGE "MYFILE"
```

is entered, the resulting file in the POLY will be:

```
10 CLS
20 FOR row = 0 TO 10
30 PRINT @(row,0) "  R";
40 NEXT row
50 REM DRAW A CAR
60 REM etc...
```

If a POLY (in TEXT mode) contains the following file:

```
100 THIS IS A TEXT FILE
200 CONTAINING ONLY
300 3 LINES
```

and the file MYTEXT.TXT contains:

```
THIS IS MYTEXT
FILE WHICH HAS
ONLY 3 LINES
```

then following the command:

```
MERGE "MYTEXT"
```

the POLY file becomes:

```
100 THIS IS A TEXT FILE
200 CONTAINING ONLY
300 3 LINES
310 THIS IS MYTEXT
320 FILE WHICH HAS
330 ONLY 3 LINES
```

2.4.10. Deleting the file being edited

The NEW command deletes the file currently being edited from memory.

For example:

```
NEW
```

If the file being edited has not been changed the Ready prompt will appear on the screen. If the file has been changed since the last SAVE the user will be prompted with

```
Save (Y/N) ?
```

or

```
Save filename (Y/N) ?
```

The filename will appear only if the file was LOADed. In the first case if Y is typed, the NEW is aborted; if N is typed, the NEW is executed. In the second case, if Y is typed the file will be SAVEd and NEW executed; if N is typed, NEW will be executed. Only Y,y,N or n will be accepted.

2.5. COMPILED POLYBASIC

A POLYBASIC program has two forms - source and compiled. These are stored on disk with the file extensions .BAS and .BAC respectively. The compiled form is best used for execution as it is more efficient and being in an encoded form which cannot be listed, provides some form of security.

2.6. STORAGE AND RETRIEVAL OF FILES (Summary)

To SAVE a file on disk enter

```
SAVE "filename"
```

To SAVE a file on disk and create a backup copy of the original file, enter

```
SAVE BACK "filename"
```

The backup file has the same filename with a .BAK extension.

To LOAD a file from disk enter

```
LOAD "filename"
```

This erases any previous file stored in the POLY memory.

To COMPILE a BASIC source program enter

```
COMPILE "filename"
```

The BASIC source program in memory will be compiled and saved on disk.

To RUN a BASIC program, whether it is stored on disk as source or compiled form, enter

```
RUN "filename"
```

.BAC is the default file extension for RUN.

To MERGE a file with the file currently in POLY memory enter

```
MERGE "filename"
```

2.7. VARIABLES

A variable is a "box" into which different values may be placed. Each "box" (or variable) is assigned a name, and the contents may be later referred to or changed by using this name.

For example:

A variable named BOX may have the value 66 placed in it by the statement:

```
LET BOX = 66
```

or simply by:

```
BOX = 66
```

Variable names must begin with a letter (A-Z) and may be followed by other letters or numbers. Variable names may be any length. Variable names containing lower case letters are different from those containing upper case. The following are valid variable names:

```
A, A8, NAME, P1234, address, Sub0
```

Words that form part of the BASIC language are referred to as reserved words (e.g. NEW, IF, TO - these are listed in Appendix 4.9). Such words may be written using both upper and lower case letters.

For example:

LET, Let, let are all representations of the same reserved word. However, as mentioned above, BOX, Box, box are different variable names and thus represent different variables.

Variable names may not, in general, contain reserved words.

For example:

Neither NEWT nor KNEW are valid.
Nor is STOAT, nor STIFF.
Nor BILLET, nor LETTER

To avoid this, the BASIC command RESOFF enables variable names to contain reserved words other than at the beginning of the name. In this mode a variable must be separated by a space from an immediately following reserved word.

By choosing meaningful variable names, programs are more easily understood.

There are three types of variables in POLYBASIC (corresponding to three types of constant) - integer, floating point or real, and string. The first two types are used to store numeric values, the string type is used to store sequences of characters.

2.7.1. Floating Point Variables

Floating point variable names are formed as described in the previous section. These variables are used for holding either numbers containing decimal points, or numbers too large or too small to store as integer variables. The number of digits displayed can be set by the DIGITS command. Unless reset in a program, only 6 digits are printed.

For example:

-0.00000123 is displayed as -1.23E-06

Valid floating point variable names are:

AB, X1, X, amount, Sum, n1549

Valid floating point values are:

-3.2, 79.1, 1E-06, 32768

Note the use of scientific notation for very small and very large numbers.

For example:

-0.0000123 will be displayed as -1.23E-05
17632468 will be displayed as 1.76325E+07

Each floating point variable (and constant) uses 8 bytes of memory.

2.7.2. Integer Variables

Integer variable names are formed using a valid variable name followed by a % sign. Integer variables may only contain whole numbers in the range -32768 to 32767. If an integer variable is assigned a value in the range 32768 to 65535, it is converted to two's complement by subtracting 65536 from it. This allows address values in this range to be handled by integer variables.

Each integer variable uses 2 bytes of memory, so where possible, use integer rather than floating point variables. Use of integer variables will also allow the program to run faster.

For example:

Valid integer variable names are:

A%, X2%, ab%, GROSS%, Tax1982%

Valid integer values are:

27, -201, 32767

2.7.3. String Variables

String variable names are formed using a valid variable name followed by a $ sign. Strings may contain up to 65535 characters, depending on the available memory space. The function FRE(-1) returns the size of the largest string which may be allocated at a given point in a program.

For example:

Valid string variable names are:

```
Ab$, A$, X1$, address$
```

Valid string values are:

```
A$="This is a string literal"
B$=""
C$= CHR$(3)+"Yellow"+CHR$(1)+"Red"
D$="12345 is part of a string"
E$="‖cYellow‖aRed"
```

Each string variable (and constant) uses 4 bytes of memory plus the memory required for the string.

2.7.4. Arrays or Tables

Tables of values may be stored using a single variable name as an array. Individual elements of an array are selected using subscripts (or indexes).

For example:

A table of 5 integers are to be stored in an array called A%. The items in this array are referenced as A%(0), A%(1), A%(2), A%(3) and A%(4).

The index of the first item in any array is always zero. Prior to the use of an array name, the size of the array must be defined using a DIM statement.

For example:

```
DIM A%(4)
```

defines an array containing 5 integer values, A%(0) to A%(4).

Multi-dimensional arrays may also be defined and individual elements referenced using more than one subscript.

For example:

```
DIM BRANCH$(2,4)
```

sets up a two dimensional string array containing 3 rows (rows 0, 1 and 2) and 5 columns (columns 0, 1, 2, 3, and 4).

BRANCH$(1,2) is the string held in row 1, column 2:

	Columns				
	0	1	2	3	4
Row 0	x	x	x	x	x
Row 1	x	x	0	x	x
Row 2	x	x	x	x	x

Arrays may have any number of dimensions.

For example:

```
DIM MULTI(4,4,4,4)
```

defines a 4 four dimensional floating point array.

Note that this would take 5 * 5 * 5 * 5 * 8 = 5000 bytes of memory. Care must therefore be taken to ensure that arrays will fit into available memory.

Variable names within each type of variable must be unique, but the same variable name may be used for different types of variable.

For example:

FINAL, FINAL%, FINAL$, final$, Final and FINAL$(6)

all represent different variables.

2.8. LITERALS AND CONSTANTS

2.8.1. String Literals

Any string of characters enclosed in double quotation marks (or single quotation marks, as long as they match) is a string literal. Single quotation marks may appear in strings delimited by double quotation marks and vice versa. A null string is written as "".

For example:

```
PRINT"ABCDEF1234"
```

displays ABCDEF1234 on the screen.

POLY uses the Teletext conventions for specifying control characters. Within strings, these may be represented by a || followed by the character corresponding to the control character where:

```
||A or ||a = 1
||B or ||b = 2
     .
     .
     .
||Z or ||z = 26
```

For example:

To print a red Hello, the string may be written either as

```
PRINT CHR$(1);"Hello"
```

or as

```
PRINT "||AHello"
```

2.8.2. Numeric Constants

Numeric constants are stored in either integer or floating point form. Integers are stored in 2 bytes, floating point numbers in 8 bytes.

For example:

Valid integer constants are:

```
32767
    2
   -7
```

Valid floating point constants are:

```
  32768    Too big for integer
    1.2    Decimal point
 -3.4E+8   Scientific notation
```

2.9. TYPES OF OPERATORS

2.9.1. Arithmetic Operators

These are:

SYMBOL	MEANING	EXAMPLE	MEANING	ANSWER
+	Add	6 + 2	Add 6 and 2	8
-	Subtract	6 - 2	Subtract 2 from 6	4
*	Multiply	6 * 2	Multiply 6 by 2	12
/	Divide	7 / 2	Divide 7 by 2	3.5
MOD	Remainder (Modulo)	7 MOD 3	The integer remainder when 7 is divided by 3	1
DIV	Integer Divide	7 DIV 3	The integer result of 7/3	2
↑<EXP> key	Exponentiation	6 ↑ 2.1	6 to the power of 2.1	43.064

When an arithmetic expression containing several of the above symbols is evaluated, it is processed in the reverse order to that shown in the list above. That is, exponentiation first, followed by multiplication and division (including MOD and DIV), and addition and subtraction last. Where there is equal priority, an expression is evaluated from left to right.

For example:

```
6 + 4 * 2 - 9 / 2 ↑ 2
    =  6 + 4 * 2 - 9 / 4      (exponentiation evaluated)
    =  6 + 8 - 2.25           (* and / evaluated)
    =      11.75              (+ and - evaluated)
```

Parentheses may be used to alter the order of evaluation. Expressions within parentheses are evaluated first.

For example:

```
(6 + 4) * 2 - (8 / 2) ↑ 2
    =  10 * 2 - 4 ↑ 2
    =      20 - 16
    =         4
```

Provided all values involved do not contain decimal points, and that the result is in the range -32768 to 32767 and is a whole number, then the result will be an integer. Otherwise it will be converted to floating point.

For example:

```
5 / 2 = 2.5
```

gives a floating point result.

Floating point results will be truncated when they are assigned to integer variables

For example:

```
I% = 3.9 will set I% = 3
I% = -3.1 will set I% = -4
```

2.9.2. Relational Operators

These allow the testing of the relationship between a values. The relational operators available in POLYBASIC are:

SYMBOL	MEANING	EXAMPLE	MEANING
=	Equal	X = 4	X is equal to 4
<>	Not equal	X <> 4	X is not equal to 4
<	Less than	X < 4	X is less than 4
>	Greater than	X > 4	X is greater than 4
<=	Less than or equal	X <= 4	X is less than or equal to 4
>=	Greater than or equal	X >= 4	X is greater than or equal to 4

The relational operations are performed after the arithmetical operations.

For example:

```
3 + 2 > 4
```

is TRUE.

See also the section on String Operators.

A TRUE result has the value -1 while a FALSE result has the value 0.

For example:

```
PRINT 8>2
```

prints the result as -1 as it is TRUE.

It is sometimes useful to set up variables TRUE = -1 and FALSE = 0 which may be used throughout the program to improve readability.

2.9.3. Logical Operators

These allow combinations of relationships and are as follows:

SYMBOL	EXAMPLE	MEANING
OR	X = 4 OR X = 10	X is equal to either 4 OR 10
AND	X > 4 AND X < 10	X is greater than 4 AND X is less than 10
NOT	X = 4 AND NOT Y = 3	X is equal to 4 AND Y is NOT equal to 3

Logical operations are performed after the arithmetical and relational operations. Priority of logical operators is NOT, AND, then OR. If there is equal priority the expression is evaluated from left to right.

For example:

```
IF X=1 OR Y=2 AND NOT Z=3 THEN GOTO 2100
IF X=1 OR (Y=2 AND (NOT Z=3)) THEN GOTO 2100
```

are identical in operation.

2.9.4. String Operators

+ may be used to join (or concatenate) two strings together.

For example:

```
B$ = "XXXX"
A$ = B$ + "ABC"
```

After this operation, A$ has the value "XXXXABC".

Relational operators may be used to compare strings.

For example:

```
IF A$ < "M" THEN GOTO 800
```

The < indicates that the first string preceeds the second string in alphabetic order.

The = indicates that the strings are equal. Note that "AB " is equal to "AB".

The > indicates that the first string follows the second string in alphabetic order.

NOTE: The alphabetic order referred to is as shown in Appendix 4.6. Note that the upper case letters precede the lower case letters, and that certain special characters precede the alphabetic characters.

2.10. EXPRESSIONS

Expressions are any valid sequence of constants, variables, functions and operators that yield a value upon evaluation. They may generally be used wherever numbers or strings are expected.

For example:

```
3 * 4 + 5 / 6
```

is an expression yielding a floating point value.

```
A$ + "XXXX" + B$
```

is a string expression.

PRINT@(A% + C%/2, SQR(B%)) A$ + B$ + LEFT$(C$,5)

shows expressions used within the PRINT@ statement.

2.10.1. Functions

Functions providing commonly used routines are specially provided in POLYBASIC to return the appropriate values. They may thus be used within expressions.

For example:

```
X1 = SQR(25) + 2
```

SQR is a function which returns the square root of its argument (or parameter, in this case 25).

The functions available in POLYBASIC are described in Section 3.

Single line functions may be programmer-defined using DEF FN. Such functions may only be used within the program in which they are defined.

2.11. STATEMENTS AND COMMANDS

Statements and commands are instructions to the computer and may generally be used either in Program mode or in Immediate mode.

For example:

```
100 PRINT@(10,5)"This is a program statement."
PRINT@(10,5)"This is an immediate command."
```

Statements form the building blocks of a program. The most common statement is the assignment statement which has already been used in examples. Other statements are usually formed using reserved words and are described in section 3. Commands are more likely to be used in Immediate mode.

For example:

```
AUTO 10,5
```

AUTO may only be used in immediate mode.

2.12. FILE CONSIDERATIONS

File names may be up to 8 characters long. The first character must be alphabetic and the remainder must be alphanumeric. File names may be followed by a "." plus a 3 letter extension. If an extension is not given it will default to:

BASIC source files	.BAS
BASIC compiled files	.BAC
Data files	.DAT
Print files	.PRT
Operating system files	.SYS
Operating system commands	.CMD
Text files	.TXT

If the file is associated with a specific drive then the drive number may be added to the filename either at the beginning or the end.

For example:

0.PROG1.BAS or PROG1.BAS.0

both refer to the file PROG1.BAS on drive 0.

It is possible to protect files by associating a password with them using the PROT command (see the POLYSYS Utilities Manual). For a discussion of file types, see section 3 under the description of the OPEN statement.

2.13. MULTIPLE SCREENS

The POLY computer has 4 screens which are displayed in the following order

1 - TEXT
2 - GRAPHICS (240 pixels across)
3 - TEXT
4 - GRAPHICS (240 pixels across)
The BACKGROUND is displayed at half intensity behind these.

At any time one screen may be selected for writing to. This is initially screen 1 but may be changed at any time by using SELECT which selects the current screen for writing but does not display it. To display a screen, the DISPLAY statement must be used. Any combination of screens may be displayed.

Note the difference between SELECT and DISPLAY - screens may be written to while being displayed or not. The background may be changed at any time using the BACKG statement.

To enable the use of fine graphics a further screen is available.

5 - GRAPHICS (480 pixels across)

Screen 5 utilizes both screens 2 and 4 to enable fine graphics.

Note that the selection of screen 2 will not affect the available memory space but the selection of screen 4 does. Screen 4 requires 8K bytes which is allocated from the user's available memory.

If PRINT commands are used while the current (i.e. selected) screen is a graphics screen, the text is written and displayed on the most recently selected text screen (whether or not it is currently being displayed).

2.14. CHOOSING COLOUR

2.14.1. Text

Colour on the text screens is selected by printing a colour control character before the data.

For example:

```
PRINT CHR$(1);"POLY SYSTEM"
```

or

```
PRINT "A POLY SYSTEM"
```

Section 4.2 contains a complete list of the control characters.

2.14.2. Graphics

2.14.2.1. Colour Choice

Each graphics screen is capable of displaying 4 colours at a time. Initially these are

RED, BLUE, GREEN and WHITE

Other colours may be obtained in either of two ways using MIX:

(a) MIXing screens.

The first option in MIX, allows the colours on screens 2, 3 and 4 to be mixed. To understand this, the way the secondary colours are obtained must be understood. Each dot on the screen is made up of 3 coloured beams, Red, Blue and Green. The colours obtained are:

```
RED                    = RED
      BLUE             = BLUE
               GREEN   = GREEN
RED + BLUE             = MAGENTA
RED          + GREEN   = YELLOW
      BLUE   + GREEN   = CYAN
RED + BLUE   + GREEN   = WHITE
```

When MIX is ON, the beams for each dot on the screen are mixed with those at the same point on the other screens, and the composite displayed. That is, to get YELLOW, Screen 2 must display a particular dot in RED and Screen 4 the same dot in GREEN (or vice versa). COLOUR allows the secondary colours to be selected and when one is selected, Screen 4 is written on (and thus selected) as well as Screen 2.

(b) Additive MIX

Each of the graphics screen may have a particular colour beam added to all dots switched on. This means that instead of RED, BLUE, GREEN and WHITE being displayed, when

RED is added then RED, MAGENTA, YELLOW, and WHITE are displayed,
BLUE is added then MAGENTA, BLUE, CYAN and WHITE are displayed,
GREEN is added then YELLOW, CYAN, GREEN and WHITE are displayed.

2.14.2.2. Colour Size

On each of the graphics screens, a colour code controls the colour of the next 6 pixels each of which may be on or off. Any that are on will thus be the same colour. This is not a severe restriction (in practice multiple screens are extremely useful) and is economical with respect to memory space. When an attempt is made to set a pixel in a set of 6 to a different colour, then all pixels in that group are turned OFF and only the pixel in the new colour displayed. If screen 5 is used then 12 pixels in a row are controlled by the same colour code.

2.15. RESETTING THE POLY UNIT AND WARM STARTS

The POLY unit may be reset to the magenta start up screen by pressing the reset button on the back of the unit.

When programming it is possible to program a loop so that there is no exit. If the <EXIT> key is deactivated (with an ON KEY 10) then the only way to get out is to press the reset button.

By holding down the W key while the reset is pressed, a WARM START back to POLYBASIC is made. Neither the source nor the current variables are lost. A WARM START after a reset or a LOGOFF will (fortunately) restore the most recent program.

Care must be exercised when turning on or resetting (either normal or warm) a POLY that is part of an active network. While the reset button is being depressed and for one second after it is released all communications through a POLY are suspended. They are also suspended for one second after a POLY is turned on. This is to allow stabilisation of the circuitry. During this time, POLYs on the daisy chain that are further away from the disk unit than the POLY being reset or turned on, will be invisible to the disk unit. If such POLYs are performing disk accesses (either read or write) they may occasionally hang up (because they do not receive correct acknowledgement to their requests). If such a hang-up occurs during a disk write, then the disk being written to should be RECOVERed. To prevent this happening

(i) ensure that all POLYs on a network are switched on at the outset

(ii) use LOGOFF rather than reset, and

(iii) if it is necessary to reset, check what others on the network are doing and don't depress the reset button for longer than necessary.

2.16. USING THE CALC KEY

The <CALC> key is activated whenever POLY is waiting for input (i.e. in Immediate mode or when the program currently running requests input). Pressing the <CALC> key at this time will enable calculations to be made on the bottom line of the screen. To return to the program the <CALC> key must be pressed again. To exit while in calculator mode, press <EXIT>.

2.17. ERROR MESSAGES

If an error occurs in a courseware module an error screen will be displayed listing diagnostic information. On pressing <NEXT> the program will CHAIN to the MENU.

If an error occurs in a program which does not have a special error routine, a short error message with an error number and the line number at which the error occurred will be displayed and the program will return to BASIC mode. Variables will not be reset, so their values can be examined.

2.18. USING THE HELP KEY

After an error message has been displayed, pressing the <HELP> key will cause a description of the error to be displayed (provided that the file ERRORS.SYS is on the system disk).

2.19. DOS COMMANDS

Some Disk Operating Commands are available in immediate mode by preceeding the command with a +. The commands available are:

CAT
COPY
DATE
FASTCOPY
FORMAT
KILL
LINK
LIST
PRINT
PROT
~~RDFORMAT~~
SCOPY

These commands are described in full in the POLYSYS Utilities Manual.

For example:

+CAT

+PRINT

Only one of these commands may appear per line, all others are ignored.

The following section describes POLYBASIC functions, statements and commands. The reserved words described in the following may not contain blanks and may be written using either upper or lower case letters. POLYBASIC accepts most standard BASIC commands but some additions and enhancements have been made in order to handle the special features of the POLY System. Functions differ from statements in that functions return values.

3.1. FUNCTIONS

3.1.1. ABS

Syntax:- ABS(numeric-expression)

The absolute value of X is returned, i.e. if X is positive then the value returned is X. If X is negative, the value returned is its positive value.

For example:

If A% has the value 16 then

