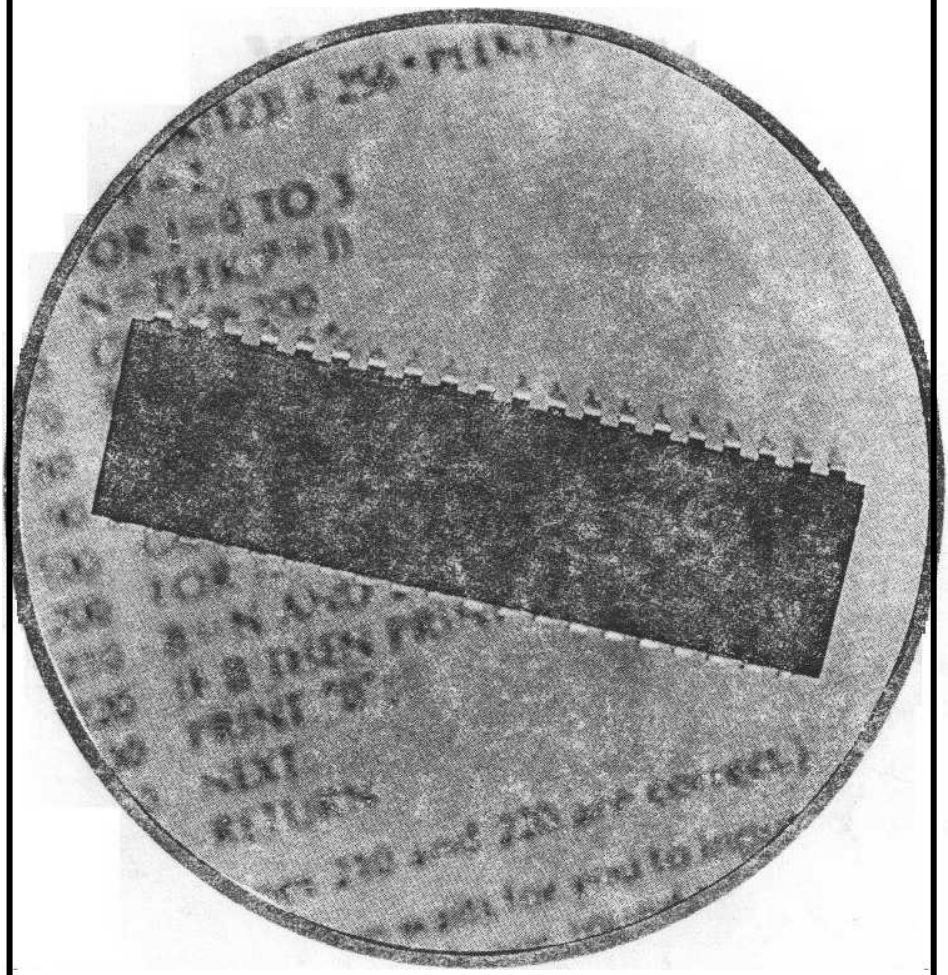
XTAL BASIC



A BASIC INTERPRETER FOR Z-80 BASED SYSTEMS INCLUDING NASCOM 1 & NASCOM 2

operating manual

XTAL BASIC

operating manual

A BASIC INTERPRETER FOR Z-80 BASED SYSTEMS
INCLUDING NASCOM 1 & NASCOM 2

Software by
ANDREW CORNISH, BSc
And
CRYSTAL ELECTRONICS

Published by CRYSTAL ELECTRONICS, 40 Magdalene Road, Torquay, Devon

ISBN 0 9506828 1 0

Edited, typeset and printed by NWL EDITORIAL SERVICES Langpool, Somerset

Introduction

XTAL (pronounced Crystal) BASIC is an interpreter written in Z.80 machine code which, while originally developed for the NASCOM microcomputer, may be easily modified to run on most Z.80 based systems. It enables even the newcomer to computing to run BASIC (Beginner's All-purpose Symbolic Instruction Code) programs with all its facilities for complex mathematical operations and String (word) handling. A comprehensive editing system and single step facility make the debugging of programs a simple matter.

Nominally an 8K interpreter, XTAL BASIC 2.2 actually occupies 7424 bytes which means that it can be used on a NASCOM with as little as 8K of extra read/write memory; however, to obtain reasonable use it is recommended that it be used with a 16K or greater system to give 9K or more of program space.

For those users 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 the auxiliary reserved word table you will be able to expand this interpreter to give the type of BASIC most suited to your own needs - in fact you may never need to buy another BASIC interpreter for the rest of your programming days! We believe that, for the time being at least, this feature is unique to XTAL BASIC 2.2 and makes it one of the most potentially powerful BASICS ever available.

In order to simplify the use and understanding of this manual all references to hardware have been restricted to the NASCOM microcomputer but XTAL BASIC is pretty well hardware independent (apart from the PRINT @ and EDIT commands which make direct access to the VDU). Sufficient information is given to allow the more experienced user of other systems based on the Z.80 CPU to make their own modifications.

XTAL BASIC 2.2 may be run with all available commercial NASCOM monitors running on either NASCOM 1 or NASCOM 2.

A reasonable knowledge of BASIC is assumed, this is not a BASIC teaching manual, but those readers unfamiliar with the more general aspects of the language are advised to refer to the bibliography on page 31 or to any of the computing hobby magazines for introductory material.

Loading XTAL BASIC 2.2

XTAL BASIC 2.2 is supplied on cassette tape in two formats allowing it to be run in conjunction with any of the commercially available monitors (B-Bug, T2, T4 and Nas-Sys) whether used on a NASCOM 1 or a NASCOM 2. Side 1 contains a fast loader program for use with T2 systems, followed by the interpreter in NASCOM 'R' format at 250 baud. Side 2 holds the interpreter in the popular CUTS (Compute Users Tape Standard - also known as the KANSAS CITY) format at 300 baud and is directly compatible with a standard Nascom 2 as well as with many other systems.

T2 users *(side 1)*
First enter the fast loader program found at the start of side 1, using the 'L' command. This will load from $0E00_{16}$ to $0EEF_{16}$ When you are satisfied that this program has been correctly loaded you should enter the command

E 0E00
(nl) before allowing the tape to continue. XTAL BASIC 2.2 will then load at about four times the 'L' speed The fast loader may be 'scrubbed' after use since it is not required during running of XTAL BASIC 2.2.

T4 and B-Bug users (side 1)
Skip past the loader program on side 1 and load the remainder of the tape under 'R' command.

Nas-Sys + Nascom 2 users (side 2)
Load side 2 of the tape under ' R' command.

Nas-Sys + Nascom 1 users (side 1)
If you have added CUTS (Kansas City) to your system then load as for Nas-Sys + Nascom 2. If you have Nas-Sys in a Nascom 1 skip the fast tape loader on side 1, load under 'R' (as for T4) and carry out the modifications shown in the section Modifying XTAL BASIC 2.2.

Non Nascom users
Load side 2 in CUTS format then study the sections Modifying XTAL BASIC 2.2 and Running XTAL BASIC 2.2. Assuming you have a reasonable grasp of your system you should not find it too difficult to make the required modifications.

Notes
XTAL BASIC 2.2 occupies memory from 1000_{16} to $2D00_{16}$ so that your back-up copy can be made using the following command:

W 1000 2D00 (nl)

or, for T2 users:

E E80 1000 2D00 (nl)

XTAL BASIC 2.2 takes about 6 minutes to load, at 250 baud.

As with all such material you are strongly advised to make a back-up copy on your own machine as soon as possible. The source tape may then be safely stored away, preferably in a metal box or cabinet well away from electromagnetic interference such as mains power lines and domestic electrical equipment.

