THE "HACK'S" GUIDE TO THE INNARDS OF THE XTAL BASIC 2.2 INTERPRETER

"Abandon hope, all ye who enter here!"

TABLE OF CONTENTS

I. LIST OF XTAL BASIC 2.2 SUBROUTINES AND ADDRESSES

In the EPROM version, which loads at E000 – FCFF, any addresses in the range 1000-2CFF should have an offset D000 added to them.

1. ADDRESSES OF COMMANDS AND FUNCTIONS

These are given in address order, i.e. the order in which they may be found within the interpreter:

KBD	1285	INCH	128D	ERR	1292	PI	1298
SPEED	12A6	NEW	140A	CALL	1572	LIST	157E
FOR	1630	STOP	16FF	END	1701	RESTORE	172D
CONT	1746	CLEAR	17B0	RUN	1812	GOSUB	181D
GOTO	182E	POP	184C	RETURN	1862	DATA	1885
REM	1887	LET	189A	ON	18F0	IF	1953
PRINT	19B2	INPUT	1A56	READ	1A7E	NEXT	1B2D
OR	1CAB	AND	1CAC	NOT	1D30	DIM	1D4A
SIZE	1ECD	POS	1F01	DEF	1F08	FN	1F2A
STR$	1F86	LEN	2157	ASC	2166	CHR$	2177
LEFT$	2187	RIGHT$	21B6	MID$	21BF	VAL	21ED
INP	2212	OUT	221D	WAIT	2223	PEEK	2260
POKE	2267	RND	2280	CMD$	22FC	CLOAD	232E
+	23B8	-	23C8	*	24F9	/	254B
LOG	24BA	SGN	2603	ABS	2619	INT	26C2
SQR	286E	↑	2877	EXP	28BB	TAN	28FB
ATN	2910	COS	2938	SIN	293E	CSAVE	2B3F
EDIT	2C00						

2. SCRATCH-PAD LOCATIONS

RNDNO	0C80-0C83	Holds last random number generated by RND.
ERRNO	0C84	Number of last error generated.
ERRMOD	0C85	Current error mode (non-zero if ON ERR.. in force).
SPEED	0C86	Delay for VDU display.
COMMAX	0C87	For PRINTing in zones, gives largest column for which Xtal BASIC will try to find new zone.
HIMEM	0C88	Lower limit of available string space.
LNNO	0C8A	Current line number.
TEXT	0C8C	Pointer to start of BASIC text area.
PRTCOL	0C8E	Current print column.
DIMFLG	0C8F	Flag used in DIM and FNDVAR routines, to indicate If we are DIMensioning an array, or just accessing it.
TYPE	0C90	Type of expression being evaluated, 0=Number, 1=String.

DATFLG	*0C91*	*In COMPRSS, flag to show when in REM / DATA statements, so that characters after these are not checked for command / function strings.*
TOPRAM	*0C92*	*Topmost RAM location available to BASIC.*
STRPTR	*0C94*	*Pointer to end of list in STRLST.*
STRLST	*0C96-0C9F*	*Storage for string sub-expressions*
STKSAV	*0CA0*	*Save SP at the start of each statement*
CHAR	*0CA2*	*Temporary string expression pointer.*
CHRADR	*0CA4*	*Address of temporary string.*
STRBOT	*0CA6*	*Pointer to bottom of used string space*
PTR	*0CA8*	*General-purpose pointer.*
DATLN	*0CAA*	*Line number of current DATA statement*
VTYPE	*0CAC*	*Variable / Array type for use in FNDVAR routine (1D53).*
RETFLG	*0CAD*	*Flag used by RETURN statement.*
RDFLAG	*0CAE*	*In READ / INPUT, flag to show which we are in (i.e, READ or INPUT statement).*
TXTPTR	*0CAF*	*Save text pointer at start of statement.*
EXPPTR	*0CB1*	*General-purpose pointer used in expression evaluations.*
LNNZ	*0CB3*	*'Old line' pointer.*
TXTPZ	*0CB5*	*'Old text' pointer.*
TXTUNF	*0CB7*	*Pointer to end of program text.*
VARPTR	*0CB9*	*Pointer to end of simple variable area.*
LOMEM	*0CBB*	*Pointer to end of array storage area.*
DATPTR	*0CBD*	*Pointer to current position in current DATA statement.*
FPA	*0CBF-0CC2*	*Floating-Point Accumulator.*
TEMP	*0CC3*	*Location used in FP calculations.*
PRTTXT	*0CC4-0CD0*	*Text area for forming numbers before printing.*
ERRLN	*0CD1*	*Line number for location of ON ERR..routine.*
	0CD3-0CD4	*Spare (but RESERVED!) locations.*
BUFFER	*0CD5-0D34*	*Buffer for input lines.*
	-0D64	*Extended buffer for EDIT.*
STACK	*0E00*	*Top of stack space.*