```
B% = ABS(A%)
```

places the value 16 in B%.

If A% has the value -16, then

```
B% = ABS(A%)
```

also places the value 16 in B%.

The statement

```
IF ABS(X) < 4 THEN GOTO 100
```

will cause a branch to line 100 only if X is between -4 and +4.

3.1.2. ASC

Syntax:- ASC(string-expression)

Returns the ASCII code (in decimal form) of the first character of the specified string. (See Appendix 4.3 for the ASCII values of characters.) If the string is a null string, then 0 is returned.

For example:

```
A% = ASC("A")
```

assigns A% the value 65.

When A$ = "POLY" then

```
A% = ASC(A$)
```

assigns A% the value 80 (the ASCII decimal value of P).

This function is useful for converting between upper and lower case as lower case is simply the upper case ASCII value plus 32. Decoding is also simpler.

For example:

If the values "A", "B", "C" and "D" are expected as input it is easier (and faster) to use

```
ON ASC(A$)-64 GOSUB 150, 200, 300, 450
```

than to test for the individual values separately.

3.1.3. ATN

Syntax:- ATN(numeric-expression)

The angle whose tangent is X is returned. The returned angle value is in radians.
To change an angle in radians to degrees, multiply by 180/PI.

For example:

```
PRINT ATN(1)
```

displays the value .785398 (in radians).

```
PRINT ATN(1) * 180/PI
```

displays the value 45 (in degrees).

3.1.4. CHR$

Syntax:- CHR$(numeric-expression)

CHR$ performs the opposite of the ASC function, creating a single character string containing the character represented by the ASCII value (in decimal form) specified. The numeric expression must have a value between 0 and 255 inclusive. Non-integer values are truncated. Any value outside this range causes an error. This function may be used when setting up Teletext control characters for the screen.

For example:

```
PRINT CHR$(4); "POLY"
```

displays the word POLY on the screen in blue characters. Appendix 4.2 lists the Teletext control characters.

```
100 A$ = INCH$
110 IF A$ <> CHR$(13) THEN 100
```

Line 100 uses the INCH$ function to test the keyboard for a key depression. If any key other than the <ENTER> key (CHR$(13)) has been pressed it is ignored.

In order to abbreviate the insertion of teletext control chracters into strings, they may be written ||x as part of the string. x is 1 character in length, where

```
||a or ||A is CHR$(1)
||b or ||B is CHR$(2)
 etc.
```

The full list is given in Appendix 4.2.

For example:

```
100 PRINT "||aPOLY SYSTEM"
```

writes POLY SYSTEM in red.

3.1.5. CLOCK

Syntax:- CLOCK

CLOCK returns a value in the range -32768 to 32767 indicating 10 millisecond intervals.

For example:

```
100 A% = CLOCK
          .
          .
          .
200 B% = CLOCK
210 IF A% <= B% THEN T = B% - A%
   ELSE T = B% + 65535 - A%
220 PRINT "TIME TAKEN = ";T/100;"SECONDS"
```

3.1.6. COS

Syntax:- COS(numeric-expression)

COS returns the COSine of X where X is in radians. To convert degrees to radians, multiply by PI/180.

For example:

```
X = COS(60 * PI/180)
```

assigns X the value 0.5. i.e. COS (60 degrees) = 0.5

3.1.7. CVT

Syntax:- CVT%$(numeric-expression)
CVT$%(string-expression)
CVTF$(numeric-expression)
CVT$F(string-expression)

The CVT functions store numeric data in strings, and vice versa, using integer or floating point formats. Integer values are stored in 2 bytes while floating point values take 8 bytes. These functions should not be confused with STR$ and VAL.

For example:

CVT%$ moves an integer value into a 2 character string (% to $).

CVT$% moves a 2 character string value back into an integer ($ to %).

CVTF$ moves a floating point number into an 8 character string (F to $).

CVT$F moves an 8 character string value back into a floating point number ($ to F).

If a string is longer than required only the first 2 or 8 characters are taken, depending on whether CVT specifies integer or floating point. These functions allow fast storage of numeric values in strings and are especially useful for the storage of numeric data in records on a file as they pack them into the minimum of disk space.

For example:

```
100 A$ = CVT%$(25665)
```

This stores the 2 byte integer value 25665 in the 2 character string A$. If printed out the string A$ would appear as:

```
dA
```

as the first character would have the ASCII decimal value 100 (25665/256) and the second character the ASCII decimal value 65.

3.1.8. DATE$

Syntax:- DATE$ [(1)]

Returns the current date in either of two forms. The date is set when the first POLY unit logs onto the system.

For example:

If the current date is the 14 January 1983, then

```
100 PRINT DATE$
```

displays the date in the form 14-JAN-83.

```
200 D$ = DATE$(1)
```

assigns to D$, as a 6 character string, the current date in the form YYMMDD. Dates used in this form can be immediately compared numerically.

```
100 A = VAL(DATE$(1))
```

gets the date and converts it to the corresponding numeric value.

3.1.9. DPEEK

Syntax:- DPEEK (address)

DPEEK returns the integer (2 character) value held at the specified address. The address must be between 0 and 65535. While in BASIC, the operating system and screen areas (except screen 4) cannot be accessed using this command.

For example:

```
A%=DPEEK(1165)
```

puts into A% the value held in addresses 1165-1166.

Note that addresses in the range 32768 to 65535 may be placed in integer variables and used, but if printed out will print as the 2's complement, i.e. with 65536 subtracted from them.

3.1.10. ERL

Syntax:- ERL

ERL returns the line number on which a program stops when an error is encountered provided ON ERROR has been set. The line number may be checked within the ON ERROR routine and the appropriate action taken.

For example:

```
5000 IF ERL<3000 OR ERL>3200 THEN RESUME
```

This line would occur within an ON ERROR routine and enables errors occurring between lines 3000 and 3200 to be processed by the following statement.

3.1.11. ERR

Syntax:- ERR

ERR returns the error number after an error has occurred. This is useful in an ON ERROR routine in sending back appropriate error messages. A full list of error numbers is given in Appendix 4.1.

For example:

```
1000 IF ERR<50 THEN RESUME 2200 ELSE PRINT ERL;ERR:END
```

This statement would probably occur within an ON ERROR routine. Errors with numbers less than 50 are sent to a routine at line 2200.

3.1.12. EXP

Syntax:- EXP(numeric-expression)

EXP returns e to the power of X. The maximum value of X allowed is 88. EXP is the inverse of the LOG function i.e. X=EXP(LOG(X)) where LOG(X) is the natural logarithm of X.

For example:

```
Y=EXP(1)
```

sets Y to 2.7182818, the value of e.

3.1.13. FILE$

Availability:- Not available in Standalone mode

Syntax:- FILE$ ("filename")

FILE$ returns a null string if the specified file does not exist. If the specified file does exist FILE$ returns a string of length 12 containing the following information:

DDMMYY - Date of creation (6 bytes)
R or S - Random or sequential
NNNNN - Size of file in sectors

3.1.14. FRE

Syntax:- FRE(0)
FRE(1)
FRE(-1)

FRE(0) returns the number of bytes of free memory. To get the maximum amount available with no program loaded enter

```
NEW
PRINT FRE(0)
```

The memory displayed is the amount available for a BASIC program. When a BASIC program is loaded, it takes up part of this area. As it runs it uses further memory for storing strings and other pointers and may run out of memory during operation. Should this occur it is necessary to reduce the size of the program and/or the amount of memory used in strings, arrays etc. Suggestions to achieve this are:

-Improve your coding.
-When a string is no longer needed reset it to null.
-Do not define oversize arrays.
-Divide your program in two and use CHAIN.
-Compile your program.
-Stop using graphics screen 4.

FRE(1) returns the value of the current high address for memory (See MEM).

FRE(-1) returns the maximum number of characters that a new string may have. This will always be less than the maximum amount of memory left as returned by FRE(0). By checking FRE(-1) it is possible to stop errors occurring due to lack of free space. However, note that FRE(-1) will return different values depending on the context of its call (e.g. if nested in function calls).

3.1.15. HEX

Syntax:- HEX (string-containing-Hex-value)

HEX converts a hexadecimal string into its decimal equivalent.

For example:

```
PRINT HEX("100")
```

displays 256.

3.1.10. INCH$

Syntax:- INCH$ [(channel)]

INCH$ reads a character from the specified channel. If a character has been input then that character is returned, otherwise CHR$(0) is returned.
(NOTE: CHR$(0) is a string containing one null character, it is not the same as "" which is an empty string.)
If no channel number is specified, then the keyboard is scanned to see if a key has been pressed. The most useful way to use this function is to test the keyboard during a loop while some other action is going on.
INCH$ neither displays a cursor nor the character received.

For example:

This example continues looping until any key is pressed.

```
100 IF INCH$ = CHR$(0) THEN 100
200 Continue processing
```

The following example repeatedly tests for characters A,B,C, throwing away any other characters received.

```
100 A$ = INCH$
200 IF A$ < "A" OR A$ > "C" THEN 100
300 ON ASC (A$)-64 GOSUB 500, 600, 700
400 GOTO 100
500 REM "A" SUBROUTINE
550 RETURN
600 REM "B" SUBROUTINE
650 RETURN
700 REM "C" SUBROUTINE
750 RETURN
```

INCH$(0) is a special form of INCH$. It also scans the keyboard, but waits until a key has been pressed before returning a value. The cursor is displayed. It is not necessary to press <ENTER> to terminate the entry.

For example:

```
10 CLS
20 A$ = INCH$(0)
25 PRINT A$;
30 GOTO 20
```

is a program which allows continuous typing and display of the characters typed.

Syntax:- INSTR(start-position, string-name, sub-string)

INSTR looks for a sub-string in a string starting at the start-position. The character position at which the sub-string is found is returned. Zero is returned if the sub-string is not found. The first character position in the string is 1.

For example:

```
10 A$="ABCDEFGHIJ"
20 B%= INSTR(3,A$,"G")
30 PRINT B%
```

A$ is searched for "G" starting at the third character. The value 7 put into B%.

```
40 PRINT INSTR(1,A$, "DEF")
```

would print the value 4.

3.1.18. INT

Syntax:- INT(numeric-expression)

INT truncates a numeric expression to the integer less than or equal to the expression.

For example:

```
INT(9)              returns  9
INT(6.7)            returns  6
INT(-1.1)           returns -2
INT(X*10+.5)/10     returns X accurate to one decimal place.
```

Note: In division, DIV may be used in place of / and INT to give the integer result. Similarly, division performed with integer variables gives an integer result.

For example:

The following all give the same integer result.

```
Z = INT(B/C)
Z% = B/C
Z = B DIV C
```

3.1.19. KVAL

Syntax:- KVAL

KVAL returns the ASCII value of the key pressed to cause transfer to an ON KEY routine. KVAL gives the value of the key pressed. If <A> was pressed KVAL would give the value 65.

For example:

```
10 ON KEY GOTO 1000
     .
     .
     .
1000 IF KVAL=65 THEN END
1010 RESUME
```

If a special key is specified (see ON KEY) then KVAL will return the ON KEY value, NOT the ASCII value.

For example:

```
10 ON KEY 4 GOTO 1000
     .
     .
     .
1000 PRINT KVAL
1010 RESUME
```

prints the value 4 if special key 4 is pressed, not 52 (the ASCII value of special key 4).

3.1.20. LDES$

Availability:- Not available in Standalone mode

Syntax:- LDES$ (line-description)

LDES$ sets up a graphics line description as a string. The line-description consists of a list of coordinates, each pair optionally surrounded by brackets. The coordinates are in row,column order.

For example:

```
100 A$ = LDES$((0,0),(100,0),(100,100),(0,100),(0,0))
```

describes a box 100 pixels square.

This may also be written as:

```
100 A$ = LDES$(0,0,100,0,100,100,0,100,0,0)
```

or

100 A$ = LDES$((0,0)(100,0)(100,100)(0,100)(0,0))

If a ; is used in place of a , between coordinate pairs, then the line is broken at that point and a new line started.

For example:

The following statement stores the line description of the box given above in a single string, such that only the left and right sides are drawn.

```
100 A$ = LDES$((100,0)(100,100);(0,100),(0,0))
```

If the line is to be used as a boundary in FILL or FILL@, the points must be specified in a clockwise order.

Two strings assigned using LDES$ may be added together and then used in a DRAW or FILL. If line is to be continuous, one of the centre end points must be duplicated in the other string.

For example:

```
10 A$=LDES$(0,0,0,5,1,7,2,10,5,12,8,14,10,7,
   11,8,12,4,14,2,9,4,8,6)
20 B$=LDES$(8,6,7,4,5,3)
30 C$=A$+B$
```

Note that the coordinate pair (8,6) is repeated at the beginning of B$.

3.1.21. LEFT$

Syntax:- LEFT$(string, numeric-expression)

LEFT$ returns the first n characters of the given string, where n is the value of the numeric expression.

For example:

```
10 A$="ABCDEFGHIJKLM"
20 X$= LEFT$(A$,5)
```

This puts the value "ABCDE" into X$.

The numeric expression must be between zero and 32767. Non-integer expressions are truncated.

If n>LEN(string) the result is padded on the right with spaces up to length n.

3.1.22. LEN

Syntax:- LEN(stringname)

LEN returns the length of the specified string including spaces and control characters.

For example:

```
10 D$="asd  fghjk"
20 L=LEN(D$)
```

This sets L to the length of D$, i.e. 10.

3.1.23. LOG

Syntax:- LOG(numeric-expression)

LOG is the natural logarithm. To calculate logs for other bases use:

LOG of X to base Y = LOG(X)/LOG(Y)

In particular, for common logs:

LOG of X to base 10 = LOG(X)/LOG(10)

For example:

```
10 PRINT LOG(2)
```

This prints .693147 .

The inverse of LOG(X) is EXP(X) i.e. X= LOG(EXP(X))

3.1.24. MID$

Syntax:- MID$(string,start-position[,numeric-expression])

MID$ returns a sub-string from the given string of length n, where n is the value of the numeric expression.

If the numeric expression is not specified, then all of the string from the start position to the end is returned.

If the start position plus the numeric expression is > LEN(string), the result is padded on the right with spaces up to n.

For example:

```
10 X$="ABCDEFGHI"
20 M1$=MID$(X$,3,5)
30 M2$=MID$(X$,4)
```

gives M1$="CDEFG" and M2$="DEFGHI"

MID$ is also available on the left hand side of assignment statements. The string must be a string variable. The string resulting from the evaluation of the right hand side replaces a string of the same length (equal to the same number of characters if specified) beginning at start-number in the string variable.

For example:

```
10 A$ = "ABCDEFG"
20 MID$(A$,3) = "XY"
30 PRINT A$
```

This prints ABXYEFG

3.1.25. NAME$

Syntax:- NAME$

NAME$ returns the initials entered when the user logged on to the POLY (using the standard LOGON program).

For example:

If the user logged on with the initials LKT then

```
PRINT NAME$
```

will display

```
LKT
```

3.1.26. PEEK

Syntax:- PEEK(address)

PEEK returns the single character value held at the address. The address must be between 0 and 65536. The value returned will be between 0 and 255 inclusive. See DPEEK.

For example:

```
A% = PEEK(1161)
```

This assigns to A% the value of the data stored at address 1161.

3.1.27. PI

Syntax:- PI

PI returns the value of PI (the number of digits depends on DIGITS - for six, PI is 3.14159).

For example:

```
A = PI*R*R
```

The area of a circle is calculated.

3.1.28. POINT

Syntax:- POINT [(]row,column[)]

POINT is used to check to see if a graphics pixel is switched on.

If the current screen is a text screen, then POINT will return 1 if the chunky pixel for the specified coordinates is on, otherwise it will return 0.

If the current screen is a graphics screen, then POINT will return the value corresponding to the colour of the specified pixel if the pixel is on, otherwise it will return a value of 0.

For example:

```
10 SELECT 2
20 COLOUR 2
30 SET (100,200)
     .
     .
     .
90 PRINT POINT (100,200)
```

This would print the value 2.

3.1.29. POS

Syntax:- POS(channel)

POS returns the current column POSition of the cursor on the specified channel. Positioning starts at zero. Channel numbers must not be greater than 12.

For example:

```
10 CLS
20 PRINT TAB(10);
30 X=POS(0)
40 PRINT X
```

Channel 0 is the screen, so this program displays the number 10 starting in column 10 (numbers have preceding and trailing spaces). Note that the ";" is essential otherwise the position returns to 0 on the next line. The POS is calculated from the beginning of the line, or from the start column of a PRINT@ statement.

For example:

```
10 CLS
20 OPEN NEW "EX3" AS 12
30 A$="AAAAAA"
40 PRINT#12,A$;
50 X=POS(12)
60 PRINT X
70 CLOSE 12
```

This OPENs a NEW sequential file, PRINTs A$ into it, then displays the number 6.

POS(-1) returns the current row number of the cursor on the screen.

For example:

```
20 row = POS(-1)
30 column = POS(0)
40 PRINT row, column
```

This prints the position of the cursor on the screen at the start of the program.

3.1.30. PTR

Syntax:- PTR(variable)

PTR returns the memory address of the specified variable. Floating point variable values are stored as 8 bytes and PTR returns the address of the first of these bytes.

Integers are stored as 2 bytes and PTR returns the address of the first of these.

A string is held as a 4 byte pointer and the actual string in a different area. The first 2 bytes of the pointer contain the start address and the last two bytes the length of the string. PTR returns the address of the first byte of the string pointer.

For example:

```
10 A$="ABCDEFGHIJ"
20 A%=PTR(A$)
30 PRINT A%
40 PRINT DPEEK(A%);DPEEK(A%+2)
```

This might display (depending on memory locations)

```
4206
4215  10
```

The 4 byte string descriptor is stored in bytes 4206-4209 . The string itself begins at address 4215 and is 10 bytes long.

3.1.31. RIGHT$

Syntax:- RIGHT$(string,numeric-expression)

RIGHT$ returns the last n characters of the given string, where n is the value of the numeric expression.

For example:

```
10 A$="ABCDEFGH"
20 AA$=RIGHT$(A$,4)
```

The value assigned to AA$ is "EFGH".

If n>LEN(string), then the entire string is returned (unpadded).

3.1.32. RND

Syntax:- RND(numeric-expression)

If the numeric-expression is greater than 0 and less than 1, this function returns a random number in that range. If the numeric-expression is negative then RND returns the same random number sequence (>=0 and <1) each time (for testing purposes). If the numeric expression is greater than or equal to 1, a random number between 1 and the value of the numeric-expression is returned.

For example:

RND(5)	returns values of 1,2,3,4,5 each with 20% probability.
RND(.1)	might return .293104
RND(-1)	might return .630542, .720439,... (for example)

3.1.33. SGN

Syntax:- SGN(numeric-expression)

SGN returns 1 if X is positive, 0 if X is 0, and -1 if X is negative.

For example:

```
10 A=-2 : B=0 : C=14
20 PRINT SGN(A);SGN(B);SGN(C)
```

This will print out the values:

```
-1 0 1
```

3.1.34. SIN

Syntax:- SIN(numeric-expression)

This is the SINe function with X in radians. Multiply degrees by PI/180 to convert to radians.

For example:

```
PRINT SIN(30 * PI/180)
```

will print the value .5

3.1.35. SQR

Syntax:- SQR(positive-numeric-expression)

SQR returns the square root of X. Negative X values will cause an error.

For example:

```
PRINT SQR(9)
```

prints the square root of 9 i.e. 3

3.1.36. STR$

Syntax:- STR$(numeric-expression)
STR$(numeric-expression,string-expression)

STR$ turns a numeric expression into a string, the string is constructed exactly as it would be printed but without a trailing space.

For example:

```
10 A = -.999 : B = 3.678
20 PRINT "X";STR$(A);"X";STR$(B);"X"
```

This prints:

```
X-.999X 3.678X
```

If a string-expression is included, the string is constructed exactly as it would be printed by
PRINT USING string-expression,numeric-expression.

For example:

```
num = STR(1.2395,"+##.##")
```

would assign +1.24 to num.

3.1.37. STRING$

Syntax:- STRING$(numeric-expression [, character])

STRING$ creates a string containing numeric-expression characters. The default character is a blank. The character may be specified either as a string or as its ASCII decimal value.

For example:

```
10 PRINT STRING$(30)
20 PRINT@(10,5)STRING$(30,"*")
30 PRINT STRING$(30,42)
```

Line 10 prints 30 blanks
Line 20 prints 30 asterisks
Line 30 prints 30 asterisks too, 42 is the ASCII decimal code for the asterisk

3.1.38. SWI

Syntax:- SWI (intno[,para1[,para2[,para3]]])

SWI acts identically to USR except that it calls a software interrupt function in the POLY unit system ROM. A full list of these and their parameters is given in the POLYSYS Utilities Manual.

NOTE: For advanced programmers only.

3.1.39. TAB

Syntax:- TAB(numeric-expression)

TAB is used in PRINT or PRINT# statements to move the cursor to the column specified. If the cursor is already in, or past the column specified by numeric-expression then the cursor does not move. TAB may not be used in any other string statements.

For example:

```
5 X=8.9 : Y=222222
10 PRINT TAB(10),X,Y,"RESULTS"
```

This displays:

```
          8.9    222222  RESULTS
```

The 8 is in column 17. The TAB moves the PRINT position to column 10, the comma moves it on to 16, and there is a leading blank before the number.

3.1.40. TAN

Syntax:- TAN(numeric-expression)

This returns the TANgent of the numeric expression which is expressed in radians. To convert degrees to radians multiply by PI/180.

For example:

```
T=TAN(45 * PI/180)
```

will assign the value 1 to the variable T.

3.1.41. TEXT$

Syntax:- TEXT$ (row, column[,length])

TEXT$ returns the block of data on the current text screen starting at the specified row, column of the specified length. If the length is not given, the single character at the current cursor position is returned.

For example:

```
10 A$ = TEXT$(20,0,40)
```

places row 20 of the current screen in A$.

3.1.42. TIME$

Syntax:- TIME$[(1)]

TIME$ returns the current time. If used without a parameter the time is returned in the form:

```
HH:MM:SS
```

where HH is a 24 hour clock. With a parameter the time is returned without the colons.

For example:

```
10 A$ = TIME$ : B$ = TIME$(1)
20 PRINT A$ : PRINT B$
```

would print

```
11:57:33
115733
```

for 57 minutes and 33 seconds after 11 o'clock. In the second form times can be compared numerically.

3.1.43. USR

Syntax:- USR(address [,[para1][,[para2][,[para3]]])
USR(stringvar[,[para1][,[para2][,[para3]]])

USR calls a machine language function stored at the specified address, or stored in the named string. Up to 3 parameters may be handed over to the subroutine. Each parameter, if not specified, defaults to value -1 (hex FFFF). Each parameter may be an integer value in the range -32768 to 65536.

Parameter 1 is placed in the D register.
Parameter 2 is placed in the X register.
Parameter 3 is placed in the Y register.

The return value must be placed in the D register. If the function has to flag an error, the carry flag must be set and the error number returned in the D register. This causes an error to occur in the BASIC.

For example:

```
Z% = USR(A$, 48, B%)
```

calls a subroutine stored in A$ and hands over two parameters, 48 and B%. The return value is placed in Z%.

To place the subroutine in A$ any string operations may be used, such as writing it as DATA statements containing the decimal value of each byte of the subroutine, and READing by "READ I% : A$ = A$ + CHR$(I%)". Or by loading it from a binary file - the first two bytes must be the length.

NOTE: For advanced programmers only.

3.1.44. VAL

Syntax:- VAL(string-expression)

VAL takes a string, evaluates it and returns its numeric value. The string may be any legal BASIC expression, not containing undefined variables.

For example:

```
A = VAL("-2.3")
F$="3.5" : D = VAL(F$)
E$="6*2+3" + "*5" : E = VAL(E$)
```

These lines assign A = -2.3, D = 3.5, E = 27

```
10 A%=6
20 PRINT VAL(2*A%)
```

will print 12, but

```
10 PRINT VAL(2*B%)
```

where B% is undefined will print 2.

Since variable names are not held in compiled programs, the first example if compiled will print 2.

If the string is illegal, VAL evaluates as much as it can from left to right.

For example:

```
B%=VAL("1234G")
C%=VAL("r123")
```

will assign B%=1234, C%=0

3.2.1. AUTO

Availability:- Not Available in Program mode.

Syntax:- AUTO [start-line] [, increment]

AUTO will automatically number lines of a BASIC program or TEXT file being entered. Line numbering will start at the first parameter value and proceed in increments as given by the second parameter. The default value for both parameters is 10. In BASIC mode, the line numbers will be printed automatically. In TEXT mode they are not printed. AUTO will not allow lines to be entered which will replace existing lines. To exit from AUTO mode, enter a null line.

For example:

```
AUTO
AUTO 30,5
AUTO ,5
AUTO 20
```

3.2.2. BACKG

Syntax:- BACKG numeric-expression

BACKG sets the half intensity background ON or OFF. The numeric expression specifies the colour where:

```
0 = off
1 = red
2 = green
3 = yellow
4 = blue
5 = magenta
6 = cyan
7 = white
```

For example:

```
BACKG 4
```

sets the background to blue.

3.2.3. BASIC

Syntax:- BASIC

BASIC is used to return the user from TEXT mode to BASIC (see TEXT). An option to SAVE a loaded file is given (see the NEW statement).

3.2.4. CHAIN

Availability:- Not available in Standalone mode

Syntax:- CHAIN string-expression[,expression-list]

CHAIN terminates the current program and loads and runs the program designated in the string-expression. The chained program may be either a BASIC or machine language program. The displayed screens are not switched off but the current screen becomes screen 1. All variables except those specified in the expression-list are lost. FETCH may be used to fetch these values into the chained program.

If no filename extension is specified, .BAC is assumed. Source (.BAS) files may also be chained, but the .BAS extension must be specified.

For example:

```
90 A$ ="FILEA"
100 CHAIN A$
```

This chains to program FILEA.BAC. All variables in the current program are lost.

```
1200 CHAIN "NEXTP.BAS", A$, B%, AX*B, D$(4)
```

This chains to program NEXTP.BAS saving the values of the specified expressions.

The program NEXTP.BAS must have a first statement in the form

```
10 FETCH A$, A%, A, A1$
```

to fetch the saved values, where, although variable names need not match, the types must.

3.2.5. CLEAR

Syntax:- CLEAR

CLEAR sets the current screen to 1, and displays it. All other screens are turned OFF, current colour is set to white and MIX mode is set OFF. The screen SPLIT is 24 and scrolling is turned ON. The background is turned OFF. All variables and dimensioned arrays are cleared. All returns are cleared. Hence do not use CLEAR within FOR...NEXT loops, subroutines or in ON ERROR, ON SEC or ON KEY routines as the return line number will be cleared. All open files are closed. It is possible that after an error has occurred (for example, disk full) that files will remain open. In order to close these, issue a CLEAR. CLEAR should also be used if the error "Illegal file Control block specified" appears. This message normally follows a user error when files are accidentally left open. CLEAR ensures that these are closed. It is not done automatically because when debugging, users often wish to

interrogate variables, etc.

Sometimes users wish to load a program from disk into the POLY memory, disconnect the POLY and then run the program (which clearly cannot use disk files). If this is to be carried out, issue a CLEAR command prior to disconnecting the POLY.

3.2.6. CLOAD

Syntax:- CLOAD

CLOAD causes a BASIC program or a text file to be loaded from tape through the optional cassette interface.

A prompt to start the tape is issued, then the tape is searched for the beginning of a file, the header string (see CSAVE) is printed, and the BASIC program or text file is loaded into memory.

CLOAD will only load BASIC programs and text files that have been saved using CSAVE.

During loading, an asterisk is printed every 256 characters received to indicate that loading is taking place.

On completion of loading, Ready is displayed. CLOAD may be terminated at any stage using EXIT. If the tape is not read correctly (error checking is implemented), the message READ ERROR will be displayed. If this occurs try again. If it still occurs, check the volume setting of the tape recorder and/or clean the tape heads.

It is possible to load RAM-based POLYBASIC extensions via cassette recorder. These generally reside on disk (in the BASIC.CMD and POLYSYS.SYS files) but are available on special cassettes obtainable from POLYCORP. Thus full POLYBASIC (except disk-referencing commands) may be used from standalone mode. Using disk-referencing commands will cause the POLY to await communication with the disk drive and you will have to reset.

3.2.7. CLOSE

Availability:- Not available in Standalone mode

Syntax:- CLOSE [KILL] [#]channel [,[#]channel]

CLOSE terminates input/output between BASIC and a file and makes the channel available for another file. The channel must have an integer value between 1 and 12, and must match a channel number specified in an OPEN statement. An error occurs if the specified channel is not OPEN.

For example:

```
100 CLOSE 3
```

CLOSEs the file which was OPEN on channel 3.

Files should be OPEN for as short a time as possible, as they maintain a buffer area in memory which is returned to the work area on CLOSING the file.

The KILL option, if present, causes the file to be removed from disk. If a file with the same name, but with the extension .BAK exists on the disk then it will be renamed with an extension the same as the file being closed. # is optional before the channel number.

For example:

```
100 CLOSE KILL 2
```

The file which was open on channel 2 is closed and deleted.

If a new serial file is opened but never written to, it will automatically be deleted on CLOSE (or END).

3.2.8. CLS

Syntax:- CLS

CLS clears the current screen, and puts the cursor in position (0,0).

For example:

```
90 SELECT 1
100 CLS
110 SELECT 2
120 CLS
```

This clears both screens 1 and 2.

3.2.9. COLOR

Syntax:- COLOR colour

See COLOUR.

3.2.10. COLOUR

Syntax:- COLOUR colour

COLOUR selects the current colour for SET, FILL, LINE and DRAW on the graphics screens. The colour codes are:

```
0 = BLACK
1 = RED
2 = GREEN
3 = YELLOW
4 = BLUE
5 = MAGENTA
6 = CYAN
7 = WHITE
```

For example:

```
100 COLOUR 2
```

selects the current colour to be green.

If no colour has been selected then WHITE is the default.

The secondary colours CYAN, MAGENTA and YELLOW may be obtained only by using MIX in either of two ways as follows:

1. By combining the colours on the two graphics screens. In this case, the drawing is put onto both screens 2 and 4 with one colour on 2 and the other on 4. The colours are then MIXed to give the secondary colour. In order to see the secondary colour, both screens 2 and 4 must be DISPLAYed, and MIX must be ON. (Remember screen 4 requires 8K of user memory.)

For example:

```
70 DISPLAY 2
80 DISPLAY 4
90 MIX ON
100 COLOUR 3
110 SELECT 2
120 SET 100,220
```

This sets point (100,220) on screen 2 in red and on screen 4 in green. The MIX command mixes the 2 colours to give yellow (colour 3). For yellow the red is set on the current screen and green on the other graphics screen

When secondary colours are selected, the colour displayed is formed as follows:

	Current Graphics Screen	Other Graphics Screen
Yellow	Red	Green
Magenta	Blue	Red
Cyan	Green	Blue

2. To each of the graphics screens, a colour may be added to give a further set of colours. This is achieved by using MIX in the second form.
The colour sets available are:

No colour added	Red	Green	Blue	White
Red added	Red	Yellow	Magenta	White
Green added	Yellow	Green	Cyan	White
Blue added	Magenta	Cyan	Blue	White

When using MIX in this form the primary colour is specified by COLOUR and the overlay colour to be added, is specified in MIX.

For example:

```
70 DISPLAY 2
80 SELECT 2
90 COLOUR 1
100 MIX 2,2
110 SET (100,210)
```

This sets the point (100,210) on screen 2 in yellow.

Remember that the primary colours are:-

```
RED  GREEN  BLUE

YELLOW  = RED  + GREEN
MAGENTA = RED  + BLUE
CYAN    = BLUE + GREEN
WHITE   = RED  + BLUE  + GREEN
```

3.2.11. COMPILE

Availability:- Not available in Standalone mode

Syntax:- COMPILE "filename"

COMPILE saves the program currently in memory on disk in compiled form. The filename should be specified in quotes. If no filename extension is specified, .BAC is assumed. A compiled program may be loaded later either with a RUN, LOAD or CHAIN. A compiled program generally takes less storage than a source file and also loads faster. A compiled program cannot be listed.

For example:

```
COMPILE "NEWPRM"
```

This compiles the BASIC program currently in memory onto the current default disk as NEWPRM.BAC.

3.2.12. CONT

Syntax:- CONT

CONT is used to restart a program after it has been stopped by a STOP statement, by pressing the <EXIT> key, or by an error. The restart is made at the statement following STOP, at the next statement after <EXIT>, or at the start of the statement causing the error.

CONT will not restart the program if any editing is performed while the program is stopped.

3.2.13. CONVERT

Syntax:- CONVERT string-variable1 TO string-variable2

CONVERT changes a line description stored in a string into a boundary description (see FILL). FILL converts a line description to a boundary description each time a FILL is performed, but for efficiency a line description may be CONVERTed prior to the use of FILL.

For example:

```
100 CONVERT A$ TO A$
110 CONVERT X$ TO Z$
```

If line descriptions have been set up in A$ and X$ then line 100 converts A$ to a boundary description and line 110 converts X$ to a boundary description storing it as Z$.

3.2.14. CSAVE

Syntax:- CSAVE string-expression

CSAVE saves the BASIC program or the text file, depending on what mode the POLY is in, currently in memory to tape via the optional cassette interface.

A prompt to start the tape is issued. CSAVE then writes out a special tape mark, followed by a heading which is the value of the string-expression, then the program or text file currently stored in memory.

Files saved using CSAVE may be re-loaded using CLOAD.

On completion of saving, the cursor will be displayed. CSAVE may be terminated at any stage using EXIT.

3.2.15. DATA

Syntax:- DATA value [,value]...

The DATA statement enables data to be stored inside a program and to be accessed by a READ statement when it is required. Each item in the list must be separated by a comma. Each time a READ is encountered, the next item in the list is read.

Strings do not need to be enclosed in quotes unless they include embedded commas or colons. Strings and numeric data may be mixed in the same DATA statement.

Multiple DATA statements may be used and need not be placed together in the program. When all the data has been read from one DATA statement, access will be made to the next DATA statement. A DATA statement must be the only statement on a line.

RESTORE may be used to set the line number of the DATA statement from which the next READ is to access.

For example:

```
 50 DIM Array$(3)
100 DATA JAN, FEB, MAR, APR, MAY
200 DATA "  100", " 1000", "10,000"
300 DATA 50,40,30,20
400 RESTORE 200
500 FOR I% = 1 TO 3
600 READ Array$(I%)
700 NEXT I%
```

This sets up Array$ with the 3 string values of 100, 1000 and 10,000.

3.2.16. DEF FN

Syntax:- DEF FN variable-name (dummy-variable) = expression

DEF FN allows the user-definition of single line functions containing one argument and returning a floating point value.

The function name is formed by preceding the variable name in the definition by FN.

For example:

```
DEF FNA(B) = SQR(B) / 2 + 10
```

defines the function FNA which takes the square root of the argument, divides it by 2 and adds 10.

FNA could then be used in an expression such as:

```
A = FNA(16) + FNA(36)
```

which would give A the value 12 + 13 = 25.

In the example, B is a dummy variable that may be used within the expression. Its value is determined when the function is called. The returned value will be floating point. String functions cannot be included within the expression.

NOTE: DEF FN must be the last statement in a line.

3.2.17. DEL

Availability:- Not available in Program mode.

Syntax:- DEL line-range [, line-range]

DEL deletes lines as specified by the parameters. A line range may be a single line, or a starting line number followed by a hyphen, followed by an ending line number specifying a group of consecutive lines.

For example:

```
DEL 10, 30-100
```

Deletes line number 10, and all lines between 30 and 100 inclusive.

3.2.18. DIGITS

Syntax:- DIGITS total-number [,number-of-decimal-places]

DIGITS specifies how many digits to print in standard numerical notation. The number must be in the range 1-10. If no number is specified, the default prints a total of 6 digits. The second argument specifies the maximum number of decimal places, and must be less than or equal to the total number. The default second parameter is equal to the first parameter. If a number will not fit the format it is printed in scientific notation. If DIGITS is not specified a number with more than 6 digits before or after the decimal point will be printed in scientific notation.

For example:

```
10 DIGITS 5,4
20 PRINT 2.7182818285
```

will print 2.7183

```
10 DIGITS 2
20 PRINT 999
```

will print 1E+03 because of rounding.

3.2.19. DIM

Syntax:- DIM variable(size) [,variable(size)]...

DIM sets the number of items allowed in each of the dimensions in an array. All arrays must be dimensioned before they can be used. There is no limit to the number of dimensions.

For example:

```
100 DIM A$(10), B%(4,2)
200 DIM C$(2,2,2,2,2)
```

Line 100 sets up a one dimensional string array with subscript elements 0 - 10, (i.e. 11 elements) and a two dimensional integer array with elements 0,0 to 4,2.

Line 200 sets up a 5 dimensional string array with elements 0 to 2 in each dimension.

Arrays cannot be redimensioned within a program.

3.2.20. DISPLAY

Syntax:- DISPLAY screen-number [ON]
DISPLAY screen-number [OFF]

DISPLAY is used to turn screens ON and OFF. The screen numbers are

1 - Text screen
2 - Graphics screen (240 by 204)
3 - Text screen
4 - Graphics screen (240 by 204)
5 - Fine graphics screen (480 by 204)

For example:

```
DISPLAY 1
```

displays the top text screen.