Modifying XTAL BASIC 2.2

The following table (Table 1) indicates all the differences between the side 1 and side 2 versions of XTAL BASIC 2.2. Side 1 is in Nascom 1 format and works directly with B-Bug, T2 and T4 while side 2 works with a Nascom 2 under Nas-Sys. Most of the differences lie in the codes used for the control characters for New Line, Clear

Screen and Backspace [(nl), (cs), (bs)] which use the normal ASCII codes ($0D_{16}$,$0C_{16}$ and 08_{16} respectively) under Nas-Sys but not under the earlier monitors.

The other major area of modification is concerned with input and output. This is now served by a single table of jump vectors located between $2BDD_{16}$ and $2BFF_{16}$. Users of non-Nascom systems will have to alter these lo suit their own requirements but sufficient information is available to make this a fairly simple operation.

The only other change is the cursor address which is at $0C29_{16}$ in the Nas-Sys monitor and at $0C18_{16}$ in the earlier monitors.

For non-Nascom users: The keyboard scan routine at $2BF7_{16}$ returns with the carry flag set if a key has been pressed and with the appropriate code in register A; otherwise the carry flag is cleared. In your character output routine ($2BF4_{16}$), you should ensure that the carry flag is cleared on return.

TABLE 1 Address differences for Nasbug and Nas-Sys Monitors

Address	*Nas-Sys*	*Nasbug*
103A	0C	1E
1049	8D	9F
1358	0D	1F
135B	8D	9F
151E	08	1D
152B	0D	1F
1561	0D	1F
16FE	0C	1E
1998	29	18
19A8	29	18
1A0A	0D	1F
1A52	8D	9F
1B0A	8D	9F
2ACE	29	18
2AE2	0C	1E
2C27	29	18
2C2C	0D	1F
2C2D	0D	1F
2C2E	8D	9F
2C35	29	18
2C3B	29	18
2C44	29	18
2C87	0D	1F
2CAB	08	1D
2CB3	11	3C
2CCC	12	3E
2CE5	13	3F

The following vectors will also have to be taken into account if modifications are being made:

Address:	*Nas-Sys*	*Nasbug*	*Remarks*
2BDD	E5 CD 38 07 E1 C9	11 3B 01 ED 53 48 0C C9	Set VDU output
2BE5	C3 B2 03	31 33 0C C3 59 03	Exit to monitor
2BEB	FF C9	FF C9	Delay routine
2BEE	C3 5B 00	C3 5E 00 (T2 & B-Bug: $005D_{16}$)	Output A to UART
2BF1	C3 00 03	C3 32 02	Print HL in hex
28F4	F7 C9	C3 4A 0C	Output character in A
2BF7	C3 54 07	C3 4D 0C	Keyboard scan
2BFA	C3 08 00	C3 3E 00	Input character to A
2BFD	C3 51 00	C3 51 00	Switch tape motor

Note: T2 and B-Bug users must change location 2BEF to $5D_{16}$. otherwise CSAVE will not work!

Running XTAL BASIC 2.2

Entering and leaving BASIC

BASIC may be entered via any one of three entry points:

E 1000 (nl)

E 1002 (nl)

E 1004 (nl)

In the first and third cases the screen will clear and the following display will appear:

XTAL BASIC 2.2

SIZE: XXXXX

O.K.

]

After E 1002 only the last three lines will appear.

XXXXX is the amount of free space (in bytes) available for program and variables. The] prompt is used to distinguish between BASIC and the monitor operating system.

Entry at 1004_{16} should be used after first loading XTAL BASIC 2.2. This clears any BASIC program and variables, and also initialises the auxiliary reserved word tables ($0E80_{16}$ - $0FFF_{16}$) - more about this in the section on user-defined reserved words.

Entry at 1000_{16} clears any BASIC program and variables, but does not affect the auxiliary reserved word tables.

Entry at 1002_{16} does not destroy the existing program or variables and would, typically, be used after making an excursion from BASIC to the monitor operating system.

lt is possible to leave BASIC and return to monitor control at any lime by entering the command NAS either directly or as part of a BASIC statement. The > prompt tells you that you are back in the realms of machine code (or the legend NAS SYS n if using Nas-Sys).

Program entry

As with most BASICs, commands may be entered in direct or deferred mode. In direct mode commands are entered without line numbers and are executed immediately upon typing (nl), giving calculator-like facilities.

Example:

PRINT 12 * 12 (nl)
144

System commands such as RUN, LIST, NEW are only of use in this mode.

In *deferred* mode commands are entered with line numbers to form programs. Line numbers may range between 1 and 65529 and may be followed by one or more commands. Each line so entered is automatically placed in its correct position with line 1 being the first line interpreted. Line numbers may start at any point in the range 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 (nl). A line may be replaced by entering a new line bearing the number of the unwanted line.

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

Example:

10 PRINT 2 * 2 : PRINT 4 * 4 (nl)
RUN (nl)
4
16

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

A line may be up lo 96 characters in length during entry, and up to 144 characters under EDIT mode so that any command abbreviations used may be expanded to their full format.

By popular demand the ? character may now be substituted for the word PRINT during entry as may many of the graphics key entries under Nas-Sys.

System control commands

NEW causes all program lines and variables to be deleted.

LIST outputs a complete listing of the program from start to finish.

LIST X lists from the specified line number X.

LIST, N lists the first N lines, pauses until any key except (cs) is pressed and then lists

the following N lines.

LIST X, N lists N lines from line X, pauses until any key except (cs) is pressed and then lists the following N lines.

Pressing (cs) during any listing will cause the listing to be abandoned, while (bs) interrupts and holds both listings and program execution.

RUN begins execution of the program starting at the lowest line number.

RUN X causes the program to execute from line X.

GOTO X causes the program to execute from line X but preserves all variables.

NAS returns control to the monitor under software control (i.e. without a RESET).

CLEAR clears all variables, arrays and strings.

CLEAR N also sets the string space size to N bytes.

CLEAR N, M also sets the location of the top end of RAM lo location M (decimal) allowing the user to preserve space for machine code routines. No check is made as to the validity of this entry although if the location selected is below the top of the variable space the message MEM FULL ERROR will appear.

CLEAR% clears the user defined reserved word table (see chapter on User defined reserved words) and the associated address tables ($0E80_{16}$ to $0F7F_{16}$ and $0F80_{16}$ to $0FFF_{16}$). The word list table is cleared so that every location contains 80_{16} and the address table so that each pair of address bytes points to location $130B_{16}$ which is the address of the routine to print CMD ERROR. Note there is already a 'safe' free RAM area from $0E00_{16}$ to $0E7F_{16}$.

EDIT causes the first line of your program to be displayed together with a non-destructive blinking cursor.

EDIT X begins editing from line X. If the line does not exist then editing begins from the next highest line number. When a line has been edited to your satisfaction, enter (nl), the corrected version will be entered into your program and the next line will be displayed. The position of the cursor at (nl) is not critical, (cs) will cause the edit mode to be abandoned without entering the last line currently on display.

Editing commands

(space) moves the cursor to the right by one position. (bs) moves the cursor to the left by one position.

(>) moves the whole line at and to the right of the cursor one position to the right. leaving a space at the cursor position so allowing the insertion of text or spaces.

(<) deletes the character at the cursor position and moves the rest of the line one position to the left to close the gap.

(?) modifies the character at the cursor position by adding 10_{16} to the ASCII value of that character without advancing the cursor.

Any other key typed enters the character typed in place of the character previously at the cursor position.

Nas-Sys users may use the arrow keys of the Nascom 2 keyboard in place of the (>), (<) and (?) keys of the Nascom 1 as follows:

(←) or (@ Q) replaces (<)
(→) or (@ R) replaces (>)
(↑) or (@ S) replaces (?)

CSAVE and **CLOAD** commands are built into the interpreter so that no distinction need be made between the various monitors.