3. ENTRY POINTS FOR XTAL BASIC

You already know the 1000, 1002 and 1004 entry points (called XBASIC, XBAS1 and INIT respectively). Another one worth knowing is at 1355, or READY.
This is like XBAS1, except that the variable space is not cleared and the stack is left alone. It is, in fact, the address to which execution is transferred after a STOP, END or LIST statement.

4. RECONFIGURING THE INTERPRETER FOR EXTRA COMMANDS / FUNCTIONS

Locations 0C80-0C8E are copied from a TABLE at 1277, when Xtal BASIC is initialised. Of particular interest here is location 1283 (HTEXT), or the 'hard text' pointer, which is copied to 0C8C (TEXT). If we add user commands and / or functions to Xtal BASIC, they may be added onto the top of the interpreter, and then HTEXT can be moved to point to the next 256-byte block ABOVE the new routines. Then, whenever Xtal BASIC is run up, the program text area will automatically start at this new position, and all of your BASIC programs will obediently load themselves in the new spot, too! This therefore provides a very easy and effective way of 'Reconfiguring' your interpreter.

5. ERROR MESSAGES

These are handled by the sub-routine ERROR which is at 1319. The only register which matters here is E, which contains the error number. This number determines which error message will be displayed, as shown on the back page of the Xtal BASIC manual. The contents of the other registers are immaterial, and it is not necessary to save any of them before entering the routine.

Three scratch-pad locations are of importance here:

ERRMOD 0C85 This is 0 for displaying normal error messages, 89 for the ON ERR GOTO .. mode, and 8DH for the ON ERR GOSUB .. mode. If we are in either of these latter two modes, the error message is not displayed, and execution transfers to location GOERR at 1927, which searches for the line number of the error handling routine.

ERRNO 0C84 This contains the error number of the last error that occurred, and is set whenever the ERROR routine is called. It is accessed when the function ERR is called.

ERRLN 0CD1 Contains the line number of the error handling routine for the ON ERR .. error modes.

Entry points for individual error routines are also available (i.e, places where the error number has simply been placed in the E register, and a jump made to ERROR), as follows:

MEMFUL	*12FE*	*Mem Full*	*SYNERR*	*1308*	*Syntax*
CMDERR	*130B*	*Cmd*	*DIVERR*	*130E*	*Division*
NXTERR	*1311*	*Next*	*DIMERR*	*1314*	*Dimension*
UFNERR	*1317*	*FN Defn*	*QTYERR*	*1787*	*Qty*
BRNERR	*1847*	*Branch*	*TYPERR*	*1B82*	*Type*
RNGERR	*1E49*	*Range*	*TAPERR*	*2A76*	*Tape*

6. USEFUL GENERAL-PURPOSE ROUTINES

GETKEY 2ACC Input character from keyboard, returning character in A, and with the carry flag set if the character is the 'CS' (BREAK) code. In that case, A=00. No other registers are affected. Note that this routine actually WAITS for a keypress. Non-NASCOM users MAY need to modify this routine, since it refers to the cursor location (to blink it under NAS-SYS).

GETK 2B00 A special routine written specifically for the NASCOM, to allow for the slowness of the NASCOM keyboard scan. This just detects the 'CS' or 'BS' characters, for breaking into a program or LIST. It may be replaced by a jump to VKBD (2BF7) on systems other than NASCOM.