```
DISPLAY 4 OFF
```

turns off the second graphics screen.

0 may be used to represent all screens.

For example:

```
DISPLAY 0
DISPLAY 0 OFF
```

displays and turns off all screens.

Following a RUN, NEW, CHAIN or CLEAR, all screens except screen 1 are turned OFF but not cleared.

Screen 5 cannot be used as well as screen 2 or 4 since it is formed by combining both of these graphics screens.

Screen 5 has the same priority as screen 2.

3.2.21. DOS

Availability:- Not available in Standalone mode

Syntax:- DOS

This places the POLY unit in the Disk Operating System (DOS) mode. A description of operation in this mode, along with a description of the full set of DOS utilities is contained in the POLYSYS Utilities Manual. An option to SAVE a loaded file is given (see NEW statement).

To return to POLYBASIC from DOS enter BASIC.

3.2.22. DPOKE

Syntax:- DPOKE address,value

DPOKE puts the 2 byte integer value at the specified address. The value must be in the range 0 to 65535.

For example:

```
DPOKE 1165, 2456
```

puts into memory locations 1165-1166 the value 2456.

Care must be taken in using this statement to ensure that the values changed are at locations where the meaning of the contents are known.

3.2.23. DRAW

Availablility:- Not available in Standalone mode

Syntax:- DRAW string-variable
DRAW # channel
DRAW # "file-name"

DRAW places onto the current screen the line whose coordinates have been defined using LDES$ and put in a string variable or onto a file, or a screen dump which has been stored in a variable or file using STORE.

A line is drawn in the current colour, on the current screen, at the place it was defined.

For a screen dump, the current screen must be a graphics screen. The screen dump is placed in the same place on the screen from which it was stored in the original colours.

For example:

```
10 CLS
100 Car$ = LDES$(115,0,115,20,100,20,100,
    70,130,70,130,0,115,0)
110 SELECT 2:CLS:DISPLAY 2
120 DRAW Car$
```

This draws the car as specified on screen 2.

```
130 STORE (100,0)(130,70)Car$
135 CLS
140 SELECT 4 : DISPLAY 4
150 DRAW Car$
```

This stores the car as a screen dump in Car$, clears screen 2 and dumps it onto screen 4 in the same place.

If a screen dump is stored on disk using STORE# the file (which may contain several screen dumps) may be opened and the screen dumps drawn using DRAW#.

For example:

```
100 OPEN OLD "DIAGRAMS" AS 1
110 CLS : SELECT 2 : CLS : DISPLAY 2
120 FOR Number = 1 TO 5
130 DRAW #1
140 NEXT
150 CLOSE 1
```

This reads 5 diagrams off the disk file DIAGRAMS.DAT and displays them on screen 2.

If there is only one screen dump in the file then DRAW#"filename" may be used to avoid the OPEN and CLOSE statements.

3.2.24. DRAW@

Availability:- Not available in Standalone mode

Syntax:- DRAW@(row, column) string-variable
DRAW@(row, column) #channel
DRAW@(row, column) #"file-name"

DRAW@ shifts the coordinates relative to the specified row, column and DRAWs the diagram in the new position.

For example:

```
50 CLS
100 Car$ = LDES$(115,0,115,20,100,20,100,
    70,130,70,130,0,115,0)
110 SELECT 2 : CLS : DISPLAY 2 : COLOUR 1
120 FOR Col = 240 TO -2 STEP -10
130 CLS
140 DRAW@(115,Col)Car$
150 NEXT Col
```

This causes the car to move from right to left across the screen.

3.2.25. DRIVE

Availability:- Not available in Standalone mode

Syntax:- DRIVE drive-number

DRIVE reassigns the current default drive for a particular POLY unit. The drive number should be in the range 0 to 4, where drives 0 to 3 are physical disk drives and 4 refers to RAMdisk.

For example:

```
DRIVE 1
```

This allows a particular POLY unit to be working on a different drive from other units using the system.

3.2.26. ELSE

Syntax:- IF condition [THEN] stmt [:stmt]...ELSE stmt [:stmt]...

ELSE must always be used in conjunction with an IF statement. All statement(s) on the same line following the ELSE are performed if the condition is FALSE. The ELSE must not be immediately preceded by a colon.

A line number immediately following ELSE is read as ELSE GOTO linenumber.

For example:

The following are valid constructs:

```
100 IF A%>3 OR B%<2 THEN GOTO 200 ELSE GOTO 300

200 IF A=4 THEN B=0 :C=0 ELSE D=0 :E=0

300 IFC<X C=X:D=OELSEIFC<YTHENC=Y:D=1ELSEC=Z:D=Z

400 IF A[%=4 PRINT A% ELSE 2000
```

The latter two examples are poor. THEN should always be used as it improves readability.

The following are NOT valid constructs:

```
500 IF A%>4 GOTO 1000 : ELSE GOTO 2000

600 IF A%=4 AND B%=6 ELSE C%=0
```

3.2.27. END

Syntax:- END

END terminates program execution. It may be placed anywhere in the program. If the last statement to be executed is also the physically last statement in a program, END need not be specified.

For example:

```
100 IF A% > 100 THEN END
```

Following END, a program cannot be restarted by using CONT, but may be RUN again. The end of a program causes the display of all screens except screen 1 to be switched off.

3.2.28. ERROR

Syntax:- ERROR numeric-expression

ERROR allows an error to be simulated during testing. ERR and ERL are set up as if the error had occurred and the ON ERROR branch is taken as if it has been set.

For example:

```
ERROR 4
```

makes the program act as if error 4 had occurred.

In the error routine, RESUME NEXT rather than RESUME must be specified as RESUME will just cause repetition of the ERROR statement.

3.2.29. EXEC

Availability:- Not available in Standalone mode

Syntax:- EXEC "filename"

EXEC loads and executes the machine language program contained in filename. Care must be taken to ensure that the program loaded does not interfere with the BASIC program running. Utilities which run in the utility command space may be executed with this command. Following execution of the program, control is returned to the BASIC program.

For example:

```
100 EXEC "CAT 1"
```

This displays the catalogue on the screen.

A complete description of utilities is given in the POLYSYS Utilities Manual.

3.2.30. FETCH

Availability:- Not available in Standalone mode

Syntax:- FETCH variable1 [,variable2]...

When a BASIC program is run from another program by the use of CHAIN, all variable values are reset. However expressions may be passed to the program being CHAINed using the FETCH statement. The expressions in the CHAIN list and the variables in the FETCH list must match by type. FETCH must be the first executable statement in the CHAINed program.

For example:

```
CHAIN "NEXTPROG",Address$,NO%
```

might CHAIN to a program in which the first statement is:

```
FETCH Address$,NUMBER%
```

3.2.31. FIELD

Availability:- Not available in Standalone mode

Syntax:- FIELD #channel, fieldsize AS stringname1
[, fieldsize AS stringname2]...

FIELD is used with random access file statements PUT and GET, to specify string subfields within the record buffer.

For example:

```
10 OPEN OLD RANDOM "TRIAL" AS 1
20 FIELD#1, 10 AS A$, 4 AS X$, 8 AS D$
30 GET #1
```

Line 10 opens the file TRIAL on channel 1.
Line 20 assigns A$ as the label for the first 10 bytes (characters) in the record buffer of channel #1, whatever these may contain, X$ as the next 4 characters, and D$ as the last 8 characters.
Line 30 reads the next record of the file into the record buffer, which will assign values to A$,X$ and D$.

After GETting a record, a FIELD statement may be used to reassign the fields within the record. If you want to retain the value of a variable which is in the buffer you must store it in another variable before reassigning the fields.

For example:

```
40 FIELD#1, 23 AS Z$, 10 AS A$, 4 AS X$, 8 AS D$
```

reassigns the labels Z$, A$, X$ and D$ to the buffer, while the data in the buffer is unchanged.

If it is required to store numeric fields in the record, the CVT function can be used to convert them.

For example:

```
5 DIM A%(125)
10 OPEN OLD RANDOM "TRIAL1" AS 9
20 GET #9
30 FOR I=0 to 125
40 FIELD #9, I*2 AS X$, 2 AS A$
50 A%(I)=CVT$%(A$)
100 NEXT I
```

This program GETs a single record from TRIAL1.DAT into the record buffer and breaks it into 126 2 character fields. These are then stored as integers in the array A%. X$ is a dummy string to put successive A$ labels at the right places. Further processing can then be done with the values stored in the array.

To store records in a random access file, FIELD, LSET, RSET and PUT are used.

For example:

```
10 OPEN NEW RANDOM(50) "TRIAL2" AS 12
20 FIELD #12,20 AS A$, 30 AS B$
30 LSET A$="AAA"
40 LSET B$="BBB"
50 PUT #12, RECORD 3
60 CLOSE 12
```

Lines 30 and 40 assign the strings "AAA + 17 spaces" and "BBB + 27 spaces" into A$ and B$ respectively. The FIELD statement in line 20 assigns these to the first 50 bytes of the buffer. Line 50 PUTs the contents of the buffer into RECORD 3 of the new file TRIAL2.DAT.

It is possible to use FIELD and GET (but not PUT) with sequential files in a similar manner provided no RECORD option is specified for the GET.

3.2.32. FILL

Availability:- Not available in Standalone mode

Syntax:- FILL [USING pixels[,shift];] string
FILL [USING pixels[,shift];] #channel
FILL [USING pixels[,shift];] #"file-name"

FILL fills a defined area on a graphics screen with a specified pattern in the current colour. The string (whether from a disk file or not), may be a boundary description or a line description (see CONVERT). When using LDES$ to create a line description to be FILLed, the points must be described in a clockwise direction. The pattern is described by two values, the pixels, and the shift. The pixels are set up in groups of 6 which are represented by the decimal value of their binary pattern. If the pixels are not specified, 63 is used. If the shift is not specified no shift is assumed.

For example:

If the pattern required is 101010 then the equivalent decimal value 42 is used. The easiest way to convert from binary to decimal is by assigning values to each digit and adding them up.

```
32 16  8  4  2  1
-----------------
 1  0  1  0  1  0 = 42
 0  1  1  1  0  1 = 29
 0  0  0  0  0  1 = 1
```

Shift is the number of pixels each row is shifted to the right.

For example:

With pixels 000001 a shift of 1 gives the following patterns on successive rows:

```
Row1 000001
Row2 100000
Row3 010000
Row4 001000
Row5 000100
```

A diagonal effect is obtained.

```
10 SELECT 2
20 CLS
30 DISPLAY 2
40 A$=LDES$((2,0)(2,100)(50,100)(50,10)(2,0))
50 COLOUR 4
60 FILL USING 1,1;A$
```

This selects screen 2, clears and displays it and fills the rectangle defined with blue diagonal lines.

The FILL is executed in two stages, the line description is first converted to a boundary description and then the area is filled. If the same line description is to be filled repeatedly, a time saving may be made by converting the line to a boundary description using CONVERT, and storing this in a string.

For example:

```
10 A$ = LDES$(2,0,2,100,50,100,50,10,2,0)
20 CONVERT A$ TO A$
30 SELECT 2 : CLS : DISPLAY 2
40 FOR I% = 1 TO 100
50 Col = RND(4) : IF Col = 3 then Col = 7
60 COLOUR Col
70 FILL A$
80 NEXT I%
```

This fills the box with a solid random colour 100 times.

3.2.33. FILL@

Availability:- Not available in Standalone mode

Syntax:- FILL@(row,column) [USING pixels[,shift];]] string
FILL@(row,column) [USING pixels[,shift];]] #channel
FILL@(row,column) [USING pixels[,shift];]] #"file-name"

This transposes the area to be filled to start at the row, column specified, operating similiarly to DRAW@.

For example:

```
100 FILL@(10,30) USING 21,1;A$
```

If A$ is as specified in the FILL example, then all points are moved by the difference between (10,30) and (2,0). That is the rows have 8 added to them and the columns 30.

(10,30)(10,130)(58,130)(58,80)(10,30)

This area is then filled.

3.2.34. FOR

Syntax:- FOR variable = start-value TO end-value [STEP increment]

FOR starts a loop so that a sequence of program statements can be executed over and over again. The loop must be terminated by a NEXT statement. The specification of the variable in the NEXT statement is optional.

For example:

```
10 FOR I% = 1 TO 6
20 PRINT I%;
30 NEXT I%
```

is the same as writing

```
10 PRINT 1;
20 PRINT 2;
30 PRINT 3;
40 PRINT 4;
50 PRINT 5;
```

```
60 PRINT 6;
```

The start and end values may be either constants, variables or expressions. The variable in the FOR statement is to count the number of times the loop is executed and care must be taken if this is changed within the loop.

For example:

```
100 FOR N%=1 TO 10
110 N%=N%*2
120 NEXT
```

Lines like 110 which change the value of the control variable should be used with caution.

STEP specifies the amount to be added to the counter variable at the end of each loop. It may be positive or negative and may contain a decimal point. If not specified, 1 is assumed. At the end of each loop, the counter is tested. If the value obtained on incrementing the counter by the step size is greater than the final value the program goes on and executes the statement after the NEXT. (If the STEP is negative, the test is less than the final value). The value of the counter after the NEXT statement is the latest value not to have failed the test.

A FOR loop may be exited by a GO TO from within the loop but a GO TO should never reenter the middle of a FOR loop (unless it has previously exited using a GO TO).

FOR loops may be nested, i.e. placed one inside another.

For example:

```
5 X=0.4
10 FOR I=1 TO X*7 STEP 0.5
20 PRINT "OUTER LOOP"; I
30 FOR J=1 TO 2
35 PRINT "INSIDE LOOP"
40 NEXT J
50 NEXT I
```

This would print

```
OUTER LOOP 1
INSIDE LOOP
INSIDE LOOP
OUTER LOOP 1.5
INSIDE LOOP
INSIDE LOOP
OUTER LOOP 2
INSIDE LOOP
INSIDE LOOP
OUTER LOOP 2.5
INSIDE LOOP
INSIDE LOOP
```

Using integer values for variables saves time and memory space. A FOR loop is always executed at least once.

3.2.35. GET#

Availability:- Not available in Standalone mode

Syntax:- GET #channel [,RECORD record-number]

GET reads a specific record from a random file and places it in the record buffer for that channel. To access the data in that record, a FIELD statement must be used. If a record number is specified, that record is read from the file. If it is not specified, then the next record on the file is read.

For example:

```
10 OPEN OLD RANDOM "TEST" AS 4
20 GET #4, RECORD 24
30 FIELD #4, 100 AS A$, 10 AS B$
40 CLOSE 4
```

Line 10 OPENs TEST.DAT on channel 4.
Line 20 reads record 24 into the buffer.
Line 30 designates the first 100 bytes of the buffer as A$, the next 10 as B$.
Line 40 CLOSEs the file.

See FIELD for further examples. GET may be used with sequential files provided the RECORD option is not specified.

3.2.36. GOSUB

Syntax:- GOSUB linenumber
GOSUB expression

GOSUB transfers program control to the subroutine beginning at the specified line number. When a RETURN statement is encountered, control is returned to the statement following the GOSUB. When an expression is used it must not start with an integer value e.g. GOSUB 20*I will be interpreted as GOSUB 20 followed by *I. However GOSUB I*20 is valid.

For example:

```
150 IF I%=1 THEN GOSUB 1000
160 PRINT "RETURN FROM SUBROUTINE"
170 END
1000 PRINT "SUBROUTINE"
1010 RETURN
```

With I% set to 1 this would display

SUBROUTINE
RETURN FROM SUBROUTINE

A subroutine may be used over and over again from various places in a program.

3.2.37. GOTO

Syntax: GOTO linenumber
GOTO expression

GOTO may also be written as two words i.e. GO TO. GOTO transfers program control to the specified line number. The line number may be given as an expression which will be evaluated. When an expression is used it must not start with an integer value e.g. GO TO 20*I will be interpreted as GO TO 20 followed by *I. However GO TO I*20 is valid.

For example:

```
100 PRINT "A";
120 GOTO 100
```

This would result in a continuous loop with the computer displaying A's. To stop the program it would be necessary to press the <EXIT> key.

