CONTENTS

0. INTRODUCTION

Nascom Extended Basic (XBASIC) is an interpreter written in Z80 machine code which has been developed by Crystal Research. It is based on experience gained with earlier versions of Xtal BASIC and Nascom ROM BASIC. Extended BASIC is significantly larger than both the earlier Xtal BASIC and Nascom ROM BASIC, but includes many new features, and existing features have been extended.

For those with some experience of machine-code programming, the ability to create user-defined reserved words must be one of the most outstanding features of this BASIC. By writing appropriate sub-routines and by inserting your own defined words in an auxiliary reserved word table, you will be able to expand this interpreter to give the type of BASIC most suited to your own needs. We believe that, for the time being at least (and we have not heard of any equivalent in over two years), this feature is unique to BASIC's from Crystal, and it makes it potentially one of the most powerful BASIC's ever available.

Extended BASIC is designed to allow the incorpoartion of disc handling commands, as well as handling cassette tape, and the file handling system has been designed with a view to dealing with both. Although we use the terms 'disc' and 'cassette tape' throughout the manual, it is as well to remember that some forms of tape, such as the 'stringy floppy' or 'floppy tape' are theoretically capable of random- access, and may hold separate 'file directories', i.e, to all intents and purposes they behave as disc drives. We therefore include all such devices under the broad term 'disc drives', to distinguish from the sequential-only 'cassette tape' drives.

Nascom Extended BASIC is available in three forms - tape cassette, NAS-DOS and CP/M. The differences involve only the media and the provision of appropriate disc access commands.

LOADING EXTENDED BASIC ON NASCOM MICROCOMPUTERS (TAPE VERSION)

XBASIC will run on any of the Nascom computers, as long as one of the NAS-SYS monitors is being used. It is supplied on tape in CUTS format, to load at 1200 baud.

To load XBASIC, type R and then press the <ENTER> key. Next, press the PLAY button on the cassette recorder. The program should be observed to load block by block until, after about two minutes, loading should be complete. XBASIC occupies the area 1000H to 40FFH (about 12 1/4 K).

To run, type in E1000 and press the <ENTER> key. This is the initialising, or 'COLD', entry to XBASIC. A 'WARM' entry is also allowed from the monitor into XBASIC by typing E1003 <ENTER>. This preserves any current BASIC program and variables. This entry point should not, however, be used unless XBASIC has already been previously entered by a COLD start.

NOTATION

In order to simplify the use and understanding of this manual and, in particular, the command and function descriptions, we have adopted a notation that explains the syntax requirements of each command/function. This consists of a single letter, which may or may not be followed by a number, enclosed thus: < >. If a command/function name has to be followed by an expression, this notation will show the type of expression that is allowed:

J — An expression, which must evaluate to a number in the range 0 to 255. If it is not an integer, the decimal part of the number will be chopped off, the integer part only being used.

I — An expression, which must evaluate to a number in the range -65535 to +65535. Again, only the integer part is actually used. In some cases, the range is restricted to 0 to 65535, or even 0 to 32767 (e.g, Array elements), but mention is made only when those cases apply.

L — A line number, in the range 0 to 65535. This must be a number only, and so may not be given as a variable.

N — Any numeric expression.

E — Any expression, whether numeric or string.

S — Any string expression.

F — A string expression, which must evaluate to give a legal file name (as defined in Chapter V.2).

U — A numeric variable, which may not be an array element.

V — A variable name, which may be of numeric or string type, and may be an array element.

SV — A string variable name, which may not be a string array element.

X — A complete Xtal BASIC statement.

Examples:

1. In Chapter III.4 we find the LEFT$ function described thus: LEFT$(<S>,<J>)

This means that LEFT$ must have two arguments separated by a comma, and enclosed within parentheses. The first argument must be a legal string expression, and the second argument must be a number in the range 0 to 255 (reasonable, since we cannot have strings longer than 255 charac- ters).

e.g. LEFT$("NAME "+X$,7) is legal.

2. In Chapter III.1, the ON..GOTO command is shown:
ON <J> GOTO <L1>,<L2>,..,<Ln>

This means that ON must be followed by a number in the range 0 to 255 followed by the word GOTO followed by one or more line numbers <L1> to <Ln>. Each of these line numbers (if more than one) must be separated by a comma.

e.g. ON X GOTO 1000,2000,3000 will simply drop to the next line if X is 0 or greater than 3, otherwise a GOTO will be executed to one of lines 1000, 2000 or 3000 according to the value of X being 1, 2 or 3 respectively.

I. GETTING STARTED WITH XBASIC

1. RUNNING UP XBASIC

Having loaded XBASIC from your tape or disc, you should be rewarded with the 'sign-on' message, i.e:

```
Nascom Enhanced BASIC Rev xx
(C)1982 Xtal
Size: yyyyy
Ok
```

where xx represents the sub-version for your machine, and yyyyy the memory size available for storage of BASIC program and variables. The Ok prompt shows that XBASIC available and is not running a program, but is waiting for a command to be typed in at the keyboard. Virtually any of the commands or statements listed in the following chapters may be typed in and executed, along with a number of special commands given in Chapter II, known as 'SYSTEM' commands. For example, we may use the machine as a calculator:

```
Example:
PRINT ATN(1)*4      You type this line
3.14159             Computer prints the result of 4 times the arc-
                    tangent of 1
```

This is known as DIRECT execution mode, since commands are executed DIRECTly they are typed.

Alternatively, commands and statements may be entered without being executed, by typing a line number in front of it. A sequence of one or more lines entered in this way forms a PROGRAM, which may be executed by means of the RUN command (see below). This is known PROGRAM mode.

Line numbers may range between 1 and 65535, and may be followed by one or more commands. Each line so entered is automatically placed in order, with line 1 at the front. Line numbers may be selected arbitrarily by the user, but it is recommended that reasonable gaps be left (say, 10) between lines, so that extra lines may be inserted if these are later found to be necessary, in the development of a program. The program may begin with any line number, but the first line to be interpreted will always be the lowest line number entered.

A line may be deleted by entering its number followed by <ENTER>, with no other information.

Several commands may be entered on a single line by separating them with colons:

```
Example:
10 PRINT 2*13 : PRINT 5+6       You type these
RUN                             lines
26                              Computer responds with the answers
11
```

Separation in this manner allows several commands to be entered in DIRECT mode as well as in a program.

2. NUMBERS AND STRINGS

There are two types of quantity allowed in XBASIC - numbers and strings. Numbers may also include floating-point numbers, integers, and hexadecimals.

2.1 Numbers

These can be whole numbers (integers) or floating-point numbers (reals). A number is stored internally as four bytes, one of which represents a signed exponent, while the other three represent a signed mantissa. This gives an exponent range from -38 to 38, with a seven-digit signed mantissa. Although the full seven digits are available for internal calculation, they are rounded off to six figures when output, the seventh figure being known as a 'guard' digit. When accuracy is at a premium, the seventh digit should always be used, if known, since Xtal BASIC can make use of it, even though only six significant figures are displayed.