CSAVE (NAME) saves the current BASIC program to cassette with a name of up to six characters. The name is not strictly necessary but is very useful as will become apparent in the section on CLOAD. The tape should be set to record and in motion before the command is entered.

CLOAD will load the first BASIC program found on tape regardless of its name.

CLOAD (NAME) will search for and load a named BASIC program from cassette. The name should consist of numbers and/or upper case letters only.

Examples:
]CSAVE PROG12
]CLOAD STARS

Programs will not necessarily load at the address in memory from which they were saved but at the address specified in location $0C8C_{16}$ (TEXT). This is normally $2D00_{16}$ on the tape as supplied but it can be changed by modifying at $0C8C_{16}$ (temporarily) or at 1283_{16} (permanently - if the interpreter is 'saved' to a new tape). The advantages are that all XTAL BASIC 2.1 programs may be loaded under XTAL BASIC 2.2, programs may be appended to one another and extensions may be added to the interpreter while keeping the entire 'saved' package in one continuous block.

Note: CLOAD sets up an end-of-text pointer after the program has been loaded so that typing CLOAD by mistake no longer causes you to lose a program.

CLOAD@ and CSAVE@ will load and save a named array allowing files to be stored without the need for including the complete program. Note that an array can only be CLOAD@'ed to an array that has already been dimensioned in your program. These commands may be used in the direct mode subject to dimensioning having previously taken place.

CLOADing errors: If an error occurs during a CLOAD or CLOAD® then loading will terminate with the message TAPE ERROR being displayed and execution will return to BASIC. This has the advantage that, for example, CLOAD® errors can now be handled from within a program although, if an error occurs under CLOAD, it will not be possible to load the whole program into memory. However, with the verification routine now available there should rarely, if ever, be such a problem.

However, if you are having a lot of trouble with CLOADing try the following:

(i) Load the program using 'R' or 'E' $0E00_{16}$, (with the fast tape loader in place) - this ignores the title.
(ii) Copy it through memory lo its correct location if necessary and make a note of the top memory location it uses.

(iii) Load the XTAL BASIC 2.2 interpreter if necessary and enter.

(iv) Enter this last memory location at $0CB7_{16}$ (TXTUNF).
(v) Make the first location of the program non-zero, type E 1002 and then type CALL 5072. This adjusts all the end-of-line pointers in the program.
(vi) If the error was not serious it should be possible to correct using LIST and EDIT,
(vii) CSAVE the program again then try to RUN it (Best of British luck!).

With reasonable care there should be few such errors, but this routine can also be useful when loading programs recorded on other systems.

CLOAD? is a very useful command which may be entered in the same format as **CLOAD** or as **CLOAD@**. It is a tape verification routine which appears to read in from tape but will never actually interfere with the program area of the interpreter, The input will be displayed on the VDU as if it were loading and it enables you to check that you have safely and correctly saved material before you move on. The following example shows the use of CLOAD?@ within a program (see line 100). Note the use of ON ERR GOTO (more on this later).

Example:

```
10  PRINT CHR$(30);TAB(10) “SQUARE ROOTS TABLE”
20  REM use CHR$(12) to clear screen with Nas-Sys
30  DIM A(99)
40  FOR I=0 to 99: A(I)=SQR(I): PRINT A(I): NEXT
50  INPUT “START TAPE AND HIT NL”; X$
60  CSAVE@A
70  REM Data now saved
80  PRINT “Now rewind and press PLAY to verify”
90  ON ERR GOTO 120
100 CLOAD?@A
110 “ALL SAFE – A PERFECT CLOAD!!”: END
120 REM Error Routine
130 IF ERR <> 21 THE PRINT CMD$(ERR); “ERROR”: STOP
140 PRINT “DATA ERROR ON TAPE, REWIND, RESET RECORDING LEVELS”
150 GOTO 50
160 REM Keep on trying!
```

Interrupting, stopping and single-stepping

The keyboard is scanned at the end of every executed statement during a program run and at the end of each line during a LIST. This means that either of these operations may be interrupted at any time as follows:

(bs) will stop the program or listing temporarily, pressing any other key will cause the program or listing to resume.

(cs) will cause the interpreter to abandon execution or listing and return to the BASIC input mode. The message ‘BREAK’ in the case of a listing or ‘BREAK IN LINE X’ in the case of a program run will be displayed. Execution can be resumed by entering CONT.

A single step facility is available allowing programs to be executed one line at a time. This extremely useful debugging aid is used as follows:
Start the program with a dummy line (5) using INCH which awaits a keyboard entry.
To run without single stepping enter any letter.
To single step enter (bs) and continue to key in (bs) as each line is executed.

Example.

```
10  X=INCH
20  A=0
30  PRINT A
40  A=A+1
50 GOTO 30
```

Numbers, strings and variables

There are two types of quantity allowed in XTAL BASIC 2.2: numbers and strings.

Numbers: These can be whole numbers (integers) or floating point numbers (reals). A number is stored initially 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 of the mantissa are available for internal calculation they are usually rounded off to a six digit figure for output. 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 .314159 3.14159 E+08 -3.14159 E-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. When accuracy is at a premium you should always enter numbers to seven significant figures since the interpreter can make use of the seventh figure even though it will only display six of them.

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.

Variables: These can be named using any combination of letters and/or numbers but they must start with a letter and (in common with most BASICs) XTAL BASIC 2.2 distinguishes only the first two characters. Variables may be either numeric variables or string variables holding numbers or strings respectively. String variables must be suffixed with $. There is no practical limit to the length of a variable name.

Examples:

A AA A2 A9 X$ X4$ ABCD$ AB123$ A1 A100

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

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 nearly always followed by a set of one or more subscripts, each subscript representing a dimension of that variable.

Examples:
A(0) TABLE(5,6) NAME$(1,0,2)

are all valid arrays where A is a array of one dimension where the subscript (a sort of address) is referring to the first element. TABLE is a two dimensional array and NAMES is a three dimensional array holding strings each of which may be up to 255 characters in length.

In XTAL BASIC 2.2 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 (q.v.) 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

The array will have 11*11*11 =1331 elements requiring over 5300 bytes to store it!

Expressions

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

The arithmetic operations allowed in XTAL BASIC 2.2 are as follows: +(add) -(subtract) *(multiply) /(divide) ** (raise to power) ** is used in the latter case in preference to ↑ since this character may not be available to some users.

Relational operators allowed are as follows:

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

Logical operators allowed are: AND OR NOT

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

Relational expressions are normally used within IF statements but can also be used within arithmetic expressions since a relational expression returns a value of -1 if true and of 0 if false.

BASIC also allows string expressions but these are restricted to concatenation (+) and comparison.

Example:

```
A$=="ABC" : BS="DEF" :  CS=A$+B$ :  PRINT C$
```
gives the output ABCDEF

and a line such as:
```
10  IF A$="HELLO" THEN PRINT "GOODBYE"
```
is also valid.

As is usual in both BASIC and mathematics generally, functions can appear within expressions in place of variables, assuming that they are of the correct type. You cannot, for example, use a function returning a string expression as part of an arithmetic expression. A full list of available functions appears in the chapter on Functions (page 16).

Operator precedence follows the usual mathematical order:

Highest precedence:	( )
	**
	* /
	+ -
	< <= = > >= <>
	NOT
Lowest precedence:	AND OR

Commands and statements

There now follows a full list of commands and statements available in XTAL BASIC 2.2 in its unmodified (by the user) version.

CALL E calls a machine code subroutine starting at the decimal address given by the expression E. This expression must be an integer in the range -32767 to +32767.

Example:
CALL 3840 will cause the program to jump to a routine at address $0F00_{16}$ ($=3840_{10}$) while CALL -3840 will jump to a routine at $F100_{16}$.

The user need not worry about storing registers, as long as the subroutine is terminated with $C9_{16}$ the return to BASIC will be automatic.