```
100 PRINT "A";
110 GO TO 1000
     .
     .
     .
1000 PRINT "B"
```

This simply transfers control to line 1000 and would display AB, because of the ";". Lines between 110 and 1000 are irrelevant to this example.

3.2.38. IF THEN

Syntax:- IF condition [THEN] stmt [:stmt] ...

If the result of evaluating the condition is TRUE, the statement(s) immediately following, or following the THEN on the same line are executed. If the expression is FALSE, control jumps to the matching ELSE statement (if there is one) or to the next line. See the ELSE statement.

A line number immediately after the THEN statement is interpreted as THEN GOTO linenumber. If THEN is omitted and an assignment (e.g. A=24) appears following the condition, a space must follow the condition.

For example:

```
100 IF A%<4 AND B%=2 THEN GOSUB 1000
200 IF X>127 THEN B%=3 : C% = 0 : GOTO 300
210 IF Z>256 INPUT A$
220 IF X%=1 THEN X%=9999
```

The following lines all perform the same action

```
300 IF A$ = "END" THEN GOTO 2000
400 IF A$ = "END" GOTO 2000
500 IF A$ = "END" THEN 2000
```

However it is good practice to always include THEN.

3.2.39. INPUT

Syntax: INPUT ["message";] variable [,variable]...

INPUT causes the program to wait until the specified number of fields are entered on the keyboard. The input statement may specify a list of string or numeric variables to be input. The items in the list must be separated by commas. When typing the strings, fields need not be enclosed in quotes unless leading or trailing spaces are to be included or the string includes a ",". <ENTER> must be pressed to terminate input.

It is an error to input a non-numeric value to a numeric variable. To trap such an error and return to the INPUT statement, ON ERROR may be used and the error tested.

INPUT displays a ? followed by the cursor in the next PRINT position. To position an INPUT to a specified place on the screen, the INPUT statement should be preceded by a PRINT@(row, column) and a semi-colon.

For example:

```
100 PRINT@(20, 15);
200 INPUT A$
```

A "prompting message" may also be included in the INPUT statement. This will be printed before the ? prompt. The statement must be enclosed in quotes and followed by a ";".

For example:

```
100 INPUT "NAME"; N$
110 INPUT "AGE IN YEARS, HOUSE NUMBER"; A%,H%
```

The following would be displayed on the screen. The data entered in is underlined.

```
NAME? JIM
AGE IN YEARS, HOUSE NUMBER? 15,113
```

If insufficient fields are entered, further ? prompts are given until all the requested values have been entered.
An empty input field, produced by typing a comma or <ENTER> with nothing between it and the previous comma or the ? prompt, leaves the previous value in the variable unchanged.

Entering spaces only followed by a comma or <ENTER> sets a string variable to null (zero length), and a number to zero. (To set a string variable to a space, enter " " with quotes). In other cases, leading spaces are ignored when inputting into a string and all spaces are ignored when inputting into a number.

For example:

```
10 INPUT A%
```

Entering

```
6 <SPACE> <SPACE> 0
```

is equivalent to entering

```
? 60
```

If <ENTER> is the only key pressed, then the variable is left with the value it had before the INPUT statement.

3.2.40. INPUT#

Availability:- Not available in Standalone mode

Syntax:- INPUT #channel, variable [,variable]...

INPUT# reads the next sequential record from a disk file on the specified channel and assigns values contained in it to the variables specified in the variable list. INPUT#0 works the same as the ordinary INPUT statement except that no ? prompt is given.

The disk file must first be OPENed and later CLOSEd. The disk file must previously have been created to match the form in the variable list. See PRINT#. To test for the end of the file, an ON END trap may be used.

For example:

INPUT and list all the records from file "NAMES.TXT". The records each contain 3 fields.

```
10 OPEN OLD "NAMES.TXT" AS 1
20 ON END #1 GOTO 1000
30 INPUT #1, A$,B%,C
40 PRINT A$
50 PRINT B%,C
60 GO TO 30
1000 CLOSE 1
```

3.2.41. INPUT LINE

Syntax:- INPUT LINE [(length)] string-variable

INPUT LINE inputs all data typed in up until <ENTER> is pressed and places it in the string variable. Only one string is allowed. Data typed in may include commas (unlike INPUT, where the comma specifies the start of a new field). Any prompting text must be printed by a previous statement. If a length is specified this is the maximum length of the string that may be entered. When the user types a line in response to INPUT LINE the cursor will refuse to move past the column of the last character which will be accepted into the string.

For example:

```
100 PRINT@(10,0) "TYPE IN YOUR NAME AND ADDRESS"
110 INPUT LINE A$
```

On the screen this would appear as:

```
TYPE IN YOUR NAME AND ADDRESS
? JOHN SMITH, 246 BERKLEY GROVE, WADESTOWN
```

All of the name and address entered by the user is assigned to A$, including the commas.

```
200 INPUT LINE (8) X$
```

This will accept no more than 8 characters. If the user types "JOHN SMITH", the "I" will be overwritten by the "T", then the "H".

If a null string is entered, the string will be set to a length of zero; it will not be left with its previous value (as it is for a null item using INPUT).

If the length required is zero, then the string will be set to zero length without waiting for the user to input a null line. A negative maximum length is an error.

3.2.42. INPUT LINE#

Availability:- Not available in Standalone mode

Syntax:- INPUT LINE [(length)] #channel , string-variable

INPUT LINE# works exactly like INPUT LINE but the string comes from the disk file which is OPEN on the specified channel. The input takes all characters up to the first line feed or up to the length specified (see INPUT LINE). INPUT LINE#0 inputs data from the keyboard similiar to INPUT, but does not give a prompt.

3.2.43. KILL

Availability:- Not available in Standalone mode

Syntax:- KILL "filename"

KILL deletes a file from disk. The defaults are the working drive and .BAS extension.

For example:

```
100 KILL "1.XXXX.AAA"
```

Line 100 KILLs file XXXX.AAA which is found on drive 1.

3.2.44. LET

Syntax:- [LET] variable = expression

LET assigns a value to a variable. The word LET may be omitted. The variable and the expression must both be numeric or both be strings. A real value assigned to an integer variable will be truncated (see the INT function).

For example:

```
5 DIM G(8,8)
10 LET X%=7
20 LET C$="ABCDEF"
30 D=SQR(5)*7.8+9
40 G(0,8)=6
50 A%="ABCDE"
```

Line 30 LET has been left out.
Line 40 assigns an array element.
Line 50 is an invalid statement because you cannot assign a string to a numeric variable.

3.2.45. LINE

Syntax:- LINE line-description

LINE is used to draw lines on to the current screen.

LINE may be used on any of the screens but if used on the text screens, the programmer must ensure that the graphics control characters are printed on the rows before using this command.

All points are specified as coordinates in the form (row, column). The coordinates may be specified with or without surrounding parentheses. If parentheses are used, the comma between points may be omitted. If parentheses are omitted, then all commas must be included.

For example:

```
LINE (10,10)(20,15)(30,20)(40,10)
LINE (10,10),(20,15),(30,20),(40,10)
LINE 10,10,20,15,30,20,40,10
```

are all legitimate forms of specification and draw a line joining the points.

Lines may be stopped and started by using a ; in place of the comma between points. The points on either side of the ; are not joined.

For example:

```
LINE (10,10) (20,15);(30,20)(40,10)
```

or

```
LINE 10,10,20,15;30,20,40,10
```

would draw 2 lines, one joining (10,10) to (20,15) and a second line joining (30,20) to (40,10).

Use of LINE on the graphics screens

COLOUR 0 (BLACK) may be used to remove a line.

The colours YELLOW, MAGENTA and CYAN can only be obtained by using MIX, either by adding a colour to the current screen or by mixing the two screens.

If a point outside the screen is specified, then the line is drawn to the edge of the screen as if it was joined to that point.

Use of LINE on the text screens

On teletext screens, the user must ensure that graphics control characters have been written prior to using LINE. A simple loop to insert these is:

```
10 FOR I%=0 TO 23: PRINT@(I%,0)"\R";:NEXT I%
```

which prints "\R" or CHR$(18) in column 0 of all rows on the screen.

Because the graphics control character is placed in column 0, chunky pixels cannot be placed in chunky graphic columns 0 or 1 on the text screen.

For example:

```
10 SELECT 1 : CLS
20 FOR I%=0 TO 23: PRINT@(I%,0) "\R";:NEXT I%
30 LINE (14,10),(42,78),(0,0)
```

draws a line joining the 3 points screen 1 in green.

For example:

```
10 SELECT 2 : DISPLAY 2
20 COLOUR 1
30 LINE (0,0),(203,0),(203,239),(0,239),(0,0)
40 A$=INCH$(0) :REM DISPLAY UNTIL KEY PRESSED
```

draws a red line around the boundary of the screen.

3.2.46. LIST

Syntax:- LIST [start line[- end line]]

LIST displays a single program line, a group of lines, or the whole program. A long listing may be stopped for examination at any time by pressing <PAUSE>. Use of the <SPACE BAR> or the <PAUSE> key allows stepping through the listing, line by line. Pressing <EXIT> terminates the listing.

For example:

LIST	Lists the whole program.
LIST 10-90	Lists lines 10 to 90.
LIST 83	Lists line 83.
LIST -200	Lists lines 1 to 200.
LIST 200-	Lists from line 200 to the end of the program.

To list a program on the printer, simply enter

```
SAVE "filename.PRT"
```

(.PRT files are automatically listed on the printer and the file is deleted after printing.)

3.2.47. LOAD

Availability:- Not available in Standalone mode

Syntax:- LOAD "filename"

LOAD loads a BASIC source program or a text file from disk into memory. The defaults are the working drive and .BAS.

For example:

```
LOAD"AAAAAA"
```

loads AAAAAA.BAS from the working drive.

```
LOAD"1.BBBB.TXT"
```

loads BBBB.TXT from drive 1.

If the POLY is in TEXT mode, the default extension is .TXT .

3.2.48. LOAD#

Availability:- Not available in Standalone mode

Syntax:- LOAD # channel, string-name
LOAD # "file-name", string-name

LOAD# loads strings previously stored as disk files using SAVE# or STORE#. In the first form, the specified channel must have already been OPENed (OLD). In the second form, the specified file is opened (as an OLD GRAPH file) and closed automatically. LOAD# may also be used to load binary files into a string (possibly to execute as an assembler subroutine).

3.2.49. LOCK

Availability:- Not available in Standalone mode

Syntax:- LOCK #channel

LOCK causes the random file attached to the specified channel to be "locked", i.e. no other user will be able to access the file (either read or write) until it is "unlocked".

3.2.50. LOGOFF

Syntax:- LOGOFF

LOGOFF returns the POLY unit to the magenta start up screen.

3.2.51. LPRINT

Availability:- Not available in Standalone mode

Syntax:- LPRINT print-list

LPRINT outputs the data specified to a sequential disk file which will automatically be printed on completion of the program, or if the program is stopped.

It is not necessary to specify a channel number, nor to open and close the disk file. (For a description of the print-list see PRINT.)

For example:

```
10 A= 1.2 : B%= 2 : C$= "SSSSSSS"
20 LPRINTA;B%,C$
30 END
```

This prints on the line-printer:

```
1.2  2    SSSSSSS
```

3.2.52. LSET

Syntax:- LSET string-variable = string-expression

LSET stores a new string value in an existing string storage location. The value is either truncated if it is too long, or has spaces added to it if it is too short (Compare RSET).
LSET (or RSET) is the only way to put a value into a variable in the I/O buffer before writing it to a disk file.

For example:

```
10 OPEN OLD RANDOM "XXXX" AS 9
20 FIELD#9,5 AS A$,6 AS B$,6 AS C$
30 LSET A$="AAAAAXXX"
40 LSET B$="BBB"
50 LSET C$="CCC"
60 PUT#9,RECORD 10
70 CLOSE 9
```

Line 10 OPENs file XXXX.DAT on channel 9. Line 20 assigns positions in the buffer to A$,B$ and C$. Lines 30-50 put the values into the strings in the buffer as follows:

```
A$="AAAAA", B$="BBB   ", C$="CCC   "
```

Line 60 PUTs the contents of the buffer into RECORD 10 in the disk file XXXX.DAT .

3.2.53. MEM

Syntax:- MEM [high-memory-address]

MEM resets the top of memory address of the BASIC programs working area. This is useful when a protected area is required for a machine language subroutine. The current memory address can be obtained using FRE(1). If the address is not specified it is reset to the initial system value.

MEM should not be used within GOSUB routines, ON ERROR routines, ON KEY routines or ON SEC routines as the stack containing the return address is stored just below high memory.

For example:

```
10 S% = FRE(1)
20 MEM S% - 100
30 REM SAVES 100 BYTES
     .
     .
     .
10000 REM END OF PROGRAM
10010 MEM
```

This "protects" 100 bytes of memory and later releases it.

3.2.54. MERGE

Availability:- Not available in Standalone mode

Syntax:- MERGE "filename"

MERGE loads a BASIC program source file from disk into memory, merging it with the BASIC program already in memory (if any). A line from the file being merged will overwrite a line in memory if they have the same line number. Defaults are .BAS and the working drive. In TEXT mode, .TXT is the default extension.

3.2.55. MIX

Syntax:- MIX [ON]
MIX [OFF]
MIX screen, colour

MIX is used for controlling:-
a) The mixing of the colours on screens 2, 3 and 4.
b) The adding of a colour to pixels ON within screens 2 or 4.

MIX ON specifies that the beams on screens 2, 3 and 4 are to be MIXed. If the current COLOUR is either YELLOW, CYAN or MAGENTA, then DRAW, FILL and LINE will draw on both screens 2 and 4, and if both screens 2 and 4 are displayed, then these colours will be displayed MIXed.

MIX OFF switches MIX mode OFF so that the screens display in their priority order.

MIX screen, colour, adds the colour to all pixels ON on the specified graphics screen (see COLOUR for examples).

Mixing Colours

The colour of each pixel on the screen is determined by which of the 3 colour beams - red, blue, and green - are switched on. These are the primary colours. The secondary colours require 2 or more beams on such that:

```
Red   + Green       = Yellow
Red   + Blue        = Magenta
Green + Blue        = Cyan
Red + Green + Blue  = White
```

All 7 colours are available on the text screens, but the graphics screens have only the primary colours immediately available. The secondary colours are created using the MIX command. When two colours are mixed the resulting colour is the composite of the two colours.

For example:

```
Red + Magenta = Red + Red + Blue
              = Red + Blue
              = Magenta
```

Note that if a beam is repeated in the MIX, it is not doubled in intensity.

3.2.56. NEW

Syntax:- NEW

NEW deletes the current program from the POLY unit. If the file in memory is a source program (or a text file) that has been altered, the user will be prompted with

```
Save (Y/N)?
```

or

```
Save filename (Y/N)?
```

The filename will appear only if the file was LOADed. In the first case if Y is typed, the NEW is aborted; if N is typed the NEW is executed. In the second case if Y is typed the file will be SAVEd and NEW executed; if N is typed, NEW will be executed. Only Y, y, N, or n will be accepted.

3.2.57. NEXT

Syntax:- NEXT [loop-variable]

NEXT terminates a FOR loop. The loop variable name need not be specified.

For example:

```
10 FOR N%=1 TO 12
20 PRINT N%
30 NEXT N%
```

This prints a list of the first 12 integers.

3.2.58. ON END

Availability:- Not available in Standalone mode

Syntax:- ON END #channel GOTO line-number

ON END allows the user to set a line number to which control will be passed when an attempt to read the file on the specified channel fails because there is no more data to be read. The file must be open when ON END is set. RESUME is not necessary after an ON END.
Warning:- When using ON END with string files saved with SAVE# or STORE#, several null strings may be input at the end of the file before ON END causes a branch.

For example:

```
10 OPEN OLD "FILLY.TXT" AS #2
20 ON END #2 GOTO 1000
30 INPUT LINE #2,A$
40 PRINT A$
50 GOTO 30
1000 CLOSE #2
1010 END
```

Line 10 OPENs FILLY.TXT on channel 2.
Line 20 sets the ON END line number for channel 2.
Line 30 INPUTs a line from the file.
Line 40 PRINTs the line from the file.
Line 50 causes the program to GOTO 30 for another INPUT.
Line 1000 CLOSEs the file and will only be executed once end of file is detected.

3.2.59. ON ERROR

Syntax:- ON ERROR GOTO linenumber

ON ERROR allows the user to trap errors and carry out whatever action is required. When an error occurs in a program without an ON ERROR GO TO, or the line number given is 0, the program terminates with a message like

```
ERROR 74 IN LINE 210
```

When an ON ERROR routine is included, the program will go to the specified line when an error occurs. The error number is saved in ERR and the line number on which the error occurs is saved in ERL. The ON ERROR statement must be executed prior to the line causing the errors. After an ON ERROR routine, control is passed back to the main program with a RESUME statement.

For example:

```
10 ON ERROR GO TO 1000
     .
     .
     .
200 FOR I%=1 TO 300
210 READ A(I%)
220 NEXT I%
230 REM program carries on
     .
     .
     .
1000 IF ERL=210 AND ERR=91 THEN RESUME 230 ELSE
      RESUME 1010
1010 PRINT ERR,ERL
```

This program READs from DATA lists. When all data has been read into array A error 91 will occur in line 210. This has been trapped and processing will resume at line 230. Other errors will be reported in line 1010.

The ON ERROR routine does not trap errors within itself.

ON ERROR routines may be tested by simulating errors using ERROR.

If an ON KEY is trapped during an ERROR routine, the ON KEY routine is not executed until AFTER resumption from the ERROR routine.

3.2.60. ON GOSUB

Syntax:- ON numeric-expression GOSUB line[,line]...

ON GOSUB allows different subroutines to be called from the same statement. Control will RETURN to the statement following the ON GOSUB statement.

For example:

```
100 ON N GOSUB 110,120,130,140,150
105 REM ....
```

If N = 1 control goes to the subroutine at line 110.
If N = 2 control goes to the subroutine at line 120 etc.

etc

If N < 0 or N > 5 control goes to the next statement, in this example, line 105.

Non-integer values for the numeric expression are truncated. Values less than or greater than the number of items in the list cause the program to continue at the following statement.

3.2.61. ON GOTO

Syntax:- ON numeric-variable GOTO line[,line]...

ON GOTO is similiar to ON GOSUB except that control is not RETURNed to the statement following the ON GOTO statement.

If the numeric-variable has a value less than 1 or greater than the number of items in the list, then the program continues at the following statement.

For example:

```
50 ON N% GOTO 200, 210, 220
60 REM ...
```

If N% = 1 control goes to the statement at line 200.
If N% = 2 control goes to the statement at line 210.
If N% = 3 control goes to the statement at line 220.
If N% < 0 or N% > 3 the program continues onto the line following line 50.

3.2.62. ON KEY

Syntax:- ON KEY [key-no-1][TO key-no-2] GOTO [linenumber]
ON KEY keynumber AS new-value

ON KEY allows the functions performed by specific keys to be programmed.

Programming the Special Keys

The special keys include the numeric keypad, the cursor keys, the editing keys and other special purpose keys. These are all assigned a special ON KEY number as follows:

Key(s)	ON KEY value
Numeric keypad numbers 0-9	0-9
EXIT	10
PAUSE	11
ENTER	12
NEXT	13
REPEAT	14
BACK	15
HELP	16
CALC	17
back arrow	18
forward arrow	19
down arrow	20
up arrow	21
INS CHAR	22
DEL CHAR	23
INS LINE	24
DEL LINE	25
. on keypad	26
SHIFT PAUSE	27
@	28
£	29
EXP	30
|	31

The ON KEY procedure works in a similar way to an ON ERROR procedure in that as soon as the key specified is pressed, a special routine in the program is performed. Control must be returned using RESUME.

During the ON KEY routine the ON KEY value of the key pressed is in KVAL. To turn off the ON KEY interrupt use:

ON KEY [key-no-1] [TO key-no-2] GOTO 0

For example:

```
20 ON KEY 12 GO TO 1000
30 REM other statements
     .
     .
     .
1000 CLS
1010 RESUME
```

This would cause the program to clear the current screen whenever <ENTER> was pressed. The program would RESUME normal operation at the statement following that during which <ENTER> was pressed. The keyboard is checked after every statement.

For example:

```
20 ON KEY 10 GO TO 1000
     .
     .
     .
1000 RESUME
```

This disables the <EXIT> key. The ON KEY routine at line 1000 does nothing except continue when the <EXIT> key is pressed.

For example:

```
10 DIM a%(9)
15 count = 0
20 ON KEY 0 TO 9 GOTO 1000
     .
     .
     .
1000 a%(count) = KVAL
1010 count = count +1
1020 IF count > 9 THEN ON KEY 0 TO 9 GOTO 0
1100 RESUME
```

This routine allows up to 10 numeric keys to be pressed during other processing. The values of these keys are stored in the array a%. When 10 keys have been pressed, the ON KEY is switched OFF.

The whole keyboard may be trapped by ON KEY by omitting the key-number.

For example:

```
20 ON KEY GO TO 1000
     .
     .
     .
1000 IF KVAL < 65 OR KVAL > 90 THEN RESUME
     .
     .
     .
1100 RESUME
```

In this case KVAL contains the ASCII value of the key pressed.

In this example, whenever any key is pressed, the ON KEY routine is executed. In line 1000, if the key pressed is not a capital letter, processing continues. Note ON KEY GOTO 0 resets the whole keyboard trap but not the special key trap.

Changing the ASCII values of the numeric keypad

ON KEY may also be used to assign a new ASCII value to the keys 0 to 9 of the numeric keypad.

For example:

It is required to enter from the keyboard the teletext characters 1/4, 1/2 and 3/4 as these are not on the keyboard. The numeric keypad numbers 1, 2 and 3 are assigned for entering these characters.