RDLN 1512 Prints the character in A, and then reads in a line at the keyboard, into the BUFFER starting at 0CD5. As each character is typed, it is echoed to the screen, with the exceptions that:

'BS' backspaces cursor and removes the last character from the buffer.

'CS' exits the routine with the carry flag set.

'NL' exits the routine with the carry flag reset, terminates the line in the buffer with a 00 byte, and leaves HL pointing to 0CD4 (BUFFER-1).

No other control-characters are allowed, and characters will not print if the line length is 96 characters (the maximum).

Registers affected: A and HL.

PR 155C 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.

Registers affected: NONE, but user's output routine MUST reset the carry flag.

CRLF 1A09 Prints CRLF, using PR.

CRLFZ 1A0E As for CRLF, except that no CRLF is printed if the cursor is already at column zero.

Registers affected: A, set to 0DH (or 00H if at column zero under CRLFZ).

PRM 2AF0 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 00-7F. Example: To print 'Hello there' we have:-

```
CD F0 2A  48 65 6C 6C 6F 20 74 78 75 72 E5
           H  e  l  l  o     t  h  e  r  e
```

Registers affected: Just the A register, which exits holding the last character printed (still with top bit set). The routine returns at the address following that character in memory.

CPHLDE 1546 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 1759 Reads the character contained at (HL) into 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 registers are affected.

7. INTERNAL PROGRAM LAYOUT.

Before describing the general-purpose text routines, it is helpful to consider the way in which a program is stored within the interpreter. Many users will already know that Xtal BASIC does not actually use a line as typed, but instead shortens each reserved word into a unique single- or two-byte code. 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 quantity is stored, which points to the address of the following line in the text.

To illustrate this point, consider the following line of program text, which, for the sake of argument, is to be stored starting at address 2D01 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 Xtal BASIC, into the following form:

```
            300   FOR I  =  0    TO 9  :    PRINT SQR( I  )  :    NEXT: END
1B 2D 2C 01 81 20 49 B0 30 20 A2 39 3A 20 98 20 B8 28 49 29 3A 20 82 3A 80 00
```

Here, the first two bytes give the address of the next line (this is at 2D1B, as you will find if you count, taking 2D01 as the address of the 1B byte) and 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 Xtal BASIC 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 Xtal BASIC program are 00. The pointer TXTUNF always points one ABOVE the last byte.

8. GENERAL-PURPOSE TEXT SCANNING ROUTINES.

In general, Xtal BASIC uses the HL register pair as the pointer to the current position in the text. The following routines will then be found useful:

IGBLK 16D8 Increments HL, then scans the program text until the first non-space character is found. Note also IGBLKI (16D9), which does the same, except that HL. is not incremented first.

Registers affected: A contains the character found, and HL points to that character. Z flag is set if at end of statement (i.e, null or ':' found). Carry set if numeric character found (0-9).

TSTC 1551 Test the character pointed to by HL in the text, ensuring that it is the same as that immediately following the subroutine call. If it is not, give a SYNTAX ERROR. This is effectively a four-byte call, e.g, CD 51 15 29 looks for a ')' (e.g, to end an argument list in a function call).

Registers affected: A contains the character found, HL finishes up pointing to the next non-blank character FOLLOWING the one tested. Note that we may also use this routine to test for a reserved word.

TSTCOM 154C Special case of TSTC, tests for a comma ',' , and thus CD 4C 15 is equivalent to CD 51 15 2C (but is one byte shorter!).

FNDLN 13EC 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). By 'start of line', we mean four bytes before the actual text in that line (see program example above).

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 13EF 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 1449 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 (0CD5). 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 (0CD4) on exit, DE points to the last byte plus two in the compressed line, and C holds the number of characters in the compressed line, plus four to take account of the space needed for the line number and pointer.

II. XTAL BASIC 2.2 FLOATING-POINT SUB-ROUTINES

1. GENERAL

A floating-point number in Xtal BASIC 2.2 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 this is at locations 0CBF-0CC2. A further byte. 0CC3, called TEMP. is used by the f.p routines for storing temporary calculations but. apart from that, only the registers and the stack are used for f.p calculations.

