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.

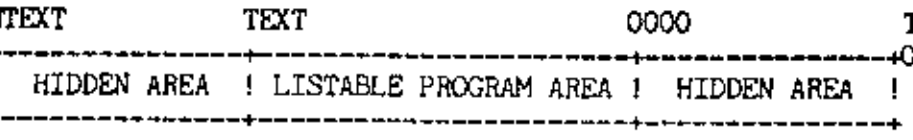
CHAIN Exactly the same as RUN, except that, in all three variations all variables are preserved, and can thus be passed from one program to another. This is an extremely useful command, particularly when it is desired to run an extremely large application, which may be split into several smaller programs 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.

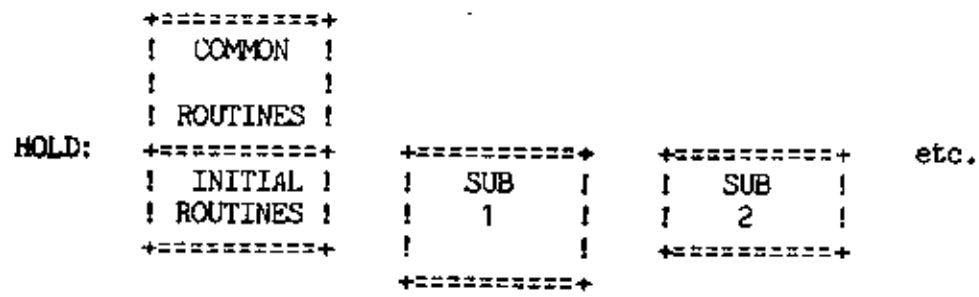
RENUM 500 Make the first line 500, increment in 10's.

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".

SCRN$(<J>) Returns the string of characters from row <J> of the VDU screen. <J> must be < the number of rows on the screen of the system (16 on a standard Nascom), and the length will always be equal to the number of columns on the screen (48 on a standard Nascom)

STR$(<N>) Returns the string representation of a numeric variable. NOTE: The format in which numbers are returned by this function is affected by the FMT command in exactly the same way as for PRINT, and also by the IOM 5 command (the trailing space optional under PRINT is not included).

Examples:
STR$(1.234) returns the string " 1.234".
FMT 2,3: A$=STR$(37.7325) places the string " 37.733" in A$.

VAL(<S>) Returns the numerical value of string <S>, up to the first non-numeric character (in this sense, the characters '+', '-', '.' and 'E' count as numeric). If the first character is non-numeric, the value 0 is returned. In addition, the character "&" is taken to indicate a hex number to follow, which can be very useful!

Examples:
VAL("1.234ABC") returns the value 1.234.
VAL("&"+"ABCD") returns the value 43981 (&ABCD).

5. USER-DEFINED FUNCTIONS

DEF FN <V1> (<V2>)=<E1> defines a user function. It must be kept to one line. <V1> may be any legal variable name, as may <V2>. <V2> is a dummy variable, which can be used within the expression <E1>. The DEF statement sets up pointers within the variable space which give the address of the expression <E1> within the program, and the address of the variable <V2> in the variable space. The line is then ignored, and execution proceeds to the next line of text as if nothing had happened.

A call to the user-defined function may now be made, as FN <V1> (<E2>). What then happens is that the current value of <V2> is saved in memory, and the value of <E2> is placed in it. The address of <E1> is then obtained, and <E1> is evaluated using the new value of <V2>. Having obtained the result of the function, the old value of <V2> is restored, as if nothing had happened to it. Thus it is, in fact, reasonably efficient in execution time, certainly better than using GOSUB statements.

<V1> must be the same type as the result of <E1>, but either or both <V1> and <V2> may be of numeric or string type. If a function call occurs before the appropriate DEF FN, a FN DEFN ERROR will occur.

Example:
```
10 DEF FN ASN(X)=ATN(X/SQR(1-X*X))
20 DEF FN ACS(X)=PI/2-FN ASN(X)
30 FOR I=0 TO .99 STEP .1
40 PRINT I,FN ASN(I),FN ACS(I)
50 NEXT
```

When RUN, this program will print out a table of values ARCSIN(I) and ARCCOS(I) for values of I between 0 and 1, in increments of 0.1. Have a go...

IV. INPUT/OUTPUT FACILITIES

1. DEVICES AND I/O ASSIGNMENT - PRINT£ and INPUT£

Special forms of the INPUT and PRINT statements allow the user to assign different I/O devices to the system, such as printers, serial or parallel devices, discs and tapes. Each device is assigned a device number in the range 0 to 254, so that we may have up to 255 output devices, and 255 input devices.

In addition, we may handle input and output to and from files, which may be stored on disc or tape, and these are handled in a slightly different way, described under sections 4 and 5 of this chapter. For your information, all file I/O is handled through device 255, which is assigned internally, and whose addresses you do not need to know.

All devices other than the file device are defined in a special table called IOLIST, the address of which is found at DEVPTR (see scratch-pad list in Appendix B). From BASIC, DEVPTR may be found or modified by means of the PTR command/function described in Chapter VII.1. The table IOLIST is set out in the following manner:

```
IOLIST: DEFB    n               ; The number of devices used.
        DEFW    INP0            ; Input device 0
        DEFW    OUT0            ; Output device 0
        DEFW    INP1            ; Input device 1
        DEFW    OUT1            ; Output device 1
        ..      ..
        DEFW    INPn-1          ; Input device n-1
        DEFW    OUTn-1          ; Output device n-1
```

The user may, of course, construct his/her own IOLIST in the above manner, and assign the address to DEVPTR (CARE!! All I/O goes through here, so can cause dire consequences if the table is incorrectly constructed). It is suggested that the user's table be located at a different address to the one provided in the Interpreter, since the one provided may not be extended. Addition of I/O routines to the user's specification is quite easy, since no registers need be saved (XBASIC has already saved any necessary registers on entering these routines). For an output routine, supply the character to be output in the A register. It does not have to be in the A register on return. For an input routine, return the input character in the A register.

Three devices are currently assigned under XBASIC as supplied:

0 OUTPUT to VDU (i.e, the 'console' output)
INPUT from keyboard. Device 0 is the only one which utilises the screen editor.

1 OUTPUT to printer (may be serial or parallel, more usually parallel). This is the device utilised for dumping the screen to printer when a CTRL/P is typed.
INPUT not assigned (may be parallel input, but currently set up as for 0).

2 OUTPUT to serial port (RS232).
INPUT from serial port.

PRINT£ <J> Assigns a new output device. All statements which produce output, such as PRINT and LIST, will now direct that output to device <J>, until another PRINT£ statement is encountered (for a different device), the program ends, or the program is aborted by an error or from the keyboard.

INPUT£ <J> Assigns a new input device. All statements requiring input, such as INPUT, INCH, and INCH$, will now receive it from device <J>, until another INPUT£ statement is encountered, the program ends, or is aborted (just as for PRINT£ above). Note that the KBD function will ONLY work from the input device 0 (the console keyboard), whichever device is used.

Both PRINT£ and INPUT£ may be used as part of normal PRINT and INPUT statements, in which case a semi-colon must be used to separate the device number from the rest of the statement. An INPUT£ statement may NOT contain a prompt, however (although subsequent INPUT statements using that device may).

Note that the CLOSE command (Chapter V.5) has the effect of including an INPUT£ 0 and a PRINT£ 0, in addition to closing files.

The END of a program (with or without the END or STOP statement) or the abandonment by interruption or error also has the effect of including an INPUT£ 0 and PRINT£ 0. This means that, in DIRECT mode, in which the line typed may be regarded as a 'miniature' program, the assignment of a device and the I/O statement required must appear in the same line, otherwise the I/O will be made through device 0!

Examples:

```
PRINT£ 1: LIST
```

This will lists the whole program to the printer device and then prints 'Ok' at the VDU (i.e an internal PRINT£ 0 has been performed). Note that the 'lines at a time' value is ignored when LISTing to a device other than 0.

```
10 PRINT£1: INPUT£2;X
20 IF X=0 THEN END
30 FOR I=1 TO X
40 INPUT A$: PRINT A$
50 NEXT
60 INPUT£0
70 PRINT£0;"Do you wish to continue?";:Y$=INCH$
80 IF Y$="Y" THEN 10 ELSE END
```

This program accepts data from input device 2, and displays it at the printer, or whichever is output device 1. First, the number of items to be read (X) is obtained from the input device, and then the items follow, being printed as they are read. After X items have been read, both input and output return to the console, so that the user may choose to terminate the program, or continue with another set of data (note that CLOSE could have been used at line 60, even though no files are being invoked).

For information on the use of PRINT£ and INPUT£ with files, see Chapter V.5.

2. DIRECT I/O PORT ACCESS

In addition to the PRINT£ and INPUT£ commands described above, users may find the following commands and functions useful, when it is desired to use I/O devices for which the appropriate machine-coded internal operating routines are not available:

OUT <J1>,<J2> sends the value of <J2> to output port <J1> of the computer.

Example:
OUT &F0,5 sends the value 5 to port F0H (240).

INP(<J>) returns the current value of input port <J> as a number in the range 0-255.

Example:
A%=INP(&F4) reads the value currently at port F4H (244)

WAIT <J1>,<J2>,<J3> Monitors the port specified by <J1>, EXCLUSIVE ORs it with <J3> (which is optional, assumed 0 if not used), and ANDs the result with <J2>, treating <J2> and <J3> as if they were eight-bit binary numbers. This will be repeated until the result is non-zero.

Examples:
WAIT 2,&40 suspends execution until bit 6 of port 2 is set.
WAIT 1,&FF,&0Fwaits until any of the 4 most significant bits are set, or until any of the 4 least significant bits are reset on port 1.

3. SPECIAL COMMANDS AFFECTING I/O AND PRINT FORMATTING

SEP <J> A very useful little command, which is not found in other BASIC's (yet, or to our knowledge!). It defines the value of the separator character used in DATA and INPUT statements. The ASCII value of the required separator must be expressed after the SEP command. Normally, this is the comma.

The usual use of SEP is SEP 0, which allows the user to put any string of characters into an INPUT or DATA statement, when only one item is required, and it is desired to allow commas to be part of the input data (in some BASIC's, a LINEINPUT command is provided to allow this). Here is another possible use:

Example:

```
10 SEP 47: REM '/' IS SEPARATOR
20  INPUT "Type in the Date as
DD/MM/YY:  ";DAY,MONTH,YEAR
30 PRINT "Day is ";DAY,"Month is ";MONTH,"Year is";YEAR
40 END
RUN
Type in the Date as DD/MM/YY: 4/12/81
Day is 4        Month is 12     Year is 81
```

Three points to watch - first, a double-quote is assumed to surround input data if it is the first non-space character in the input line, and will be subsequently removed. Secondly, certain characters will not work well as separators, particularly if numeric data is desired (e.g, the '.', although there are no problems here with string input). Finally, do not forget that SEP affects DATA statements too! Normal operation may be restored by means of a SEP 44.

The current separator may be obtained at any time by using SEP as a function.

Example:
A=SEP puts the ASCII value of the current separator into A.

FMT <J1>,<J2> Formats numeric output, for PRINT statements or **STR$** functions. The expressions <J1> and <J2> set up the number of figures to be printed in front of and behind the decimal point respectively. If the actual number of figures in front of the decimal point is less than that specified, leading spaces are used, while overflow will cause a default output in scientific notation. Trailing zeroes are always printed (except in the 'normal mode', see below), so that the output may be shown right-justified.

Scientific notation may be forced by setting <J1> to 15, in which case <J2> still gives the number of trailing figures. Otherwise, the sum of <J1> and <J2> may not be greater than 8 (remember that the maximum precision of the system is only 7 significant figures!).

'Normal' output format may be restored by means of FMT 0,0. In this format, output is to 6 significant figures, scientific notation is forced if the magnitude of the number is > 1E6 or < 1E-2, and trailing zeroes are suppressed.

Examples:

FMT 3,3: PRINT 567.9876	displays as 567.988
PRINT .00124	displays as 0.001
PRINT -1.73205	displays as - 1.732
PRINT 7895	displays as 7.895000E+03
FMT 15,2: PRINT 567.9876	displays as 5.68E+02
FMT 0,0: PRINT 0.056	displays as .056

Note that the sign is not counted with the figures, but appears in the leading space at the start of the entire number (when positive, this space may be removed by means of IOM, see below).

All in all, although a PRINTUSING facility (a command for formatting output, found on some other BASIC's) is not provided with XBASIC as supplied, the FMT command provides a flexible way of 'tidying-up' output, and for putting such numbers into strings (something that PRINTUSING cannot do).

IOM <J1>,<J2> Sets bit <J1> of the IOMOD word in the scratch-pad area. This consists of sixteen one-bit flags, of which seven are used in XBASIC, the others being reserved for future expansion. Normally, all sixteen bits are set (1), indicating a mode on, and <J2> may evaluate to either 0 or 1 only. The modes for <J1> are described as follows:

Bit 0 - Edit mode. On for SCREEN EDIT mode, off for LINE EDIT mode.

Bit 1 - Echo mode. On if all input characters are to be echoed to the output device, otherwise off. With this mode off, LINE EDIT mode is automatically selected, whatever the setting of bit 0 (otherwise the whole system would lock up!).

Bit 2 - Switch mode. Normally, we swap to LINE EDIT mode when a program is RUN, and back again to SCREEN EDIT when the program ends or is abandoned. By setting this bit to 0, this 'switching' is prevented, i.e, the edit mode, once set, will stay in that state until it is set to the other mode.

Bit 3 - Breaks mode. On if <ESC> is to be allowed to interrupt a program, and <EOF> to indicate end-of-file. If this bit is off, program interruptions will NOT be allowed (great for demonstrations!), and an end-of-file will only be indicated by the last block in a file being detected (so there may be some junk within that last block that is 'undesired' data). This is useful for reading files that may contain ANY characters as part of the data (e.g, program files).

Bit 4 - Trail space mode. On if a trailing space is to be printed after any numeric output, otherwise numbers will all run into one another, as strings already do! Some BASIC's always print trailing spaces (e.g Nascom ROM BASIC and Xtal BASIC 2), while others never do, so the object of this and Bit 5 is to make program compatibility easier to achieve.

Bit 5 - Leading space mode. Similar to bit 4, many BASIC's will print a leading space on numeric output of positive numbers, while printing a negative sign for negative numbers, while others omit the leading space. Again, this bit is on for a leading space, off for no leading space.

Note - these last two bits only affect NUMERIC output, NOT string output (even if the strings consist entirely of numbers).

Bit 6 - Auto LF mode. When set, outputs a line feed character whenever a newline is output, but just sends a <CR> if reset. By 'newline', we mean that XBASIC thinks that it is sending a <CRLF>, e.g, at the end of each line in a LIST. A case such as PRINT CHR$(13);CHR$(10); would, however, print a <CR> and an <LF> whatever the value of this bit. The setting of this bit has no effect on device 0.

Note - this is useful for file output, since the <LF> code is ignored in input of a file using INPUT#, so that the file size may be reduced by resetting bit 6 prior to output. Of course, some files may need the <LF> codes, when bit 6 should be set. This facility is also necessary when using some printers, which may not have an auto line-feed enable/disable option.

Bit 7 - Expand TAB's. When set, TAB's are expanded and, when reset, the actual TAB character itself is output (ASCII code =9). When a PRINT is performed using the comma separator (for printing in 'zones'), a TAB character is actually printed, but normally is internally expanded into spaces. Thus, by performing an IOM 7,0 , the actual TAB character is printed (ASCII code =9), which is useful for sending output to a printer, for example, which may have its own special tab settings.

IOM may be called as a function, to give the appropriate bit setting.

```
Example:
IOM 0,0:  PRINT  IOM(0)
```

Prints the result 0, after setting up for line edit mode. Screen edit mode will be turned on as soon as direct mode is re-entered and, if the above example had been executed from direct mode, IOM(0) would now return 1 again!

SPEED<J> Sets a delay in the character output to the current output device. 0 gives the slowest (VERY slow!) speed, while 255 is normal (fastest) speed. SPEED may also be used as a function, to return the current set speed.

Example:
SPEED 200: A=SPEED results in A containing 200.

NULL <J> Sets the number of nulls to be printed after every <CR> character. This command is designed for operating with slow serial devices, where the <CR> code may take a little over the time allowed for one character to print. The correct setting for particular devices will be found by experiment, although 1 will usually be enough for most applications (up to 255 is allowed, however!). The default setting is 0.