```
20 ON KEY 1 AS 123
30 ON KEY 2 AS 92
40 ON KEY 3 AS 125
```

If a KEY is trapped during a KEY, SEC or ERROR routine, the ON KEY routine for that key is not executed until after resumption of the main code following completion of the routine. Multiple KEY traps are stacked.

3.2.63. ON SEC

Syntax:- ON SEC [interval] GOTO line-number

ON SEC acts in a manner similar to ON ERROR or ON KEY except that the ON SEC routine is performed at specified time intervals. The interval is specified in seconds.

For example:

```
100 ON SEC 5 GOTO 1000
      .
      .
      .
1000 PRINT@(0,32)TIME$;
1010 RESUME
```

During this program, the time would be displayed every 5 seconds on the top line of the screen.

3.2.64. OPEN

Availability:- Not available in Standalone mode

Syntax:- OPEN OLD [RANDOM [(record-length)]] "filename"
AS [#]channel
OPEN NEW [BACK] [GRAPH] [RANDOM[(record-length)]]
"filename" AS [#]channel

In general before a file may be used in BASIC, it must be OPENed. Sequential files are either written from beginning to end (NEW) or read from beginning to end (OLD), whereas RANDOM files may be read or written in any order and may previously exist or not. Sequential files are accessed using PRINT# and INPUT#. For RANDOM files, a record length (an expresion yielding an integer between 1 and 252) may be specified. PUT and GET then refer to buffers of this length rather than 252.

A BACK option is available when a NEW file is opened. If BACK is not specified and the specified file already exists, then it will be deleted. If BACK is specified and the specified file already exists then it will be renamed with a .BAK extension (if a .BAK file with the same name also exists it will be deleted). If the NEW file is not written to, only the .BAK file will exist.

If the GRAPH option is specified when opening a sequential file, space compression will be turned off. This mode is necessary if control characters are to be input or output. The GRAPH option is only allowed to be used with sequential files. Control characters can be input and output from RANDOM files without any special considerations.

It is possible to have up to 12 files open (for both read and write) at any given time. The OPEN statement associates the specified file with an input/output channel.

The # is optional before the channel number.

For example:

```
10 OPEN NEW "AAAA" AS 7
20 A$="DD" : B=9.2 : C%=99
30 PRINT#7,A$,",",B,",",C%
40 CLOSE 7
50 OPEN OLD "AAAA" AS 2
60 INPUT#2,W$,X,Y%
70 PRINT W$,X,Y%
80 CLOSE 2
```

The display shows:

```
DD      9.2     99
```

Line 10 OPENs a new file on channel 7.
Line 30 PRINTs data into the file.
Line 40 CLOSEs the file on channel 7.
Line 50 OPENs an existing file on channel 2.
Line 60 INPUTs data from the file.
Line 70 PRINTs the data on the POLY screen.
Line 80 CLOSEs the file on channel 2.

The variable names are not stored with the data. Default extensions are the working drive and .DAT . Channel numbers may be variables or expressions with a value between 1 and 12 inclusive.

Random files must be OPENed with the RANDOM option. Random files are accessed using GET# and PUT#.

The use of PUT# and GET# is shown in the following example.

For example:

```
10 OPEN OLD RANDOM (20) "EXAMPLE" AS 11
20 GET#11,RECORD 2
30 FIELD#11,5 AS A$,15 AS B$
40 PRINT A$;B$
```

```
50 LSET A$="AAA"
60 LSET B$="BBB"
70 PUT#11,RECORD 2
80 CLOSE 11
```

The old file EXAMPLE.DAT is opened, record 2 is read into the buffer and the fields within it are defined. Strings A$ and B$ are printed then new data is moved to them in the buffer. Then the whole contents of the buffer is PUT back in to the same record on the disk. Lastly the file is closed.

The size of a random file may be calculated as follows.
If the record length is r%, then the number of records per disk sector is th e integer value of 252/r% i.e. INT(252/r%).
If the maximum record number you PUT to the file is max%, then the number of sectors allocated to the file is

2 (for the index) + max%/INT(252/r%)

If this value is not an integer, it must be rounded up.
For example, if r%=252 and you PUT record 92, the file will be 94 sectors long. If r%=20 and you PUT record 200, the file will be 19 sectors long.

NOTE (i) If no data is written to a new sequential file before it is closed, then when it is CLOSEd (by CLOSE or END) it will be removed from disk. If no data is written to a new random file before it is closed, it will exist.
(ii) GET and FIELD may be used with sequential files (but not PUT).
(iii) OPENing and using files in RAMdisk (drive 4) will substantially increase the speed of access.

3.2.65. POKE

Syntax:- POKE address, numeric-expression

POKE stores a single character integer value at the specified address. The address must be between 0 and 65535. The numeric-expression must have a value between 0 and 255 inclusive. See DPOKE.

For example:

```
POKE 1161,1
```

This puts the value 1 into address 1161.

3.2.66. PRINT

Syntax:- PRINT print-list

The print-list is a list of items to be printed. PRINT causes items in the print-list to be displayed starting at the current PRINT position. This position may be varied by the punctuation in the PRINT statement. ? may be used as an abbreviation for PRINT.

No punctuation after an item causes the current position to move to the start of the next line.

For example:

```
10 PRINT "AAA"
```

A comma after an item causes the next item to be PRINTed starting in column 0,8,16,24, etc (whichever is the next column). On the screen, column 40 is the same thing as column 0 of the next line.

For example:

```
20 PRINT "0","8","16","24","32"
```

This displays the numbers each starting in the given column. i.e.

```
0       8       16      24      32
```

A semi-colon after an item causes the next item to be PRINTed immediately following the previous one.

For example:

```
30 PRINT "A";"B"
```

This will display

```
AB
```

with no spaces between.

An exclamation mark causes the next data to be printed on the next line starting in the same column as the start of the last data printed.

For example:

```
20 PRINT@(10,10)"Line 1"!
30 PRINT"Line 2"!
40 PRINT "Line 3"!
```

would be printed as, starting on line 10, in column 10.

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

Commas, semi-colons and exclamation marks may be mixed in a single PRINT statement.

Numbers are printed with one trailing space, and one leading space unless this is filled with a minus sign. All strings, including 'number' strings, have no leading or trailing blanks.

For example:

```
100 X$="ABC" : B=4.2 : C%=-7
110 PRINT X$ : PRINT B : PRINT C%
120 PRINT X$,B,C%
130 PRINT X$;B;C%
```

This program displays:

```
ABC
4.2
7
ABC     4.2      -7
ABC 4.2 -7
```

3.2.67. PRINT@

Syntax:- PRINT@(row,column)[,]print-list

PRINT@ specifies the point where the PRINTing starts. The rows are numbered from 0 to 23, and the columns from 0 to 39. When printing on row 23 a semi-colon is necessary to stop scrolling of the screen.

For example:

```
10 X=555
20 PRINT@(10,4)"X =";X
30 PRINT@(23,0)"This is the bottom line";
```

This displays: "X = 555" starting at position (10,4), and "This is the bottom line" starting at position (23,0).

3.2.68. PRINT USING

Syntax:- PRINT USING string, print-list

PRINT USING uses various special strings to format the data displayed. The string is an image of the required output, but with special characters in place of the actual characters. The print list is similar to that of the PRINT statement. Literal characters, i.e. ones that are not special, may be inserted in front of the string of special characters. Special characters are: !,",#,$,*,!, and sometimes a comma.

For example:

```
10 PRINT USING"! ! !","Joe","E","Bloggs"
```

This prints the initial letters with a single space between each pair. i.e. J E B. The "!" denotes a single character string field.

```
20 PRINT USING"!!!","Joe","E","Bloggs"
```

This prints the initial letters without spaces. i.e. JEB

```
30 PRINT USING"%1234%","ABCDEFGH"
```

This prints the first 6 letters of the string i.e. ABCDEF. The pair of "%"'s denote a string field with a length equal to the total length of the image string. The middle characters are arbitrary.

```
40 PRINT USING"####.###",123.4567
```

This prints 123.457 in a total field width of 8. The "#" characters denote a number field. Decimal points are lined up and the number is rounded to fit the format.

```
50 PRINT USING"$###.##",-9.87
```

This prints $ -9.87 in a total field width of 7 (a dollar sign, up to 3 digits, one of which may be used for a minus sign, a decimal point and 2 digits).

```
50 PRINT USING"$$##.##",-9.87
```

This prints -$9.87 i.e. the dollar sign is moved right to be immediately in front of the first digit when two dollar signs are used. "$$##.##" will handle numbers up to 999.99, just as "$###.##" will, the second dollar sign acts as a "#" in reserving space for a digit.

```
60 PRINT USING"$$###.##-",-10.1234
```

This prints $10.12- .

```
70 PRINT USING"**###.#",1.23
```

This prints ****1.2 . Two asterisks cause all leading spaces to be filled.

```
70 PRINT USING"**###.#",-1.25
```

This prints ***-1.3.

```
80 PRINT USING"$**###.##",67.89
```

This prints $***67.89 . The $ is a 'literal' character preceding the asterisks.

```
90 PRINT USING"###,#.##",2345.67
```

This prints 2,345.67 . A single comma embedded somewhere in the numeric field will cause the number to be printed out with commas separating every three digits.

```
100 PRINT USING"##.###↑↑↑↑",123456
```

This prints 1.235E+05. Four up arrows are needed after the numeric field to print in scientific notation.

NOTE: If the number is too big to fit into a given format a % sign will be printed followed by the number in a format as close as possible to that specified.

3.2.69. PRINT@ USING

Syntax:- PRINT@(row,column) USING string, print-list

PRINT@ may be combined with USING. Refer to PRINT USING.

For example:

```
10 A$="###.##" : X=4.5
20 PRINT@(10,15)USING A$,X
```

This will print 4.50 in row 10 with the decimal point in column 18.

3.2.70. PRINT#

Availability: Not available in Standalone mode

Syntax:- PRINT#channel, print-list

PRINT# outputs the data specified to a sequential access disk file. The print-list format is as described in PRINT.

For example:

```
10 OPEN NEW"AAAA" AS 5
20 PRINT#5,A$,",",B,",",C%
30 CLOSE 5
```

Be careful to insert "," between variables if the file is to be read using INPUT, because INPUT expects commas to separate fields. This is very easy to overlook.

Note that channel 0 corresponds to the screen.

3.2.71. PRINT# USING

Availability:- Not available in Standalone mode

Syntax:- PRINT#channel, USING string, print-list

PRINT# may be combined with USING. Refer to PRINT USING and PRINT#.

For example:

```
10 OPEN NEW"AAAA" AS 5
20 A=1 : B=23.456 : C=206
30 PRINT#5, USING "$###.##", A,",",B,",",C
40 CLOSE 5
```

Be careful to insert "," between variables if the file is to be read using INPUT.

3.2.72. PUT#

Availability:- Not available in Standalone mode

Syntax:- PUT #channel [, RECORD numeric-expression]

PUT# is used to PUT data from an I/O buffer into a random access disk file. FIELD is used to define the buffer area and LSET and RSET must be used to place the data in the buffer.

For example:

```
10 OPEN NEW RANDOM "NNNNN" AS 3
20 FIELD#3,6 AS G$,5 AS H$,4 AS I$
30 LSET G$="GG" : LSET H$="HHH" : LSET I$="IIIIII"
40 PUT#3,RECORD 8
50 CLOSE 3
```

This has now PUT into record 8 of file NNNNN the data:

```
"GG    HHH  IIII"
```

Each time a PUT or GET statement is actioned and the RECORD number is not specified, the RECORD number is automatically increased by one.

For example:

```
10 OPEN NEW RANDOM(18) "PPPP" AS 6
20 FIELD#6, 5 AS X$, 6 ASY$, 7 AS Z$
15 FOR N%=1 TO 10
30 LSET X$="XX" : LSET Y$="YY" : LSET Z$="ZZ"
40 PUT#6
45 NEXT N%
50 CLOSE 6
```

This PUTs the same three strings into the first 18 bytes of RECORDs 1 to 10.

GET# is used to read the data from the file.

3.2.73. RANDOM

Syntax:- RANDOM

RANDOM generates a new seed for the RND function. This means that each time RANDOM is used, a new sequence of RND values is started.

For example:

```
20 PRINT RND(10)
```

will always give the same result each time after switching the POLY unit ON

```
10 RANDOM
20 PRINT RND(10)
```

ensures that the value is different each time.

3.2.74. READ

Syntax:- READ variable-list

READ is used to READ data specified in DATA statements and place it in specified variables. The data must match the variable type.

For example:

```
300 READ A,B%,C$
500 DATA 32.4,55,AAAA
```

This will take the next 3 items in a DATA list and store them in variables A, B% and C$.

Trying to READ DATA items when there is no more data causes an error. This may be trapped by using ON ERROR GO TO and testing for ERR = 91. To use the same data again the RESTORE statement may be used.

3.2.75. REM

Syntax:- REM [remarks]

Any text in a REMark statement is ignored by the program and is used purely for documentation to make the program more understandable.

For example:

```
230 REM Put explanations here ...
240 X%=0 : REM X% counts sheep
```

3.2.76. RENAME

Availability:- Not available in Standalone mode

Syntax:- RENAME "filename1","filename2"

RENAME may be used to change the name of a file on disk. The default drive is the work drive. The default extension for filename1 is .DAT; the default for filename2 is whatever filename1 had.

For example:

```
RENAME "1.AAAA","1.BBBB.BAS"
RENAME"AB123","AB100"
```

File 1.AAAA.DAT is renamed 1.BBBB.BAS and file AB123.DAT is renamed AB100.DAT.

3.2.77. RENUM

Availability:- Not available in Program mode, nor Standalone mode.

Syntax:-

RENUM [start-line[,increment[,first-line[-last-line]]]]

RENUM will renumber the lines of a BASIC program commencing at the line number given by the first parameter (default 10), and proceeding in increments as given by the second parameter (default 10).

The last two parameters specify that part of the program to be renumbered, (defaults are the actual first line and last line of the program i.e. the whole program).

Renumbering will not take place if interleaving and duplication of line numbers would occur. It is not possible to re-order the program with RENUM, use MERGE to do this.

All line numbers contained within the program are changed to their corresponding new line numbers. Line numbers within statements that do not have a corresponding line will be listed and left unchanged, (they would ultimately give rise to errors when the program is run).

For example:

```
RENUM
RENUM 10,5
RENUM 100,10,60,400
```

Warning:- Use of line numbers in expressions (e.g. in a comparison with ERL) will not be located.

3.2.78. RESET

Syntax:- RESET [(]row,column[)]

RESET switches off the specified pixel. On text screens, RESET switches off the chunky graphic pixel. (See SET)

For example:

```
100 RESET 24,26
200 RESET (24,27)
```

3.2.79. RESOFF - RESON

Syntax:- RESOFF or RESON

RESOFF will allow variable names to contain sequences of characters corresponding to a reserved word, except at the beginning of the variable name. RESON cancels RESOFF and is set by default. With RESOFF a variable must be separated from an immediately following word by a space.

For example:

With RESOFF, XLEN and AGOTO are valid variable names, LENX is not. With RESON none are valid variable names.

3.2.80. RESTORE

Syntax:- RESTORE [line-number]

If the same DATA is to be READ in more than once, a RESTORE statement will allow the program to go to the beginning of the DATA, or to the first DATA item after the specified line number.

For example:

```
200 READ A,B,C
210 READ D,E,F
220 RESTORE 280
230 READ G,H,I
240 RESTORE
250 READ J,K,L
260 PRINT A,B,C,D,E,F,G,H,I,J,K,L
270 DATA 2.3,3.4,4.5
280 DATA 6.7,7.8,8.9
```

This program reads in the 6 data items on lines 270 and 280 and assigns them to variables A to F. The data is RESTOREd to line 280, so G, H, and I take on the values 6.7, 7.8, 8.9 respectively. The data is then RESTORED to the first DATA statement so J, K, and L take on the values 2.3, 3.4, and 4.5.

3.2.81. RESUME

Syntax:- RESUME [line-number]
RESUME NEXT
RESUME LINE

The trapping of an error, processing of a trapped key, or expiry of a certain amount of time are known as interrupts. Interrupts may be thought of as occurring at the beginning of a BASIC statement (i.e. if a statement contains an error, the interrupt may be thought of as occurring at the beginning of the statement containing the error).

ON ERROR, ON KEY and ON SEC routines are known as interrupt routines and RESUME must be used to return control from such interrupt routines.

If the line number is omitted or is 0, control will be returned to the point at which the interrupt occurred. For ON ERROR, the statement containing the error will thus be re-executed. For ON KEY and ON SEC, statements will continue to be executed in sequence.

If a line number is specified, then control is returned to the line specified.

If LINE is specified, then control is returned to the beginning of the line on which the interrupt occurred.

If NEXT is specified, then control is returned to the next statement. For ON ERROR, the statement containing the error will thus be omitted. For ON KEY and ON SEC, the next statement will be omitted and since this is likely to be undesirable, RESUME NEXT should not be used with ON KEY or ON SEC.

For example:

```
20 ON ERROR GO TO 300
30 READ A$
40 REM Process data
     .
     .
     .
150 GO TO 30
300 IF ERR=91 THEN PRINT "NO MORE DATA"
310 RESUME 320
320 REM End Data
REM ...
```

For example:

```
20 ON KEY 0 GOTO 1000
30 REM rest of the program
     .
     .
     .
1000 RESUME
```

This disables the 0 on the numeric keypad.

3.2.82. RETURN

Syntax:- RETURN

When a subroutine is completed, control is returned to the main program with RETURN. This sends control to the statement following the corresponding GOSUB.

For example:

```
100 GOSUB 2000
     .
     .
     .
2000 REM SUB 2000 STARTS HERE
     .
     .
     .
2060 IF R=99 THEN RETURN
2060 REM rest of subroutine...
```

3.2.83. RSET

Syntax:- RSET string-variable = string-expression

RSET is similar to LSET except that the inserted string is right justified, i.e. it is packed on the left with spaces to make it fit. Truncation is the same as for LSET.

For example:

```
10 OPEN OLD RANDOM "FFFF" AS 4
20 FIELD#4, 6 AS A$, 6 AS B$
30 X$="123" : Y$="12345678"
40 RSET A$=X$
50 RSET B$=Y$
60 PUT#4
70 CLOSE 4
```

This sets A$=" 123" and B$="123456" in the disk file buffer area. This is then written into the file FFFF.DAT.

3.2.84. RUN

Syntax:- RUN ["filename"][linenumber]

After LOADing a BASIC source program from disk into a POLY unit it may be RUN without specifying the filename. Specifying the filename causes a program to be loaded from disk and started automatically. The filename extension defaults are .BAC and the working drive. After a RUN, all screens except screen 1 are switched OFF and variables are reinitialized.

For example:

```
RUN
```

Executes the BASIC program loaded in POLY memory.