> Example:
> The PI function actually uses 3.141593, even though it displays as 3.14159 (to show this, try PRINT PI-3).

Leading and trailing zeroes are suppressed on output, so that integers are actually printed as such without long rows of zeroes.

> Examples:
> 3 3.14159 314.159 .0314159 3.14159E+08 -3.14159E-37

These are all possible forms in which numbers may be output. The last two, for those not familiar with them, are in SCIENTIFIC notation, a form only used when the output is too large or too small to be conveniently printed in any other way. Numbers may be INPUT in this form, if required.

2.2 Integers

XBASIC supports 16-bit integers, i.e, whole numbers in the range -32768 to +32767. Integers outside that range may only be stored in ordinary numeric variables (see next section, on variables), but integers in this range may be stored internally in two bytes instead of four (for simplicity of design of the interpreter, we actually store integers in four bytes for simple variables, and two bytes per element within arrays, where the greatest savings may be made).

We use an additional convention with integers, however, that numbers in the ranges -65535 to -32769, and 32768 to 65535, may be accepted by integer variables, since it is often useful to do so. In these cases, the values are internally converted to lie in the ranges 1 to 32767 and -32768 to -1 respectively (since we should otherwise need 17 bits to store such numbers).

2.3 Hexadecimal numbers

The allowance of hexadecimal numbers in XBASIC greatly increases the ease of linkage to machine-code routines and locations, and they may normally be used in any expressions requiring numeric quantities. The only limitations are that only integers are allowed for, in the range &0 to &FFFF. To indicate a

hexadecimal number, a leading ampersand '&' symbol is supplied, followed by a string of characters, each of which may a number in the range 0 to 9, or a letter in the range A to F. When encountered within a numeric expression, a hexadecimal number is internally converted into a decimal integer and the result of such an expression will still always be a normal number. The hexadecimal number may consist of 1 to 4 digits. More may be entered, but all except the last four will then be ignored.

Examples:
&1F34 represents 7988 in decimal.
&A7 represents 167 in decimal.
&91F34 still represents 7988 in decimal **(the first digit is ignored).**

To obtain the hexadecimal equivalent of a decimal number, the HEX$ function may be used (see Chapter VII.1).

2.4 Strings

These are combinations of ASCII characters representing letters, numbers and symbols, useful for storing names, titles and text, although their intrinsic data can be extracted by the interpreter and they are frequently used to hold numeric values as well. A string can be any combination of up to 255 characters, usually shown in quotes " " in order to prevent confusion with numbers or variables.

Examples:
"TREVOR" "Trevor" "12345.6" "Oh! *$%" are all valid strings.

3. VARIABLES

A variable is a combination of letters and/or numbers, the first character being a letter. XBASIC distinguishes the first FIVE characters (most BASICs distinguish only the first two). Variables may be of either numeric, integer or string type, and hold numbers, whole numbers and strings respectively. Integer variables must be suffixed with a '%', and string variables with a '$'. There is no theoretical limit to the length of a variable name, although the length of the input line will clearly limit it!

Examples:
A AA X9% Z9J% X$ F4$ ABCD$ AB123$ KRAZY KRAZY10

are all valid variables,although BASIC would be unable to distinguish between the names of the last pair, since their first five characters are the same.

Care must be taken to ensure that variable names do not contain reserved words, otherwise SYNTAX ERRORs will result.

Examples:
TONE LETTER COST EXPENSE PINCH TERROR TO LET COS
EXP INCH ERR (and OR)
will all cause problems.

Keeping variable names to two characters will solve this problem (the only 2-character names in XBASIC are IF, TO and FN) and this also saves space.

Integer variables may contain only integers in the range -65535 to +65535, but they may also contain hexadecimal numbers. However, values returned by integer variables are in the range -32768 to +32767, using the most-significant bit as a SIGN flag (these may be regarded as sixteen-bit numbers).

Example:

```
A%=65535: PRINT A%        will display the value -1
LOC=&9678: PRINT LOC      will display the value 38520
LOC%=&9678: PRINT LOC%    will display the value -27016
```

4. ARRAYS

In addition to simple numeric and string variables, we can use numeric and string arrays. An array is, in effect, a table full of variables, each of which can be uniquely identified. Naming of arrays takes exactly the same form as for simple variables, except that they are followed by a set of one or more subscripts, each subscript representing one of the dimensions of that variable.

Examples:

```
A(0)    TABLE%(5,6)    NAME$(1,0,2)
```

are all valid arrays, where A is an array of one dimension, the subscript (0) referring to the FIRST element. TABLE% is a two-dimensional integer array and NAME$ is a three-dimensional array holding strings, each of which may be up to 255 characters in length.

In XBASIC all array subscripts number from zero.

In order for BASIC to know how much space to allocate to an array, the array in question must be dimensioned with a DIM statement (see Chapter III.1) before being brought into use. However, if all subscripts in an array have maximum values of 10 or less, then that array may be used without a DIM statement.

Example:

```
AA(7,4,6)=56
```

will dimension that array exactly as though the following had been written:

```
DIM AA(10,10,10):AA(7,4,6)=56
```

assuming that no previous DIM statement has been used for AA. This array will have 11*11*11=1331 elements, requiring over 5320 bytes to store it! NOTE: If we had used an integer array AA%, we should only require about 2670 bytes to store it.

5. EXPRESSIONS

Expressions consist of variables, numbers, string variables or strings in any combination, and related by means of arithmetic and/or logic operations.

5.1 Arithmetic operators

The arithmetic operations allowed in XBASIC are as follows:

```
+ (add)          - (subtract)     * (multiply)     / (divide)
↑ (raise to power)                MOD (remainder)
```

The ↑ operator has the following conventions:
X ↑ 0 = 1 for X>=0, and 0 ↑ Y = 0 for Y>0 (so 0 ↑ 0 = 1!).
X ↑ Y is undefined for X<0 or for X=0 and Y<0.

The MOD operator is the remainder from a division, and can be defined as follows:

```
X MOD Y = X - Y * INT(X/Y)
```

Examples:
5 MOD 3 returns 2
-5 MOD 3 returns 1 (see definition of INT, Chapter III.3).

5.2 Relational and logical operators

RELATIONAL operators are used for comparisons and the evaluation of conditions, particularly for IF statements. The ones allowed are:

```
> (greater than)          >= (greater than or equal to)
< (less than)             <= (less than or equal to)
= (equal to)              <> (not equal to)
```

LOGICAL operators allowed are:

```
NOT     AND     OR      XOR (exclusive-OR)
```

Example:

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

Expressions involving relational operators and logical operators are normally used within IF statements (Chapter III.1), but can also be used within normal arithmetic expressions, since a relational expression returns a value -1 if it is TRUE, and 0 if FALSE. In some cases, quite a lot of space can be saved.