The current number of nulls may be found by using NULL as a function.

WIDTH <J> Sets the width of the current output device, so that an automatic CRLF is generated as soon as the column count reaches <J>. This is useful on certain types of printer (e.g Teletypes and Creed Teleprinters), on which overprinting would normally occur when the print head reaches the end of a line. Normally, the width is set to 0, when no automatic CRLF is produced. The current width can be obtained at any time, by using WIDTH as a function.

ZONE <J1>,<J2> Sets the print zone (tab) width (<J1>), and the largest column for which printing to the next zone will stay on the same line (<J2>), known as the ZONE LIMIT. The default settings for these are 14 and 36 respectively.

V. FILE MANAGEMENT IN XBASIC

1. GENERAL

The management of files in XBASIC has deliberately been kept as simple as possible, while maintaining flexibility. In particular, the commands for handling cassette tape files and disc files are exactly the same, the only differences being that cassette files do not allow the following:

a. Random-access (since cassette tape is a sequential medium).
b. Changing from read to write mode (or vice versa) while accessing.
c. File input after a CREATE, or file output after an OPEN.

Disc files allow ALL of these facilities.

Disc and tape drives are allocated single letters of the alphabet, the letters A to S inclusive being allocated to 'disc' drives (i.e, devices which are capable of supporting a file directory area and random-access), while T to Z inclusive are allocated to 'tape' drives.

For those who did not read the start of Chapter 0 (tut, tut!), we remind you that 'disc files' are those which belong to devices which allow random-access and a directory of files, whereas 'tape files' belong to devices which support sequential access only.

2. FILE NAMING CONVENTIONS

The file naming convention used in XBASIC is based upon that of the CP/M disc operating system (produced by Digital Research Inc.), since it is now very well known and widely used, and the file name convention may be adapted to other tape or disc operating systems.

We specify an optional 1-character drive name, a file NAME of up to 8 characters, and a file TYPE of up to 3 characters.

The drive name, if used, is a single letter from A to Z, as explained in the previous section. If not given, the default drive (the one which is assumed to be specified if none is given) is set up as A in a system which runs discs (or discs and tape), and T in a system which runs tape only. The default drive may be changed at any time by means of the DRIVE command, described in section 5.

Both the file name and type may consist of any combination of ASCII characters, with the exception of those with ASCII codes greater than 127, and the characters . , " < > ; : = ? *

The file types recognised are these:

XBS - XBASIC Source, a normal program file. If the file type is not given, .XBS is assumed. Many BASIC's use .BAS as the source file type, but it is felt that .XBS will cause less confusion with other versions of BASIC (it IS possible that someone is using .XBS for some obscure application, but we do not know of any at the time of writing!).

ASC - ASCII program file, that is an uncompressed source file (.XBS files contain tokens for reserved words, whereas .ASC files contain them exactly as they appear in a LIST).

OBJ - OBJECT file, or machine-code subroutines/data. A special area may be set up within the memory map for the storage of machine-code routines, utilising the CLEAR command, and anything stored here may be SAVEd as a .OBJ file, and LOADed into this area.

Any other combination is treated as a data file, that is, a sequence of ASCII characters, divided into one or more records, which may be accessed by a BASIC program. In the broadest sense, this includes files of the three special types .XBS, .OBJ, and .ASC (particularly the .ASC files, since they are pure text).

In some disc applications, it is necessary to specify more than one file (an AMBIGUOUS file reference), e.g, in specifying the files for a DIRectory list. Here, the ? character matches any character in that position in the file name found. In addition, the * character matches all of the characters at and after its position in the file name or type in which it appears. This is best explained by some examples:

X?Z.ASC	matches XYZ.ASC, XAZ.ASC, X9Z.ASC, etc.
?X.D?T	matches AX.DAT, BX.D7T, 4X.DZT, etc.
.	matches ANY file, and is the same as ????????.???
*.XBS	matches any .XBS file.
*	also matches any .XBS file, being the default type.
PROG*.ASC	matches PROG1.ASC, PROG10.ASC, PROGABC.ASC, etc.

Ambiguous references are not used (or allowed!) in cassette tape operations.

3. THE FILE DESCRIPTOR

Before describing the file-handling commands, it would be helpful to mention what we believe to be a new concept in file-handling. Most BASIC's have some method of assigning a storage area for use by a file for the time that it is open. This area usually contains a buffer, and also some information to describe the file and where it is stored on disc, etc. The problem that arises is that this area has to be fixed and set aside before running the program, and this means that the space set aside may not be used for anything else when the file is closed. It also places a constraint upon the number of files which may be open at one time.

XBASIC overcomes this by assigning a special string variable to a file when it is opened for access, and dropping this variable when the file is closed. This string variable is known as the FILE DESCRIPTOR for the file. It is always 168 characters long, contains a 128-byte buffer, and 40 bytes of special file information.

The layout of a file descriptor string is as follows:

FDESC:	DRIVE	1 byte	Disc drive name, 01=A, 02=B, etc.
	FILNAM	8 bytes	File name
	FILTYP	3 bytes	File type
	INFO	21 bytes	Internal allocation information (disc only).
	RECORD	2 bytes	Record number in the range 0-65535 (applies
	+	1 overflow byte	to disc only).
	FILPTR	1 byte	Pointer to current byte in buffer for I/O.
	RWFLAG	1 byte	Read/Write Flag 0=Input, 1=Output.
	RECLEN	2 bytes	Random Record Length (Random access only).
	FILBUF	128 bytes	128-byte file buffer.

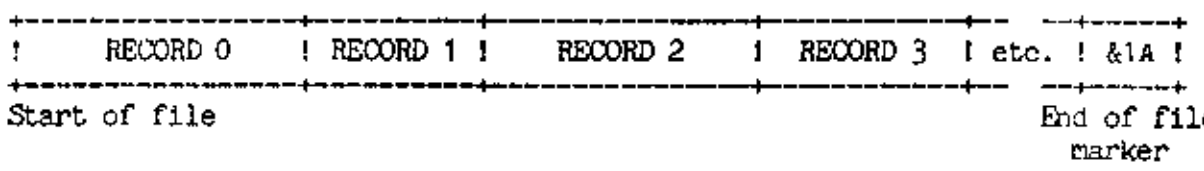
The File Descriptor (which, for the sake of abbreviation, we shall refer to as an FDESC in the rest of this discussion) must be a simple string variable (it may not be part of a string array, but all of the parts of it are accessible by normal string functions. For example, the buffer contents may be inspected by doing a RIGHT$ of the last 128 bytes of the FDESC.

A file descriptor may not be modified by LET statements, etc. If it is, a File Error will occur the next time it is used in a PRINT£ or INPUT£ statement.

4. SEQUENTIAL AND RANDOM ACCESS METHODS

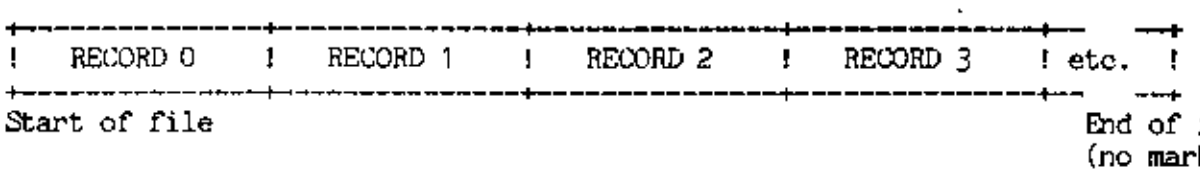
Before explaining the various file-handling commands, let us give a short description of the two alternative methods of accessing files allowed directly under XBASIC:

a. Sequential Access.



Sequential Access is most often used for the manipulation of text or index files, where records may be of variable length, and need to be scanned sequentially, i.e, one after the other, starting from the beginning of the file until the desired record location is reached. Each record should normally have a terminator, such as a carriage return code, and a special code to mark the end-of-file (EOF). XBASIC supplies an EOF code when closing a sequential file if the last operation was a write, and will normally detect the end-of-file marker on reading. For disc files, an end-of-file condition will also occur if an attempt is made to read beyond the last allocated sector of the file. Note that tape data files MUST have an EOF mark present within the file, otherwise the end-of-file will not be detected!

b. Random-access (Disc files only).



Records stored in random-access mode will normally be of fixed length, or of variable length within fixed-length blocks. The record length is specified when the file is opened, and a record number is specified whenever a file input or output is required. This means that we may move about the file in a completely RANDOM fashion, accessing only the desired records. Having accessed a record, reading or writing may continue from that point in the file, so that it is possible to read or write sequentially in the file, even though a random record length has been specified. It is even permissible to write a file with one record length, and to read or write to the same file with a different record length (if the programmer has a good reason for wanting to do so!).

An EOF marker is NOT supplied when closing a random-access file, since it is not convenient - the end-of-file condition will occur when an attempt is made to read a sector that does not exist. However, it will not always occur on a non-existent record, since the disc space may have already been created for it as a side-effect of writing another record which uses the same physical disc sector. In that case, it will be as if the record is simply empty.

This may all be as clear as mud, so read the following section, and then study (and try out!) the examples at the end of the chapter.

5. FILE-HANDLING COMMANDS

DRIVE Followed by a single letter, sets up the default disc or tape drive for any subsequent file-handling commands. If the drive specified is not available on the user's system, a DRIVE SELECT ERROR will occur.

```
Example:
DRIVE B                 Selects disc drive B as the default drive.
OPEN "DATA.TXT",F$      will now open the file DATA.TXT on drive B.
```

OPEN <F>,<SV>,<I> Opens a file named <F>, and assigns internal file information and buffer space to the FDESC <SV>. The random record size is given by <I>, which must be in the range 0-65535 if specified, but is not required if the file is to be accessed sequentially. In fact, a random record length of 0 indicates that sequential access is to be performed.

For tape files, only reading of the file is allowed after an OPEN statement. Otherwise, both reading and writing are allowed (but this is usually only advantageous when using random access!).

For disc files, a No File error occurs if the file is not present on the specified drive.

```
Example:
OPEN "A:SILLY.DAT",FD$,15
```
Opens the file SILLY.DAT on the disc drive A, assigns variable FD$ as the file descriptor, and sets it up for random-access with 15-character records.

CREATE <F>,<SV>,<I> The format is exactly the same as for OPEN, except that an existing file named <F> is first deleted if present, and a new empty file of the same name is opened. For cassette tape drives, this is the command to be used when writing to a file.

CLOSE <SV1>,<SV2>,...,<SVn> Closes the files given by the FDESCs <SV1> to <SVn> inclusive, writes the remaining contents of the appropriate buffers to their files and stores directory information (for disc files). A buffer will only be written out if the last operation performed on it was a write. The FDESCs are then set to null strings, which effectively releases the space for use by variables or other files. A File error will occur if any of the FDESC's given is not active, or is an ordinary string (note that FDESC's are internally marked so that XBASIC can distinguish them from normal strings).

CLOSE by itself is allowed, in which case ALL files currently open will be closed, and no error is given if no files are open.

Note: As a side-effect, CLOSE does an automatic PRINT£ 0: INPUT£ 0, so that all input and output will go through the console. A CLOSE command may be executed at any time when these two statements are required (it is shorter!).

APPEND <F>,<SV> This command is used for disc files only, and is very similar to OPEN. The difference is that the internal file pointer moves to the end of the file instead of the start of the file, and no record length is supplied. This command is used to write extra information at the end of a sequential file, when to OPEN the file and read up to the end would be most inefficient!

A No File error will occur if the file <F> does not exist on the disc.

PRINT£ <SV>,<I>; <expression list> Outputs the <expression list> to the file given by the FDESC <SV>, from the start of record number <I> in the file. The location relative to the start of the file is calculated as <I> multiplied by the record length which was given when the file was opened. This, of course, is if the file was opened for random access, and this is not allowed for tape files. Hence a File Error will occur if <I> is specified with <SV> specifying a tape file.

For sequential access, leave out the ',<I>' but keep the ';', and then output will start from the current place in that file (XBASIC does not lose its place in a particular file even when several files at once may be open for output). In fact, the only purpose of the record number is to define the point within the file at which input or output is to begin, and so it will be assumed that we start 'from where we left off' if the record number is not given. With disc files that have been opened for sequential access, the internal file pointer may always be set to the start of the file by specifying a record number (any will do, since it will be multiplied by the zero record length!).

The <expression list> is similar to that in a normal PRINT statement, and remember that the data output will be EXACTLY as for thát. Hence a CRLF is output at the end of the statement, unless terminated by a semi-colon.

Note that all subsequent statements supplying output will now go to a file, until another PRINT£ or CLOSE statement is encountered.

PRINT£ <SV>,<I> or PRINT£ <SV> are allowed, setting up the specified file for output. Nothing is output by either of these (No, not even a CRLF!), that being reserved for subsequent output statements (e.g, PRINT or LIST).

As you might expect, a stream of data without CRLFs may be output to a file by simply terminating each such PRINT statement with a ';'. Remember also that you may need to suspend the automatic tab expansion (where CHR$(9) is expanded to spaces) by using the IOM 7,0 command. The output of strings which contain machine-code may now be contemplated, for example.

INPUT£ <SV>,<I>; <variable list> Takes input from the file given by the FDESC <SV>, starting at the first character of record number <I> in the file. As for PRINT£, the <I> may be omitted, in which case the file is read from the last point reached (or from the beginning if it has just been opened). The <variable list> is as for the normal INPUT command (Chapter III.1), and items are assigned to the variable names given in the same way.

Note that all subsequent statements requiring input (e.g, INPUT, INCH, INCH$ and INCH$(N)) will now try to gather input from the file, until another INPUT£ or CLOSE statement is encountered.

INPUT£ <SV>,<I> or INPUT£ <SV> is allowed, setting up the specified file for input. Nothing will actually happen until the next INPUT statement, which need not then have the file specification given.

INCH$ and INCH$(N) These are mentioned again here, since they are very important when files containing all sorts of control characters are required to be input (e.g, machine-code files). Remember that INPUT ignores most control characters, and terminates on a CR or null character (and also on the input terminator defined by SEP). INCH$ suffers from none of these disabilities, and so may be used for this purpose. INCH$(N) may be used even more effectively, since it creates a string of length N and is usually much faster than INCH$ on its own. An EOF condition on INCH$(N) causes the truncation of the string to the length reached at the time when the EOF occured, so that all of the information may still be passed into an expression before the EOF error is actually flagged. The very next input from that file will then flag the EOF condition in the usual way (i.e, 'End of text Error' or ON ERR/ON EOF routine).

6. FILE-HANDLING EXAMPLES

6.1 A text file display program.

This program allows the display of data or .ASC files on the screen. It may be used under the tape version of XBASIC by removing line 35 (which is useful on disc systems, when the name of the required file has been forgotten!). Users of CP/M will notice that this performs virtually the same function as the TYPE command, and that it works at about the same speed!

```
10 REM TEXT FILE DISPLAY PROGRAM
20 N=128: REM No. of characters read at a time
30 INPUT "File to display?"; NAME$
35 IF NAME$="" THEN DIR: GOTO 30
40 ON EOF GOTO 80
50 OPEN NAME$,FD$
60 INPUT£ FD$
70 PRINT INCH$(N);: GOTO 70
80 CLOSE FD$
90 END
```

Try replacing line 70 with the following, noting how much slower it is:
70 PRINT INCH$;: GOTO 70 or try smaller values of N in line 20.

6.2 A simple Mailing List (Sequential Access)

The program below is a simple mailing list program suitable for either tape or disc drives, showing as it does the use of sequential access for reading and writing files. In this case, the data file is read into a large string array M$ at the start of the program, and rewritten to the file SMAIL.DAT at the end. This means that access to particular customers is very quick, but at the expense of keeping the entire file in memory at once. Moreover, the maximum number of customers that the system can handle is limited by memory size, and the size of M$ as dimensioned in line 9000.

The information under each customer consists simply of his/her name, telephone no. and address, the address being stored in two lines, or fields. The array CUST$ holds these items temporarily when being accessed by one of the program options.