```
RUN"AAAA"
```

Loads the compiled program AAAA.BAC from disk and executes it.

```
RUN 100
```

Executes the BASIC program loaded in POLY memory starting at line 100.

3.2.85. SAVE

Availability:- Not available in Standalone mode

Syntax:- SAVE [BACK] ["filename"] [startline[-endline]]
SAVE

After creating or editing a file (BASIC or TEXT) it may be SAVEd on disk. If BACK is not specified, any existing file of the same name will automatically be deleted. If BACK is specified, any existing file of the same name will be renamed with the extension .BAK (if one of these .BAK files also exists it will be deleted). The default drive is the working drive and the default extension is .BAS from BASIC mode and .TXT from TEXT mode. If a line number range is specified, only those lines will be SAVEd.

For example:

```
SAVE"1.AAAAAA."
```

This SAVEs file AAAAAA.BAS on the disk in drive one.

```
SAVE BACK "XYZ" 100-200
```

This SAVEs file XYZ.BAS, lines 100 to 200 and renames any existing XYZ.BAS as XYZ.BAK.

SAVE may be used without a filename if the file has previously been LOADed from disk. In this case the user will be prompted with

Save filename (Y/N) ?

where filename is the name of the file that was LOADed.

3.2.86. SAVE#

Syntax:- SAVE# channel, string
SAVE# "file-name", string

Any string may be saved on disk using the SAVE# statement. Note that only one string is permitted in the argument list. In the first form, the specified channel must have been already OPENed (NEW). It is possible to write more than one string into a given channel using more SAVE# statements. In the second form, the specified file is opened (as a NEW GRAPH file, the old file with the same name, if any, being deleted) and closed automatically. SAVE# is especially useful for saving text screens.

For example:

```
20 SAVE# "TEXT1", TEXT$(10,0,44)
```

Note that strings written onto disk files using SAVE# and STORE# are preceded by two bytes representing their length. Also since such strings may contain control characters, files should be GRAPH files. The LOAD# statement may be used to reload strings previously saved using SAVE#.

3.2.87. SCROLL

Syntax:- SCROLL ON
SCROLL OFF

If SCROLL is ON, when the bottom of the page is reached, during printing or with the cursor arrow, then all lines move up one line and the top line is lost. Moving off the top of the screen with the up arrow causes the lines to SCROLL down.

When SCROLL is OFF, when the bottom of the screen is reached, printing continues at the top of the screen. The cursor arrows simply move the cursor from the bottom line of the screen to the top line (or vice versa).

SCROLL is turned ON on a RUN, CHAIN, CLEAR or NEW.

For example:

```
100 SCROLL OFF
```

Also see examples in SPLIT.

3.2.88. SELECT

Syntax:- SELECT screen-number

SELECT sets the current screen for writing. The screens available are

1	Top Text Screen (24 by 40)
2	Top Graphics Screen (204 by 240)
3	Bottom Text Screen (24 by 40)
4	Bottom Graphics Screen (204 by 240)
5	Fine Graphics Screen (204 by 480)
5TEXT	Fine Graphics and Text Screen (204 by 480)

At the start of a program, the current screen is always 1. SELECT does not DISPLAY the screen. This must be done with a separate command.

The graphics commands SET, RESET, LINE and POINT refer to the current screen, whether graphics or text. The graphics commands DRAW and FILL refer to current screen, which must be a graphics screen.

At the END of a program, all screens except 1 are turned OFF but not cleared.

On an error or <EXIT>, all screens are left exactly as they are. Commands may be typed in and these are displayed on screen 1, but the displays they produce are placed on the currently selected text screen.

For example:

```
10 CLS
20 SELECT 2
30 CLS
40 DISPLAY 2
50 COLOUR 1
60 LINE 0,0,0,239,203,239,203,0,0,0
70 PRINT@(10,10) "SCREEN 1"
80 GOTO 80
```

Line 10 clears screen 1
Line 20 selects the graphics screen 2
Line 30 clears screen 2
Line 40 displays screen 2
Line 50 selects the colour red
Line 60 draws a box around the screen in red on screen 2
Line 70 prints on screen 1, the words SCREEN 1
Line 80 loops so that display is not turned off

The PRINT command places text on the most recently selected text screen, whether 1, 3 or 5TEXT.

Selecting screen 5TEXT means that screen 5 is selected for both graphics and text. In this case, characters written to screen 5 using PRINT will be software generated. This software must be specially provided by POLYCORP and may be used to provide alternate character sets, e.g. 60 characters per line. The software for generating alternate character sets is stored in a file called POLYACS.SYS. If this file does not exist then selecting 5TEXT will cause screen 1 to be selected.

If screen 5TEXT has been selected and another graphics screen (either 2 or 4) is selected, printing will produce strange results as screen 5 is a combination of screens 2 and 4.

Since screen 5 is not a teletext screen, colour is selected using the COLOUR and MIX commands as for screen 5 graphics, rather than by using control characters. In general characters of different colours should not be placed adjacent to one another on screen 5TEXT.

For example:

```
10 CLS
20 SELECT 5TEXT
30 DISPLAY 5
40 CLS
50 COLOUR1
60 LINE 70, 150, 70, 330, 98, 330, 98, 150, 70, 150
70 COLOUR2
80 PRINT@ (10,21) "This is an example"
90 GO TO 90
```

This will display a red box enclosing blue text produced using the alternate character generator.

The use of screen 4 (and hence screen 5) reduces the available program memory. If screen 4 is selected late in a program it may be found to have 'noise' on it since the memory space was already being used as a working area. Either select screen 4 at the start of the program or clear screen 4 with CLS immediately after selection. If screen 4 is selected, the size of the largest possible program that may be chained to is 100 disk sectors.

3.2.89. SET

Syntax:- SET [(]row,column[)]

SET switches on the specified pixel on the current screen. On graphics screens, the pixel is turned on in the current colour. If the colour is a secondary colour, the specified pixel is also turned on in the other graphics screen. MIX must be ON and both screens 2 and 4 displayed for this to be displayed in the correct colour (See COLOUR and MIX).

On text screens, a teletext graphics control character must have previously been placed on the start of the line. If this is missing the character equivalent of the graphics character is displayed. The colour is as set up in the graphics control character.

For pixels to be displayed using SET

Rows should be between 0 and 71 for screens 1 and 3.
Rows should be between 0 and 203 for screens 2, 4 and 5.
Columns should be between 2 and 79 on screens 1 and 3.
Columns should be between 0 and 479 on screen 5.

For example:

```
100 CLS
110 FOR row = 0 TO 23:PRINT@(row,0)" ,";:NEXT row
120 SET 15, 20
130 SET (64,70)
140 SELECT 2:CLS:DISPLAY 2
150 SET (120,230)
160 COLOUR 1
170 SET 121,230
```

Line 110 prints graphics green control characters on screen 1
Lines 120-130 set points on
Line 140 selects, clears and displays screen 2
Line 150 sets a point on in WHITE
Line 160 sets COLOUR to red
Line 170 sets a point on in RED

3.2.90. SOUND

Syntax:- SOUND[pitch [, length]]

SOUND without parameters produces a beep. With parameters, SOUND produces a pitched sound of the specified length. The parameter "pitch" should be calculated as (502400/frequency)-1 e.g. to get a 200Hz frequency for 1 second you could write "SOUND (502400/200)-1,100" or "SOUND 2511,100". Equivalent musical note values are given in Appendix 4.5. The length must be specified in 10 millisecond lengths.

For example:

```
10 C=1919:G=1281:A1=1141:B1=1016:C1=959
    .
    .
    .
100 SOUND C,50
110 SOUND C,50
120 SOUND G,50
130 SOUND G,50
140 SOUND A1,25
150 SOUND B1,25
160 SOUND C1,25
170 SOUND A1,25
180 SOUND G,100
```

This plays the first line of "BAA BAA BLACK SHEEP".

3.2.91. SPLIT

Syntax:- SPLIT number-of-lines[,cursor-action]

Each of the text screens may be split into 2 independently scrolling screens.

The number-of-lines specifies the number of lines in the top section of the screen.

The cursor-action, if specified must contain either 0 or 1. If this is 0 or not defined, then the cursor is left in place after each PRINT. If 1 is specified, after each PRINT, the cursor is returned to the top section of the screen to where it was after the last PRINT.

For example:

```
100 CLS : SPLIT 10
110 GOSUB 200
120 PRINT@(10,0);
130 GOSUB 200
140 END
200 FOR I = 0 TO 20
210 PRINT I
220 NEXT
230 RETURN
```

Line 100 splits the screen
Line 110 Prints the numbers 0 to 20 in the top section
Line 120 moves the cursor to the top line of the bottom section
Line 130 Prints the numbers 0 to 20. This shows the independant scrolling of the two sections. If the line

```
105 SCROLL OFF
```

is inserted, then the return to the top of each section can be seen.

With a split screen, CLS only clears the section of the screen on which the cursor is currently placed.

For example:

Add the line

```
135 CLS
```

to the program in the above example and only the bottom section of the screen is cleared.

For example:

```
100 CLS : SPLIT 23,1
110 FOR I = 0 TO 100
120 PRINT I
130 PRINT@(23,0) I;
140 NEXT I
```

Line 100 clears the screen, placing the cursor in (0,0) and then splits the screen specifying that the cursor is to return to the top section. In the loop (lines 110-140), the number is printed in the top section and then in the bottom section. Following the PRINT@(23,0) in the bottom section, the cursor is returned to its position in the top section for the next PRINT in line 120.

On a RUN, NEW, CHAIN or CLEAR, SPLIT is reset to SPLIT 24,0.

3.2.92. STOP

Syntax:- STOP

The program terminates and a STOP AT LINE message will be displayed. The program can be restarted from just after the STOP statement with a CONT command. STOPs are useful for diagnostic purposes.

For example:

```
250 IF X=10 THEN STOP ELSE PRINT X
```

If X = 10 the program will halt displaying STOP AT LINE 250.

3.2.93. STORE

Availability:- Not available in Standalone mode

Syntax:- STORE (r1,c1),(r2,c2) string-name
STORE(r1,c1),(r2,c2)#channel
STORE(r1,c1),(r2,c2)#"file-name"

STORE allows a rectangular area of a graphics screen to be stored in a string and then be redrawn anywhere on the screen using either DRAW or FILL. It will store the rectangle between r1,c1 and r2,c2 either in the string named, in the NEW file opened on the specified channel, or in the file specified (opening and closing is automatic in this last case). After drawing a complicated picture on the screen it may be stored and reprinted anywhere on the screen.

For example:

```
1000 STORE (0,0),(20,40) house$
1010 CLS
1020 DRAW@(40,100) house$
```

Diagrams stored on disk may be retrieved and re-displayed on the screen by using DRAW#.

For example:

```
10 OPEN NEW "ABC" AS 1
20 STORE (0,0),(20,40)#1
30 CLOSE 1
40 CLS
50 OPEN OLD "ABC" AS 2
60 DRAW#2
70 CLOSE 2
```

3.2.94. SWAP

Syntax:- SWAP variable1, variable2

SWAP exchanges the values of two variables of the same type i.e. 2 strings or 2 reals or 2 integers.

For example:

```
10 SWAP A$,B$
```

This exchanges the contents of the two strings.

3.2.95. TEXT

Syntax:- TEXT

TEXT is used to enter TEXT mode so that text files may be edited. Following a TEXT command an option to SAVE a loaded file is given.

Normally after entering TEXT mode an AUTO or a LOAD command would be used. If the AUTO command is given lines of text may be entered one after the other. Line numbers will not be displayed on the screen. Once the text has been entered, a null line will exit AUTO mode. A LOAD command will cause the specified file to be loaded . LISTing a file in TEXT mode will display line numbers but these are not really part of the file, they are for use in editing, (e.g. overwriting existing lines and deleting lines). Lines may be added by using line numbers. A SAVE command in TEXT mode causes the program to be stored on disk without line numbers. To return to normal BASIC mode, use the BASIC command. In TEXT mode, defaults for SAVE, LOAD and MERGE are .TXT . After each line press <ENTER>.

For example:

```
TEXT

 Ready
```

```
AUTO

program SHOW(INPUT, OUTPUT)
begni
watch ('TEST PROGRAM')
end.
<null line to exit AUTO mode.>

LIST

10 program SHOW(INPUT, OUTPUT)
20 begni
30 watch ('TEST PROGRAM')
40 end.

20 begin
     <Will fix this one>
35 watch ('************')
     <Will add a line>

SAVE"TEST.PAS"
     <Will save the program without line numbers.>

LOAD"TEST.PAS"
     <Will load it again>

BASIC
     <Goes to BASIC>

  Ready

DOS  <Goes to DOS>

  DOS
```

3.2.96. TROFF

Syntax:- TROFF

This turns the TRace OFF. See TRON.

3.2.97. TRON

Syntax:- TRON

TRON turns the TRace ON. The trace displays the line numbers as the program is executed and can be useful during debugging. When the trace is on, text will scroll out of sight but fine graphics will not move. Use the <PAUSE> key and <SPACEBAR> to examine the execution of the program line by line. These numbers are printed one after the other, from wherever the cursor happens to be.

For example:

The screen might appear as follows:

<1500><1501><1502><400><401><402><1503>

The program has gone to and returned from a subroutine at line 400.

3.2.98. UNLOCK

Availability:- Not available in Standalone mode

Syntax:- UNLOCK#channel

UNLOCK causes the random file attached to the specified channel to be "unlocked", i.e. it may be accessed by other users.

3.2.99. WAIT

Syntax:- WAIT length

WAIT suspends the program for the specified number of 10 millisecond intervals.

For example:

```
120 WAIT 100
```

suspends the program for 1 second.

Interrupts occurring during a wait are queued until the beginning of the next statement.

4. APPENDICES

4.1. ERROR MESSAGES

NUMBER	MEANING
0	EXIT key pressed
1	Illegal file request
2	The requested file is in use
3	The file already exists
4	File could not be found
5	System directory error
6	System directory full
7	All disk space has been used
8	End of file error
9	Disk file read error
10	Disk file write error
11	File or disk is write protected
12	File is protected, access denied
13	Illegal file control block specified
14	Illegal disk address encountered
15	Illegal drive number specified
16	The disk drive is not ready
17	File is protected, access denied
18	File not opened in the correct mode
19	Data index range error
20	File management system inactive
21	Invalid filename syntax
22	File close error
23	Sector map overflow, the disk is too segmented
24	Non-existent record number specified
25	Record number match error or the file is damaged
26	Syntax error in command
27	Lost communications with the disk drive
31	Illegal software interrupt function
32	Disk drive door was opened while file open for write
33	Cannot lock a sequential file
34	File is locked, access denied
36	Pirated software !
40	Unbalanced parentheses
41	Illegal character or reserved word in statement
42	Source file is not present in memory
43	The line is too long, 255 characters is the limit
44	Syntax error on compile
50	Invalid syntax
51	Invalid syntax in function
52	An invalid character is present at the start of line
53	An invalid statement start
54	An invalid statement terminator
55	A label was expected
56	A numeric result was expected
57	A string result was expected
58	A left parenthesis "(" was expected
59	A comma "," was expected
60	A right parenthesis ")" was expected
61	Missing or invalid item in expression

62	String and numeric expressions mixed
63	Too many temporary strings
64	The array subscript is negative or out of range
65	Incorrect number of subscripts with array reference
66	Undimensioned array referenced or misspelt function
67	The expression result is <0 or >255
68	a string variable was expected
69	Different string lengths
70	RETURN without a corresponding GOSUB
71	NEXT without a corresponding FOR
72	RESUME is not in an interrupt routine
73	Cannot continue, variables have been re-initialised
74	The line number was not found
75	Auto mode will not overwrite existing lines
76	Line number too large
77	Fatal renumbering error
80	Arithmetic overflow has occurred
81	The real number is too large to convert to an integer
82	Cannot calculate the LOG of zero or a negative number
83	Cannot calculate the SQR of a negative number
84	Cannot divide by zero
85	Argument too large
90	Argument out of range
91	Out of data for READ
92	Data type mismatch in PRINT USING
93	Illegal format-string in PRINT USING
94	Attempt to access outside text screen area
95	SWAP arguments must be the same type of variable
96	An invalid parameter in a SWI or USR function call
97	The array has already been dimensioned
98	The FN function has not been defined
99	The array dimension was negative or too large
100	The clock is not running
101	No room for stack, program is too large for memory
102	The memory limit has been set too low or too high
103	Not enough room for a new string or array
120	Non-numeric data in input
121	Number input is too large for integer variable
122	Number input is too large for integer variable
130	Specified file is not a compiled file
131	An invalid channel number was specified
132	The specified channel was not open
133	The specified channel is already in use
134	Invalid use of the FETCH statement
135	Compiled files cannot be merged
136	Sum of field sizes exceeds the declared record size
137	Illegal DOS command from BASIC or TEXT mode
138	Cannot access random files with sequential methods
139	Sequential files cannot be accessed by random methods
140	Random or graphic files cannot have a .PRT extension
150	A graphics screen has not been selected
151	Cannot use a null string for graphics

ASCII Decimal Value		Representation In Strings	Function
0		@ or space	Not used
1		A or a	Starts RED characters
2		B or b	Starts GREEN characters
3		C or c	Starts YELLOW characters
4		D or d	Starts BLUE characters
5		E or e	Starts MAGENTA characters
6		F or f	Starts CYAN characters
7	*	G or g	Starts WHITE characters
8		H or h	Starts FLASHING
9	*	I or i	Ends FLASHING
10		J or j	Not used
11		K or k	Not used
12	*	L or l	Normal height
13		M or m	Double height (see below)
14		N or n	Shift into ASCII characters
15	*	O or o	Shift into Teletext characters
16		P or p	Reverse video on
17		Q or q	Starts RED graphics
18		R or r	Starts GREEN graphics
19		S or s	Starts YELLOW graphics
20		T or t	Starts BLUE graphics
21		U or u	Starts MAGENTA graphics
22		V or v	Starts CYAN graphics
23		W or w	Starts WHITE graphics
24		X or x	CONCEAL display on rest of line
25	*	Y or y	Contiguous graphics
26		Z or z	Separated graphics
27	*	← or ;	Reverse video off
28	*	½ or <	No background to characters
29		→ or =	Set background to current colour
30		↑ or >	Print graphics characters over control characters
31	*	# or ?	Print space for control characters

EACH CHARACTER MUST BE PRECEDED BY A ||.

To include a single || in a print string use || ||.
Each of the control characters takes up ONE screen position except the reverse video on and off characters and the shift characters for ASCII.
All control characters are reset at the beginning of each line to those with an * beside them. Reverse video is switched off at the end of each PRINT.
The control characters in strings are always converted to the first of the two options listed above, i.e. "||a" is converted to "||A".
Double height may be used on screen 1 but not screen 3. Double height characters extend down to the following line. If double height is used anywhere on a line the following line is not displayed. Anything printed on screen 3 "behind" a line containing a double height character will be displayed in normal height on both rows.

ASCII Decimal Value	Function
0	Not used
1	Insert character
2	Delete character
3	Not used
4	Not used
5	Scroll up
6	Scroll down
7	BEEPs the speaker
8	Moves cursor 1 space to the LEFT
9	Moves cursor 1 space to the RIGHT
10	Moves cursor 1 space DOWN and scroll
11	Moves cursor 1 space UP and scroll
12	Clears screen and moves cursor to HOME position (0,0)
13	Moves cursor to start of screen line
14	Not used
15	Starts TELETEXT characters
16	Reverse video on
17	Not used
18	Not used
19	Not used
20	Not used
21	Not used
22	Not used
23	Not used
24	Not used
25	Not used
26	Not used
27	Reverse video off
28	Not used
29	Not used
30	Clear to end of line
31	Initialise line editor

NOTE

To use ASCII screen control characters, they must be preceded by Teletext control character 14 (Shift In) and followed by Teletext control character 15 (Shift Out).

4.4. SPECIAL FUNCTION KEYS

Function Key	ON KEY Value	ASCII Decimal value
Numeric keypad 0	0	48 *
1	1	49 *
2	2	50 *
3	3	51 *
4	4	52 *
5	5	53 *
6	6	54 *
7	7	55 *
8	8	56 *
9	9	57 *
Numeric keypad .	26	46
EXIT	10	26
PAUSE	11	28
ENTER	12	13
NEXT	13	19
REPEAT	14	20
BACK	15	21
HELP	16	
CALC	17	
←	18	08
→	19	09
↓	20	24
↑	21	25
CHAR INS	22	01
CHAR DEL	23	02
LINE INS	24	17
LINE DEL	25	18
SHIFT/PAUSE	27	27
@	28	64
£	29	35
↑ or EXP	30	94
|	31	124

The function keys marked with * may be assigned "soft key" values by using ON KEY <exp> AS <exp>

e.g. ON KEY 4 AS 16

This assigns a new ON KEY value, key 4 now "looks like" a key with the ASCII value of 16.

The HELP and CALC keys cannot be tested by checking their ASCII values. To disable or trap these keys use ON KEY.

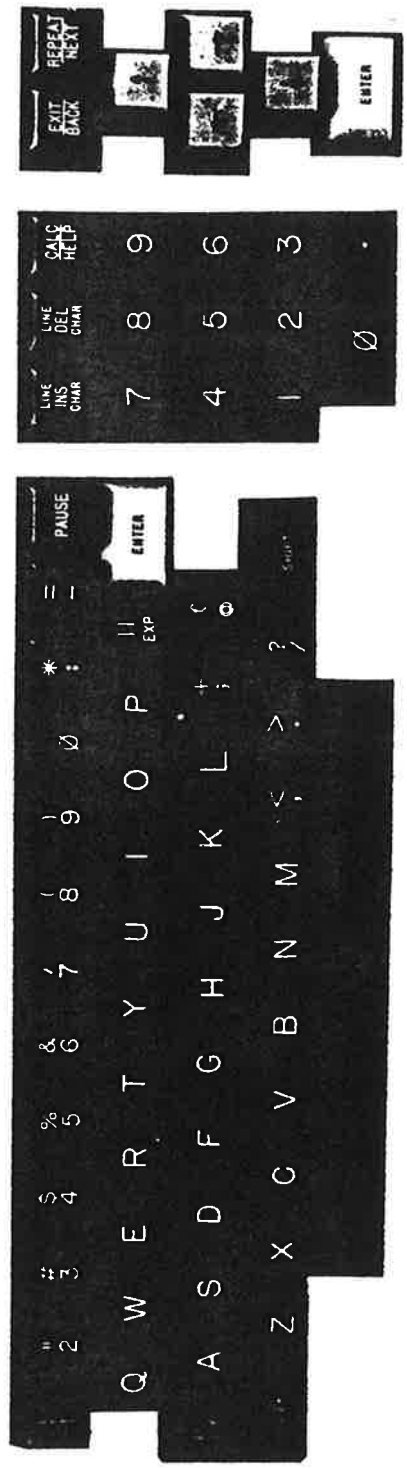
This scale has the A above middle C defined as having a frequency of 440 Hz. N1 is the 'pitch' value to be used with the SOUND statement.

Note	Freq	N1	Freq	N1
A	55	9134	880	570
A#	58	8621	932	538
B	62	8137	988	508
C	65	7680	1047	479
C#	69	7249	1109	452
D	73	6842	1175	427
D#	78	6458	1245	403
E	82	6096	1319	380
F	87	5753	1397	359
F#	92	5430	1480	338
G	98	5126	1568	319
G#	104	4838	1661	301
A	110	4566	1760	284
A#	117	4310	1865	268
B	123	4068	1976	253
C	131	3840	2093	239
C#	139	3624	2217	226
D	147	3421	2349	213
D#	156	3229	2489	201
E	165	3047	2637	190
F	175	2876	2794	179
F#	185	2715	2960	169
G	196	2562	3136	159
G#	208	2418	3322	150
A	220	2283	3520	142
A#	233	2154	3729	134
B	247	2033	3951	126
C(middle)	262	1919	4186	119
C#	277	1812	4435	112
D	294	1710	4699	106
D#	311	1614	4978	100
E	330	1523	5274	94
F	349	1438	5588	89
F#	370	1357	5920	84
G	392	1281	6272	79
G#	415	1209	6645	75
A	440	1141	7040	70
A#	466	1077	7459	66
B	494	1016	7902	63
C	523	959	8372	59
C#	554	905	8870	56
D	587	854	9397	52
D#	622	806	9956	49
E	659	761		
F	698	718		
F#	740	678		
G	784	640		
G#	831	604		

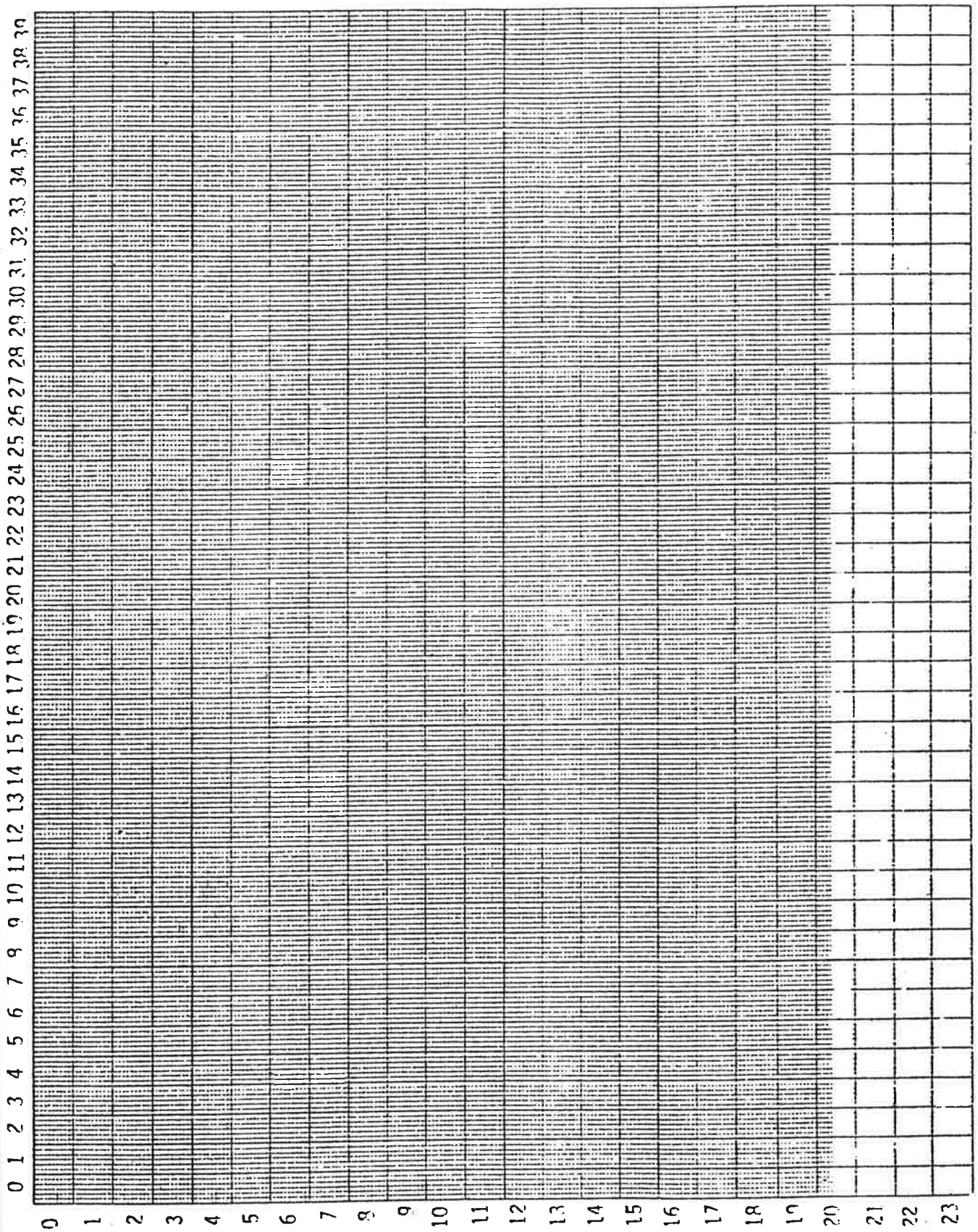
4.6. TELETEXT CHARACTERS AND GRAPHICS

ASCII DECIMAL VALUE	CHAR-ACTER	GRAPHICS	ASCII DECIMAL VALUE	CHAR-ACTER	ASCII DECIMAL VALUE	CHAR-ACTER	GRAPHICS
32	SPACE		64	@	96	-	
33	!		65	A	97	a	
34	"		66	B	98	b	
35	£		67	C	99	c	
36	$		68	D	100	d	
37	%		69	E	101	e	
38	&		70	F	102	f	
39	'		71	G	103	g	
40	(		72	H	104	h	
41	)		73	I	105	i	
42	*		74	J	106	j	
43	+		75	K	107	k	
44	,		76	L	108	l	
45	-		77	M	109	m	
46	.		78	N	110	n	
47	/		79	O	111	o	
48	0		80	P	112	p	
49	1		81	Q	113	q	
50	2		82	R	114	r	
51	3		83	S	115	s	
52	4		84	T	116	t	
53	5		85	U	117	u	
54	6		86	V	118	v	
55	7		87	W	119	w	
56	8		88	X	120	x	
57	9		89	Y	121	y	
58	:		90	Z	122	z	
59	;		91	←	123	¼	
60	<		92	½	124	‖	
61	=		93	→	125	¾	
62	>		94	↑	126	▬	
63	?		95	#	127	■	

4.7. DIAGRAM OF POLY KEYBOARD



4.8. SCREEN LAYOUT CHART



4.9. RESERVED WORDS

Words which are used in POLYBASIC commands, statements or functions must not be in general included in variable names (see RESON, RESOFF). Reserved words may be in either upper or lower case, and must not contain spaces.
The following is a list of the reserved words.

ABS	END	LPRINT	RND
AND	ERL	LSET	RSET
AS	ERR	MEM	RUN
ASC	ERROR	MERGE	SAVE
ATN	EXEC	MID$	SCROLL
AUTO	EXP	MIX	SELECT
BACK	FETCH	MOD	SEC
BACKG	FIELD	NAME$	SET
BASIC	FILE	NEW	SGN
CHAIN	FILL	NEXT	SIN
CHR$	FN	NOT	SOUND
CLEAR	FOR	OFF	SPC
CLOCK	FRE	OLD	SPLIT
CLOSE	GET	ON	SQR
CLS	GO	OPEN	STEP
COLOR	GOSUB	OR	STOP
COLOUR	GOTO	PEEK	STORE
COMPILE	GRAPH	PI	STR
CONT	HEX	POKE	STRING$
CONVERT	IF	POINT	SWAP
COS	INCH$	POS	SWI
CVT	INPUT	PRINT	TAB
CVTF	INSTR	PTR	TAN
DATA	INT	PUT	TEXT
DATE$	KEY	RANDOM	THEN
DEF	KILL	READ	TIME$
DEL	KVAL	RECORD	TO
DIGITS	LDES$	REM	TROFF
DIM	LEFT$	RENAME	TRON
DISPLAY	LEN	RENUM	UNLOCK
DIV	LET	RESET	USING
DOS	LINE	RESOFF	USR
DPEEK	LIST	RESON	VAL
DPOKE	LOAD	RESTORE	WAIT
DRAW	LOCK	RESUME	CLOAD
DRIVE	LOG	RETURN	CSAVE
ELSE	LOGOFF	RIGHT$	

4.10. ADVICE

1. Corrupted Disks - Prevention and Recovery

Corrupted disks can result from files being opened for write and not subsequently closed. If this happens the structure of the free space on the disk is corrupted and subsequent use of the disk may cause problems.

To prevent the corruption of disks, remember the following rules.

(i) Before switching off or resetting a POLY, either LOGOFF or enter CLEAR from BASIC.

(ii) If your BASIC program ends with an error or stops for any reason with files open, execute CLEAR to close all files (if you need to examine variables for debugging purposes do so first).

(iii) Do not reset or switch off the disk unit until all POLYs have closed their files as above.

(iv) Do not remove a disk from the disk unit, or even open the door unless you are sure that all users have closed all the files they have open on that disk.

NOTE A file is open for write if it is opened NEW or RANDOM. A file is also open for write if LPRINT is used (though this is automatically closed if an error occurs) and when SAVE is being executed. Many utilities have files open for write as well so execution of these should not be interrupted.

The RECOVER utility should be used to clean up a disk under any of the following conditions.

(i) If any of the above rules are broken.

(ii) If a disk door opened error is given (the disk which was in the drive whose door was opened should be RECOVERed).

(iii) If, after executing the CAT command, the "SECTORS=" number (the number of sectors used by the files) plus the "FREE=" number (the number of sectors not used by the files) do not add up to 2280 for a double-sided disk or 1140 for a single-sided disk. Note that even if these numbers do add up to 2280 or 1140 as appropriate, the disk is not necessarily uncorrupted. Also note that sometimes when a disk is FORMATted, the "TOTAL SECTORS=" number may be less than 2280 or 1140 as appropriate due to physically crashed sectors (see below). We suggest that you throw away such disks.

2. Crashed Disks

Disks may crash for a variety of reasons - physical damage, wear, dirt, power surges, etc. Normally such disks will contain one or more unreadable sectors and access will result in disk read or disk write errors (errors 9 and 10). Cleaning disk heads every so often may help (cleaning kits are available from POLYCORP) but remember the following rules.

(i) Back up your disks regularly.
(ii) Once you have noticed a disk crash error, do not continue to use the disk.

If you do have a crashed disk then the following procedure will recover as much as possible.

(i) Get a catalog of the crashed disk (if this isn't possible the disk is irretrievable; a partial catalog will allow some files to be recovered).
(ii) Decide which files you want to recover.
(iii) Copy as many of these files as possible, one at a time to another system disk using PCOPY, COPY, or SDC.
(iv) Copying a file containing an unreadable sector will present problems. For example, the copy may stop in the middle, in which case only part of the file will be recovered. Some times the file may have unwanted lines in it and these will have to be edited out. Sometimes, the file may appear infinitely long and an attempt to copy it will continue for ever (this is a bad one, the network controller will have to be reset to turn off the disk drive so the disk can be extracted and the file deleted).
(v) After copying the wanted files, reformat the crashed disk and if you're sensible, back up your recovered disk.

3. Deadly Embrace

Several precautions must be taken when sharing files. If a file is opened for write by more than one program, and one of the programs is to get a record, alter it, and then put it back, then the other programs must be denied access to that record while it is being changed. To accomplish this, the updating program must LOCK the whole file (i.e. all records) before it GETs the required record and then UNLOCK the file after it has PUT the record.

In the simple cases this is all very well, but consider a more complex example.

Program A opens two files, say X and Y, locks X and tries to read Y. In the meantime, program B opens X and Y, locks Y and tries to read X. Using the error trap as in

```
ON ERROR GOTO 1000

1000 IF ERR=34 THEN RESUME
```

the two programs will wait on each other forever. This is known as "Deadly Embrace" and must be carefully avoided.

4. Security

If you have files you want to protect from accidental or purposeful access by other users, log onto the POLY with your initials and a password that you will remember. Then run the PROT utility specifying the P option and the password will be

associated with the file. Future access to the file will only be permitted to users logging on with the same initials and password.

If you forget your password, an OVERRIDE program (with limited distribution) exists so that the password may be removed.