The high byte of the four is the exponent (0CC2), 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 NEGATIVEI). 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↑(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 added 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 problem with conversion from binary to decimal, as well as the accuracy of the method used for calculating powers).

2. FLOATING-POINT FUNCTIONS

The addresses of the single-argument f.p functions are as follows. In each case, the argument is expected to be found in the FPA:

LOG	24BA	EXP	28BB	SQR	286E
SIN	293E	COS	2938	TAN	28FB
ATN	2910	RND	2280	ABS	2619
SGN	2603	INT	26C2		

Notes: COS is performed by the identity COS(X) = SIN(X + PI / 2)
TAN is calculated as: TAN(X) = SIN(X) / COS(X)
SQR is calculated as: SQR(X) = X ↑ 0.5

3. FLOATING-POINT OPERATORS

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	23B8	ADD1	23CD	SUB	23C8	SUB1	23CA
MULT	24F9	MULT1	24FB	DIVIDE	254B	DIV1	254D
POWER	2877	POWER1	2879	ADDN	23BF	SUBN	23C4

*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.*

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. Xtal BASIC stores a large table of constants in the area 29AE - 2A21, and here are some of the more useful ones:

HALFPI	*29AE*	*Pi / 2*	*HALF*	*29B2*	*0.5*
TWOPI	*29CB*	*Pi*2*	*QTR*	*29B6*	*0.25*
ONE	*29FD*	*1*	*NEGONE*	*2A1A*	*-1*

4. ADDITIONAL USEFUL ROUTINES

STKFPA	*2625*	*Returns with the FPA on the stack, in the form shown above. Destroys the DE registers.*
LDFPR	*265F*	*Copies the FPA to the FPR, leaving HL pointing to TEMP (0CC3).*
STFPR	*2635*	*Copies the FPR to the FPA, without affecting any registers.*
HLTOFPA	*2632*	*Copies the four bytes starting at (HL) into the FPR AND FPA, leaving HL pointing to the byte following the block of four.*
HLTOFPR	*2662*	*Copies the four bytes starting at (HL) into the FPR, leaving HL as above, but not affecting the FPA.*
FPATOHL	*263E*	*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.*
DETOHL	*2641*	*As above, but copies the four bytes starting at (DE) to those starting at (HL).*
CHKSGN	*25F4*	*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	*261D*	*Changes the sign of the FPA, turning it from a positive to a negative number, or vice versa. This affects A and HL. Note also the ABS function (2619) which sets the sign to positive.*

5. POLYNOMIAL EVALUATION

Xtal BASIC 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 *298E* *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 *297F* *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.

12.

SINTAB 29BA LOGTAB 29CF ATNTAB 29DC EXPTAB 2A01 are tables used within Xtal BASIC, but they wont 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 whole range of values.

III. EXPRESSIONS AND INTEGER FUNCTIONS

1. GENERAL

This is all very fine, but how do we get a number, or, more important, a complicated expression containing numbers, functions and operators, into the f.p format described in the preceding paragraphs? In fact, there is a set of very powerful routines which are available for just this purpose. 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	*1B8B*	*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 TYPE (0C90) 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	*1B77*	*As for EXPR, but only accepts a numeric expression, and returns 'Type Error', if a string expression is found.*
PARNZ	*1C36*	*As for EXPR, but expects the expression to be enclosed inside parenthesis (), returning 'Syntax Error' if not.*
PARN	*1B87*	*As for PARNZ, but only looks for a left bracket '(', so that more expressions can be evaluated, perhaps separated by commas (use TSTCOM (154C) to test for separating commas), finally finishing with a TSTC ')' (CD 51 15 29) to test for the right bracket.*
FCHNUM	*26F3*	*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* *The result is returned in the FPA.*
GETNM	*178C*	*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.*

UEXINT	*1761*	*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	*1769*	*As for UEXINT, but restricts the range to 0 to +65535.*
I255	*2250*	*Here, we restrict the range to 0 to +255, and the result is returned in A as well as DE (D=00, of course).*

NOTES:

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

Some users may still have copies of Xtal BASIC 2.2 which restrict integers to +/- 32767. If this is so, location 1776 should be modified from 90 to 91, when it will be found that the extended range is then available.