The options supported by the program are to add a customer to the list, to access a customer from the list for modification, and to list all customers to the screen or printer.

```
 10 REM *** SIMPLE MAILING LIST PROGRAM (SEQUENTIAL ACCESS) ***
 20 REM
 30 GOTO 9000
 98 REM
 99 REM *** COMMON ROUTINES ***
198 REM
199 REM *** OPEN DATA FILE ***
200 PRINT: PRINT "Do you have a file to load (Y/N)?";: Y$=INCH$
210 PRINT Y$: IF Y$="N" THEN RETURN
220 CLS: PRINT@4,10;"Reading in data file..."
230 OPEN FILE$,FD$
240 INPUT£ FD$; NCUST: REM Get No. of customers on file
250 IF NCUST=0 THEN 290
260 FOR I=0 TO NCUST-1
270 FOR J=0 TO 3: INPUT M$(I,J)
280 NEXT J,I
290 CLOSE
295 RETURN
298 REM
299 REM *** HEADING DISPLAY ***
300 CLS: PRINT@8,0;HEAD$
310 PRINT@3,2;"Number of customers on file: ";NCUST: PRINT
320 RETURN
398 REM
399 REM *** WRITE NEW DATA FILE ***
400 PRINT: PRINT "Do you wish to save the file (Y/N)?";: Y$=INCH$
410 PRINT Y$: IF Y$="N" THEN RETURN
420 CLS: PRINT@4,10;"Writing New Data file..."
430 CREATE FILE$,FD$
440 PRINT£ FD$; NCUST
450 IF NCUST=0 THEN 490
460 FOR I=0 TO NCUST-1
470 FOR J=0 TO 3: PRINT M$(I,J)
```

```
480 NEXT J,I
490 CLOSE
499 RETURN
798 REM
799 REM *** MENU DISPLAY ***
800 CLS: PRINT@8,0;"SIMPLE MAIL LIST PROGRAM"
810 PRINT@4,3;"Options:"
820 PRINT@4,5;"0. Exit Program"
830 PRINT@4,7;"1. Enter Customers"
840 PRINT@4,9;"2. Modify Customers"
850 PRINT@4,11;"3. List Customers"
870 PRINT@4,13;"Which? ";: N$=INCH$(1): PRINT
880 N=VAL(N$): IF N<0 OR N>3 THEN PRINT BEL$: GOTO 870
890 IF N=0 THEN GOSUB 400: CLS: PRINT@8,0;"GOODBYE!";BEL$: END
898 REM
899 REM *** SELECT OPTIONS ***
900 ON N GOTO 1000,2000,3000
910 STOP: REM SHOULD NEVER GET HERE!
998 REM
999 REM *** END OF COMMON ROUTINES ***
1000 REM *** MSUB1 -- Enter Customers ***
1010 HEAD$="ENTER CUSTOMERS"
1020 GOSUB 300
1030 PRINT"Any more customers to add (Y/N)?";:Y$=INCH$: PRINT Y$: PRINT
1040 IF Y$<>"Y" THEN 800
1050 FOR I=0 TO 3
1060 PRINT PRMPT$(I);: INPUT CUST$(I)
1070 NEXT
1080 FOR I=0 TO 3: M$(NCUST,I)=CUST$(I): NEXT: NCUST=NCUST+1
1090 GOTO 1020
1999 REM
2000 REM *** MSUB2 -- Modify Customers ***
2010 HEAD$="MODIFY CUSTOMERS"
2030 GOSUB 300
2040 INPUT "Customer No.?";CN$: IF CN$="END" THEN 800
2050 CN=VAL(CN$): IF CN=0 OR CN>NCUST THEN 2040
2060 CN=CN-1
2070 FOR I=0 TO 3: CUST$(I)=M$(CN,I): NEXT
2080 PRINT@3,8;"Customer No. :",CN+1
2090 FOR I=0 TO 3
2100 PRINT I+1;PRMPT$(I),CUST$(I)
2110 NEXT: PRINT
2120 PRINT "Any changes for this item (Y/N)?";: Y$=INCH$: PRINT Y$
2130 IF Y$<>"Y" THEN 2180
2140 PRINT "Which Line (1-4)?";: Y$=INCH$: PRINT Y$: PRINT
2150 I=VAL(Y$)-1
2160 IF I<0 OR I>4 THEN 2030 ELSE PRINT PRMPT$(I);: INPUT CUST$(I): CLS
2170 GOTO 2080
2180 FOR I=0 TO 3: M$(CN,I)=CUST$(I): NEXT
2190 GOTO 2030
2999 REM
3000 REM *** MSUB3 -- List Customers ***
3010 HEAD$="LIST CUSTOMERS"
3020 GOSUB 300: IF NCUST=0 THEN 800
3030 PRINT "To Screen or Printer (S/P)?";: PF$=INCH$: PRINT PF$: PRINT
3040 IF PF$="P" THEN PRINT#1
3050 FOR CN=0 TO NCUST-1
```

```
3060 PRINT "Customer No. :",CN+1
3070 FOR I=0 TO 3: PRINT PRMPT$(I),M$(CN,I): NEXT: PRINT
3080 IF PF$<>"P" THEN INPUT "Type <CR> to go on:";Y$: PRINT
3090 NEXT CN
3100 PRINT£ 0: GOTO 800
8998 REM
8999 REM ** INITIALISING STUFF **
9000 SEP 44: REM Use separator for DATA below
9010 BEL$=CHR$(7): REM The bells, the bells!
9020 CMAX=100: REM Max. No. of customers allowed
9030 DIM M$(CMAX-1,3),PRMPT$(3),CUST$(3)
9040 FOR I=0 TO 3: READ PRMPT$(I): NEXT
9050 FILE$="SMAIL.DAT": REM File name
9060 SEP 0: REM Allow commas in input text
9070 ZONE 28,20: REM Set up zone width
9080 GOSUB 200: REM Read in data file
9090 GOTO 800:  REM Go and do your stuff!
9098 REM
9099 REM *** DATA FOR FIELD PROMPTS ***
9100 DATA "Customer Name:","Telephone No.:"
9110 DATA "Addr. Line 1 :","Addr. Line 2 :"
```

6.3 A simple Mailing List (Random Access).

The program suite below is given to illustrate both the use of random-access files and the 'semi-CHAIN' facility outlined at the end of Chapter II. It does the same job as the single program at example b., but with much less memory, and shows how the random-access method improves the file-handling capability. The limit on the number of customers is now dictated only by the free disc space available, and the array M$ of example b. is dispensed with. The suite consists of four programs, the common and setting-up routines, and the three sub-programs which deal with the three options currently supported (see example b. above).

A record length of 75 characters is used, and this limits the amount of information that may be held on each customer, checks being needed to ensure that the total lengths of the fields entered (NB, including CR and LF codes!) do not exceed this length. Such checking may be found at lines 1090-1100 in MSUB1, and 1170-1180 in MSUB2 below. This kind of check is not necessary with a sequential file.

The first record contains the total number of records on file (NCUST), and provides a useful way of preventing access above the limit available.

Finally, note the use of the ON ERR routine at 100, which makes special checks for CHAINing to a non-existent sub-program, and allows the user to create a new data file if one is not present.

```
 10 REM *** SIMPLE MAILING LIST PROGRAM (RANDOM ACCESS) ***
 20 REM *** COMMON ROUTINES ***
 30 ON ERR GOTO 100
 40 GOTO 1000
 98 REM
 99 REM *** ERROR ROUTINE ***
100 IF ERL=900 THEN PRINT"CANNOT INVOKE DESIRED OPTION";BEL$: GOTO 800
```

```
110 IF ERR<>25 THEN PRINT ERR$;" Error in line ";ERL: END
120 PRINT "No data file -- Create (Y/N)?";: Y$=INCH$
130 IF Y$="Y" THEN CREATE FILE$,FD$: PRINT£ FD$;"0": CLOSE
140 GOTO 800
198 REM
199 REM *** OPEN DATA FILE ***
200 OPEN FILE$,FD$,RL
210 INPUT£ FD$,0;NCUST: INPUT£ 0: REM Get No. of customers on file
220 RETURN
298 REM
299 REM *** HEADING DISPLAY ***
300 CLS: PRINT@8,0;HEAD$
310 PRINT@3,2;"Number of customers on file: ";NCUST: PRINT
320 RETURN
798 REM
799 REM *** MENU DISPLAY ***
800 CLOSE: CLS: PRINT@8,0;"SIMPLE MAIL LIST PROGRAM"
810 PRINT@4,3;"Options:"
820 PRINT@4,5;"0. Exit Program"
830 PRINT@4,7;"1. Enter Customers"
840 PRINT@4,9;"2. Modify Customers"
850 PRINT@4,11;"3. List Customers"
870 PRINT@4,13;"Which? ";: N$=INCH$: PRINT N$
880 N=VAL(N$): IF N<0 OR N>3 THEN PRINT BEL$: GOTO 870
890 IF N=0 THEN CLS: PRINT@8,0;"GOODBYE!";BEL$: END
898 REM
899 REM *** CHAIN TO OTHER SUB-PROGRAMS ***
900 HOLD 1000: CHAIN "MSUB"+N$
910 STOP: REM SHOULD NEVER GET HERE!
998 REM
999 REM *** END OF COMMON ROUTINES ***
1000 REM ** INITIALISING STUFF **
1010 SEP 44: REM Use separator for DATA below
1020 BEL$=CHR$(7): REM The bells, the bells!
1030 DIM CUST$(3),PRMPT$(3)
1040 FOR I=0 TO 3: READ PRMPT$(I): NEXT
1050 FILE$="RMAIL.DAT": RL=75: REM File name & record size
1060 SEP 0: REM Allow commas in input text
1070 ZONE 28,20: REM Set up zone width
1080 GOTO 800: REM Go and do your stuff!
1098 REM
1099 REM *** DATA FOR FIELD PROMPTS ***
1100 DATA "Customer Name:","Telephone No.:"
1110 DATA "Addr. Line 1 :","Addr. Line 2 :"
1000 REM *** MSUB1 -- Enter Customers ***
1010 HEAD$="ENTER CUSTOMERS"
1020 GOSUB 200
1030 GOSUB 300
1040 PRINT"Any more customers to add (Y/N)?";:Y$=INCH$: PRINT Y$: PRINT
1050 IF Y$<>"Y" THEN 800
1060 FOR I=0 TO 3
1070 PRINT PRMPT$(I);: INPUT CUST$(I)
1080 NEXT
1090 L=0: FOR I=0 TO 3: L=L+LEN(CUST$(I))+2: NEXT
1100 IF L>RL THEN PRINT "RECORD TOO LONG";BEL$: GOTO 1030
1110 PRINT£ FD$,NCUST+1
1120 FOR I=0 TO 3: PRINT CUST$(I): NEXT: NCUST=NCUST+1
1130 PRINT£ FD$,0; NCUST: PRINT£ 0: REM Update No. of customers
1140 GOTO 1030
```

```
1000 REM *** MSUB2 -- Modify Customers ***
1010 HEAD$="MODIFY CUSTOMERS"
1020 GOSUB 200
1030 GOSUB 300
1040 INPUT "Customer No.?";CN$: IF CN$="END" THEN 800
1050 CN=VAL(CN$): IF CN=0 OR CN>NCUST THEN 1040
1060 INPUT£ FD$,CN
1070 FOR I=0 TO 3: INPUT CUST$(I): NEXT: INPUT£ 0
1080 PRINT@3,8;"Customer No. :",CN
1090 FOR I=0 TO 3
1100 PRINT I+1;PRMPT$(I),CUST$(I)
1110 NEXT: PRINT
1120 PRINT "Any changes for this item (Y/N)?";: Y$=INCH$: PRINT Y$
1130 IF Y$<>"Y" THEN 1170
1140 PRINT "Which Line (1-4)?";: Y$=INCH$: PRINT Y$: PRINT
1150 I=VAL(Y$)-1: PRINT PRMPT$(I);: INPUT CUST$(I): CLS
1160 GOTO 1080
1170 L=0: FOR I=0 TO 3: L=L+LEN(CUST$(I))+2:NEXT
1180 IF L>RL THEN PRINT "RECORD TOO LONG";BEL$: GOTO 1080
1190 PRINT£ FD$,CN: FOR I=0 TO 3: PRINT CUST$(I): NEXT: PRINT£ 0
1200 GOTO 1030
```

```
1000 REM *** MSUB3 -- List Customers ***
1010 HEAD$="LIST CUSTOMERS"
1020 GOSUB 200
1040 GOSUB 300
1050 PRINT "To Screen or Printer (S/P)?";: PF$=INCH$: PRINT PF$: PRINT
1060 IF PF$="P" THEN PRINT£1
1070 FOR CN=1 TO NCUST
1080 INPUT£ FD$,CN: REM Read Customer record from file
1090 FOR I=0 TO 3: INPUT CUST$(I): NEXT: INPUT£ 0
1100 PRINT "Customer No. :",CN
1110 FOR I=0 TO 3: PRINT PRMPT$(I),CUST$(I): NEXT: PRINT
1120 IF PF$<>"P" THEN INPUT "Type <CR> to go on:";Y$: PRINT
1130 NEXT CN
1140 GOTO 800
```

This example has been given to give an idea of what may be done, but is no doubt greatly extendable (for example, there ought to be a facility to delete a customer entry, and facilities for searching and sorting under a given field). No doubt a useful exercise for the user...

VI. ERROR HANDLING

1. LIST OF ERROR MESSAGES

After an error occurs (whether resulting from a direct command or from within a program), one of the following messages will be output and execution will terminate (unless, of course, an ON ERR statement is in force, as described in section 2).

The forms of error messages are:

xxxxxx Error — in direct mode
xxxxxx Error in <l> — in program mode

where xxxxxx will be one of the following:

Bad Data — A checksum error has been detected while loading or verifying a program/data file from disc or tape.

Branch — Reference has been made to a non-existent line number.

Cmd — An attempt has been made to reference a reserved word which does not exist in the system. It may be that one user is trying to run a program developed on another user's system, but does not have all of the necessary commands on his/her own system.

Cont — An attempt has been made to CONTinue a program after an error occurred, or after alterations have been made to the program.

Data — A READ statement has been presented with insufficient data from DATA statements.

Dimension — An attempt has been made to redimension an array. An array may only be DIMensioned once in a program. This includes arrays of under 10 elements that have not been formally DIMensioned.

Division — An attempt has been made to divide a number by 0.

Drive Select — A tape or disc drive has been selected which is not available on the system.

End of Text — An end-of-file marker has been encountered in a data file, or the last block of the file has been read. This error may be handled specially by means of the ON·EOF command.

File — An attempt has been made to open a file which is already open, or to read or write from/to a file which is not open.

File Type — A file of one type has been specified, when one of another has been expected.

Fn Defn — A user-defined function has been referred to, without having first been DEFined, or CALL has been used as a function without the USRLOC having first been set up.

Mem Full An attempt has been made to execute a command which would need more memory than is available.

Next A NEXT has been encountered which cannot be matched to a FOR statement.

Operand An operand has been omitted after an operator.

Example:
PRINT 2.3*4+

Ovfl A numeric overflow has resulted from a calculation.

Qty A parameter in an array, command or function is out of range.

Examples:
A(X) where A is an array and X<0 or X>65535.
LOG(X) where X<=0.
SQR(X) where X<0.
OUT X,Y where X or Y are <0 or >255.
Note: Reference to the sections describing the commands and functions concerned will usually reveal the cause of this error message.

Range An attempt has been made to access an element of an array outside its previously defined dimensions.

Return An attempt has been made to execute a RETURN or POP without a corresponding GOSUB.

Stack Full FOR loops, GOSUBs and/or parentheses in expressions have been nested too deeply, causing a stack overflow.

Str Ovfl An attempt has been made to exceed the maximum length of a string (255 characters).

Str Complex A string expression is too long or complex and needs to be broken into smaller sections.

Syntax A typing error has been made, or a command/function has been wrongly formatted.

Type A numeric quantity has been found where a string type was expected, or vice versa.

The following error messages apply only to the Disc version:

Dir Full The directory is full up.

Disc Full No more space is available on the disc.

Disc Locked An attempt has been made to write to a write-protected disc (or under CP/M to a read-only disc.

Disc Seek An attempt has been made to access a sector that is not on the disc (usually in random-access files, when the record required is off the disc).

File Exists — An attempt has been made to REName a file to an already existing one.

File Locked — An attempt has been made to write to a file that has been LOCKed.

No File — The required file cannot be found in the directory (not given by cassette tape).

A complete list of the error messages, together with their error numbers, is given at Appendix A of this manual.

2. ERROR HANDLING WITHIN BASIC

ON ERR GOTO <L> ON ERR GOSUB <L>

Special forms of the ON statement, which do, however, work in an entirely different manner. These two commands are used for handling error routines from within a BASIC program rather than forcing abandonment of execution.