CONT causes an interrupted program (page 8) to resume without clearing the variables. It may be used after an integral STOP command in the program has caused a stop. During the stopped period the user may look at or alter variables without causing any harm, but any attempt to modify the program itself will cause the error message CONT ERROR to appear on resumption. CONT may also be used after an interrupt using (cs). This is a particularly useful aid to debugging in, for example, the tracing of an infinite loop.

LET takes the form:

LET V = E where V is a variable and E is an expression; the value of the expression is assigned to the variable. The word LET is optional but advisable.

Example:
```
LET AA= 1+2*3/4
```
assigns the value 4.5 to variable AA.
```
NAME$ ="JOHN"
```

assigns the string JOHN to variable NAME$ and shows use of the formal without the word LET.

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

STOP like END terminates a program and displays the message BREAK IN N where N is the line number in which termination occurs. Several STOP commands may be used in a program and execution can be restarted from this breakpoint by use of CONT provided that no program alterations have been made during the break.

REM causes the remainder of the line to be ignored by the interpreter. Its main use is for entering programming notes during the construction of a program.

Example:
10 REM THIS LINE WILL BE IGNORED

PRINT takes the general form PRINT El, E2 ..... E_N where El, E2 through to EN are numeric or string expressions.

The separators between expressions can be as follows:

, (comma) moves the (imaginary) print-head lo the start of the next 16 column zone of which there are three per line (48 characters).

; (semi-colon) moves the (imaginary) print-head to the right by one character position.

If no separator is used at the end of a PRINT command the print-head moves to the start of the next line as though it had received a 'carriage return-line feed' instruction.

Examples:

```
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 space which is reserved for a sign but which only holds one if the number is negative.

PRINT *may be abbreviated to ?* although it will still LIST and EDIT as PRINT. It will stay as ? in REM, CLOAD, CSAVE and DATA statements as well as between double quotes.

PRINT @ allows printing of expressions at specified points on the screen using coordinates. For this command the screen is divided (internally and automatically) into 48 (0-47) columns and 16 (0-15) rows. The X co-ordinate must be an integer from 0 to 47 but the Y co-ordinate may be an integer between 0 and 255 with the remainder after division by 16 being automatically selected as the Y row. Co-ordinates and the expression to be printed should all be separated by commas as shown in the example.

A PRINT @ command should be followed by , or ; to avoid issuing a 'carriage return-line feed' which will scroll the screen up (under Nasbug).

Example:
10 PRINT@ 0,0,"x"; @ 47,0,"x": @ 0,15,"x": @ 47,15,"x";
will print an 'x' in each corner of the screen.

PRINT @ can be abbreviated to ?@ or even to @ alone, as shown, provided a PRINT or ? appears in the first instance.

INPUT causes a request for keyboard input to be displayed and takes the form INPUT "Prompt"; V1 .... V_N. The prompt is optional but must be a string in quotes followed by a ; if used. If no prompt is used then BASIC prompts with a ?

Data entered as a result of an INPUT command may be in the form of numbers, strings or strings in quotes. In the case of more than one variable being filled the entries must be separated by commas.

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 numerical data is required the message ?RE-ENTER ALL appears and the input list must be re-entered from the start.

Example:

```
10  INPUT "Name, Rank and Number"; NAMES, RANKS, N
20  PRINT NAMES, RANKS; N
RUN
Name, Rank and Number? Cornish, Lt, 506659
Cornish         Lt  506659
```

INPUT cannot be used in the direct mode.

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

READ V1 .... V_N reads in data from a list stored in the program in DATA statements. XTAL BASIC 2.2 maintains a pointer which remembers the last item of data read so that subsequent READ instructions 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 the message DATA ERROR IN L (L = line number) appears.

DATA D1 .... D_N specifies the items of data to be read. These items may be numbers, strings in quotes, or strings without quotes provided they contain no spaces or commas. The user may have as many DATA statements as he likes 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.

RESTORE sets the DATA pointer back to the first item of data in the program.

RESTORE L sets the DATA pointer to the first item of data following line number L.

GOTO L transfers program execution to line L If line L does not exist the program stops with the message BRANCH ERROR IN L.

GOSUB L transfers program execution to line L, execution continues from this point until a RETURN instruction is encountered whereupon execution is returned to the line immediately following the original GOSUB command.

RETURN terminates a subroutine accessed by a GOSUB. If RETURN is encountered without having been preceded by a GOSUB then the message RETURN ERROR IN L is displayed and execution stops.

POP has the same effect as a RETURN but without the branch. It pops one address off the stack of return addresses so that the next RETURN will branch one statement beyond the second most recently executed GOSUB.

If has several forms:

IF E THEN L | these two are equivalent
IF E GOTO L |

IF E THEN S1: S2: .... S_N

E is normally a logical expression. In the first two cases execution is transferred to line L if E is true; if E is false then execution transfers to the next line. In the third case execution stays on the same line (statements S2 to S_N) only if E is true, otherwise it transfers lo the next line.

Example:

```
10  IF X=Y THEN PRINT "HELLO": GOTO 30
20  PRINT "GOODBYE"
30  END
```

ON E GOTO LI, L2, ... L_N
ON E GOSUB LI, L2, ... L_N

In both cases expression E is evaluated and must return a number between 0 and 255. Execution then transfers to line L1 if E=1, L2 if E=2 and so on. If E = 0 or E = a number greater than the number of lines in use execution remains on the same line. The transfer takes the form of a GOTO or a GOSUB as specified.

FOR V=E1 TO E2 ..... NEXT
FOR V=E1 TO E2 STEP E3 ..... NEXT

In both cases V is a numeric variable which is initialised to the value of expression E1 while E2 represents a numeric limit,

E3 represents a numeric increment and if it is not present it is assumed to be of value 1. El, E2 and E3 may be positive or negative.

Execution of this command sets V to E1 until a NEXT is encountered

NEXT may be in either of three forms:

NEXT
NEXT V
NEXT V1, V2 . . .. V_N

This command adds the value of E3 to V (adds 1 if STEP was omitted) and compares V with E2. If V>E2 (or if V<E2 in the case of a negative STEP) execution is transferred back to the statement immediately following the FOR statement, otherwise execution continues from the point following the NEXT statement.

In the first form (NEXT) the most recently executed FOR statement is dealt with so that no variable need be specified. In the second form (NEXT V) if V does not correspond to the V in the last FOR statement that FOR loop will be abandoned and the next most recently executed one checked, and so on until the correct FOR statement is found. If V does not correspond with any of the current FOR loops, or if there is no active FOR loop, the program stops and displays the message NEXT ERROR IN L

The third form (NEXT V1, V2 .... V_N) is a useful and short way of dealing with nested loops which would otherwise have to take the form NEXT V1: NEXT V2: .... : NEXT V_N

Example:

```
5   DIM A(7.7)
10  FOR X = 0 TO 7
20  FOR Y = 0 TO 7
30  A(X,Y)=0
40  NEXT Y, X
```

When RUN, this routine will set all elements of an 8 by 8 array (A) to zero (line 30).

DIM is used to reserve Storage for numeric or string arrays. It takes the form DIM ARRAYNAME (V1, V2, ... V_N) where each array is shown as ARRAYNAME (E1, E2 .... E_N). The ARRAYNAME may be any legal variable name while E1 to E_N are positive integers 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 cannot be dimensioned more than once in each program without the program stopping and displaying the message DIMENSION ERROR IN L.

POKE E1, E2 will place the computed value of E2 in memory location E1 .El must lie in the range -32767 to + 32767 and E2 must have a value between 0 and 255. E1 and E2 are of course, expressed in decimal. See also PEEK, under Functions.

SPEED E sets a delay in the character output to the VDU. E must lie between 0 and 255 and 0 gives the slowest (very slow) speed while 255 is normal (fastest) speed.

OUT E1, E2 sends the value of E2 to output port E1. This is designed for Z80 systems using I/O port addressing as opposed to memory mapped port addressing. Both E1 and E2 must have values between 0 and 255. See also INP under Functions.

WAIT has two forms:

WAIT E1, E2
WAIT E1, E2, E3

where all Es must be integers in the range 0 to 255. This instruction monitors the port specified by E1, EXCLUSIVE ORs it with E3 (if used) and ANDs the result with E2 treating E2 and E3 as binary numbers.

Example:

WAIT 2, 64 suspends execution until bit 6 of port 2 is set.
WAIT 1, 255, 15 waits until any of the 4 most significant bits are set or until any of the least significant bits are reset on port 1.

ON ERR GOTO L
ON ERR GOSUB L

These two commands are used for handling error routines from within your 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 a GOSUB will be made to line L where a routine will perform what ever action has been programmed (by you) 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 take place and is then handled by a subroutine.

If an ON ERR GOSUB statement is used then 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 after a program using it has terminated just POKE 3204,0. This can, of course, also be done from within a program.

Example;

```
10  REM Program to print list of all error messages and reserved words
20  SPEED 210: REM so that you can take it all in!
30  FOR I =1 TO 255
40  ON ERR GOSUB 100
50  PRINT CMD$(I)
60  NEXT
70  SPEED 255: POKE 3204,0: END
80  REM Now for the error handling routine
100 REM Test for end of error message list
110 IF I<64 THEN I=63: RETURN
120 REM Test for end of user defined reserved word list
130 IF I<128 THEN I=127: RETURN
140 REM Must beat end of function list
150 POP: GOTO 80
```

Functions

ABS(X) returns the absolute value of expression X.

Example:
ABS(-314159) returns 314159

ASC(A$) returns the ASCII value of the first character in the string.

Example:
ASC("ABC") returns the value 65 (remember BASIC is decimal).

ATN(X) returns the arctangent of X in radians ranging from –pi/2 to + pi/2

Example:
ATN(1) returns 0.785398 which is pi/4.

CHR$(X) returns a single character string whose ASCII value is X. X must be an integer (if it isn't then BASIC will round down) between 0 and 255.

Example:
PRINT CHR$(30) returns a clear screen character and then clears the screen.
PRINT CHR$(69) returns an E (decimal - remember?).

CMD$(X) returns d string of the reserved word or error message corresponding to

the argument X (which must be between 1 and 255).

if X is less than 64 but greater than 0 then the error string corresponding to X (see index of error messages on appendix page vi) is returned.

Example:
PRINT CMD$(2) returns the string SYNTAX.

If X is 128 or more a reserved word is output.

Example:
X$=CMD$(142) puts RETURN into X$.

If X lies between 64 and 127 then a user defined reserved word is returned, according to its position in the auxiliary table.

If the argument refers to a non-existent message or word the error message CMD ERROR will be displayed.

COS(X) returns the cosine of X where X is in radians.

DEF .... FN is used to define a user function. It takes the form DEF FN NAME(V) = E and must be kept to one line. NAME can be any legal name and V can be any legal variable name. V is a dummy variable and can be used within the expression E. The DEF statement is ignored (as if it were a REM) until a function call is encountered in the program. A function call takes the form FN NAME (E).

NAME identifies the appropriate DEF FN statement and then E is assigned to variable V whenever it occurs within the DEF FN statement with the result being returned to the calling expression.

If a function call occurs before the appropriate DEF FN then the program stops and displays the message FN DEFN ERROR IN L. DEF FN is not allowed in direct mode.

Example:
```
10  DEF FN ASN(X) = ATN(X/SQR(1-X*X))
20  DEF FN ACS(X) = 1.570796-FN ASN(X)
30  FOR I=0 TO 1 STEP .1
40  PRINT I, FN ASN(I), FN ACS(I)
50  NEXT I
```
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.I. Have a go...

ERR returns the number of the last error that occurred. This function is particularly useful within ON ERR routines (see chapter on Commands and statements) to find out what error actually occurred.

Example:
If the last error was a syntax error (it often is!) then PRINT ERR will output the value 2.

EXP(X) raises e (value 2.71828) to the power of X. If X is greater than about 87 the error message OVFL ERROR is output.

INCH waits for a keyboard entry and returns the ASCII value of that entry. Very useful for single character interactive responses such as Y/N?

Example:
```
10  PRINT "Type in a character"
20  A$=CHR$(INCH)
```

30 PRINT "You typed a: "; A$: END

Note; see also KBD.

INP(X) returns the current value of input port X as a number from 0 to 255. X must be an integer between 0 and 255.

INT(X) returns the largest integer below or equal to X.

Example:
PRINT INT(3.14159) returns the value 3.

KBD scans the keyboard to see if a key has been pressed. It returns zero if no key has been pressed or the ASCII value if one has. *It does not wait for a keypress.*

LEFT$(A$,X) returns the leftmost X characters of string A$.

Example:
LEFT$("ABCD",2) will return the string AB.

LEN(A$) returns the length of the string A$ including punctuation marks, control characters and spaces.

Example:
LEN("ABCD") returns the value 4.

LOG(X) returns the natural logarithm of X where X must be greater than 0.

MID$(A$,X) returns the rightmost characters of A$ remaining after the Xth character.

Example:
MID$("ABCD",2) returns the string BCD.

MID$(A$,X.Y) as above, but returns only Y characters from the Xth.

Example:
MID$("ABCD",2,1) returns the string B.

PEEK(X) returns an integer from 0 to 255 representing the contents of the memory location X (specified in decimal). X must be an integer from -32767 to+32767 (as for CALL).

POS(X) returns the value of the present column position on the output device. The value of X is immaterial but should be legal.

PI returns the value 3.14159 and is faster than using a variable to hold the number.

RND(X) returns a random number from 0 to 1. If X<>0 a new random number will be returned; if X=0 the previous number returned is returned. 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.

RIGHT$(A$,X) returns the X rightmost characters of string A$.

Example:
RIGHT$("ABCD",2) returns the String CD.

Note: X must be an integer in the range 0 to 255.

SGN(X) returns the sign of X: if X>0 it returns +1, if X<0 it returns -1 and if X=0 it returns 0.

SIN(X) returns the sine of X where X is in radians.

SIZE returns the size of memory available for programs, variables and pointers. On a 16K system it will initially return 8907 bytes. If your system is greater than 32K a *negative* number is returned.

SIZE$ returns the size of memory available for strings. When the system is initialised via E 1000 or E 1004, SIZES returns value 50. (See CLEAR to alter this).

Note: the use of $ on this function is not strictly proper since a number rather than a string is being returned, but as a mnemonic its use outweighs its correctness (we think!).

SPC(X) prints X spaces and is only valid within a print statement. X must be an integer between 0 and 255.

SQR(X) returns the square root of X. X must be greater than 0.

STR$(X) returns a string representation of a numeric variable.

Example:
If the value of the variable X is 1.234, STR$(X) returns the string "1.234".

TAB(X) causes spaces to be printed until the (imaginary) print-head reaches column X on the output device. Only valid within PRINT statements.

TAN(X) returns the tangent of X where X is in radians.

VAL(A$) returns the numerical value of string A$. If the first character of A$ is not a **.**,**+**,**-**, or a digit then VAL(A$) returns the value 0.

Example:
VAL("1.234") returns the value 1.234.

User defined reserved words

XTAL BASIC 2.2 has a capability which, it is believed, is at the time of writing unique to this version 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.

When a line of BASIC is entered it is not used by the interpreter in the form in which it was typed, instead, each reserved word is shortened to an appropriate single byte code.

For example: END (ASCII codes 45_{16} (E), $4E_{16}$ (N) 44_{16}(D)) becomes simply 80_{16} while PRINT and ? become 98_{16}. Since variable names and numbers normally use ASCII codes (from 0 to $7F_{16}$) we can use codes 80_{16} to EE_{16} to represent our own reserved words - which, incidentally, is why graphics characters can be used in place of reserved words on the Nascom 2.

This rule is not followed within REM, DATA. CLOAD or CSAVE statements, nor

between double quotes where any combination of characters, including graphics, are allowed.

LIST and EDIT both 'blow up' the reserved word codes into the actual words normally used so that the user is not normally aware that all this is going on. As a result of this process programs are considerably compacted without any loss of clarity.

In XTAL BASIC 2.2 all user defined words are internally shortened to two bytes, the first one always being FF16 to distinguish them from the inbuilt reserved words. These words are stored in the *Auxiliary Reserved Word Tables* with their addresses, in turn, stored in *the Auxiliary Address Table*.

Auxiliary reserved word table at: $E80_{16}$ - $F7F_{16}$
Auxiliary address table at: $FB0_{16}$ – FFF_{16}

To create a new reserved word requires only a few simple steps. First of all the user defined (auxiliary) reserved word table and the auxiliary address table must be cleared either by entering BASIC from the monitor via E 1004 (see page 4) or by issuing a CLEAR% command from within BASIC (see page 6). The following procedure should then be followed:

(i) Find a free area to place the machine code routine. The area $0E00_{16}$ to $0E7F_{16}$ is free but only 128 bytes in length. Far more useful is the fact that the area from $2D00_{16}$ onward can be used so long as the location TEXT (see appendix page iii) is set to a point above the new routine(s). This enables you to save the interpreter and its new additions as a continuous block. *(see also P.7 under CLOAD)*

(ii) The name of the routine, its reserved word, must now be written into the auxiliary reserved word table ($0E80_{16}$ to $0F7F_{16}$) as a set of ASCII codes, *the first letter having its top bit set.* The easiest way to do this is to look up the ASCII (hex) code for the first letter, add 8_{16} to the first (hex) digit and enter this. Thus, for example, the letter E, ASCII 45_{16} would be entered as $C5_{16}$ since 4_{16} (the first digit)+ 8_{16} = C_{16}. The first word in the table starts at $0E80_{16}$ and each word is entered contiguously with the next (i.e. there are no spaces between words) with the table, no matter how short or long, always ending in 80_{16}.

(iii) The appropriate address in the auxiliary address table is then modified so that, for example, the first address (relating to the first word) points to the address at which the first machine code routine starts.

If an entry is now made into BASIC via E 1000 or E 1002 (but not E 1004 which will clear out the newly created tables!) your reserved word and its routine will behave exactly as though it had always been a part of XTAL BASIC 2.2.

Example.1
XTAL BASIC 2.2 does not have a reserved word for CLEAR SCREEN. This has been deliberately omitted to encourage users lo have a go at creating their own routines and this example shows you how to develop this command.

To clear the screen under Nas-Sys monitor it is only necessary to load register A with $0C_{16}$ and then do a RST 30. The full machine code routine is thus:

```
3E 0C      (load A with (cs) character code)
F7         (RST 30)
C9
```

Find a suitable spot - we shall use $0E00_{16}$ - for your routine and enter it using the Nascom ' M' command. (If you are using one of the earlier monitors then use the following routine: 3E 1E C3 3B 01 in place of the above. Note that (cs) is $1E_{16}$ in earlier monitors.)

Now you must enter the new reserved word in the auxiliary reserved word table. We shall use the word CLS as the new command. CLS in ASCII is 43_{16} (C) $4C_{16}$ (L) 53_{16} (S) which becomes C3 4C 53 once we have set the top bit of the first letter as described above.

The final step is to modify the auxiliary address table at location $0F80_{16}$ (since this is going to be the first entry) so that it points to our routine. Using the 'M' command you can now enter

M 0F80
0F80 xx> 00 0E

Now re-enter BASIC via E 1000 (or E 1002 if you have a program to be saved) and try filling the screen with a bit of garbage. Type in a (nl) then enter the following line:

CLS (nl)

whereupon your screen should clear and the cursor will appear in its normal starting position.

Now you know how to do it for a simple command, you can go on to create routines as simple or as complex as you like and they wilt all behave as though they had always been a part of XTAL BASIC 2.2.

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 then the word must end with a "(" (ASCII 28_{16}) to indicate that an argument is to follow (see the next worked example on DEEK and DOKE).

In a command routine the HL register pair should be stacked if it is likely to be modified during execution of the routine since HL holds the memory address of the position directly following the command word. HL can then be 'popped' back (using RET or $C9_{16}$ to get back to BASIC).

In a function routine, on the other hand, the pointer to the text position has already been stacked and should be 'popped' and incremented to find the value of the argument. The routine will have a special end since a right bracket must follow the argument expression (again see the worked examples on DEEK and DOKE).

Note: If an auxiliary reserved word has been defined and used in a program but has subsequently been cleared from the table, you will still be able to LIST the program, whereupon all references to that word will LIST as a decimal number preceded by two question marks (e.g. ??64).

Example.2
DEEK and DOKE are two byte equivalents of PEEK and POKE first seen in the NASCOM 8K BASIC interpreter. They are very useful for reading and storing 16 bit quantities such as addresses.

First the machine code routines:

```
0E 00. CD 61 17  DOKE   CALL UEXINT   ; Fetches integer expression in
   03. D5                PUSH DE      ; range ±32768 in DE
   04. CD 4C 15          CALL TSTCOM  ; Look for '.' Between expressions
   07. CD 61 17          CALL UEXINT  ; Fetch 2nd integer expression
   0A. E3                EX HL,(SP)   ; Put 1st expression in HL
   0B. 73                LD (HL), E   ;
   0C. 23                INC HL       ; Store second expression at
   0D. 72                LD (HL), D   ; address given by first.
   0E. E1                POP HL       ; Restore text pointer
   0F. C9                RET
0E 10. E1          DEEK(  POP HL      ; Retrieve text pointer
   11. 23                INC HL
   12. CD 61 17          CALL UEXINT  ; Read 2 bytes at (DE)
   15. 1A                LD A, (DE)
   16. 13                INC DE
   17. 47                LD B, A
   18. 1A                LD A, (DE)
   19. E5                PUSH HL      ; Stack text pointer, then convert
   1A. C3 A4 2B          JP FNENDI    ; number in AB to f.p and test
                                      ; for ) to end
```

Now the reserved words are entered into the tables:

```
M 0E80
0E80 80>C4 4F 4B 45 C4 45 45 4B 28                    DOKEDEEK(
```

Followed by their addresses:

```
M 0F80
0F80 0B>00 0E 10 0E                    Addresses 0E00₁₆ and 0E10₁₆
```

And DEEK and DOKE are now part of the interpreter. Run up XTAL BASIC 2.2 via E 1000 or E 1002 and try this test:

```
] DOKE 3024,0
```

Two square boxes should appear at the top of the screen.

Then try:

```
] ?DEEK(3024)
```

And the answer 0 should appear.

Example.3

The function RAD is a degree to radian conversion function which takes a floating point expression (in degrees) and converts it to radians by multiplying it by pi/180.

First the machine code routine:

```
0E 1D. E1          RAD(  POP HL       ; Retrieve text pointer
   1E. 23                INC HL
   1F. CD 77 1B          CALL EXNMCK  ; Fetch an f.p. expr in 0CBF₁₆ to
                                      ; 0CC2₁₆
   22. 01 0E 7B          LD BC, 7B0E  ; BCDE contains pi/180
   25. 11 35 FA          LD DE, FA35
   28. E5                PUSH HL      ; Save text pointer again
   29. CD FB 24          CALL MULTI   ; and do multiplication
   2C. C3 AA 2B          JP FNEND     ; Test for ) and end.
```

Then the reserved word table.