2. 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:

FORMNUM	*1EED*	*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.*
FM1BYT	*1F04*	*As above, but converts a one-byte integer (0-255) into an f.p number.*

You will probably have already noticed the methods given in the Xtal BASIC manual for ending the user-defined function reserved words -- FNEND in the example RAD(, and FNENDI in the example DEEK(, on page 22. These were added to Xtal BASIC in order to make it easier for users to finish their own routines with a single jump, but FNENDI simply uses FORMNUM, and then continues on into FNEND with the integer value now sitting in the FPA.

3. TYPE CHECKING ROUTINES

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

NUMCHK	*1B7A*	*Ensures that the expression just evaluated is a number.*
STRCHK	*1B7B*	*Ensures that the expression just evaluated is a string.*
TYPMCH	*1B7C*	*Checks that the type of one expression matches another. This is done by making the carry flag represent the type of the first expression (Reset if numeric. Set if string).*

In all of these cases, we return a TYPE ERROR if the wrong type was found, and the location TYPE (0C90) contains the type of the expression last evaluated. Only the A register and flags are affected by these routines.

4. 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 (at 0CBF-0CC0). In order to process the string correctly, we use the following routine:

FCHSTR	*2126*	*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 from 0C96 to 0CA1, and. simply 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 at 0CA2-0CA5 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	*215B*	*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.*
ASC0	*216A*	*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. See MUL$ in chapter V for an example of its use.*

STRSPC 200C 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 apace, a 'housecleaning' 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 1FAB As for STRSPC, but then assigns this string space to the 'temporary string accumulator' (CHAR), writing the length to 0CA2, and the start address to 0CA4 (CHRADR). 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 1FD9 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, and an example of its use is given in the MUL$ function in chapter V.

If the sub-expression list from 0C96 is full, a STR COMPLEX ERROR is returned. This is a rare occurrence, 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 TYPE (0C90) 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!

IV. STRINGS AND VARIABLES UNDER XTAL BASIC

1. 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. Of course, it is necessary to do a CLEAR N command before the DIM statement, if N bytes of string space are required for later use.

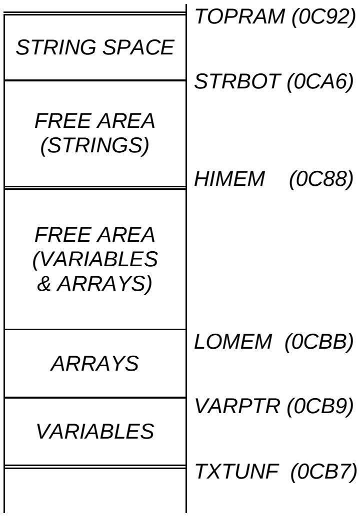
Even when a string is assigned, it may still not use up any string space, since the string may actually be a part of the program, e.g:

10 A$="NAME" or 100 DATA JACK,JILL,HILL

In these cases, when the string assignment is made, no string space is used, the pointer being made to point to the actual address within the program where the string sits. In addition, if we assign another string variable to the same string, its pointer is simply made to point to the same string, rather than create a duplicate string, and this can save a good deal of space.

2. INTERNAL STORAGE OF VARIABLES AND ARRAYS

The Xtal BASIC memory map above the program end looks like this:



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 (TXTUNF) to (VARPTR). If it is not found, the list is extended by increasing VARPTR by six bytes (and moving the arrays up six bytes, if necessary), and then inserting the following information:

First two bytes: The first two characters of the variable name, in the order second byte followed by first byte. In addition, the variable type is stored using the top bit of the first byte stored, this being 1 if the variable is a string. We had better give some examples:

A	stores as:	00 41
AB	stores as:	42 41
AB$	stores as:	C2 41
XYZ$	stores as:	D9 58

...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 it 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 41 00 00 40 82

XYZ$="hello" : D9 58 05 00 50 2D, where we are assuming that the string "hello" if stored at 2D50.