They simply set an internal flag so that, if an error occurs AFTER the command, a GOTO or GOSUB will be made to line <L>, where a routine will perform whatever action has been programmed (by the user) to overcome that error. This allows us to forget about, for example, testing for division by zero within a program; the error is simply allowed to occur and is then handled by a subroutine.

If an ON ERR GOSUB statement is used, the last statement in the error-handling routine should be a RETURN, as with other GOSUBs (or use POP and go where you will!). Execution returns to the statement FOLLOWING that where the error occurred.

Notes:
(i) Any error must occur AFTER the ON ERR statement.
(ii) The ON ERR flag reverts to normal after the first error (in case you have an error in the error routine!), so this should be set again by another ON ERR statement, either at the end of the error routine, or soon after re-entering the main program.
(iii) To restore the ON ERR flag to normal within a program, or after a program has terminated with a STOP (as opposed to an END), use OFF ERR as previously mentioned. The flag reverts automatically upon normal termination of a program.
(iv) Within the error-handling routine, the number of the error that has occurred (see section 1 of this chapter) is passed in ERR, while the line on which the error occurred is passed in ERL.

ON EOF GOTO <L> ON EOF GOSUB <L>

As for ON ERR above, except that this deals specifically with the encounter of an end-of-file on reading. There is an error message (End of Text Error) which deals with this occurrance, and so this could easily be handled by means of an ON ERR statement. However, since it is often useful to handle the end-of-file condition separately, the ON EOF statement has been included. The only difference in execution of an ON EOF routine is that the ON EOF flag is

NOT reset - it stays in force, so that a subsequent EOF will also invoke the same routine (unless another ON EOF statement has been encountered). For this reason, an OFF EOF statement should be used when the end-of-file condition no longer needs to be handled. Note also that, if both ON ERR and ON EOF are in force, ON EOF has priority for end-of-file conditions.

OFF ERR or OFF EOF Turns off the ON ERR and ON EOF modes respectively, if either of these have been previously turned on. OFF ERR will cause any errors which subsequently occur to be displayed at the console, while OFF EOF will cause any subsequent end-of-file to give an END OF TEXT ERROR, or to be routed through an ON ERR routine, if that mode is still on. Nothing happens if the appropriate mode is already off, and both modes are automatically turned off if the program ends in a normal way.

ERR Returns the number of the last error that occurred. This function is particularly useful within ON ERR routines, to find out what error actually occurred.

ERR$ Returns the error STRING message, without the word 'Error', corresponding to the last error that occurred. This saves having to flag every possible kind of error within an ON ERR routine, when one in particular may be expected.

> Example: Suppose the last error was a SYNTAX ERROR (it often is!).
> ERR returns the number 2.
> ERR$ returns the string "Syntax".

ERL Returns the line number at which the last error occurred.

3. ERROR MESSAGE TABLE CONSTRUCTION AND EXTENSION

As well as command and function extension (see Chapter VIII), XBASIC allows the addition of user-defined error messages. The error messages are normally formed in a table pointed to by ERRTAB (see PTR, Chapter VII.1), in a similar way to the reserved words:

```
        B  r  e  a  k  N  e  x  t  S  y  n  t  a  x  R  e  t  u  r  n ....
        C2 72 65 61 6B CE 65 78 74 D3 79 6E 74 61 78 D2 65 74 75 72 6E
Code:   0              1           2              3
```

Any standard ASCII characters may be included here, including spaces, unlike the case for reserved words above, so that an error message may consist of several words, if need be. Again, an &80 code must terminate the table.

The word ' Error' is supplied automatically on the end of the message (except for 'Break', which is not strictly an error, but is trapped in the same manner as an error). To call up an error, a machine-code jump should be made to the routine ERROR (see Appendix C under Useful Internal Routines), passing the error code number in the E register. If the error message is in the table, it will be printed out in the usual way, and the user returned to

Direct mode (unless, of course, an ON ERR statement is in force!). If the error number is larger than the number of messages, the number itself is printed, followed by 'Error'.

Example: Suppose a PTR 4,&9F00 command is performed, and location &9F00 contains &80. Then a subsequent syntax error will result in the message:

2 Error appearing on the screen.

Up to 128 error messages may be contained in this table, which is known as the Standard Error table. In addition, up to a further 128 error messages, numbering from &80 to &FF, may be supplied in an Auxiliary Error table, pointed to by AUXERR (accessed by PTR(5)). This table contains a single &80 code as supplied, so that the user may add error messages without affecting the standard table, if desired (the standard error table as supplied allows no room for expansion by the user, although the user may expand it by recreating the table elsewhere and adding to it, or even replace the messages by those of his/ her own choosing, when fed up with the messages supplied - How about a 'Not Understood Error' instead of 'Syntax Error'?!)

VII. MACHINE-CODE LINKAGE TO Xtal BASIC

1. MACHINE-CODE RELATED COMMANDS/FUNCTIONS

XBASIC contains extensive facilities for allowing access of machine-code routines and data, over and above the command/function extension capability described in Chapter VIII. The following commands and functions are available for this purpose:

CALL <I> Calls a machine-code subroutine starting at the address given by the expression <I>. The user need not worry about pushing registers, as long as the routine is terminated with a C9H (RET) code, which will automatically return control to BASIC. Note that the pointer to the current position in the program text will be available at the top of stack, if needed.

Example:
CALL 3840 will cause the program to jump to a routine at location &0F00 (note that CALL &F00 would have the same effect).

CALL(<E>) CALL may also be used as a FUNCTION, in which case the working is very much altered. Here, we may pass any expression as an argument to the function, having previously set the location USRLOC in the scratch-pad (set it by means of the PTR 9,<I> instruction, described later in this section). This defines the location of the required machine-code routine. The argument is passed into the FPA (Floating-Point Accumulator), from which it may be accessed by one of the routines described in Appendix C. On return, any result may be stored in the FPA, and this is then returned as the result of the function.

CALL used in this form is very much like USR in many other BASIC's. We feel that to allow CALL in both forms is both more flexible and more easily adapted to use with other BASICs.

POKE <I>,<J1>,<J2>,..,<Jn> Places the values of the expressions <J1> to <Jn> into memory starting at location <I>. Each of these expressions occupies a single byte, so must be in the range 0-255.

Example:

(i)	POKE 16384,132	puts 132 (=&84) into location 16384 (=&4000).
(ii)	POKE &5100,&77,&34	results in the numbers &77 and &34 being placed into locations &5100 and &5101 respectively.

DOKE <I>,<I1>,<I2>,..,<In> Places the values of the expressions <I1> to <In> into memory starting at location <I>. This is like POKE, but each expression is placed into TWO bytes, the first byte being the lower significant byte.

Examples:
(i) DOKE 16384,5764 results in the numbers &84 and &16 being placed into the locations &4000 and &4001 respectively (16384=&4000, 5764 =&1684).
(ii) DOKE &5100,&77,&1234 results in the numbers &77, &00, &34, and &12 being placed into locations &5100 to &5103 consecutively.

PEEK(<I>) Returns an integer in the range 0-255, which represents the contents of the memory location (<I>).

DEEK(<I>) Returns an integer in the range -32768 to 32767 representing the contents of memory locations <I> and <I+1>. The byte <I+1> is taken as the most significant byte.

Example: Suppose location &4000 contains &C4, and &4001 contains &06. DEEK(&4000) returns 1732 (or &06C4).

PTR <J>,<I> Allows the user to set selected scratch-pad locations, without using POKE or DOKE, but using the number <J> to select the location, and <I> to be the new value. <J> may be chosen as follows:

0	HTEXT	Default or 'hard' pointer to start of BASIC program.
1	TEXT	Pointer to start of BASIC program (modified by HOLD).
2	SCMD	Pointer to standard reserved word table.
3	AUXCMD	Pointer to auxiliary (user) reserved word table.
4	ERRTAB	Pointer to normal error message table.
5	AUXERR	Pointer to auxiliary error message table.
6	SADR	Pointer to standard address table.
7	SFNADR	Pointer to standard function table.
8	AUXADR	Pointer to auxiliary address table.
9	USRLOC	Pointer to user machine-code routine (CALL as funct.)
10	DEVPTR	Pointer to list of available I/O devices.
11	DEFLST	No. of lines to 'LIST' at a time.
12	BUFPTR	Pointer to start of input buffer.
13	BUFLEN	Length of input buffer.
14	TXTTOP	Pointer to end of BASIC program.
15	VARTOP	Pointer to end of simple variable space.
16	ARRTOP	Pointer to end of array space.
17	STRBOT	Pointer to bottom of string space.
18	STKBOT	Pointer to bottom of stack area.
19	VRAM	Pointer to bottom of 'internal VDU' area.
20	LIMIT	Pointer to top of RAM used by Xtal BASIC.
21	TOPRAM	Pointer to top byte of RAM available to user.
22	LNNO	Current line number being executed.
23	DATLN	Line number of current DATA statement (undefined before a READ statement has been done).
24	DATPTR	Pointer to current position in DATA statement (if using READ statements). Can be moved to specified line by RESTORE <L> statement.

Any value for <J> outside this range will result in a RANGE ERROR. The advantage of using this command rather than the more usual method of POKEs or DOKEs is that the same command may be used in different versions of XBASIC without modifying the programs in which it is used, even though the scratch-pad area may sometimes be in a different place (e.g, the scratch-pad area for NAS-SYS/NAS-DOS is at 1000H, on the CP/M version it is at 0100H, and so on). It may not entirely achieve compatibility, since users are bound to use some other scratch-pad locations not in this list, but will reduce the modifications needed.

CARE!! Like POKE and DOKE, this command can be 'lethal' if applied indiscriminitely, since no checks can be made to see if the alterations are

being made to non-existent tables, locations within areas already used, etc. For example, the CLEAR command (see Chapter II.3) should be used to set up the LIMIT and STKBOT locations, not PTR 20,<I> and PTR 18,<I> respectively.

The current value of any of the PTR locations may be accessed by using PTR as a function, with the argument representing the desired location.

Example:
A=PTR(12) puts the start address of the current input buffer area into variable A.

HEX$(<I>,<J>) Returns the Hexadecimal string corresponding to the number <I> (the integer part only, I'm afraid!), as a string of <J> characters, where <J> must be <=4. If <J> is omitted, a value of 4 is assumed. The number is 'padded' with leading zeroes, if needed and, if <J> is too small for the number to be returned, only the lower <J> significant digits will be returned. This 'fixed' format is preferred to the 'floating' format used for numbers, since most applications with hexadecimal numbers require 2- or 4-digit output.

Examples:
HEX$(1234) returns the string "04D2".
HEX$(100,2) returns the string "64".
HEX$(356,2) also returns the string "64".

2. LOADING AND SAVING MACHINE-CODE FILES

As previously described in several places within this manual, a protected area may be set aside for the storage of machine-code routines and data, by means of the CLEAR command. With this in mind, the facility exists for loading and saving memory within this area, using the normal LOAD and SAVE commands. This may be achieved by using the file type "OBJ" to specify that it is machine-code routines or data to be loaded or saved, as opposed to a BASIC program.

Examples:
LOAD "T:ROUTINES.OBJ" Loads the machine-code routines or data from the file "ROUTINES.OBJ" on tape drive T, into the area reserved for them in the memory map (reserved by the CLEAR command). If the size of the file is larger than the area reserved, a MEM FULL ERROR will occur.

SAVE "A:MCSTUFF.OBJ",<I1>,<I2> Saves the area of memory starting from address <I1> and ending at address <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.

VIII. COMMAND/FUNCTION EXTENSION

In 1979 Crystal introduced in Xtal BASIC a capability which is still, at the time of writing, unique to their versions of BASIC. It allows the creation of an auxiliary reserved word table of up to 64 extra reserved words. This means that machine-code routines can be written and added to the interpreter as if they were commands and functions already built into the language. Some knowledge of machine-code programming is needed to take real advantage of this facility, and users who have not yet experienced machine code are advised to get studying! The ability to create what is, in effect, a personalised BASIC conforming to your own requirements is an extremely powerful tool indeed.

1. PROGRAM STORAGE

Before describing the method of adding auxiliary reserved words, it would be helpful to consider the way in which a program is stored within the text area. Many users will already realise that XBASIC does not actually use a line as typed, but instead shortens each reserved word into a unique single- or two-byte 'token'. This speeds up program execution, and also saves storage space. In addition, a null byte is appended to each line, so that we have a delimiter between each line of text (i.e, each numbered line). The line number is stored as a two-byte quantity (hexadecimal), and an additional two byte number is stored, which gives the offset to the start of the next line in the program text.

To illustrate this point, consider the following line of program text, stored in memory:

```
300 FOR I=0 TO 9: PRINT SQR(I): NEXT:END
```

A normal text editor would store this line in memory in the form of ASCII codes thus:

```
3  0  0     F  O  R     I  =  0     T  O     9  :     P  R  I  N  T     S  Q
33 30 30 20 46 4F 52 20 49 3D 30 20 54 4F 20 39 3A 20 50 52 49 4E 54 20 53 51

R  (  I  )  :     N  E  X  T  :  E  N  D  <CR>
52 28 49 29 3A 20 4E 45 58 54 3A 45 4E 44 0D
```

This would be abbreviated by XBASIC, into the following form:

```
      300   FOR   I  =  0     TO 9  :    PRINT  SQR(  I  )  :    NEXT:  END
1B 00 2C 01 8F 20 49 7E 30 20 72 39 3A 20 A2 20 D9 28 49 29 3A 20 9B 3A 8E 00
```

Here, the first two bytes give the offset to the next line (this is &001B, as you will find if you count, starting from 0 at the first byte of the offset). The next pair gives the line number (&012C= 300). Finally, you will note that the spaces are significant, and remain in the text. They make virtually no difference to the operating speed of XBASIC programs, and allow the user to lay out programs in the way that suits him/her. Removing them does, of course, save space, but this should be not be done at the expense of readability unless absolutely necessary.

Note that even '=' is treated as a reserved word, although it has only one

character anyway. This is so that execution will be faster when scanning for relational operators (including '<' and '>').

The above format still applies if the line is the last in the program, since we always indicate the end of the program text by means of a null pair, i.e, the last THREE bytes of a XBASIC program are 00. The pointer TXTTOP always points one ABOVE the last byte.

Since variable names and constants use ASCII codes from 0 to &5F (lower case variable names are internally converted to upper case immediately after entry), we may use codes &60 to &FF to represent our reserved words, and XBASIC actually uses codes &6F to &E9.

Within REM and DATA statements and between double quotes, however, this compression does not occur, so that all ASCII codes, including lower case letters and graphic characters, may be included in these cases.

LIST 'blows up' the reserved word codes (but not within quotes, REM or DATA statements!) into the actual words used, so that the user is not normally aware that all of this is going on.

2. RESERVED-WORD CONSTRUCTION

The reserved word table appears **within the interpreter as a long string of** reserved words held together, and separation is achieved by setting the top bit of the first byte in each word. In XBASIC the start of the table looks like this:

```
          S  P  C  (   S  T  E  P   T  A  B  (   T  H  E  N  .....etc.
      7B  D3 50 43 28  D3 54 45 50  D4 41 42 28  D4 48 45 4E .....
Token:    6F          70          71          72
```

The first byte of this table is the total number of reserved words allocated (&7B, or 123 in this case), in case a corrupted program should happen to contain a non-existent token.

This is, in fact, not the best place to look at the table, since all of these words are special, in that they are not commands/functions in their own right, but appear only in certain statements. If we look a bit further through the table, we pass through the arithmetic and relational operators, and finally arrive at the commands:

```
          A  U  T  O   C  H  A  I  N   C  L  E  A  R   C  L  O  S  E  ...etc.
          C1 55 54 4F  C3 48 41 49 4E  C3 4C 45 41 52  C3 4C 4F 53 45 ...
Token:    80          81          82          83
```

Associated with each command or function is an address, where the routine for executing it may be found. All of these addresses build into an address table which, at RUN time, is indexed according to the token supplied. This means that:

a. The reserved word table **is NOT** used (or needed) at **RUN time, only** the address table.
b. All commands/functions **may be accessed** at RUN time at the **same speed,** so that the order in which they appear within the tables **is immaterial.**

3. THE AUXILIARY TABLES

This much is done in a similar way on many BASIC's - the point about XBASIC is that it has TWO reserved word tables, one of which is empty, and may be expanded by the user. All user-defined reserved words are stored as TWO- byte tokens, the first one always being &FF, to distinguish them from the inbuilt reserved words. These words are stored in an AUXILIARY reserved word table, with their addresses being stored in an auxiliary address table. Both of these occupy no space within the interpreter, and so the user must create extra space in memory for the tables, in addition to that needed for the actual routines themselves.