Example:

```
IF X>15 THEN A=0: ELSE A=1      can be replaced by:
A= -(X>15)
```

5.3 Bit manipulation

Logical operators may also be used for bit manipulation, provided that the sub-expressions on either side evaluate to results in the range -65535 to 65535 (i.e, they can be thought of as sixteen-bit quantities). Then AND, OR, XOR and NOT will all work upon the individual respective bits of the two expressions.

Example:

```
PRINT 1234 AND 3412     outputs the result 1104:
0000 0100 1101 0010 (1234 = &04D2)
0000 1101 0101 0100 (3412 = &0D54)
0000 0100 0101 0000 (1104 = &0450)
```

5.4 Operator precedence

Operator precedence follows the usual mathematical order. We also include the relational and logical operators here, so that they may be used within arithmetic expressions with the correct precedence:

Highest precedence:	() (parentheses).
	^
	* / MOD
	+ -
	< <= = <> >= >
	NOT
	AND
Lowest precedence:	XOR OR

5.5 String expressions

XBASIC also allows string expressions, but the only operator is CONCATENATION, represented by '+'.

```
Example:
A$="ABC": B$="DEF": C$=A$+B$: PRINT C$
outputs the result
"ABCDEF"
```

String comparison may also be performed for alphabetic sorting, since "B">"A", for example. In comparing two strings, the comparison is done character by character, until a position is found in which the two differ. The 'greater' string is then the one whose character has the greater ASCII code. If no differences are found, but one string is longer than the other, the longer string is considered to be the greater.

```
Examples:
"GOLIATH" is greater than "DAVID"
"ANDY" is greater than "ANDREW"
"BROTHERHOOD" is greater than "BROTHER"
"Hello" > "Goodbye" returns the numeric result -1 (true).
```

II. THE SYSTEM EDITOR AND SYSTEM COMMANDS

1. SCREEN CONTROL CODES

The following VDU control codes are used by XBASIC on the standard 48x16 VDU. Note, incidentally, that all 16 lines scroll when running XBASIC.

Ctrl-A	&01	HOME cursor to top left corner of screen.
<TAB>	&09	TAB cursor to next print ZONE, by printing spaces. However, see also IOM command in Chapter IV.3.
<LF>	&0A	LINE FEED, or move cursor DOWN. Scroll screen at bottom.
<CS>	&0C	CLEAR SCREEN and Home cursor to top left corner.
<CR>	&0D	CARRIAGE RETURN, without line feed.
Ctrl-P	&10	PRINT SCREEN to printer (device £1, see Chapter IV.1).
Ctrl-Q	&11	Move cursor LEFT.
Ctrl-R	&12	Move cursor RIGHT.
Ctrl-S	&13	Move cursor UP.
Ctrl-T	&14	Move cursor DOWN (same as <LF>).

2. THE XBASIC EDITOR

This powerful facility, available to you the moment that XBASIC is entered, has been designed in an attempt to make program entry and debugging more of a pleasure rather. Input lines may be up to 127 characters long, and note is kept at all times of where the start and finish of the line is. So, if you have several lines in a listing, you may move the cursor up the screen to that line and make modifications to it, even if it occupies two or more rows on the screen. If the line is extended so that it will apparently run into the next one, the lines below simply move down one row to make room for it. Note that the modified line is only entered into the program when the <ENTER> key is pressed while the cursor sits in one of the rows of the screen containing that line.

The following special key functions are available, the ones in brackets indicating the equivalents for the Nascom 1 keyboard:

Ctrl-A (@A)	HOME cursor to top left corner of screen.
s<BS> or <CS>	CLEAR screen and Home cursor.
' ' (@R)	Move cursor RIGHT.
' ' (@Q)	Move cursor LEFT.
' ' (@S)	Move cursor UP.
' ' (@T)	Move cursor DOWN (scroll screen at bottom).
<BS>	DELETE character to the LEFT, but moving rest of line one place to the left.
s' ' (@U)	DELETE character from the RIGHT, moving rest of line one place to the left.
s' ' (@V)	INSERT space at cursor, moving rest of line one place to the right, and moving lines below it one row down, if required. NOTE: An insertion done at the bottom line of the screen will cause an immediate scroll, moving the cursor up with it. This has no ill effects, apart from being a bit disconcerting when first observed.

Ctrl-W (@W)	ERASE whole line. This differs from Ctrl-X in that the cursor is returned to the start of the line before clearing it.
Ctrl-X (@X)	ERASE to end of line (even if it occupies 2 or more rows), the current cursor position.
Ctrl-O (@O)	ERASE to end of screen from current cursor position.
Ctrl-P (@P)	PRINT SCREEN contents to printer.
<ESC> (s<NL>)	Abandons a line (though you could just use an arrow key or a Ctrl-W!) and prints the 'Ok' prompt.
<CR> or <NL>	ENTER the current line on which the cursor sits into BASIC. cursor will end up sitting at the start of the next line (i.e, not necessarily the next ROW of the screen). Leading and trailing spaces are ignored, and lines of greater than 127 characters will be truncated to 127 (this being the size of the buffer area).

If this is all as clear as mud(!), the best thing to do is to 'play'!

3. THE LINE EDITOR

In addition to the **screen editor**, a **'Line edit' mode** is also available, primarily for use within programs, when to use the screen editor could cause some irritation (since the INPUT prompt would also be assumed to be part of the input line! On the other hand, this could also be very useful in certain applications).

In this mode, cursor movement keys are not available, except that ' ' and <BS> both delete the last character from the line, Ctrl-P still gives a screen dump to printer, <ESC> abandons the line, and <CR> enters it into BASIC.

For the reasons outlined above, screen edit mode is 'switched on' automatically in direct mode, and LINE EDIT mode turned on for programs. In addition, the user may use the IOM command (see Chapter IV.3), inside or outside a program to change the editing mode: IOM 0,1 gives SCREEN EDIT mode, IOM 0,0 gives LINE EDIT mode. In direct mode, IOM 2,0 should be used before IOM 0,0 , otherwise screen edit mode will be reselected on completion of the statement.

LINE EDIT mode shows itself by means of a prompt at the start of the line (']' in direct mode and '?' in an INPUT statement with no specified prompt string).

SPECIAL NOTE: In spite of the declaration above that lines are limited to 127 characters in length, it is possible to move the buffer area to other areas in the memory space, and to change the buffer length up to 254 characters maximum! This may be done by means of the PTR command (see Chapter VII). Care must then be taken over selection of the area used to contain the buffer, and it is recommended that an area created by means of a CLEAR command be used.

4. SYSTEM COMMANDS

The following commands are normally intended for use in direct mode, although some (such as RUN and LIST) can also be used to advantage within programs, and CHAIN is used almost entirely within programs. Because they all affect modification and overall control of programs and of the system, they are all referred to as SYSTEM commands:

MON Takes control back to the operating system, or the monitor. This is the command to use when you wish to leave XBASIC.