HOWEVER, if the variable does not exist, but appears on the RHS of an assignment, or is part of an expression, Xtal BASIC does NOT create it in memory, but instead returns zero, or a null string, as necessary.

b). Storage of defined functions.

There is one other type of 'variable', although it may not appear as such, and that is the DEF FN function. Sere, the function is defined within the variable space just as a numeric variable, except that the SECOND byte now has its top bit 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 RHS 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. In this way, it is a reasonably quick matter, on the call of this function, to directly stack and substitute the argument, and then evaluate the expression, without having to do a long-winded search through the whole program for the required DEF statement. Example: Suppose we have a DEF statement as the first line of a program, that the text starts at 2D01, and the variable space at 3000:

```
10 DEF FN HSN(X)=(EXP(X)-EXP(-X))/Z          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:

```
3000: 53 C8 10 2D 08 30 00 58 xx xx xx xx,      where the xx's contain some number.
            ^     ^
            |     |_____Note: 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. <u>Storage of arrays.</u>

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 two bytes: As for variables.

Bytes three and four: Give an offset to the start of the next array in memory.

Byte five: Gives the number of dimensions in the array. Let us call this number N.

*Bytes six to 2*N+5: 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 Xtal BASIC!). This should look like this:*

59 5B CF 0A 03 05 00 06 00 17 00 xx xx ....etc.

Here are the three dimensional 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 node 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).

3. <u>ROUTINES FOR ACCESSING VARIABLES DIRECTLY</u>

It is often necessary to access a numeric or string variable directly, rather than allow any type of expression (e.g, the CSAVE@/CLOAD@ commands) and, indeed, to return a SYNTAX ERROR if an expression is attempted instead of just a variable name.

FNDVAR 1D53 General routine for accessing variables, depending on value of VTYPE (0CAC).

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, examples of which are CSAVE@ and CLOAD@, in which we refer to the array as a whole, without any parentheses commands). 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 for 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.

V. EXAMPLE COMMANDS/FUNCTIONS

To round off our description of the workings of Xtal BASIC 2.2, source listings of some extra commands / functions are given below, and we are sure that all of these will be of considerable use in your own programs, as well as providing a better indication of how to use the routines described in this booklet. In all cases, it is necessary to relocate the routines, and they are all show starting at 0000.

1. SWAP

This command is used to swap the contents of two variables of the same type, i.e, two numeric variables or two string variables. The form of the command is:

SWAP A,B or SWAP A$,B$

Array elements may appear as either one or both of the parameters. This command obviates the need for an extra 'dummy' variable that is normally needed when doing this, i.e, it replaces:

T=A:A=B:B=T or T$=A$:A$=B$:B$=T$

However, there is a more important advantage with strings; only the POINTERS to the strings get swapped, so that no test is required for string space, and the strings themselves are not moved at all. SWAP therefore saves an enormous amount of time when used in a sort routine.

```
                           ORG    0000       ; CHANGE TO WHERE YOU WANT IT!
0000:  CD 53 1D   SWAP     CALL   FNDVAR     ; FIRST VARIABLE
       3A 90 0C            LD     A,(TYPE)
       D5 F5               PUSH   DE,AF      ; SAVE ADDR & TYPE
       CD 4C 15            CALL   TSTCOM
       CD 53 1D            CALL   FNDVAR     ; SECOND VARIABLE
       F1                  POP    AF
       E3                  EX     (SP),HL    ; STACK TEXT PTR, GET
       1F                  RRA               ; ADDR & TYPE OF 1st VAR
       3A 90 0C            LD     A,(TYPE)
       CD 7C 1B            CALL   TYPMCH     ; CHECK THE TWO TYPES
       06 04               LD     B,04
0019:  4E         SWAP1    LD     C,(HL)     ; SWAP THE TWO VARIABLES
       1A                  LD     A,(DE)
       77                  LD     (HL),A
       79                  LD     A,C
       12                  LD     (DE),A
       23                  INC    HL
       13                  INC    DE
       10 F7               DJNZ   SWAP1
```

```
        E1                  POP   HL        ; RESTORE TEXT PTR & RETURN
        C9                  RET
0024:                                       ; SIZE 36 BYTES
```

2. MUL$(

This is a string function, and is a good example because it illustrates how to fetch both a numeric and string expression, and return a string result. MUL$ allows us to create a string which is a multiple of another string,

e.g, PRINT MUL$(4,"Hello") would print HelloHelloHelloHello on the screen.