Earlier versions of Xtal BASIC used a fixed area of RAM for holding the tables, but XBASIC uses the two pointers AUXCMD and AUXADR in the scratch -pad area, which may be set up by means of the PTR command (Chapter VII.1). Hence the user may make his/her tables as big or as small as desired, the only requirement being that the last byte of the auxiliary reserved word table MUST be an 80H code, and MUST have the correct total of reserved words given at its start.

4. COMMANDS AND FUNCTIONS

There is an important distinction to be borne in mind when creating commands or functions and each will be checked by BASIC for correct syntax when being used. If the reserved word is to be used as a function, the word MUST end with a '(' (ASCII &28) to indicate that an argument is to follow.

In a command routine, the HL register pair is treated as the text pointer and, on entry, holds the memory address of the first non-space character following the command word in the program text. On exit, HL should point to the statement separator (':') or the end-of-line byte 00. A simple RET instruction may be used to get back to BASIC. No other registers need to be preserved.

In a function routine, on the other hand, the text pointer has already been PUSHed onto the stack, and should be POPped and incremented to find the value of the argument. The routine almost always has a special end, since a closing parenthesis ')' MUST follow the argument expression.

Note: If an auxiliary reserved word has been defined and used in a program, but has subsequently been cleared from the tables (or the tables themselves have been re-initialised), the program will still be LISTable, but all references to that word will display as a decimal number preceded by a question-mark (e.g, ?64).

5. HOW TO ENTER EXTRA RESERVED WORDS

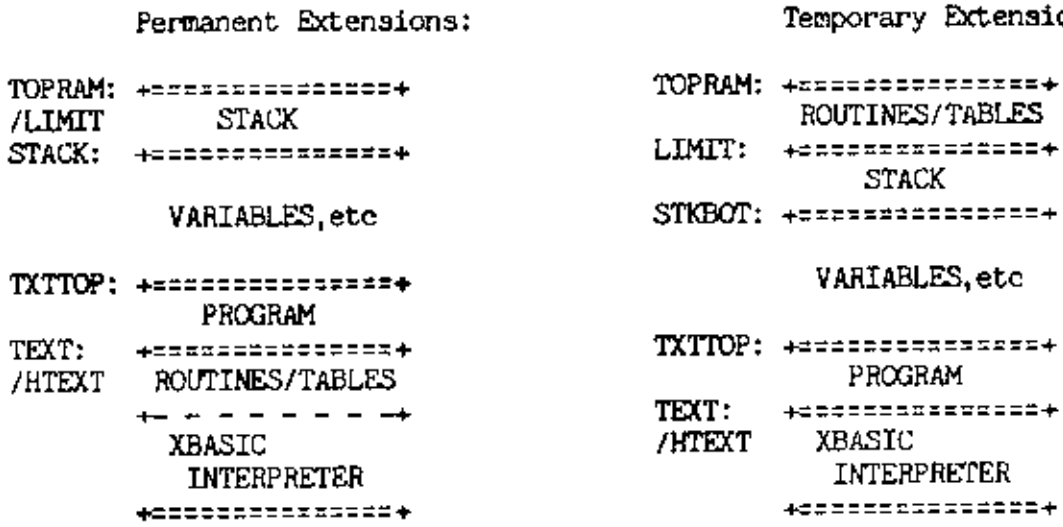
Without any further ado, let us now give the step-by-step method for adding extra words to XBASIC:

a. Decide whether the new words are to become a permanent part of XBASIC, or are just to be added on temporarily. For example, you may have some 'Tool-kit' type commands which may be required to assist with development of

a BASIC program. You may then wish to drop those routines later, so that the space may be utilised by the program developed. To do all of this, use the CLEAR command to set aside a machine-code area at the top of the memory space, put your routine(s) and tables in there, either storing them in a .OBJ file (see Chapter VII.2) or POKE/DOKEing them from a BASIC program. Temporary extensions may be removed by executing a 'Cold Start' to XBASIC. It is quite in order for a BASIC program to define its own reserved words, which it will use itself later on within the program, and then to remove these extra words on completion.

If, on the other hand, you wish to make a permanent addition to the system, this may be done by moving up the HTEXT pointer (using a PTR 0,<I> command), so that the routine(s) and tables may be placed in the area created. They then become a natural extension to the interpreter, which may subsequently be saved to disc or tape (as your operating system allows). In this case, it is advisable to make the auxiliary tables larger than required, so that additional extensions may then use the same tables.

The simplified memory maps below illustrate the two methods:



For the remainder of the discussion, it is assumed that only one command or function is being entered, although clearly the same instructions apply to the addition of several words at once.

b. Having found our free area, write the machine-code routine for performing the command/function within this area. This may be POKEd in from XBASIC, or entered from within the machine-code monitor of your machine.

c. The name of the routine, its reserved word, must now be written into the Auxiliary Reserved Word Table (pointed to by AUXCMD) as a set of ASCII codes, the first letter having its top bit set, as shown in section 2 of this Chapter. Do not forget to set up or modify the first byte of the table for the number of reserved words in the table, otherwise the command/function will return an error when later invoked! The address of the table held in AUXCMD may be entered from within BASIC by means of a PTR 3,<I> command, if desired.

d. The appropriate address in the Auxiliary Address Table (which is pointed to by AUXADR) is then set up for the start of the newly entered machine -code routine so that, when the command or function is invoked, this routine will

be executed. The address of this table may be set up in AUXADR by means of a PTR 8,<I> command, if desired. NOTE: This also applies to c., above. Do NOT use the PTR command to set up the Auxiliary Tables when making permanent extensions, because the next 'Cold Start' will simply remove them! For permanent extensions, set the pointers in the 'default scratch-pad' (which is copied to the scratch-pad area whenever a 'Cold Start' is executed). The necessary addresses are given at the end of Appendix D.

e. If permanent extensions have been made, save a new copy of XBASIC onto tape or disc before running it up, not forgetting to include the area added to the end of the Interpreter!

f. Re-enter XBASIC via either the COLD START or the WARM START entry points given in Chapter 0, if the above operations were carried out from within the monitor of your machine.

Your new reserved word will now behave exactly as though it had always been a part of XBASIC (if there are no bugs in the routine!!). Now is the time to try the examples of extra reserved words given in Appendix D, which should illustrate these instructions.

AWFULLY IMPORTANT NOTE: In scanning the reserved word tables for the compression of text typed in, the Auxiliary Reserved Word Table is scanned before the Standard Reserved Word Table, so that it is possible to use complete words from the existing table as part or whole of Auxiliary words. Thus the following would be perfectly acceptable as reserved words:

PRINTUSING SINH DELAY VALUE

and would not affect the appropriate existing reserved words which they replace.

However, if READ was included in the Auxiliary tables, it would assume priority over the existing word READ, with rather interesting results! In particular, a .XBS file containing READ statements would continue to execute the existing READ statement, but any lines added to that program would correspond to the new READ command, if READ was typed into any of those lines. This option should therefore be used with some care.

APPENDIX A -- INDEX TO RESERVED WORDS AND ERROR MESSAGES

1. RESERVED WORDS

A complete list of reserved words is given below, together with their associated 'tokens' and main pages on which descriptions of them may be found. The TYPE column tells whether the word is a Command, Function, Separator (i.e, it appears only as a part of another statement, e.g, THEN in an IF statement), or Operator, and CF indicates that the word may be used either as a Command or Function. An asterisk indicates that the word is only available in the disc version of XBASIC.

	CODE			PAGE
	HEX	DECIMAL	TYPE	
+	74	116	O	5
-	75	117	O	5
^	76	118	O	5
*	77	119	O	5
/	78	120	O	5
>	7D	125	O	6
=	7E	126	O	6
<	7F	127	O	6
ABS	C4	196	F	27
AND	7A	122	O	6
APPEND	B1	177	*C	48
ASC	C5	197	F	31
ATN	C6	198	F	27
AUTO	80	128	C	12
CALL	B8	184	CF	65
CHAIN	81	129	C	14,17
CHR$	C7	199	F	31
CLEAR	82	130	C	14
CLOSE	83	131	C	47
CLS	84	132	C	18
CONT	85	133	C	18
COS	C8	200	F	27
CREATE	86	134	C	47
DATA	87	135	C	24
DEEK	C9	201	F	66
DEF	88	136	C	34
DEL	89	137	C	11
DIM	8A	138	C	18
DIR	B2	178	*C	26
DOKE	8B	139	C	66
DRIVE	8C	140	C	46
ELSE	8D	141	S	20
END	8E	142	C	18
EOF	E2	226	S	62,63
ERA	B3	179	*C	26
ERR	E0	224	F	62,63
ERL	E1	225	F	62,63
EVAL	CA	202	F	27
EXP	CB	203	F	27
FN	E3	227	F	34
FMT	B0	176	C	39
FOR	8F	143	C	18

GOSUB	90	144	C	19
GOTO	91	145	C	19
HOLD	92	146	C	15,17
HEX$	CC	204	F	68
IF	93	147	C	20
INCH	E4	228	F	28,31,49
INP	CD	205	F	38
INPUT	94	148	C	21,36,49
INT	CE	206	F	28
IOM	B9	185	CF	10,40
KBD	E5	229	F	28,31
LEFT$	DD	221	F	32
LEN	CF	207	F	32
LET	95	149	C	21
LIST	96	150	C	11
LN	D0	208	F	28
LOAD	97	151	C	12
LOCK	B4	180	*C	26
LOG	D1	209	F	28
MGE	98	152	C	15
MID$	DE	222	F	32
MOD	79	121	O	5,6
MON	99	153	C	11
MUL$	E6	230	F	32
NEW	9A	154	C	11
NEXT	9B	155	C	19
NOT	E7	231	F	6
NULL	BA	186	CF	42
OFF	9C	156	C	63
ON	9D	157	C	22,62
OPEN	9E	158	C	47
OR	7B	123	O	6
OUT	9F	159	C	38
PEEK	D2	210	F	66
PI	E8	232	F	28
POINT	D3	211	F	28
POKE	A0	160	C	65
POP	A1	161	C	20
POS	D4	212	F	28
PRINT	A2	162	C	22,36,49
PTR	BB	187	CF	66
READ	A3	163	C	23
REM	A4	164	C	24
REN	B5	181	*C	26
RENUM	A5	165	C	15
RESET	A6	166	C	24
RESTORE	A7	167	C	24
RETURN	A8	168	C	20
RIGHT$	DF	223	F	32
RND	D5	213	F	29
RUN	A9	169	C	14
SAVE	AA	170	C	13
SCRN$	D6	214	F	32
SEP	BC	188	CF	39
SET	AB	171	C	24
SGN	D7	215	F	29
SIN	D8	216	F	29
SIZE	E9	233	F	29
SPC	6F	111	S	30
SPEED	BD	189	CF	42

SQR	D9	217	F	29
STEP	70	112	S	18
STOP	AC	172	C	25
STR$	DA	218	F	33
SWAP	AD	173	C	24
TAB	71	113	S	30
TAN	DB	219	F	30
TO	72	114	S	18
THEN	73	115	S	20
UNLOCK	B6	182	*C	26
VAL	DC	220	F	33
VERIFY	AE	174	C	14
WAIT	AF	175	C	38
WIDTH	BE	190	CF	42
XOR	7C	124	O	6
ZONE	BF	191	CF	42

2. ERROR MESSAGES

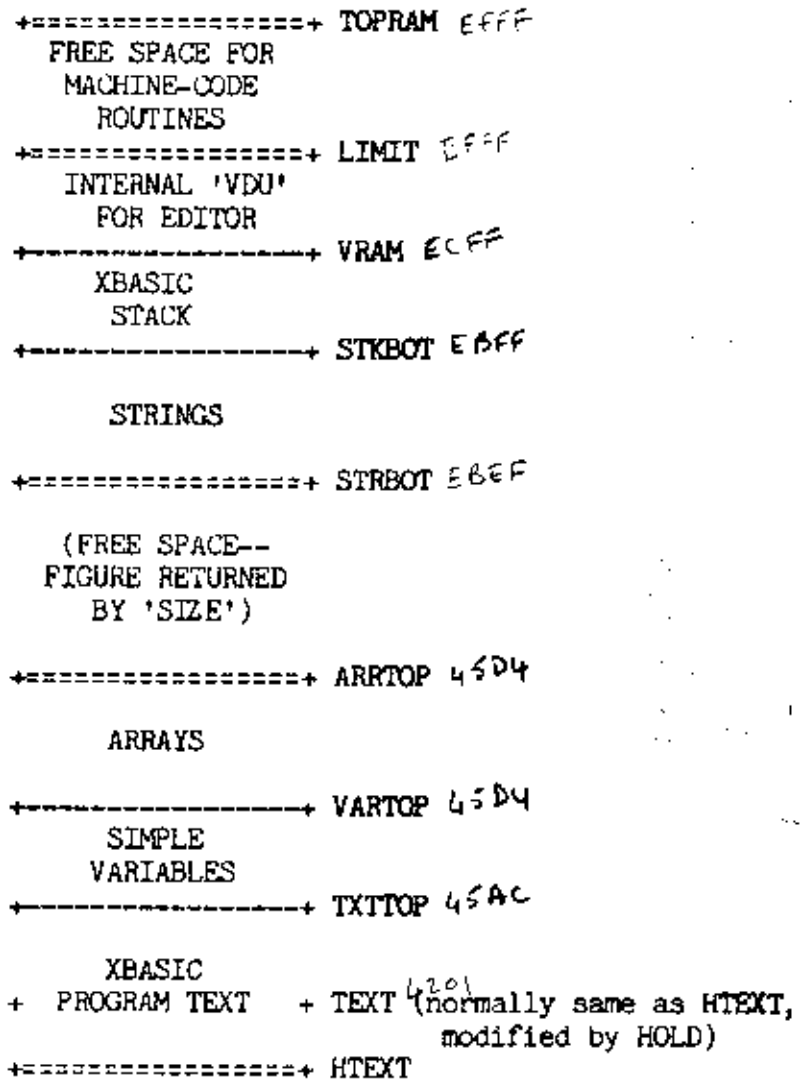
	CODE	
	HEX	DECIMAL
Break	00	0
Next	01	1
Syntax	02	2
Return	03	3
Data	04	4
Qty	05	5
Ovfl	06	6
Mem Full	07	7
Branch	08	8
Range	09	9
Dimension	0A	10
Division	0B	11
Stack Full	0C	12
Type	0D	13
Cmd	0E	14
Str Ovfl	0F	15
Str Complex	10	16
Cont	11	17
Fn Defn	12	18
Operand	13	19
Bad Data	14	20
End of Text	15	21
File	16	22
Drive Select	17	23
File Type	18	24

DISC ERRORS: Only available in the disc version.

No File	19	25
File Exists	1A	26
File Locked	1B	27
Disc Locked	1C	28
Disc Seek	1D	29
Disc Full	1E	30
Dir Full	1F	31

APPENDIX B -- HARDWARE CONFIGURATION (CASSETTE VERSION)

1. MEMORY MAP FOR XBASIC



The interpreter itself occupies the area &1000 to &41FF.