NEW Causes all program lines and variables, if any, to be deleted.

DEL <L1>,<L2> Deletes all lines from the program in the range <L1> to <L2>. Both start and finish lines should be specified, but will default to 10 if not given! If <L1> is larger than <L2>, or if <L1> is larger than the largest line present, a RANGE ERROR will occur.

Example:
DEL 100,199 deletes all lines with numbers from 100 to 199 inclusive

LIST <I1>,<I2>,<I3> Lists the program to the current output device. The listing starts from the first line after <I1> and ends at line <I3> or the first line after <I3> if that line is not present. <I2> gives the number of lines to list at a time. After <I2> lines have been listed, there will be a pause. The user then presses a key, and the listing continues with another <I2> lines (except for some special keys, given in note (iii)). Any or all of the expressions may be omitted, but the appropriate commas should be present if <I2> and/or <I3> are specified.

Examples:

LIST	Lists whole program
LIST ,5	Lists whole program, 5 lines at a time
LIST 100,7	Lists 7 lines at a time starting from 100
LIST 200	Still lists 7 lines at a time, starting from line 200
LIST 100,,199	Lists 7 lines at a time from 100 to 199
LIST ,4,299	Lists 4 lines at a time from the start to line 299
LIST ,,199	Lists 4 lines at a time from the start to line 199
LIST 300,5,999	Lists 5 lines at a time from 300 to 999

Notes:

(i) BASIC remembers the last value of <I2> given and keeps using it until LIST is used with a different value. When BASIC starts up, <I2> is assumed to be 65535, until given.
(ii) A listing may be abandoned at any time, whether paused or not, by pressing <ESC>.
(iii) When paused, the cursor movement keys may be used to abandon the listing and, at the same time, move the cursor in the direction of the key pressed. This only works in SCREEN EDIT mode (see section 3 of this Chapter). The purpose of this is to allow quick exit to the Editor, without having to remember to press <ESC> first!
(iv) Unlike many BASIC's, the LIST command may be used within a program as a normal statement, and note also that <I1>, <I2> and <I3> may all be EXPRESSIONS. This can be extremely nice!

Example:
X=100: Y=50: LIST X,Y,X+99 Lists from line 100 to 199, 50 lines at a time.

AUTO <L1>,<L2> Automatic line-numbering while entering programs. This command requires a start line <L1> and increment <L2>, and both of these default to 10 if not given.

Examples:

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

Each line number is displayed just as if it had been typed from the keyboard, and the user may enter the usual program statements at that point. On pressing <CR> in that line, the text is entered with its line number in the usual way, and then the next line number appears. The user then continues with this line. When finished, just type <ESC> to abandon, whereupon normal direct mode will be re-entered. Any error (BRANCH ERROR is common, when the user just presses <CR> without entering any statement and the line does not exist) will also cause a return to normal direct mode. The editing mode is not affected by this command.

LOAD <F> Loads a file from tape OR disc (depending which is available, either if both are available) whose file name is <F>. The file name convention is described in full in Chapter IV.4, so the user is referred to that.

Examples:

LOAD "TEST" Loads the program file "TEST.XBS" from the current default disc or tape drive. Any existing program in memory is deleted, but note that variables are NOT destroyed.

LOAD "B:TEST.ASC" Loads the ASCII program file "TEST.ASC" from disc drive B, whatever the current default drive. In this mode, the user may actually observe the file loading, appearing line-by-line on the screen. Again, variables are NOT destroyed, but neitner is the existing program. Thus the user may add extra routines to existing programs, and the added lines will appear at their correct positions in relation to those already present. Note, nowever, that if a new program is to be loaded as a .ASC file, a NEW command must first be executed.

LOAD "T:ROUTINES.OBJ" Loads the machine-code routines or data from the file "ROUTINES.OBJ" on tape drive T, into the area previously reserved for them in the memory map (by means of the CLEAR command). The start address will be assumed to be the first location above this CLEARed area (e.g, if a CLEAR &9FFF has been done, the file will load starting at &A000). See also Chapter VII.2.

In all three cases, if the size of the file is larger than the area available, a MEM FULL ERROR will occur. If the file is not present,a NO FILE ERROR will occur if a disc drive is being searched, while no result will be returned if a tape drive is being searched - the user simply has to abandon the tape load, as explained in Appendix B.

If a type other than XBS, ASC or OBJ is specified, a FILE TYPE ERROR will occur (this also applies to SAVE. If it is desired to load or save data files, use the file access commands described in Chapter V).

SAVE <F> As for LOAD, but saves a file named <F> to tape or disc.

Examples:
SAVE "T:TEST" Saves the program file "TEST.XBS" to tape drive T, whatever the current default drive.
SAVE "TEST.ASC",<I1>,<I2>,<I3> Saves the program in ASCII format from lines <I1> to <I3>. The value of <I2> has no effect here, but should be a legal integer quantity 0-65535. The format is, in fact, like that of LIST, except that nothing appears on the screen, and NO pauses are made at every <I2> lines.
SAVE "TEST.ASC" by itself will save the whole program in this form.
SAVE "A:MCSTUFF.OBJ",<I1>,<I2> Saves the area of memory starting from <I1> and ending at <I2> to disc drive A. Both <I1> and <I2> MUST be specified, and <I2> must be larger than <I1>, otherwise nothing will actually be saved. Although intended for saving routines for use in the 'machine-code area' (see memory map, Appendix B), there is no restriction on the actual area of memory saved.

VERIFY <F> Verifies the file named <F> on tape or disc, reporting a checksum error as a BAD DATA ERROR. This command works in the same way as LOAD, except that program files are not loaded into memory, but treated as if they were data files. Any valid file name may be specified. As for LOAD, if an attempt to verify a non-existent disc file, a NO FILE ERROR will result.

CLEAR <I1>,<I2> Clears all variables and arrays from the system, and clears out all strings.

When specified, <I1> and <I2> set up the the topmost location of memory available to BASIC (<I1>) and size of the stack (<I2>), additional to clearing the variables.

The stack is usually 256 bytes, and may not be set to a smaller value.

It will not normally be necessary to increase the size of the stack, unless a large number of nested FOR loops, subroutines and expressions are used (if you encounter STACK FULL ERRORs, this is usually because subroutines are being entered and not RETURNed from (i.e, something naughty is being done!). If <I1> is not specified, the stack size will remain unaltered.

The top of memory is set in order to leave space for OBJ files, that is machine-code routines or data. Normally, none is reserved, and the value reserved is left unchanged if <I2> is omitted. <I2> may not be set above the top of the RAM space - any attempt to do so, or to set it too low, or to set too large a stack size, will result in a MEM FULL ERROR.

Examples:
CLEAR ,500 sets 500 bytes of stack space.
CLEAR &7FFF sets the top location of RAM for BASIC programs and variables to &7FFF, so that machine-code stuff can be placed in the area from &8000 up. The stack size is unaffected.
CLEAR &AFFF,300 sets 300 bytes of stack space, and the top location to &AFFF.