```
M 0E89
0E89 80>D2 41 44 28                    RAD(
```

And the address table:

```
M 0F84
0F84 0B>1D 0E
```

Now run up BASIC and try the following:

```
? SIN(RAD(30))
```

and the result should give the sine of 30 degrees, 0.5.

Example.4

SETVID is a routine in XTAL BASIC 2.2 which will restore normal VDU output after, for example, using a printer with a different format. SETVID lives at 11229_{10} so we could access it using either:

```
CALL 11229
```

or

SETVID 11229 followed at any time by CALL SETVID

but both these methods are wasteful of space compared to making use of the two bytes required for a reserved word. Since the routine is already in the interpreter we need only modify the tables as follows:

```
M 0E8D
0E8D 80>D3 45 54 56 49 44              SETVID
```

```
M 0F86
0F86 0B> DD 2B
```

And SETVID is now a reserved (two byte) word instead of being a seven byte CALL command.

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).

The forms of error messages are:

XXXXXX ERROR (in direct mode)

XXXXXX ERROR IN L (in deferred mode, where L is a line number)

and XXXXXX will be one of the following:

SYNTAX A typing error has been made or a command has been wrongly formatted.

MEM FULL An attempt has been made to execute a command which would require more memory than is available.

OVFL A numeric overflow has resulted from a calculation.

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

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

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

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

DATA A READ statement has been presented with insufficient data.

OPERAND An operand has been omitted after an operator.
Example: PRINT 2+3+4+

QTY A parameter in an array or function is out of range.

Examples:
A(X) where A is an array and X<0 or not an integer.
LOG(X) where X<=0
SQR(X) where X<0

Note: reference lo the chapters on commands and functions should 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.

STR OVFL Maximum string length is 255 characters.

STR SPC All RAM space available for strings has been used up. See CLEAR command to overcome but beware of using this in mid program!

TYPE An attempt has been made to use a number in place of a string (or vice versa).

DIMENSION An attempt has been made to re-dimension an array. An array can only be DIMmed once in a program. This includes arrays of under ten elements that have not been formerly DIMmed.

DIRECT INPUT and DEF statements cannot be used in direct mode.

CONT An attempt has been made to continue a program after an error occurred, after alterations have been made to the program or if no program existed.

FN DEFN An attempt has been made to refer to a user defined function that has not yet been defined (see DEF under Functions).

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

CMD An attempt has been made to reference a user defined command which does not exist in the system - a common cause for this is that the user tables have been inadvertently CLEARed.

Alternatively, in the CMD$ function, a code has been selected for which there is no error message or reserved word string.

TAPE A checksum error has been detected while loading or verifying a program.

Appending BASIC Programs

As outlined previously in the section on CLOAD, it is possible lo append one program to another in XTAL BASIC 2.2, making use of the facility to load programs wherever we please. The following assumes that we wish to append a program PROG2 on the end of another program PROG1:

(i) Ensure that the two programs have no line numbers which would appear in the wrong place when joined (i.e. the smallest line number in PROG2 must be greater than the largest in PROG1).

(ii) Type CLOAD PROG1.

(iii) Now do DOKE 3212. DEEK(3255)-2 it you have added DEEK and DOKE to your system. Otherwise, do:

```
A=PEEK(3255): B=PEEK(3256)
IF A<2 THEN A=A+256: B=B-1
POKE 3212,A-2: POKE 3213.B
```

This step simply brings TEXT up to TXTUNE-2. If a LIST is now done, you will find that PROG1 will seem to have disappeared. It hasn' ŧ it is simply being 'held' out of the way, so that following CLOADs don't destroy it.

(iv) Type CLOAD PROG2.

(v) Finally, do DOKE 3212.11520
or POKE 3212.0: POKE 3213,45
This restores TEXT to $2D00_{16}$, or, with other values, to wherever your TEXT location normally points.

NOTE: *These POKE addresses are for Nascom and identical systems.*

You should now have a program with no visible join which may now be LISTed, RUN or (preferably first) CSAVEd. If you wish to append a third program, you may return to step (iii), if several programs are to be appended to the first, ignore step (v) until all of the programs have been appended, then do step (V) at the end.

Creating Useful Subroutines

Using the cassette appending facility described above, the user can build his/her own library of subroutines in BASIC for incorporation into programs. A subroutine so written should have larger line numbers than those in a program likely to incorporate it, so that it may be CSAVEd as a routine and appended as required to the user's program.

The subroutine below suspends the printing of (nl) characters, so that the user can, for example, remain on the same line after an INPUT (under the NAS8UG monitors, an automatic screen scroll will normally occur at that point). All printing and positioning of the cursor can then be performed by means of the PRINT@ command. Note especially the use of PRINT@ in lines 40 and 50, to position the cursor without printing anything.

The routine at 10000 puts a machine-code subroutine in the space from $0E79_{16}$ to $0E7F_{16}$, the codes being stored in a DATA statement within the program. The first number in the DATA statement gives the number of bytes -1 to be put into memory.

```
5    SETVID=11229:   REM Sub-routine within BASIC
6    REM                    to restore normal VDU vector
10   GOSUB 10000:    REM Set up sub-routine
20   POKE OV, OL:    POKE OV+1, OH:          REM Set up output vector
30   PRINT CHR$(30); @ 15,0,"NON-SCROLL TEST"
40   PRINT@ 1,1, INPUT "Your move"; A$
50   PRINT@ 1,14, INPUT "Your next": B$
60   CALL SETVID: REM Resets VDU output
70   END
```

```
10000  REM Generate machine-code subroutine at 0E79(16)
10010  OP=3705: OV=3147: READ L
10020  OH=INT(OP/256): OL=OP-OH*256
10030  FOR I=OP TO OP+L: READ N: POKE I,N: NEXT
10040  RETURN
10050  DATA 5,254,31,200,195,59,1
```

The little machine-code routine entered by this subroutine is as follows

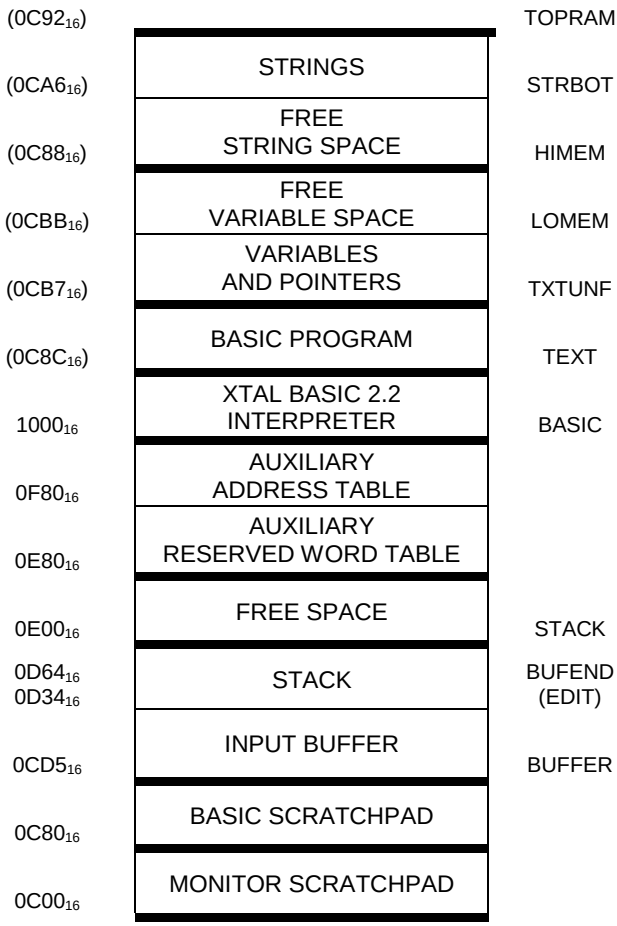
0E79 - FE 1F C8 C3 3B 01

Hex Dump of Fast Tape Loader