2. XBASIC SCRATCH-PAD LOCATIONS

The scratch-pad is divided into two parts - that part which must be initialised with certain default values, according to the hardware being used, and the part which simply requires setting to zero. The 'default' area is in fact copied from an area within the interpreter called the HARD scratchpad, starting at HTEXT, which may be found at 3A7CH. The default settings for this area are shown in the table below, given for the tape version:

```
1000    XCOLD:  JP      XCOLD1  ; ENTRY TO 'COLD START' ROUTINE
1003    XWARM:  JP      XWARM1  ; ENTRY TO 'WARM START' ROUTINE
```

```
LOC.    NAME     DEFAULT VALUE     REMARKS

1006    TEXT:    &4201             ; PTR TO START OF BASIC PROGRAM (PTR(1))
1008    SCMD:    &3E35             ; PTR TO STANDARD CMD TABLE     (PTR(2))
100A    AUXCMD:  &3FC9             ; PTR TO AUXILIARY CMD TABLE    (PTR(3))
100C    ERRTAB:  &3D8E             ; PTR TO STANDARD ERROR TABLE   (PTR(4))
100E    AUXERR:  &3E34             ; PTR TO AUXILIARY ERROR TABLE  (PTR(5))
1010    SADR:    &3FCB             ; PTR TO STANDARD ADDR TABLE    (PTR(6))
1012    SFNADR:  &4053             ; PTR TO STANDARD FN ADDR TABLE (PTR(7))
1014    AUXADR:  0                 ; PTR TO AUXILIARY ADDR TABLE   (PTR(8))
1016    USRLOC:  &15CA             ; PTR TO USER 'CALL' ROUTINE    (PTR(9))
1018    DEVPTR:  &342D             ; PTR TO START OF I/O DEVICES   (PTR(10))
101A    DEFLST:  65535             ; LINES TO 'LIST' AT A TIME     (PTR(11))
101C    BUFPTR:  &0080             ; PTR TO INPUT BUFFER AREA      (PTR(12))
101E    BUFLEN:  127               ; LENGTH OF INPUT BUFFER        (PTR(13))
101F    RNDMOD:  1                 ; MODE OF RANDOM No. (INTEGER/REAL)
1021    RNDNO:   &8047B709         ; HOLDS LAST RANDOM No. (0.780148)
1025    NDISCS:  0                 ; No. OF DISCS ALLOWED
1026    NTAPES:  1                 ; No. OF TAPES ALLOWED
1027    XLEN:    48                ; VDU COL SIZE
1028    YLEN:    16                ; VDU ROW SIZE
1029    VDUSIZ:  &0300             ; SCREEN SIZE (XLEN*YLEN)
102B    STKSIZ:  &0100             ; STACK SIZE
102D    DEFDRV:  &13 ('T'-'A')     ; CURRENT TAPE/DISC DRIVE
102E    IOMOD:   &FFFE             ; O/P MODE, USED BY EDITOR, ETC (IOM(0-15))
1030    TXFIGS:  0                 ; NUMERIC DISPLAY FORMAT FOR FMT
1031    SEPRTR:  &2C (',')         ; SEPARATOR FOR 'INPUT'         (SEP)
1032    TABCHR:  &20 (' ')         ; CHARACTER FOR 'TAB' FUNCTION
1033    WIDTHT:  36                ; END OF VDU LINE FOR ZONING    (ZONE(0))
1034    ZWIDTH:  14                ; PRINT ZONE WIDTH              (ZONE(1))

        ABOVE IS COPIED AREA - BELOW IS AREA INITIALISED TO ZERO

1035    WIDTH:   DEFS   1          ; WIDTH FOR PRINTOUT            (WIDTH)
1036    PRTCOL:  DEFS   1          ; PRINT COLUMN                  (POS(0))
1037    ROWCOL:  DEFS   2          ; CURRENT CURSOR COORDs         (POS(1 or 2))
1039    CURPOS:  DEFS   2          ; LOCATION OF INTERNAL CURSOR
103B    VSPEED:  DEFS   1          ; DELAY FOR CHARACTER O/P       (SPEED)
103C    NULCNT:  DEFS   1          ; NULLS AFTER CR?               (NULL)
103D    KEYIN:   DEFS   1          ; INPUT CHARACTER FROM QUIKCK
103E    PRTCHR:  DEFS   1          ; LAST OUTPUT CHARACTER
103F    ODEV:    DEFS   1          ; CURRENT OUTPUT DEVICE
1040    IDEV:    DEFS   1          ; CURRENT INPUT DEVICE
1041    TXTTOP:  DEFS   2          ; PTR TO END OF BASIC PROGRAM   (PTR(14))
1043    VARTOP:  DEFS   2          ; PTR TO END OF VAR SPACE       (PTR(15))
1045    ARRTOP:  DEFS   2          ; PTR TO END OF ARRAY SPACE     (PTR(16))
1047    STRBOT:  DEFS   2          ; PTR TO BOTTOM OF STR. SPACE   (PTR(17))
1049    STKBOT:  DEFS   2          ; PTR TO BOTTOM OF STACK AREA   (PTR(18))
104B    VRAM:    DEFS   2          ; PTR TO START OF VDU AREA      (PTR(19))
104D    LIMIT:   DEFS   2          ; PTR TO TOP OF USED AREA       (PTR(20))
104F    TOPRAM:  DEFS   2          ; PTR TO TOP BYTE OF RAM        (PTR(21))
1051    LNNO:    DEFS   2          ; CURRENT LINE NO.              (PTR(22))
1053    DATLN:   DEFS   2          ; LINE NO. OF CURRENT DATA STATEMENT  (PTR(23))
1055    DATPTR:  DEFS   2          ; CURRENT POSN. IN DATA STATEMENT     (PTR(24))
1057    TXTPTR:  DEFS   2          ; SAVE TEXT PTR AT START OF STATEMENT
1059    EXPPTR:  DEFS   2          ; SAVE TEXT PTR IN EXPRESSION
105B    LNNZ:    DEFS   2          ; 'OLD' LINE PTR FOR CONT
105D    TXTPTZ:  DEFS   2          ; 'OLD' TEXT PTR FOR CONT
105F    ENDLST:  DEFS   2          ; LAST LINE FOR LIST
```

```
1061    PTR:     DEFS    2       ; GENERAL-PURPOSE POINTER
1063    PTR1:    DEFS    2       ; GENERAL-PURPOSE POINTER
1065    DIMFLG:  DEFS    1       ; FLAG FOR DIM/FNDVAR ROUTINES
1066    NTYPE:   DEFS    1       ; TYPE OF EXPRESSION EVALUATED
1067    VTYPE:   DEFS    1       ; VAR/ARRAY TYPE USED BY FNDVAR
1068    STRFLG:  DEFS    1       ; FLAG TO INDICATE HOUSE-CLEAN DONE
1069    GARPTR:  DEFS    2       ; GARBAGE-COLLECT POINTER
106B    ASNPTR:  DEFS    2       ; TEMP. PTR FOR 'LET'
106D    STRPTR:  DEFS    2       ; PTR TO END OF STRLST
106F    STRLST:  DEFS    10      ; STR SUB-EXPRESSION LIST
1079    CHAR:    DEFS    2       ; TEMP. STRING 'ACCUMULATOR'
107B    CHRADR:  DEFS    2       ; ADDR. ASSOCIATED WITH CHAR
107D    RDFLAG:  DEFS    1       ; IN READ/INPUT, FLAG TO SHOW WHICH
107E    FPA:     DEFS    4       ; F.P ACCUMULATOR
1082    TEMP:    DEFS    1       ; LOC. USED IN F.P CALCULATION
1083    PRTTXT:  DEFS    17      ; TEXT AREA FOR FORMING NUMBERS
1094    TXNEXP:  DEFS    1       ; TEMP. EXPONENT VALUE FOR FORMATTING NUMBERS
1095    TXNBUF:  DEFS    8       ; DIGIT BUFFER FOR FORMATTING NUMBERS
109D    ZERMOD:  DEFS    1       ; COPY OF CURRENT ERROR MODE (FOR 'STOP')
109E    ERRMOD:  DEFS    1       ; CURRENT ERROR MODE
109F    ONERRLN: DEFS    2       ; LINE NO. OF ON ERR STATEMENT
10A1    ERRNO:   DEFS    1       ; NO. OF LAST ERROR GENERATED
10A2    ERRLN:   DEFS    2       ; LINE NO. OF LAST ERROR
10A4    EOFMOD:  DEFS    1       ; CURRENT EOF MODE
10A5    ONEOFLN: DEFS    2       ; LINE NO. OF ON EOF STATEMENT
10A7    STKSAV:  DEFS    2       ; SAVE STACK FOR ON ERR.. STUFF
10A9    VDATTS:  DEFS    32      ; 'START-OF-LINE ROW' LIST FOR INTERNAL 'VDU'
10C9    ASCFLG:  DEFS    1       ; FLAG TO SHOW WE ARE HANDLING .ASC FILE
10CA    FDESC:   DEFS    4       ; CURRENT FILE DESCRIPTOR NAME
10CE    OTDESC:  DEFS    4       ; O/P FILE DESCRIPTOR NAME
10D2    INDESC:  DEFS    4       ; I/P FILE DESCRIPTOR NAME
10D6    FILTYP:  DEFS    1       ; STORE TYPE OF FILE IN SAVE & LOAD
10D7    FSPEC:   DEFS    28H     ; FIXED FILE SPEC. AREA, FOR SAVE, LOAD, etc
10FF    FBUFF:   DEFS    80H     ; BUFFER FOR FSPEC
```

3. INPUT/OUTPUT

As described in Chapter IV, the I/O device specification is given by IOLIST, pointed to by DEVPTR. As supplied, the devices are as follows:

0 - Normal Nascom VDU, 48x16, but with all 16 lines scrolling. This overcomes the 'top-line printing' problems apparent when using NAS-SYS 1, where printing a character to the non-scrolling line resulted in the cursor moving to the bottom of the screen, accompananied by a scroll of the screen!

Input device 0 is the NASCOM keyboard.

1 - Output to serial printer, using bit 7 of port 0 as DTR connection. This allows handshaking, and a 1 on bit 7 indicates that the printer is ready to receive data from the RS232 port.

The input device 1 is currently as for device 0.

2 - Output to and Input from the RS232 port, in the usual way (i.e, with no special handshake connections).

The user can, of course, define his/her own IOLIST, for the various other I/O possibilities, and is referred to Chapter IV.1 for the relevant details.

4. GRAPHICS SUPPORT

SET, RESET and POINT have already been described in Chapter III.1 and III.3. All that need be said here is that the resolution provided by these 'dot' graphics is 96 by 48 (0-95 and 0-47 being the respective ranges). Coordinates outside these ranges may be specified, up to a maximum of 255, with 'wrap-around'.

Those used to Nascom ROM BASIC should note that SET and RESET may NOT be used with parentheses around the coordinate pair, and that the vertical coordinate works more logically (i.e, 0 is at the very top of the screen, with 47 at the very bottom), rather than making special exceptions for the 'non- scrolling row'.

APPENDIX C - XBASIC USEFUL SUBROUTINES

Note: All of the addresses given within this appendix are specified in Hexadecimal. This appendix has been provided for assisting the generation of extra reserved words in an efficient manner. It is not complete, but we think that the most useful routines are all present in this list!

1. ERROR MESSAGES

Not much more need be said about this than has already been covered in Chapter VI, except to give the address of the routine ERROR, which actually handles errors, and may be found at 15CF. The only register which matters here is E, which contains the error number, as defined in Chapter VI. It is not necessary to CALL this routine, just jump to it!

2. USER-FUNCTION TERMINATION ROUTINES

```
1185    FNBIT:      ; RETURN BIT VALUE IN CARRY FLAG.
1188    FNENDB:     ; RETURN BYTE VALUE IN A.
118B    FNENDI:     ; RETURN INTEGER VALUE IN AB  (high byte in A).
118E    FNENDF:     ; RETURN NUMERIC VALUE IN FPA (Floating-point).
1191    FNEND:      ; RETURN EXPR. VALUE IN FPA  (may be a string).
```

These routines should **NOT** be CALLed, but used to terminate your function routine. The routines all assume that the text pointer is on stack, so that the registers may contain anything on entry to these routines (except, of course, for the ones returning results!). Also, see section 8 of this appendix for the use of STREND, the usual way of returning string results.

3. GENERAL-PURPOSE TEXT SCANNING ROUTINES

As explained in Chapter VIII, XBASIC uses the HL register pair as the pointer to the current position in the program text. The following routines make use of this:

RDLN 3587 Reads in a line of text from the keyboard or current input device to the BUFFER, pointed to by BUFPTR. This makes use of the editing facilities described in Chapter II, according to the IOM setting currently in use. On entry, if 'Line Edit' mode is in force, the character contained in A is printed as a prompt at the start of the line.

On exit, the carry flag is set if the line has been abandoned by <ESC>, but is reset if <CR> has been used to complete the line. In this case, the line in the buffer is terminated with a 00 byte, and HL is left pointing to one byte before the start of the buffer. Registers affected: A and HL.

PR 34E7 Print character in A register, to VDU or current output device. The side-effect of this is that the location PRTCOL is adjusted to give the correct column on the screen/printer, for TABs, etc. In addition, a delay is imposed if the SPEED command has been used to slow down the print rate.

PRTNUM 11FC Prints the contents in the HL register pair as an integer in the range 0-65535. All registers may be affected.

PRM 11B9 Prints the message immediately following the sub-routine call, terminated by having the MSB of the last character set. This means that all other character codes must have ASCII values in the range 0-&7F. Thus, to print "Hello there", we do:

```
CD B9 11  48 65 6C 6C 6F 20 74 78 75 72 E5
          H  e  l  l  o     t  h  e  r  e
```

On exit, A holds the last character printed, still with its top bit set, and the return address is that immediately following the last character in the message. No other registers are affected.

CPHLDE 1197 Compare HL and DE and return flags set as follows: Carry - Set if HL<DE, reset if HL>=DE. Zero - Set if HL=DE. Registers affected: A.

LTRCHK 11DF Places the character contained at (HL) in A, and tests to see if it is a letter in the range A - Z (i.e, a capital letter). Carry is Reset if it is a capital letter, and Set if it is any other character. No other register is affected.

LWRTST 11D0 Loads character from (HL) into A and, if in the range &60-&7F, converts it to upper-case (in the range &40-&5F). Only A and the flags are affected.

IGBLK 11AA Increments HL, until the first non-space character is found. On return, A contains the character found, and HL points to that character. Z flag is set if at the end of statement (null or ':' found), and C flag set if numeric character found (0-9).

TSTC 11A2 Test character at (HL), ensuring that it is the same as that specified immediately after the call. If not, a Syntax Error occurs. This is effectively a four-byte call, e.g, CD xx xx 28 looks for a '('. A contains the test character, and HL points to the next non-blank character following the tested one. Note that we may also use this routine to test for a reserved word token.

TSTCOM 119D
RPARN 1192 Special cases of TSTC, test for comma ',' and right parenthesis ')' respectively. These only require 3 bytes instead of four, though!

FNDLN 123B Searches for the line in the program text given by DE, from the start of text. Returns with the following conditions:

Carry and Zero set: Line found, BC points to start of line, HL points to start of following line (or to 0000 if the line found is the last in the text), as described in Chapter VIII.1 .

Carry reset, Zero set: Line not found, and end of text reached. BC then points to the start of the last line of text, and HL=0000.

Carry and Zero reset: Line not found, but we have found a line with a number larger than that searched for, BC pointing to that line, and HL pointing to the next line (or 0000).

Other registers affected: A will be affected, but DE will remain unchanged.

NXTLN 123E As for FNDLN above, but this time searches for the line given in DE from the current position in the text, given in HL.

COMPRSS 1761 Routine to take a line of text in the buffer starting at the location given in HL, and terminated by a 00 byte, and which generates the same line in the compressed format given above, in the input buffer (BUFFER). Note that the new line is ALWAYS shorter than the original. In normal use, when entering a line of text into a program, the compressed line overlays the input line, since the pointer to the original text is always in front of that to the compressed text. In addition, the line number is not considered here, since HL is pointing at the next non-blank character after the line number (if one has been used). COMPRSS does NOT generate a compressed line number nor the pointer to the next line.

Registers affected: All. HL points to one byte before the start of the buffer on exit, DE points to the last byte plus two in the compressed line, and C holds the number of bytes in the compressed line, plus four to take account of the space needed for the line number and pointer.

4. FLOATING-POINT FEATURES

a. Representation of floating-point numbers.

A floating-point number in XBASIC is stored in four consecutive bytes. There are four bytes reserved within the scratch-pad, used for floating- point calculations, called the Floating-Point Accumulator (FPA), and a further byte TEMP is used by the f.p routines for storing temporary calculations. Apart from these, only the registers and the stack are used for f.p calculations.

The high byte of the FPA is the exponent, which is a signed power of two. Note that the sign bit is 0 if NEGATIVE, 1 if POSITIVE (for a reason which will become apparent later). The lower 3 bytes form a signed mantissa, the top bit of the top byte being the sign (this time 0 if POSITIVE, 1 if NEGATIVE!). The mantissa is a number between 0 and 1, with the binary point coming above the top bit.

If we let e = Exponent byte, and m = Mantissa bytes, we express any f.p number N as:

$$N = (1 + m) * 2\uparrow(e - 1),$$

with the added convention that any number with a zero exponent is taken as 0. Now we see why 1 is used for a positive sign on the exponent - e=01 must represent 2↑(-128), and 0 is clearly smaller than this (not much!). Note that e=80 represents 2↑(-1), or 0.5 up to 1 (depending on the value of m). The advantage of using this convention for 0 is that we can initialise variables and arrays simply by filling them with 0's (each element is then zero).