RUN Begins execution of the program currently in memory, starting at the lowest line number, and clearing all variables. The following variations are also available:

RUN <L> - Begins execution at line number <L>.
RUN <F> - Equivalent to a LOAD <F> followed by a RUN. This can be used within a program as well, to link from one program into another.

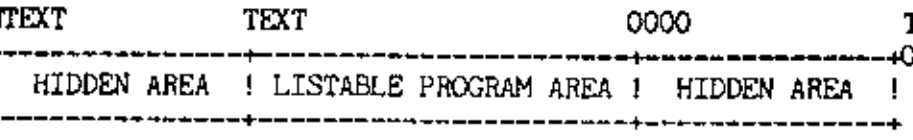
CHAIN Exactly the same as RUN, except that, in all three variations all variables are preserved, and can thus be passed from one program to another. This is an extremely useful command, particularly when it is desired to run an extremely large application, which may be split into several smaller programs sharing the same variables. See also section 4 of this chapter for a discussion of applications of this command.

HOLD <L1>,<L2> 'Holds' a range of lines for view in a program, so that another program may be appended to it, or so that this range may be renumbered, and thus moved to another part of the program. The effect is that the rest of the program seems to have disappeared. In fact, it is still present in memory, but cannot be found by a LIST command, nor executed by RUN, etc. Both <L1> and <L2> may be omitted, their default values being 0 and 65535 respectively. Thus, HOLD by itself has no effect.

Examples:

HOLD 100,199	Leaves only lines 100-199 inclusive 'in view'.
HOLD 100	Leaves all lines from 100 up in view.
HOLD ,199	Leaves all lines up to and including 199 in view.

What actually happens is rather 'sneaky'. The normal text pointer TEXT is moved up to point to the start of line <L1>, while the 'start of next line' pointer held within the line immediately above <L2> is set to a pair of nulls. Note that TXTTOP is still pointing to the real end of text. The program memory map then looks like this:



Note that the listable area can be modified and even RUN without affecting the hidden areas. LOADing another program DOES destroy the upper hidden area, but does not affect the lower one.

MGE Restores sanity to a 'held' program. MGE does not just replace text correctly and restore the removed line pointer - it does a true 'merge' of the held area, so that the lines of the resulting program follow their correct order. MGE takes no account of two or more lines having the same line number, and both lines would then appear in the text together.

RENUM <L1>,<L2> Renumbers a 'held' program, or the whole of it if no HOLD command has previously been used. <L1> is the new starting line, and <L2> the increment. All references following GOTO, GOSUB, RUN, THEN, ELSE and RESTORE commands are modified to their new line numbers. Only the line numbers within the listable area (see HOLD) are modified, but references to modified lines are checked throughout the whole program. This means that, by using HOLD, followed by a RENUM, and finally doing a MGE, whole sections of the program may be moved into a different area of the program.

Note that both <L1> and <L2> may be omitted, each defaulting to 10, as for the AUTO command.

Examples:

RENUM 1000,5 Renumber, making the first line become 1000, and incrementing in 5's.

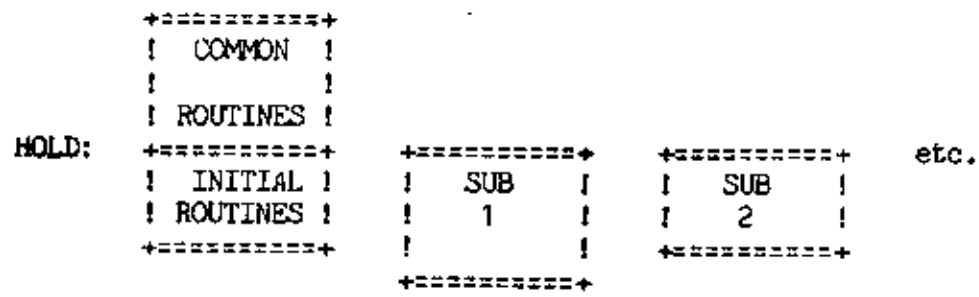
RENUM 500 Make the first line 500, increment in 10's.

RENUM ,20 Make the first line 10, increment in 20's.

5. CHAINING AND 'SEMI-CHAINING' PROGRAMS

The RUN & CHAIN commands have already been mentioned in the previous section. In addition to allowing the use of RUN and CHAIN commands from direct or deferred mode, Xtal BASIC 3 allows the 'semi-CHAIN' of programs, so that several programs may use a common 'pool' of sub-routines, without having to keep the same set of routines within each sub-program. This saves file space, and greatly improves the efficiency of a CHAIN, by speeding up the loading of each sub-program.

To do this, we use the HOLD command before executing a RUN or CHAIN. The RUN and CHAIN commands both restore a 'held' program by setting TEXT back to HTEXT as soon as the program has loaded. However, execution of the resulting program will commence at the start of the added section, NOT at the start of the program. The only restriction is that the common sub-routines must have line numbers smaller than those in any of the sub-programs. The following simplified memory map should help to explain what we are trying to do:



By 'INITIAL' routines, we mean those which set up arrays, variables and memory space, such as DIM and CLEAR statements, which only need to be executed once (indeed, the initial routines could be contained in a separate sub-program which would CHAIN to that containing the COMMON routines). As may be seen, just ONE sub-program actually contains the common routines. When SUB2 or SUB3 are CHAINED, they may use the common routines, just as SUB1.

III. COMMANDS, STATEMENTS AND FUNCTIONS

1. COMMANDS/STATEMENTS

There now follows a list of commands and statements available in XBASIC in its unmodified (by the user) version:

CLS Clears the screen on the current output device, or sends a form feed code, if the output device is a printer.

CONT Causes an interrupted program to resume without clearing the variables. It may be used after a program has terminated with a STOP command. During the stopped period, the user may look at or alter variables without doing any harm, although any attempt to modify the program itself will cause a CONT ERROR to occur. CONT may also be used after a program interrupt using <ESC>. This is a particularly useful aid to debugging in, for example, the tracing of an infinite loop.

DIM This is used to reserve storage for numeric or string arrays. It takes the form DIM A1(I1,I2,..,In),A2(..),...,An(..) , where A1 to An are names of one or more arrays, and I1 to In are numeric expressions in the range 0-65535 representing the maximum size of each dimension in the array. If an array is referenced without having first been dimensioned, it is assumed to have a maximum subscript of 10 for each dimension referenced.

The DIM statement thus defines the amount of storage, the number of dimensions and the size of each dimension in the array.

An array may not be dimensioned more than once in each program - an attempt to do so will result in a DIMENSION ERROR.

END Terminates execution of a program. It is not strictly necessary when the end of the program coincides with the end of the highest line number.

FOR <U>=<N1> TO <N2> STEP <N3> Allows us to set up program LOOPs, for the repetition of sequences of one or more statements.