```
Addr.      1  2  3  4  5  6  7   8  9  A  B  C  D  E  F

0E00-  CD E9 0E CD 51 00 CD 3E  00 FE FF 20 0D 06 03 CD
0E10-  3E 00 FE FF 20 04 10 F7  18 12 FE 1E 20 E8 06 03
0E20-  CD 3E 00 FE 1E 20 E2 10  F7 C3 51 00 CD 3E 00 6F
0E30-  CD 3E 00 67 CD 3E 00 5F  CD 3E 00 57 0E 00 CD 70
0E40-  0E CD 3E 00 B9 20 12 43  0E 00 CD 3E 00 77 81 4F
0E50-  23 10 F7 CD 3E 00 B9 28  0A EF 45 52 52 4F 52 1F
0E60-  00 18 08 CD 40 02 AF BA  CA 51 00 18 99 00 00 00
0E70-  D5 CD 32 02 E3 CD 32 02  E3 D1 C9 00 00 00 00 00
0E80-  CD E9 0E CD 51 00 AF 47  FF 10 FD 2A 0E 0C ED 5B
0E90-  10 0C EB 37 ED 52 DA 51  00 EB AF CD 5D 00 06 04
0EA0-  3E FF CD 5D 00 10 F9 AF  BA 20 02 43 04 58 7D CD
0EB0-  5D 00 7C CD 5D 00 7B CD  5D 00 7A CD 5D 00 0E 00
0EC0-  CD 70 0E 79 CD 5D 00 0E  00 7E 81 4F 7E CD 5D 00
0ED0-  23 10 F6 06 0B 79 CD 5D  00 AF 10 FA CD 40 02 18
0EE0-  AD 00 00 31 33 0C C3 59  03 D1 21 E3 0E E5 D5 C9
```

XTAL BASIC 2.2 Memory Map



Note that the T2 tape loader/dumper loads from $0E00_{16}$ to $0EFF_{16}$, but will be corrupted as soon as XTAL BASIC 2.2 is run up, since the auxiliary reserved word tables occupy that space. This should not normally cause problems, since XTAL BASIC 2.2 has its own tape routines, and the tape loader can be relocated to, say. $0D00_{16}$, by changing the following locations from $0E_{16}$ to $0D_{16}$ (for Nascom users), and then copying down to $0D00_{16}$:
$0E02_{16}$, $0E40_{16}$, $0E82_{16}$, $0EC2_{16}$ and $0EEC_{16}$.

VDU Map for PRINT @ command

Hex Address		00	04	08	12	16	20	24	28	32	36	40	44	47
														X coordinate
0BCA	0													
080A	1													
084A	2													
088A	3													
08CA	4													
090A	5													
094A	6													
098A	7													
09CA	8													
0A0A	9													
0A4A	10													
0A8A	11													
0ACA	12													
0B0A	13													
0B4A	14													
0B8A	15													
	Y coordinate													

The above diagram should help the user to understand how the PRINT@ command relates to the VDU Map shown in the Software Notes on the Nascom 1. This command allows the user to print characters at any specified point on the screen. The '@' can appear as many limes as desired within a PRINT command, for 'plotting' several points at once.

Example:

10 A$=CHR$(127): PRINT@ 0,0,A$;@46,0,A$;@ 0,15,AS; @ 46,15,A$;

This would print a block at each of the four corners of the screen.

There are one or two restrictions on the use of this command:

First: it is not very useful in direct commands (unless used on Row 0,which does not scroll), since the screen is always scrolled at least twice after a direct command. *Second:* if text is printed at the end of the bottom line (i.e. @47,15), this will also scroll the screen.

The screen will also scroll, of course, when the program ends or breaks.

A ';' should follow any expression PRINTed by this command, again so that the cursor remains in the right place after the printing.

Incompatibilities with XTAL BASIC 2.1

Happily, there are very few and, in general, your earlier programs wil1 run very nicely in version 2.2.

Any programs which used SIZE or SIZE$ must be EDITed at the appropriate places - you will find the word CMD$ or CMD$$ and these should be changed back to SIZE or SIZE$ respectively.

Some locations within the interpreter which were used within programs will have moved - e.g. SETVID. which was at location 5484_{10} in 2.1 is now at 11229_{10} (this is the routine to restore normal VDU output - see subroutines appendix page i). In the BASIC scratch-pad, BUFFER now starts at $0CD5_{16}$ the location of the SPEED parameter is at $0C86_{16}$. and the scratch-pad itself starts at $0C80_{16}$ (to allow for the expanded monitor scratch-pad under NAS-SYS).

PCW Benchmarks

For those who are interested in such, and to show that we have achieved a reasonable speed improvement with XTAL BASIC 2.2, here are the timings, taken on a NASCOM 1 computer running at 2 MHz, of the execution of the eight test programs shown by Personal Computer World magazine (January 1978 issue):

	BM1	BM2	BM3	BM4	BM5	BM6	BM7	BM8
XTAL BASIC 2.1	2.9	12.5	24.7	26.1	30.4	50.3	72.4	10.2
NASCOM 8K BASIC	2.2	10.8	22.2	23.2	25.2	38.6	55.2	10.4
XTAL BASIC 2.1	1.7	9.9	20.9	22.2	24.8	36.9	52.8	9.4

These timings (all in seconds) are, of course, halved on a machine running at the full 4 MHz of the Z-80A processor.

Error Message and Reserved Word Lists

ERROR MESSAGES		RESERVED WORDS									
					CONT	99		**INT**	B3		
Next	01	**END**	80		**LIST**	9A		**ABS**	B4	*180*	
Syntax	02	**FOR**	81		**CLEAR**	9B		**CMD$**	B5		
Return	03	**NEXT**	82	*130*	**CLOAD**	9C		**INP**	B6		
Data	04	**DATA**	83		**CSAVE**	9D		**POS**	B7		
Qty	05	**NAS**	84		**NEW**	9E		**SQR**	B8		
Ovfl	06	**INPUT**	85		**SPEED**	9F		**RND**	B9		
Mem Full	07	**DIM**	86		**POP**	A0	*160*	**LOG**	BA		
Branch	08	**READ**	87		**TAB**	A1		**EXP**	BB		
Range	09	**LET**	88		**TO**	A2		**COS**	BC		
Dimension	0A	*10* **GOTO**	89		**FN**	A3		**SIN**	BD		
Division	0B	**RUN**	8A		**SPC**	A4		**TAN**	BE	*190*	
Direct	0C	**IF**	8B		**THEN**	A5		**ATN**	BF		
Type	0D	**RESTORE**	8C	*140*	**NOT**	A6		**PEEK**	C0		
Str Spc	0E	**GOSUB**	8D		**STEP**	A7		**LEN**	C1		
Str Ovfl	0F	*15* **RETURN**	8E		**+**	A8		**STR$**	C2		
Str Complex	10	**EDIT**	8F		**-**	A9		**VAL**	C3		
Cont	11	**REM**	90		******	AA	*170*	**ASC**	C4		
Fn Defn	12	**STOP**	91		*****	AB		**CHR$**	C5		
Operand	13	**OUT**	92		**/**	AC		**LEFT$**	C6		
Cmd	14	*20* **ON**	93		**AND**	AD		**RIGHT$**	C7		
Tape	15	**CALL**	94		**OR**	AE		**MID$**	C8	*200*	
		WAIT	95		**>**	AF		**SIZE**	C9		
		DEF	96	*150*	**=**	B0		**INCH**	CA		
		POKE	97		**<**	B1		**KBD**	CB		
		PRINT	98		**SGN**	B2		**ERR**	CC		
								PI	CD	*205*	

Decimal representations of some of the codes are given in italic for ease of calculation.