This is still probably as clear as mud(!), so let's have a few examples, to illustrate the system:

Decimal Number	Hex (f.p) representation	Remarks
0	00 00 00 00	Zero
1	81 00 00 00	2↑0
2	82 00 00 00	2↑1
3	82 40 00 00	1.5*2↑1
-3	82 C0 00 00	
3.141593	82 49 0F DB	Pi
0.6931472	80 31 72 18	Ln(2)
65536	91 00 00 00	2↑16

The RANGE over which we can operate is determined by e, and is thus: 2↑(-128) < N < 2↑127, which is 2.938736 * 10↑(-39) to 1.701412 * 10↑38.

The ACCURACY of calculations is determined by the length of m, which in this case represents 1 part in 2 24, or an error of < 5.960464 * 10↑(-8), which is better than 7 sig. figs. However, to try and account for rounding errors, we allow one guard digit, and so you will note that all numbers are printed to 6 sig. figs (even this does not ALWAYS account for ALL errors, and you will note, for instance, that 3↑4 is displayed as 81.0001, and not 81, as it should be! This is mainly due to problems with conversion from binary to decimal, as well as the accuracy of the method used for calculating powers).

b. Floating-point functions and operators.

The addresses of the single-argument f.p functions are as follows. In each case, the argument is taken from the FPA on entry, and the result returned in it on exit:

LOG	2A51	LN	2A5D	EXP	2D14
SIN	2D94	COS	2D8E	TAN	2D7A
ATN	2D54	RND	2ED0	ABS	2C2A
SGN	2C36	INT	2C8A	SQR	2CC7

By 'operators' we mean those in which we are dealing with TWO f.p quantities. In general, we do a calculation in the form a = b o a, where a = contents of FPA, b = contents of top four bytes of stack, and o is the operation performed. On the stack, the top pair of bytes represent the exponent (high byte) and top byte of mantissa. For each operator, there is another entry point (given a suffix '1'), in which b is stored in the BCDE registers. Here, B contains the exponent, C the high byte of the mantissa, and DE the rest of the mantissa. We call the set of four registers used in this way the Floating -Point Register (FPR). The result of any of these operations is, of course, returned in the FPA.

ADD	295B	ADD1	2970	SUB	296B	SUB1	296D
MULT	2A9C	MULT1	2A9E	DIV	2AEE	DIV1	2AF0
POWER	2CD0	POWER1	2CD2	ADDN	2962	SUBN	2967
MOD	2B75	MOD1	2B77	MUL10	2BA4	DIV10	2AE2

Note: POWER is actually calculated as: X ↑ Y = EXP (Y * LOG(X)), with the convention that X ↑ 0 = 1 for X>=0 and 0 ↑ Y = 0 for Y>0, and X ↑ Y is not defined for X<0 or for X=0 and Y<0.

MUL10 and DIV10 respectively multiply and divide the contents of the FPA by 10, leaving the result in the FPA.

ADDN and SUBN are like ADD1 and SUB1, except that HL points to a memory location at which b may be found. You can place a constant here, or even a

temporary result, if you wish. XBASIC stores a large table of constants within the Interpreter, and here are some of the more useful ones:

HALFPI	2F0B	Pi/2	HALF	2F0F	0.5
TWOPI	2F07	Pi*2	QTR	2F13	0.25
ONE	2ECD	1	NEGONE	2EC9	-1

c. Other useful F.P routines.

STKFPA 2C51 Returns with the FPA on the stack, in the form shown above. Destroys the DE registers.

LDFPR 2C5E Copies the FPA to the FPR, leaving HL pointing to TEMP.

STFPR 2C76 Copies the FPR to the FPA, without affecting any registers.

HLTFPA 2C73 Copies the four bytes starting at (HL) into the FPR AND FPA, leaving HL pointing to the byte following the block of four.

HLTFPR 2C61 Copies the four bytes starting at (HL) into the FPR, leaving HL as above, but not affecting the FPA.

FPATHL 2C7F Copies the FPA into the four bytes starting at (HL), leaving HL as above, DE pointing to TEMP, B=00 and A= exponent of FPA.

FPRTHL 2C6A Copies the FPR into the four bytes starting at (HL), leaving HL as in HLTFPA, but no other registers affected.

DETOHL 2C82 As above, but copies the four bytes starting at (DE) to those starting at (HL).

CHKSGN 2C1B Test sign of FPA, returning A=00 if FPA=0, A=01 if FPA>0 and A=FF if FPA<0. This does not change any other registers.

CHGSGN 2C2E Changes the sign of the FPA, turning it from a positive to a negative number, or vice versa. This affects A and HL.

d. Polynomial evaluation.

XBASIC uses routines called POLY and POLY1 to evaluate polynomials for the transcendental functions LOG, EXP, SIN, and ATN. All of the others are derived from these 'big four'. Both of these functions use HL on entry to point to a table of coefficients, and these are then used to form the required polynomial. The first byte of the table gives the number of coefficients, and each coefficient then follows (highest order coefficient first), stored in four bytes as usual. The result is, of course, returned in the FPA. Now, let us assume that the FPA holds a number X, on entry, and Y on exit to/from these routines, and that there are n+1 coefficients C0-Cn:

POLY1 2DED Returns an evaluation of a polynomial of the form:

Y= C0 + C1* X + C2* X↑2 + C3* X↑3 + ... + Cn* X↑n

The table looks like this:

n+1 Cn C3 C2 C1 C0

HL points here on entry.

POLY 2DD1 Returns an evaluation of a polynomial of the form:
Y= C0* X + C1* X↑3 + C2* X↑5 + ... + Cn* X↑(2*n+1),
and the table looks the same as above.

All other registers may be affected by these routines.

SINTAB 2EF6 LOGTAB 2EA3 ATNTAB 2ED1 EXPTAB 2EB0 are the ones used within XBASIC, but they won't look like they do in your standard mathematics books, because we use a special method known as CHEBYSHEV economisation to calculate these functions to the required degree of accuracy and the same degree of efficiency over the appropriate range of values.

5. EXPRESSIONS AND FUNCTIONS

To get a number or a complicated expression containing numbers, functions and operators, into the f.p format described in the preceding paragraphs, we use a set of very powerful routines to evaluate them. In all of the following cases, HL points to the position in the text where the expression is to be found and, unless otherwise stated, all register contents may change:

EXPR 255A The general expression evaluation routine, for calculating both numeric AND string expressions. The numeric result (or string pointer in the case of string expressions) is simply returned in the FPA, and NTYPE contains the type of expression returned (0 for numeric, 1 for string). The expression can be as simple or as complicated as desired, and may even contain logical or relational operators.

EXNMCK 2541 As for EXPR, but only accepts a numeric expression, and returns 'Type Error', if a string expression is found.

PARNZ 261C As for EXPR, but expects the expression to be enclosed inside parentheses (), returning 'Syntax Error' if not.

PARN 2556 As for PARNZ, but only looks for a left bracket '(', so that more expressions can be evaluated, perhaps separated by commas (use TSTCOM to test for separating commas), finally finishing with RPARN to test for the right bracket.

FCHNUM 2F34 Tests for a f.p number (NB, NOT an expression, just a numeric constant), leaving HL pointing to the first non-numeric text character. Examples of values accepted by this routine are:

```
1          2.34     -51.76548 (rounded to -51.7655)
-1.23E-07           &7FE (hexadecimal value, taken as 2046)
```

The result is returned in the FPA.

GETNM 251C Like FCHNUM, but this time the number must be an integer in the range 0-65529, and 'Syntax Error' is returned if it is not in this range. The number is returned in DE, and HL again points to the first non-numeric character. This routine is mainly used for fetching line numbers in the text (e.g, after GOTO or GOSUB statements). This routine leaves BC unaffected.

TXTNUM 2FD4
TXT1 2FD7 Converts the number in the FPA into an ASCII format number, starting at PRTTXT (or at the position given by HL in the case of TXT1). The format in which the number is returned depends upon the FMT statement that is in force (i.e, the number of leading and trailing figures given in the number). This is also dictated by the scratch-pad location TXFIGS, which contains the number of leading figures allowed in its top half, and the

number of trailing figures in its bottom half (e.g, if TXFIGS contains 42H, we have 4 leading figures and two decimal places.

The number stored is terminated by a 00 byte, and the routine returns with HL at its original value.

UEXINT 24E7 As for EXNMCK, but this time makes the expression into an integer, which must be in the range -65535 to +65535, returning the result in DE, as a signed 16-bit quantity. Note that, due to the range allowed, equivalent positive and negative values may be used interchangeably, e.g, -65535 is equivalent to +1.

INTEXP 24F4 As for UEXINT, but restricts the range to 0 to +65535.

I255 250D Here, we restrict the range to 0 to +255, and the result is returned in A as well as DE (D=00, of course).

In these last three routines, 'Qty Error' is returned if the number is not in the correct range described.

6. ROUTINES TO PRODUCE NUMERIC RESULTS

It is often necessary, after obtaining one or more numeric expressions and doing some manipulation, to return a numeric result. If the result is an f.p number, there is no problem - we just return the result in the FPA. If we have an integer result, we can use the following routines to return the result in the FPA suitably converted. Note that these may be CALLed, unlike the function terminating routines described earlier.

FORMNUM 146B Converts a two-byte integer (-32768 to 32767) into an f.p number. The integer is stored in the A and B registers (high byte in A), and all of the other registers are affected.

FORMPOS 13E1 As for FORMNUM, but returns the number unsigned, i.e, it assumes the number to be in the range 0-65535. In this case, the integer is taken from HL on entry.

7. TYPE CHECKING ROUTINES

There are three routines provided for checking the type of variable returned by a sub-expression or expression:

NUMCHK 2544 Ensures that the expression just evaluated is a number.

STRCHK 2547 Ensures that the expression just evaluated is a string.

TYPMCH 2549 Checks that the type of one expression matches another. This is done by making the Accumulator represent the type of the first expression (0 if numeric, 1 if string).

In all of these cases, we return a TYPE ERROR if the wrong type was found, and the NTYPE contains the type of the expression last evaluated. Only the A register and flags are affected by these routines.

8. STRING EXPRESSIONS

We already know that we may use EXPR to return the pointer to a string expression in the first two bytes of the FPA. In order to process the string correctly, we use the following routine:

FCHSTR 2309 This does a call to STRCHK (to ensure that the expression just evaluated was a string), and exits with HL pointing to the length byte of the string expression. It also checks to see whether the string was a 'temporary' sub-expression. String sub-expressions are stored at STRLST in the scratch-pad, and serve to stack the pointers to strings which are created within an expression and then forgotten about when the expression has been completely evaluated. We use CHAR to store the current 'temporary string' (for example, the result of concatenating several strings, which, until assigned to a variable, would have nowhere to keep its pointer).

Registers affected: Apart from HL, the contents of all of the registers are modified, but their values are not important.

LEN1 233E If you want to use LEN, you should in fact use this routine, which calls FCHSTR, and then returns the length of the string in A. HL still points to the length byte. TYPE is set to 0, to indicate a numeric result, and so is D.

ASC1 234D Similarly, use this routine where you want to use ASC. This calls LEN1, returns the address of the start of the string in DE, and the first character in A. HL is left pointing to the LAST byte of the string pointer, not the first, as it was in the above two cases.

STRSPC 2105 Creates space for a new string within the string space, the required space being given by A. All other registers are affected. If there is insufficient string space, a 'house-cleaning' operation is initiated, which removes all strings to which there is no longer a pointer (i.e, the string variable which was pointing to it has now been assigned to another string). DE is left pointing to the first byte of this free space, and STRBOT is lowered by the appropriate amount.

ASNSTR 2173 As for STRSPC, but then assigns this string space to the 'temporary String accumulator' (CHAR), writing the length to CHAR, and the start address to CHAR+2. This is thus the routine to use if it is desired to create a string in a user-defined function, since it is now an easy matter to copy your string into this space, and then use STREND (see below). Registers affected: All, but HL finishes pointing to CHAR, DE still points to the start of the created space, and A contains its length.

STREND 21A6 Sets the first two bytes of the FPA to the next position in the sub-expression list, and then moves the temporary string pointer from CHAR into that position, thus freeing CHAR for another string, if necessary. This provides the correct way to end a user-defined string function. If the sub-expression list at STRLST is full, a STR COMPLEX ERROR is returned. This is a rare occurrance, since the only types of string manipulation that occur do not require stacking (e.g, you DON'T need to do this:

```
A$="HELLO"+(A$+(B$+E$))
```

It is allowed, however, so we must allow for 'idiots' within the programming fraternity!)

This routine also sets NTYPE to 1, indicating a string result. Registers affected: All. This routine should never be called as a Sub-routine, since it expects to find the text pointer on stack, and this will be found in HL at the end of the routine. SO, ensure that the text pointer is immediately available on stack, and then JUMP to this routine, when you use it!

9. DYNAMIC ALLOCATION OF STRING SPACE

A string may have any length from 0 to 255 characters, or 0 to 255 bytes, whereas a numeric variable occupies just 4 bytes, a fixed length. In order to make storage allocation more efficient, we therefore use a separate 'string space' area in addition to the 'variable space'. The variable space contains pointers to the various strings used, while the string space contains the actual strings themselves. No separators are needed within the string space to tell us where one string ends and the next one starts, because the pointers contain both the start address and the length of the string (this needs only 3 bytes, but we actually use 4 in order that string pointers occupy the same space as numeric variables).

When we DIMension a string array, we use up variable space in setting up the pointers, but we do NOT at that stage use up any string space, since no strings have actually been assigned.

10. INTERNAL STORAGE OF VARIABLES AND ARRAYS

a. Storage of variables.

Let us first look at the storage of variables, both string and numeric. Each string and number, as it is defined in the program, is searched for in the list from (TXTTOP) to (VARTOP). If it is not found, the list is extended by increasing VARPTR by eight bytes (and moving the arrays up eight bytes, if necessary), and then inserting the following information:

First four bytes: The first five characters of the variable name, in reverse order. In order to 'squeeze them in', we use five bits to represent the first character (in the range &00 to &1A corresponding to A - Z), and six bits for subsequent characters (the range being &01 to &0A and &12 to &2B corresponding to 0 - 9 and A - Z. Note that &00 represents no character, for variable names of less than five characters).

The first character thus occupies the top five bits of byte four, while the other characters, if used, are placed in the first three bytes.

The bottom three bits of byte four are reserved for the TYPE of the variable, bit 0 being set for string variables, bit 1 set for integer variables (both bits reset thus represent ordinary numerics!), and bit 2 is set if the 'variable' is actually a user-defined function (of the DEF FN variety!) - more about that later.

	Examples:	
A	stores as:	00 00 00 00
AB	stores as:	13 00 00 00
AB$	stores as:	13 00 00 01
XYZ$	stores as:	AB 0A 00 B9
HOUSE%	stores as:	16 69 82 3A

...and so on.

Remaining four bytes: These contain the number or string, stored in the same manner as they would be in the FPA, i.e, High byte is exponent, lower three are mantissa. In the case of strings, the high pair give the start of the string in the string space area, while the bottom byte actually gives the length of the string.

Here are some complete examples:

A=3 stores as: 00 00 00 00 00 00 40 82
XYZ$="hello" : AB 0A 00 B9 05 00 FB 8B, where we are assuming that the string "hello" is stored at &8BFB.

b. Storage of defined functions.

There is a special type of 'variable', although it may not seem as such, and that is the DEF FN function. Here, the function is defined within the variable space just as a numeric variable, except that bit 2 of the fourth byte is set (to distinguish it from a numeric or string variable), and the other four bytes contain two pointers. The first pointer gives the address within the program at which the expression on the right-hand side of the DEF statement may be found, while the second gives the address within the variable space at which the argument variable of the DEF statement may be found. Example: Suppose we have a DEF statement as the first line of a program, that the text starts at &4201, and the variable space at &4300:

```
10 DEF FN HSN(X)=(EXP(X)-EXP(-X))/2      This is the Hyperbolic sine
                    This is at the address stored in the variable space.
```

If the program is RUN, the variable space should look like this:
4300: 1F 09 00 3C 10 42 0C 43 00 00 00 B8 xx xx xx xx
This is the address of the CONTENTS of the argument.
And here is the address of the expression shown above (as an exercise, work it out and verify it!).