> <U> is known as the CONTROL VARIABLE, which MUST be a simple numeric variable.
> <N1> is the INITIAL VALUE to which the control variable is set.
> <N2> is the LIMIT VALUE, which, when passed, ends the loop.
> <N3> is the optional STEP VALUE, which is the amount by which <U> is changed on each iteration of the loop. If STEP <N3> is omitted, a step value of 1 is assumed.

The statement(s) within the loop follow(s) the FOR statement. To indicate the end of the loop, we use the NEXT statement,which takes the form:

NEXT <U1>,<U2>,..,<Un> where <U1> to <Un> represent control variables of n nested FOR loops, and is equivalent to the sequence of statements

NEXT <U1>: NEXT <U2>: .. : NEXT <Un> .
NEXT <U> adds the value of <E3> (or 1, as the case may be), and then compares <U> with <N2>. If <U> is greater than <N2> (or LESS, if <N3> was negative), execution continues on after the NEXT statement, otherwise execution transfers back to the statement immediately following the FOR statement corresponding to <U>. If <U> does not correspond to an active FOR loop, a NEXT ERROR will occur, otherwise, if it does not correspond to the last FOR statement, that one will be abandoned, as will any others, until the specified one is found. Note: If no variable is specified, the last FOR statement is assumed to be the desired one.

Example 1: We may wish to print out square roots of numbers between 1 and 10. We can do this:

```
10 FOR I=1 TO 10
20 PRINT SQR(I)
30 NEXT I
```

Example 2:

```
 5 DIM A(7,7)
10 FOR X=0 TO 7
20 FOR Y=0 TO 7
30 A(X,Y)=5
40 NEXT Y,X
```

When RUN, this routine sets all elements of an 8x8 array A to 5 (line 30).

GOTO <L> Transfers program execution to line <L>. If <L> does not exist, a BRANCH ERROR will occur.

GOSUB <L> Transfers program execution to line <L>. Execution continues from there until a RETURN statement is encountered, whereupon execution is returned to the line immediately following the original GOSUB statement. In this way, subroutines may be implemented. If <L> does not exist, a BRANCH ERROR will occur.

RETURN Terminates a subroutine accessed by a GOSUB statement. If a RETURN is encountered without having been preceded by a GOSUB in this way, a RETURN ERROR will occur.

POP Removes, or 'pops' one address off the stack of GOSUB addresses, so that the next RETURN will branch one statement beyond the SECOND most recently executed GOSUB. As with RETURN, a RETURN ERROR will occur if no GOSUBs are currently active.

IF Allows the evaluation of conditions, so that the machine may make a choice depending on whether a condition is true or false. The most general form is: IF <N> THEN <X1>: <X2>: ..: <Xn>: ELSE <Xn+1>: ..: <Xm>

The expression <N> is evaluated and, if non-zero (TRUE), execution continues with the statement(s) <X1> to <Xn> following THEN. In this case, the ELSE statement and the rest of the line after it is ignored. If <N> IS zero (representing FALSE), execution continues with the statements following ELSE, ignoring the ones between THEN and ELSE.

ELSE is optional, and execution transfers to the next line if <N> is false and ELSE is not in the line. ELSEs may not be nested (or rather, they MAY be, but the result will be that only the first one will have any significance. The following DOES, however, work: IF .. THEN ... ELSE IF .. THEN .. ELSE ..

Other forms of IF statement allowed are:

IF <N> THEN <L1> ELSE <L2>
IF <N> GOTO <L1> ELSE <L2>, which is equivalent. Again, ELSE is optional in both cases. If <N> is true, execution transfers to line <L1>, otherwise to <L2>.

It is also possible to mix the two forms, replacing either <L1> or <L2> with statements (if <L1> is replaced by statements, there MUST be a statement separator (:) between the last statement and the ELSE), but a line number must, of course, follow the GOTO, if that form is used.

INPUT Used for getting input, from the keyboard, from a file, or from some other input device. The last two are described in Chapters IV and V.

The usual form is as follows:
INPUT "<Prompt>"; <V1>,<V2>,..,<Vn>

The prompt is optional, but must be a string in quotes followed by a ; if used. If no prompt is used, BASIC prompts with a ?. However, if the system is in screen edit mode (see Chapter II.1) and no prompt has been used, no question-mark will appear (so that a line may be input without any junk in front of it!).

Data entered as a result of an INPUT command may be in the form of numbers, strings, or strings within quotes. In the case of more than one variable being filled, the entries must be separated by a special character, NORMALLY a comma (but see SEP command in Chapter IV.3).

If the number of entries typed in exceeds the required number for the INPUT statement, then only the first values entered will be used, followed by the displayed message EXTRA IGNORED. If insufficient data is entered, a further prompt ? will appear.

If the user attempts to enter a string when numeric data is required, the non-numeric data will be ignored, and 0 will be assumed if the first character is non-numeric.

```
Example:
10 INPUT "Name, Rank and Number: "; NAME$, RANK$, N
20 PRINT RANK$,NAME$;N
RUN
Name, Rank and Number: CORNISH,Capt,506659
Capt     CORNISH 506659
```

LET <V>=<E> or <V>=<E> Assigns the value of <E> to the variable <V>. The word LET is optional, but is REALLY more correct!

Example:

LET AA=1+2*3/4	assigns the value 4.5 to variable AA
LET TEMP%=12	assigns the value 12 to integer variable TEMP%
NAME$="JOHN"	assigns the string JOHN to variable NAME$, and shows use of the format without the word LET.

It is perfectly permissible to assign integer variables to ordinary real variables, and even to assign floating-point quantities to integer variables. In the latter case, however, the result MUST be in the range -65535 to 65535 (not forgetting that numbers in the ranges -65535 to -32769 and 32768 to 65535 will be converted as shown in Chapter I.2b). Moreover, any floating-point part will be lost, as if an INT function (section 3 of this Chapter) had been performed before assigning the result.

Example:
A%=PI is the same as A%=INT(PI), and assigns the value 3 to A%.

ON <J> GOTO <L1>,<L2>,..,<Ln>
ON <J> GOSUB <L1>,<L2>,..,<Ln> In both cases, expression <J> is evaluated, and execution transfers to line <L1> if <J>=1, <L2> if <J>=2, and so on. If <J>=0 or >n, execution continues with the next statement. The transfer takes the form of a GOTO or GOSUB as specified and, in the case of a GOSUB, execution will continue with the statement following the ON statement after returning. NOTE: A Qty Error occurs if <J> is negative!

Example:

```
 10 INPUT "Type in the day of the week (1-7)";DAY
 20 PRINT "It is ";
 30 ON DAY GOSUB 110,120,130,140,150,160,170
 40 PRINT " today."
 50 END
100 REM DAYS OF THE WEEK
110 PRINT "SUNDAY";: RETURN
120 PRINT "MONDAY";: RETURN
130 PRINT "TUESDAY";: RETURN
140 PRINT "WEDNESDAY";: RETURN
150 PRINT "THURSDAY";: RETURN
160 PRINT "FRIDAY";: RETURN
170 PRINT "SATURDAY";: RETURN
RUN
Type in the day of the week (1-7): 3
It is TUESDAY today.
```

PRINT Used for sending out to the screen, printer, a file, or to some other output device. Special formats for output to other devices and files are described in Chapters IV and V). The usual form is to follow the command PRINT with a list of expressions, each separated by one of a selection of separators. The expressions may be numeric or string types.