*This is probably not the most usual mode of its use -- it is most useful for producing repeating patterns, e.g, *************** or +--+--+--+--+--+--+--+--+*

(done by PRINT MUL$ (15,"") and PRINT MUL$(8,"+— ");"+" respectively).*

Of course, either or both parameters may be complete expressions, so this is quite a powerful function to use.

If the resulting string is longer than 255 characters, a STR OVFL ERROR is returned, and a QTY ERROR is returned if the numeric expression is negative or greater than 255. A null string expression is not allowed, although a null result CAN be returned.

Note the use of the integer multiply routine here: this is provided partly in the interests of efficiency, and partly because it is quite a useful routine to have available -- there is no such routine in Xtal BASIC at present.

```
                            ORG   0000
0000:   E1          MUL$(   POP   HL
        23                  INC   HL
        CD 50 22            CALL  I255      ; GET MULTIPLIER
        08                  EX    AF,AF’
        CD 4C 15            CALL  TSTCOM
        CD 8B 1B            CALL  EXPR      ; GET STRING EXPRESSION
        CD 51 15 29         CALL  TSTC ‘)’  ; CHECK FOR CLOSING BRACKET OF
0010:   E5                  PUSH  HL        ; FUNCTION, AND STACK TEXT PTR
        CD 6A 21            CALL  ASC0
        D5                  PUSH  DE        ; GET ADDR OF STRING START IN
        2B                  DEC   HL        ; DE, THEN GET HL BACK TO LEN
        2B                  DEC   HL        ; BYTE OF POINTER.
        2B                  DEC   HL
        6E                  LD    L,(HL)    ; GET LENGTH IN L
        08                  EX    AF, AF’
        67                  LD    H,A       ; AND GET BACK MULTIPLIER IN H
        E5                  PUSH  HL
        CD 3E 00            CALL  IMULT     ; HL=H*L
        24                  INC   H
```

```
0020: 25                DEC   H          ; ENSURE H=0 (NEW LENGTH ,<256)
      1E 0F             LD    E, 0F
      C2 19 13          JP    NZ,ERROR ; 'Str Ovfl Error'
      7D                LD    A,L
      CD AB 1F          CALL  ASNSTR     ; CREATE NEW STRING
      C1 E1             POP   BC,HL
      78                LD    A,B
      B7                OR    A
      28 0B             JR    Z, MUL2    ; IF NEW STRING NULL, DONE
      06 00             LD    B,00
0032: C5 E5     MUL1    PUSH  BC,HL
      ED B0             LDIR             ; KEEP COPYING OLD STRING INTO
      E1 C1             POP   HL,BC      ; NEW ONE
      3D                DEC   A
      20 F7             JR    NZ, MUL1
003B: C3 D9 1F  MUL2    JP    STREND     ; RETURN STRING RESULT (NB,
                                         ; TEXT PTR STILL STACKED
003E: C5 D5     IMULT   PUSH  BC,DE      ; SUBROUTINE TO MULTIPLY H,L
      EB                EX    HL,DE      ; RETURNING RESULT IN HL
      7A                LD    A,D
      21 00 00          LD    HL,0000
      54                LD    D,H
      06 08             LD    B,08
      29        IMUL1   ADD   HL,HL
      87                ADD   A
      30 01             JR    NC, IMUL2
      19                ADD   HL,DE
      10 F9     IMUL2   DJNZ  IMUL1
      D1 C1             POP   DE,BC
      C9                RET
0052:                                    ; SIZE 82 BYTES
```