Note that, if the argument variable name already exists (X in this case), that variable will be used (we do not create a new one!), but its value is stacked away before the function is evaluated.

c. Storage of arrays.

An array is just an ordered set of variables, so, as we would expect, each array element is stored in the same way as a numeric or string variable, in four bytes. However, some extra overhead is needed to define the type and extent of the array, and this is done as follows:

First four bytes: As for variables.
Bytes five and six: Give an offset to the start of the next array in memory.
Byte seven: Gives the number of dimensions in the array. Let us call this number N.
Bytes eight to 2*N+7: Pairs giving the size of each dimension in turn, used to calculate the required offset to obtain a particular array element, and to ensure that an array access is within the required bounds.
The remaining bytes: Contain the elements of the array.

As this is rather complicated, let us have an example, of the array created by means of the following DIM statement:

```
10 DIM XY(22,5,4)
```

This is a three-dimensional array, containing a total of 23*6*5=690 elements (remember, we count from zero in XBASIC!). It should look like this:

```
2A 00 00 B8 CF 0A 03 05 00 06 00 17 00 xx xx ....etc.
                        Here are the three dimension pairs.
                    The number of dimensions.
                And this is the offset to the next array (or to the end
```

of the list if there are no more arrays).

We calculate the offset as: <No. of elements>*4 + 2*N + 1, where the No of elements is found by multiplying together all of the dimension pairs. Note that the dimension pairs are stored in the opposite order to that in which they were given in the DIM statement, and that the actual numbers stored are one greater than those given. Note also that, in the case where we are not using a DIM statement, the dimension pairs are each made equal to 000B (10 +1), and the number of dimensions worked out from the number of expressions given in the subscripts.

Finally, when an array is set up in the above manner, the space set aside for the elements is filled with 00s which means that each element IS, effectively, set to zero (or made to point to a null string, in the case of a string). Note that an array is set up, if it does not exist, whichever side of an assignment it appears on, unlike variables (see a. above).

11. ROUTINE FOR ACCESSING VARIABLES DIRECTLY

It is often necessary to access a numeric or string variable directly, rather than allow any type of expression and, indeed, to return a SYNTAX ERROR if an expression is attempted instead of just a variable name.

FNDVAR 277D General routine for accessing variables, depending on value of VTYPE.

a. Simple variable or array element expected. VTYPE=0 on entry.
DE points to the contents of the variable on return.

b. Entire array expected. VTYPE=1 on entry.
This is the case in which we refer to the array as a whole, without any parentheses. On return, BC points to the location containing the no. of dimensions and DE contains the offset to the next array.

c. Simple variable ONLY expected. VTYPE>1 on entry. Otherwise as a.
An example of this case is in the FOR statement, where we have a SYNTAX ERROR if the control variable is given as an array element. The routine itself does not actually return the error in this case -- it simply leaves HL pointing to the '('.

In all of these cases, HL starts pointing to the first character of the variable/array name, and finishes pointing to the first character AFTER the end of the name. If this routine is called with VTYPE non-zero, you should make it zero again sometime before returning from the routine in which you call FNDVAR.

APPENDIX D -- EXAMPLES OF EXTRA COMMANDS/FUNCTIONS

To help illustrate the method for command/function extension given in Chapter VIII, let us try a few examples. It is assumed that we are constructing a permanent set of additions to XBASIC, the Auxiliary Reserved Word Table starting at location &4200, the Auxiliary Address Table at &4240 and the actual routines starting from location &4260. Clearly, these addresses are given purely for the sake of example, and the user may care to use different areas.

Before entering the monitor to add these extras, move HTEXT up first to &4400, by doing a PTR 0,&4400, to reserve space for all of the 'extras' described below.

1. HOME

XBASIC does not have a reserved word to home the cursor to the top left-hand corner of the screen, although it does have a command to clear the screen (CLS). XBASIC can be made to perform a <HOME> by means of a PRINT CHR$(1); , where the semi-colon would be very important here, since a <CRLF> would otherwise be output as well.

However, it is only necessary to write a short routine to output the <HOME> code, and then to define a new reserved word HOME to execute it, and we may then use this command whenever required. The routine used is as follows:

```
4260: 3E 01     HOME:   LD      A,01            ; <HOME> CODE IN ACCUMULATOR
4263: C3 E7 34          JP      PR              ; OUTPUT IT
4265:                                           ; 5 BYTES TOTAL
```

The reserved word table is next constructed:

```
4201: C8 4F 4D 45       H O M E
```

Lastly, the address table is set up:

```
4240: 60 42             Point to HOME routine.
```

PR is the routine internal to XBASIC that outputs a single character given in the Accumulator to the current output device. It is one of a set of useful general-purpose routines that the user may wish to utilise for his/her own additions. The list of these routines and their addresses are given at Appendix C.

2. RAD

RAD is a degree to radian conversion function. It takes a floating- point expression in degrees and converts it to radians by multiplying it by PI/180. First, the machine-code routine:

```
4265: E1        RAD:    POP     HL              ; RETRIEVE TEXT POINTER
      23                INC     HL
      CD 41 25          CALL    EXNMCK          ; FETCH F.P EXPR IN FPA
      01 0E 7B          LD      BC,7B0EH        ; GET PI/180 INTO FPR
```

```
      11 35 FA          LD      DE,FA35H
      E5                PUSH    HL              ; SAVE TEXT POINTER AGAIN
      CD 9E 2A          CALL    MULT1           ; DO MULTIPLICATION
      C3 91 11          JP      FNEND           ; TEST FOR ')' AND RETURN
4277:                                           ; 18 BYTES TOTAL
```

Next, the reserved word table:

```
4205: D2 41 44 28       R A D (
```

Finally, the address table is extended:

```
4242: 65 42             Point to RAD routine.
```

On re-entering BASIC in the usual way, try the following:

```
PRINT SIN(RAD(30))
        .5      the sine of 30 degrees.
```

3. EXTRA TRANSCENDENTAL FUNCTIONS

By using mathematical identities, we can easily obtain a host of extra functions, with no great use of memory. The advantage of having them done in this way is that we can save time which would otherwise be wasted in scanning text, e.g, it is much better to do TAN(X) than to do SIN(X)/COS(X).

The following identities are employed:

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

Although HTN(X) could be done as HSN(X)/HCS(X), we need only do 1 call of EXP by the method adopted, rather than the four needed otherwise. Some more useful routines are included here, and are explained as follows:

```
4277: CD E8 42  ASN:    CALL    TFN             ; ASN(X)
427A: CD 51 2C  ASN1:   CALL    STKFPA          ; STACK X
      CD 5E 2C          CALL    LDFPR
      CD 9E 2A          CALL    MULT1           ; X↑2
      21 CD 2E          LD      HL,ONE
      CD 67 29          CALL    SUBN            ; 1-X↑2
      CD C7 2C          CALL    SQR             ; SQR(1-X↑2)
      C1 D1             POP     BC,DE           ; UNSTACK X
      3A 81 10          LD      A,(FPA+3)       ; SPECIAL CASE FOR ASN(1)=PI/2
      B7                OR      A
      28 0C             JR      Z,ACS2
      CD F0 2A          CALL    DIV1            ; X/SQR(1-X↑2)
      C3 54 2D          JP      ATN             ; ATN(X/SQR(1-X↑2))

429A: CD E8 42  ACS:    CALL    TFN
      CD 7A 42  ACS1:   CALL    ASN1
      21 0B 2F  ACS2:   LD      HL,HALFPI
```

```
        C3 67 29          JP      SUBN            ; PI/2 -ASN(X)

42A6:   CD E8 42  HSN:    CALL    TFN
        CD DC 42  HSN1:   CALL    HSN2
        CD 6D 29          CALL    SUB1            ; EXP(X)-EXP(-X)
42AF:   21 81 10  HALVE:  LD      HL,FPA+3        ; DIVIDE-BY-2 BY JUST
        7E                LD      A,(HL)          ; DECREMENTING EXPONENT
        B7                OR      A
        C8                RET     Z               ; NOT IF FPA=0
        35                DEC     (HL)
        C9                RET

42B7:   CD E8 42  HCS:    CALL    TFN
        CD DC 42  HCS1:   CALL    HSN2
        CD 70 29          CALL    ADD1            ; EXP(X)+EXP(-X)
        18 ED             JR      HALVE

42C2:   CD E8 42  HTN:    CALL    TFN
        CD FC 42  HTN1:   CALL    DOUBLE          ; X*2
        CD 14 2D          CALL    EXP             ; EXP(X*2)
        21 CD 2E          LD      HL,ONE
        E5                PUSH    HL
        CD 62 29          CALL    ADDN            ; 1+EXP(X*2)
        CD F4 42          CALL    RECIP           ; 1/(1+EXP(X*2))
        CD FC 42          CALL    DOUBLE          ; 2/(1+EXP(X*2))
        E1                POP     HL
        C3 67 29          JP      SUBN            ; 1-2/(1+EXP(X*2))

42DC:   CD 14 2D  HSN2:   CALL    EXP             ; GET EXP(X) AND EXP(-X)
        CD 51 2C          CALL    STKFPA
        CD F4 42          CALL    RECIP           ; DO EXP(-X) AS 1/EXP(X)
        C1 D1             POP     BC,DE
        C9                RET

42E8:   E1        TFN:    POP     HL              ; ROUTINE TO EVALUATE THE
        E3                EX      (SP),HL         ; EXPRESSION BETWEEN THE
        23                INC     HL              ; BRACKETS, FOR USER-DEFINED
        CD 41 25          CALL    EXNMCK          ; FUNCTIONS.
        11 91 11          LD      DE,FNEND        ; WILL EVENTUALLY RETURN TO
        E3                EX      HL,(SP)         ; FNEND
        D5                PUSH    DE
        E9                JP      (HL)            ; JUMP TO RETURN ADDRESS

42F4:   01 00 81  RECIP:  LD      BC,8100         ; CALCULATE RECIPROCAL
        51                LD      D,C
        59                LD      E,C             ; FPR=1
        C3 F0 2A          JP      DIV1

42FC:   21 81 10  DOUBLE: LD      HL,FPA+3        ; DOUBLE FPA BY INCREMENTING
        7E                LD      A,(HL)          ; EXPONENT
        B7                OR      A
        C8                RZ                      ; NOT IF FPA=0!
        34                INC     (HL)
        C0                RNZ
        1E 06             LD      E,06
        C3 CF 15          JP      ERROR           ; OVERFLOW IF EXPONENT=FF
4309:                                             ; SIZE 146 BYTES
```

Next, the function names:

```
4209: C1 53 4E 28 C1 43 53 28   ASN( ACS(
4211: C8 53 4E 28 C8 43 53 28   HSN( HCS(
4219: C8 54 4E 28               HTN(
```

And, finally, the addresses:

```
4244: 77 42 9A 42 A6 42 B7 42 C2 42
424E:
```

4. LOC

It is often useful to be able to obtain the location in memory where the contents of a variable may be found. A variable name or array element MUST be specified as the argument, and the memory location of the start of the contents of the variable is returned as a result. In the case of a string variable, the address of the start of the actual string is returned.

```
4309: E1          LOC:    POP     HL
      23                  INC     HL
      CD 7D 27            CALL    FNDVAR
      E5                  PUSH    HL
      3A 66 10            LD      A,(NTYPE)
      B7                  OR      A
      EB                  EX      DE,HL
      28 0A               JR      Z,LOC1
      23                  INC     HL      ; IF STRING, GET ACTUAL STRING
      23                  INC     HL      ; ADDRESS, NOT JUST POINTER
      7E                  LD      A,(HL)
      23                  INC     HL
      66                  LD      H,(HL)
      6F                  LD      L,A
      AF                  XOR     A
      32 66 10            LD      (NTYPE),A
      CD E1 13  LOC1:     CALL    FORMPOS
      C3 91 11            JP      FNEND
4326:
```

The function name:

```
421D: CC 4F 43 28       LOC(
4221:
```

And its address:

```
424E: 09 43
4250:
```

5. UCS$

This function returns a string corresponding to the argument (which is also a string), with all lower-case letters converted to upper-case:

```
4326: E1          UCS$:   POP     HL
      23                  INC     HL
      CD 5A 25            CALL    EXPR
      CD 92 11            CALL    RPARN           ; GET CLOSING BRACKET
      E5                  PUSH    HL              ; AND PUSH TEXT POINTER
      CD 4D 23            CALL    ASC1
      2B                  DEC     HL
      2B                  DEC     HL
      2B                  DEC     HL
      7E                  LD      A,(HL)
      D5                  PUSH    DE
      CD 73 21            CALL    ASNSTR          ; MAKE NEW STRING
      E1                  POP     HL
      47                  LD      B,A
      7E          UC1:    LD      A,(HL)
      CD D0 11            CALL    LWRTST          ; CONVERT LOWER-CASE LETTER
      12                  LD      (DE),A          ; AND PLACE IN NEW STRING
      13                  INC     DE
      23                  INC     HL
      10 F7               DJNZ    UC1
      C3 A6 21            JP      STREND
```

4348:
The function name:

```
4221: D5 43 53 24 28    UCS$(
4226: 80
```

And its address:

```
4250: 26 43
```

The number of reserved words should now be added at location &4200, this being 9, if all of these extra commands and functions have been entered. Also, do not forget to put the code &80 at the end of the Auxiliary Reserved word Table.

Finally, the pointers to the Auxiliary Tables should be changed to point to the new tables, by modifying the DEFAULT Auxiliary pointers:

```
3A88: 00 42
3A92: 40 42
```

Note that this cannot be done by means of the PTR command in BASIC, owing to the fact that the next 'Cold Start' would restore the old values.

We hope that this set of examples will give the user many more ideas!

APPENDIX E - TRANSLATOR FOR NASCOM ROM AND TAPE BASIC PROGRAMS

XBASIC will not run programs written for ROM BASIC or tape BASIC as they stand, for two reasons; the reserved word 'tokens' are different, and the tape loading format is different.

A small program is therefore provided, on the reverse side of the XBASIC cassette, called NLOAD.XBS. This is a BASIC program, which loads the actual translator routine (in a file called NLOAD.OBJ, which follows NLOAD.XBS on the tape). As well as loading the translator routine, NLOAD.XBS also sets up the user command and address pointers for the extra command ("NLOAD") that is added to the system.

You should therefore load XBASIC and execute it at 1000 in the usual way. Then type LOAD "NLOAD.XBS", press <ENTER> and press 'PLAY' on the cassette recorder to load the NLOAD program. When this has loaded type RUN, and turn the cassette to 'PLAY' again to load the file NLOAD.OBJ. When this has loaded (Ok will be displayed) you can load a ROM BASIC program by simply typing NLOAD and then pressing <ENTER>. Now press 'PLAY' on the cassette recorder. The program name will be displayed, in a similar format to the normal ROM BASIC display, and the program will then be loaded in the same way as for XBASIC programs (a Bad Data Error occurs, if a bad block is read). A short pause will then occur, while the program is being 'translated'. On the completion of the translation, the 'Ok' prompt will reappear, and the program may now be RUN, LISTed or modified as a normal XBASIC program.

Note the following 'incompatibilities' which are dealt with by the translator:

a. SCREEN X,Y commands are converted to PRINT@ X,Y. NASCOM BASIC treats the top line of the screen as line 16, whereas Xtal BASIC treats it as line 0, but, happily, this will cause no problems, since XBASIC 'wraps around' in both directions, and so no further modification is needed.

b. The LINES command, if found, will be converted to a ':', since that command is not provided (or needed) in XBASIC.

c. SET(X,Y) and RESET(X,Y) are translated to SET X,Y and RESET X,Y respectively since the brackets would cause a Syntax Error in XBASIC. The translator allows for expressions containing brackets within these two commands, so that SET(X(I),Y(I)) would become SET X(I),Y(I) , for example.

Another incompatibility here is that, in ROM BASIC, SET and RESET use coordinates in the ranges 1-96 and 1-48, whereas XBASIC uses 0-95 and 0-47. However, XBASIC again allows these to 'wrap around' so that the only effect will usually be to shift the graphic display one position to the right.

d. USR(X) is translated to CALL(X). However, this may still not work, since machine-code routines used formerly with USR may call internal BASIC sub-routines, which will no longer be available in those locations under XBASIC. These routines should therefore be modified to run under XBASIC.

Of course, the user will now probably want to take advantage of some of the extra features of XBASIC, to improve the efficiency of the programs, but should in all but a few cases require no modifications to run programs that have previously been created under ROM BASIC.