The separators between expressions may be as follows:

; leaves the (imaginary) print-head where it is, so that the next expression will print directly from the end of the previous one.

, moves the (imaginary) print-head to the start of the next tab-point, of which there are several per line, normally 14 columns apart (but this may be modified by means of the ZONE command, as may the 'tab limit'). If the print column is already past the tab limit, a CRLF is printed before the next expression.

@ allows printing of expressions at specified points on the screen using coordinates. For this situation, the screen is divided (internally and automatically) into columns and rows (see Appendix B for the number of columns and rows in your own system). Both coordinates must be specified as a number between 0 and 255 but if either is greater than the number of columns or rows (as appropriate), a 'wrap-around' will occur. Thus if, on a 48x16 screen, for example, we do a PRINT @57,24 the cursor actually moves to 9,8. Coordinates must both be given and separated by a comma, while the separator between the coordinates and the expression following may be a comma OR a semi-colon (in this case, the separator has no effect). This last separator is not needed if no expression follows the coordinate specification.

Except in the case of a coordinate specification coming at the end of a PRINT statement, a CRLF is printed at the end of a PRINT statement unless a ';' or ',' separator appears at the end of the statement.

By the same token, a PRINT statement by itself will just print a CRLF.

```
Example:
10 PRINT "HELLO";"GOODBYE","TO YOU";987,
20 PRINT 1234
RUN
HELLOGOODBYE    TO YOU 987    1234
```

Note that all numbers are printed with a leading and trailing space, the leading space being reserved for a sign which is only shown if the number is negative. Both of these spaces may, however, be removed, when desired, by means of the IOM command (Chapter IV.3), for convenience and compatibility with some other BASICs. Moreover, numeric printout may be specially formatted on printout by means of the FMT command (in the same section), and the user should consult this section for information about the various forms in which numbers may be displayed.

PRINT may be abbreviated to ? when typed in as a line of program text, although it will still LIST as PRINT (except, of course, that ? stays as such within REM and DATA statements, or within quotes).

READ....DATA....RESTORE are used for storing and using data from within a program as opposed to data entered by the user.

READ <V1>,<V2>,..,<Vn> Reads in data from a list stored in the program within one or more DATA statements. BASIC maintains a pointer which remembers the last item of data read, so that subsequent READ statements will continue from that point. The format is very like that of the INPUT statement (without a prompt) and if there is insufficient data available, a DATA ERROR will occur.

DATA <data> Specifies the items of data to be read. These items may be numbers, strings in quotes, or strings without quotes, provided they contain no leading spaces or separators. The user may have as many DATA statements as

are desired within a program, each containing as many or as few items as are convenient. DATA statements may appear at any position in a program and will be read as though they were all in one block.

DATA statements are ignored when encountered during the running of a program (just like REM statements)

As with INPUT, the separator (normally ',') may be modified by means of the SEP command (see Chapter IV.3), and it must be remembered that this command affects both INPUT and READ statements.

RESTORE <L> Restores the internal data pointer to the first DATA statement following line <L>. <L> is optional and, if omitted, the pointer is restored to the very first DATA statement in the program. In this way, DATA statements may be re-read several times within the same program, without requiring to be stored in variables throughout the execution of the program.

REM Causes the remainder of the line to be ignored by the Interpreter. Its main use is for entering programming notes during the development of a program, so that it may be more easily understood by anyone reading it.

SET <J1>,<J2> RESET <J1>,<J2> Graphics commands, for turning on (and off) graphics points on the display screen. The standard screen is arranged as 96 points horizontally and 96 points vertically.

STOP Like END, terminates a program, but also displays the message BREAK IN <L>, where <L> is the line number in which the termination occurs. Several STOP commands may be used in a program, and execution may be restarted from this break point by means of the CONT command, provided that no program alterations have been made during the break.

SWAP <V1>,<V2> Swaps the contents of variables <V1> and <V2>, which may be numeric or string variables or array elements. Clearly, they must be of similar type, otherwise a TYPE ERROR will occur. This command is very useful in sorting algorithms, being much faster than the follow- ing:

```
Example:
SWAP A(I),A(I+1) replaces T=A(I): A(I)=A(I+1): A(I+1)=T
```

where T is an extra variable which would otherwise be needed to hold one of the other variables. The speed of this command becomes very apparent when string sorting is done, since only the POINTERS are swapped, not the actual strings themselves.

2. DISC COMMANDS

The following are available only in the disc version of XBASIC:

DIR <F> Displays the directory, showing the files specified by <F>.

If <F> is not given, or is given as "*.*", all files are listed on the default disc drive. Locked files are indicated by a * in front of their names. The actual number of files per line shown varies according to the value of the zone limit (see ZONE, Chapter IV.3).

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

DIR "*.ASC"
: XYZ     .ASC :*INVADERS.ASC
```

ERA <F> Erases the file given by <F>. Only one file at a time may be erased (to discourage wholesale slaughter when you don't really mean it!). A No File error occurs if <F> does not exist, and a File Locked error if the file is locked.

REN <F2>,<F1> Renames the file given by <F1> to the name <F2> (note the order in which the names appear!). A File Exists error occurs if the name <F2> is already present, and a No File error occurs if <F1> does not exist. If <F1> is locked, a File Locked Error occurs.

LOCK <F> Locks the file named <F>, so that it may not be written on, ERAsed or RENamed. A No File error occurs if <F> does not exist. Locked files are shown in a DIRectory display with a leading "*".

UNLOCK <F> Unlocks the file <F> previously LOCKed, so that it may be written to, ERAsed or RENamed.

3. STANDARD FUNCTIONS

Note: Where we say that a function 'returns' a value we mean, of course, 'returns for use within an expression'. If you wish to try out the examples given below put the command PRINT in front, to display the desired result.

```
Example:
PRINT ABS(-3.14159)     will display the result 3.14159
```

ABS(<N>) Returns the Absolute value of <N>

```
Example:
ABS(-3.14159)  returns 3.14159
```

ATN(<N>) Returns the Arctangent of <N> in radians ranging from -PI/2 to +PI/2

```
Example:
ATN(1)returns 0.785398, which is PI/4
```

COS(<N>) Returns the cosine of <N>, where <N> is in radians.

EVAL(<S>) Returns the result of evaluating the text in the string expression <S>, as if it were part of the normal program text. This is particularly useful when it may be desired to INPUT an expression for evaluation. The expression <S> must be syntactically correct, otherwise a SYNTAX ERROR will occur.