3. <u>EXTRA TRANSCENDENTAL FUNCTIONS</u>

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 / (EXP(X*2)+1)) 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:

```
                              ORG   0000
0000: E1          TFN         POP   HL         ; ROUTINE TO EVALUATE THE
      E3                      EX    (SP),HL    ; EXPRESSIONS BETWEEN THE
      23                      INC   HL         ; BRACKETS, FOR USER-DEFINED
      CD 77 1B                CALL  EXNMCK     ; FUNCTIONS.
      11 AA 2B                LD    DE, FNEND; WILL EVENTUALLY RETURN
      E3                      EX    HL,(SP)    ; FNEND
      D5                      PUSH  DE
      E9                      JP    (HL)       ; JUMP TO RETURN ADDRESS

000C: CD 00 00    ASN(        CALL  TFN        ; ASN(X)
000F: CD 25 26    ASN1        CALL  STKFPA     ; STACK X
      CD 5F 26                CALL  LDFPR
      CD FB 24                CALL  MULT1      ; X↑2
      21 FD 29                LD    HL,ONE
      CD C4 23                CALL  SUBN       ;1-X↑2
      CD 6E 28                CALL  SQR        ; SQR(1-X↑2)
      C1 D1                   POP   BC,DE      ; UNSTACK X
      3A C2 0C                LD    A, (FPA+3) ; SPECIAL CASE FOR ASN(1)=PI / 2!
      B7                      OR    A
      28 0C                   JR    Z, ACS2
      CD 4D 25                CALL  DIV1       ; X / SQR(1-X↑2)
      C3 10 29                JP    ATN        ; ATN(X / SQR(1-X↑2))
002F: CD 00 00    ACS(        CALL  TFN
      CD 0D 00    ACS1        CALL  ASN1
      21 AE 29    ACS2        LD    HL,HALFPI
      C3 C4 23                JP    SUBN       ; PI / 2 –ASN(X)
003B: CD 00 00    HSN(        CALL  TFN
      CD 71 00    HSN1        CALL  HSN2
      CD CA 23                CALL  SUB1       ; EXP(X)-EXP(-X)
0044: 21 C2 0C    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
004C:  CD 00 00  HCS(      CALL  TFN
       CD 71 00  HCS1      CALL  HSN2
       CD CD 23            CALL  ADD1       ; EXP(X)+EXP(-X)
       18 ED               JR    HALVE
0057:  CD 00 00  HTN       CALL  TFN
       CD 85 00  HTN1      CALL  DOUBLE     ; X*2
       CD BB 28            CALL  EXP        ; EXP(X*2)
       21 FD 29            LD    HL,ONE
       E5                  PUSH HL
       CD BF 23            CALL  ADDN       ; 1+EXP(X*2)
       CD 7D 00            CALL  RECIP      ; 1/(1+EXP(X*2))
       CD 85 00            CALL  DOUBLE     ; 2/(1+EXP(X*2))
       E1                  POP   HL
       CD C4 23            JP    SUBN       ; 1-2/(1+EXP(X*2))
0071:  CD BB 28  HSN2      CALL  EXP        ; GET EXP(X) AND EXP(-X)
       CD 25 26            CALL  STKFPA
       CD 7D 00            CALL  RECIP      ; DO EXP(-X) AS 1 / (EXP(X)
       C1 D1               POP   BC,DE
       C9                  RET
007D:  01 00 81  RECIP     LD    BC,8100    ; CALCULATE RECIPRICAL
       51                  LD    D,C
       59                  LD    E,C        ; FPR=1
       C3 4D 25            JP    DIV1
0085:  21 C2 0C  DOUBLE    LD    HL,FPA+3   ; DOUBLE FPA BY INCREMENTING
       7E                  LD    A,(HL)     ; EXPONENT
       B7                  OR    A
       C8                  RZ               ; NOT OF FPA=01
       34                  INC   (HL)
       C0                  RNZ
       C3 66 24            JP    OVFLO      ; OVERFLOW IF EXPONENT=FF
0090:                                       ; SIZE 144 BYTES
```

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

INDEX OF ROUTINE & SCRATCH-PAD NAMES