```
Example:
10 X=5
20 INPUT "Type in expression:";A$: Y=EVAL(A$)
30 PRINT "Result is: ";Y
RUN
Type in expression: 1+X-EXP(X/3)
Result is: .70551
```

EXP(<N>) Raises e (value 2.71828..) to the power of <N>. If <N> is greater than about 87, an OVFL ERROR will result (since the result will be greater than 1 E39!).

INCH Returns the ASCII value of the next input character, which it must first wait for. This is very useful for pausing between pages of instructions, for example. See also INCH$ and INCH$(N) in section 4 for the version to use with strings.

INT(<N>) Returns the largest integer less than or equal to <N>. This definition is important, since it applies also to negative numbers.

```
Examples:
INT(3.14159)   returns the value 3.
INT(-3.14159)  returns the value -4.
```

KBD Similar to INCH, but only scans for input. It returns 0 if no character is available, or the ASCII value if one has. It does not wait for a character. See also KBD$ in section 4, the version for use with strings.

LN(<N>) Returns the natural (base e) logarithm.
LOG(<N>) Returns the base 10 logarithm.

Care is needed when using these, since many BASICs use only LOG, and that for natural logarithms only. If <N> is less than or equal to zero, a QTY ERROR will occur.

```
Examples:
LN(2)  returns 0.693147
LOG(2)  returns 0.30103
```

PI Returns the value 3.14159, and is faster than using a variable to hold the number pi.

POINT(<J1>,<J2>) Used in conjuction with the special graphics commands SET and RESET, returns 1 if the graphics point at <J1>,<J2> is lit, otherwise 0. See Appendix B.

POS(<J>) Used to obtain the current output column or row position, according to the value of <J>.

POS(0) returns the 'print column' count. This is independent of screen size, and is only zeroed when a CR, HOME or CLEAR SCREEN/FORM FEED code is output, or if the column count exceeds 255. This is designed mainly for use with printers.

POS(1) returns the current column position of the cursor on the VDU.

POS(2) returns the current row position of the cursor on the VDU. Both of these are designed to be used in conjunction with the PRINT@ facility (see PRINT in section 1).

RND(<I>) Returns a random number, depending upon the value of <I>.

RND(1) returns a random number in the range 0 to 1, as a floating-point number.

RND(<I>) with <I> in the range 2-65535, will return an integer random number, ranging from 0 to <I>-1. E.g, RND(9) returns a number in the range 0-8. This achieves compatibility with many integer BASIC's, and obviates the need, for example, for INT(9*RND(1)), which is often seen in other BASIC's.

RND(0) returns the last random number produced, whether integer or real

The random number generator uses the Z80 refresh register several times during the routine to give far more random results than a 'pseudo' random number generator. Hence the RANDOMIZE statement found in many BASIC's is not required in XBASIC.

SGN(<N>) Returns the sign of <N>. If <N> < 0, it returns -1, if <N> = 0, it returns 0, and if <N> > 0, it returns 1.

SIN(<N>) Returns the sine of <N>, where <N> is in radians.

SIZE Returns the size of memory available for the program, variables, pointers and strings, as a positive number in the range 0-65535.

SQR(<N>) Returns the square root of <N>. If <N> is less than 0, a QTY ERROR will occur.

SPC(<J>) Prints <J> spaces. This function is only valid within a PRINT statement.

TAB(<J1>,<J2>) Prints characters until the (imaginary) print head reaches column <J1> on the output device. This function is also only valid within a

PRINT statement. The value of <J2> represents the ASCII value of the character printed, and <J2> is optional. If omitted, the character specified in a previous TAB function will be used, or a space character if none has been previously specified, thus being 'upward-compatible' with TAB on most BASICs.

This 'tab-character' feature is somewhat unusual, and is provided for two reasons. First, a few BASIC's, for example, PET BASIC and SHARP BASIC, use a 'cursor RIGHT' instead of a space as the TAB character, with the advantage that headings and margins on the screen may be 'printed over' without removing parts of the screen that may still be required. In these cases, work in 'translating' such a program to run under XBASIC is eased by specifying the ASCII code for 'cursor-RIGHT' at the first occurrence of a TAB function within a program.

Secondly, by using other characters, patterns for lining up margins and headings may easily be produced.

```
Example:
10 PRINT "Name";TAB(20,46)
20 PRINT "Address";TAB(20)
RUN
Name.................
Address..............
```

The user may well 'dream up' some other applications!

NOTE: If the print column is past or at column <J1>, no TAB will occur.

TAN(<N>) Returns the tangent of <N>, where <N> is in radians.

4. STANDARD STRING FUNCTIONS

ASC(<S>) Returns the ASCII value of the first character of the string <S>.

Example:
ASC("BCD") returns the value 66 (decimal for 42H, the code for "B").

CHR$(<J>) Returns the single character string whose ASCII value is <J>.

INCH$ Waits for an input character, and then returns it as a one-character string. This is very handy for single-character reponses such as Y/N?

```
Example:
10 PRINT "Type in a character:";: A$=INCH$
20 PRINT: PRINT "You typed a: ";A$
30 END
RUN
Type in a character:      (type a "B")
You typed a: B
```

Note that neither this nor the INCH function will actually echo the key back to you, so either PRINT the string as soon as it is input, or use INCH$(1) instead (see next paragraph).

INCH$(<J>) Waits for an input string of <J> characters. Each character will be echoed as input, unless the IOM command has been used to suspend echoing of characters. No special characters are recognised, and EXACTLY <J> characters must be input. This function is mainly useful for file input, since it does not react to selected characters (unlike INPUT), and may thus be used to read program or machine-code files.

See also Chapter V on file-handling, for the use of these functions with files.

KBD$ Again, like INCH$, but returns a null string if no character is available, otherwise the character as a one byte string. Note: KBD and KBD$ work only with the console keyboard, whatever device is currently selected for input.

LEFT$(<S>,<J>) Returns the leftmost <J> characters of the string <S>.

Example:
LEFT$("HELLO",2) returns the string "HE".

LEN(<S>) Returns the length of the string A$ including punctuation marks, control characters and spaces.

Example:
LEN("HELLO") returns the value 5.

MID$(<S>,<J1>,<J2>) Returns <J2> characters starting from the <J1>th character position in string <S>. <J2> may be omitted, in which case the whole string starting from the <J1>th character will be returned.

Example:
MID$("HELLO",3,2) returns the string "LL".
MID$("HELLO",3) returns the string "LLO".

MUL$(<S>,<J>) Returns a string <S> repeated <J> times. This is 'string multiplication'. The string returned must be no longer than 255 characters. This is particularly useful for displaying repeating patterns.

Example:
MUL$("*",15) returns the string "***************"
MUL$("+--",8) returns the string "+--+--+--+--+--+--+--+--"

RIGHT$(<S>,<J>) Returns the rightmost <J> characters of the string <S>.

Example:
RIGHT$("HELLO",2) returns the string "LO".