TARBELL BASIC

Tarbell
950 DOVLEN PLACE • SUITE B • CARSON, CALIFORNIA 90746

TARBELL BASIC   Table of Contents   October 19, 1978

TARBELL CASSETTE BASIC

Written by Tom Dilatush of REAL TIME MICROSYSTEMS, 2240 Main St.
No. 18, Chula Vista, CA 92011 for Don Tarbell of TARBELL ELECTRONICS,
950 Dovlen Place, Suite B, Carson, CA 90746.

This manual describes TARBELL BASIC in such a way as to be understood by those having previous experience with other BASIC's. It is not intended as a tutorial, as there are several good BASIC texts (see Appendix K). Where reasonable it is upward compatible from ALTAIR* 8800 BASIC release 4.0 8k version. Items which differ significantly from that version are marked with an asterisk. Items enclosed in angles (<item>) are defined in Appendix B. Items enclosed in brackets ([item]) are optional.

Keyboard Control Characters:

7F	rubout	deletes the last character entered (except in EDIT mode)
15	control-U	deletes the current line being entered
03	control-C	stops the program from running or a listing operation
09	control-I	tabs 8 spaces to the right
13	control-S	stops the program or printing temporarily until another key is pushed

Modes of Operation:

Direct Mode:
Most TARBELL BASIC statements may be entered and executed while in command level. This statement may be only one line, but may be any length up to the limits of memory. Statements which would modify allocated memory, such as DIM & LET, are not allowed in direct mode entries. Statement names are not allowed. Multiple statements per line may be seperated by colons (:).

Entry Mode:
This mode is entered by typing "ENTER" or ":", and is used for creating lines of program text from the keyboard. it is also used for inserting lines. See ENTER command.

Edit Mode:
This mode is entered by typing "EDIT" and a line descriptor. It is used for making changes to existing lines without having to retype the whole line. See EDIT command.

Run Mode:
This is the normal, programmed mode. The stored TARBELL BASIC program begins executing when a "RUN" command is entered.

To acquire a better feel for the modes of operation, and for using TARBELL BASIC in general, see the sample run in Appendix F.

* ALTAIR is a trademark/tradename of Pertec Computer Corp.

.* Line Descriptors:

In TARBELL BASIC, line descriptors may not only be line numbers, as in conventional BASIC'S, but also may be any alphanumeric string of characters (including numbers), except spaces or punctuation. If the descriptor is in a statement that is referencing another statement, it may have an offset appended. The offset is indicated by the symbol "+" or "-". This feature may be used to greatly increase readability, and thus increase maintainability of programs.

Line descriptors need only be used on lines which are referred to by another statement, such as a GOTO, GOSUB, GOPROC, RESTORE, etc. Line descriptors are used in a similar fashion to the labels in assembly language. A line descriptor may be a number, just as in normal BASIC's, but need not be in any order. Line descriptors usually are chosen with names that mean something in the program, so that it will be easy for the programmer to remember the name of a particular line or subroutine.

See Appendix G for more examples of the use of line descriptors.

Examples:

```
SORT A=B+6        "SORT" is the descriptor, in this case, the
                  name of the statement.
GOTO SORT+1       "SORT+1" is the descriptor, indicating a
                  transfer to the statement following "SORT". That
                  statement may or may not have a name of it's own
GOSUB SORT - 5    "SORT - 5" is the descriptor, indicating a trans
                  subroutine call to the statement 5 lines before
                  the statement named "SORT".
10 FOR N=1 TO 5           Line numbers can still be used, but need
05 PRINT N,SQR(N)         not be in order, and are not used to
20 NEXT N                 edit in the same manner as other BASIC's
```

Running ALTAIR BASIC programs under TARBELL BASIC:

1. First, since the internal form of ALTAIR BASIC differs drastically from the internal form of TARBELL BASIC, and since this is the form that ALTAIR BASIC programs are saved onto cassette, these programs will not directly load into TARBELL BASIC, even though they may be stored on TARBELL cassette format. There are a few different ways to handle this problem. The simplest, but also the most time-consuming, is to retype the whole program into TARBELL BASIC from the keyboard. The second, which requires the use of a paper-tape punch and reader, is to punch out the programs to paper tape, then read them into TARBELL BASIC from paper tape instead of the keyboard The third, which requires some technical know-how, is to replace the ALTAIR BASIC console output routine with the cassette output routine provided as part of the I/O section of TARBELL BASIC. The programs could then be read directly into TARBELL BASIC by using LOAD.

2. The IF statement in ALTAIR BASIC evaluates a variable as true if it is not zero. In TARBELL BASIC, true must be a minus one.

3. ALTAIR BASIC's CLOAD and CSAVE are replaced by GET, PUT, LOAD, SAVE, BGET, BPUT, BLOAD, and BSAVE in TARBELL BASIC.

4. In TARBELL BASIC, strings must be quoted in a DATA statement.

Commands: (can only be used from command mode)

* BYE
Causes a jump to location 0000 in memory. Example: BYE

CLEAR [<expression>]
Sets all program variables to zero. Sets all strings to nulls. Releases all string and array space. In ALTAIR BASIC, "CLEAR <<expression>>" defines the amount of space to allot for strings. TARBELL BASIC will automatically allot all space not being used for actual programs to strings and arrays. To maintain compatability with ALTAIR BASIC, "CLEAR <expression>" will be processed identically to the CLEAR command.
Examples: CLEAR   CLEAR 2000 (does the same thing)

CONT
Continues program execution after a control/C has been input or after a "STOP" or "END" statement has been executed. Execution resumes at the statement following the break, unless an input from the terminal was interrupted. In the latter case, execution resumes with the interrupted statement. Execution cannot be continued if the program was modified by direct mode entries. Example: CONT

* DELETE <line descriptor> [<line descriptor>]
Eliminates the line(s) indicated from the stored program. If only the first <line descriptor> is present, only that one line is deleted. If both <line descriptor>'s are present, both those lines, and all lines in between are deleted. If there is no such line descriptor, an error message is issued.
Examples: DELETE START+4   DELETE LOOP LOOP+5

* ENTER [<line descriptor>] or :[<line descriptor>]
Causes TARBELL BASIC to go to program entry mode. Any input after this statement is interpreted as program statements. A carriage return delimits each line. The "ENTER" statement is provided to allow statements without names to be entered. If <line descriptor> is ommitted, entry begins after the last statement currently in memory. If <line descriptor> is present, entry begins immediately before the line indicated. Entry mode is terminated by two carriage returns in a row. Multiple statements per line are allowed if separated by a colon. The colon shown in the command format above, however, is a shorthand form of ENTER, the same way that "?" is a shorthand form of PRINT. The first line in a program should always have a label.
If any of the following commands are invoked from entry mode, they will be performed, then command mode will be reentered:
LIST,DELETE,EDIT,RUN.
Examples: ENTER   ENTER ADDC+3   ENTER LOOP   : START+1

LIST [<line descriptor>] [<line descriptor>]
Lists the program starting from the statement corresponding with the first <line descriptor>, until the end is reached, if there is no second <line descriptor>, or until a control/C is entered.
If neither <line descriptor> is present, the whole program is listed.
Examples: LIST   LIST START+10   LIST LOOP END1

NEW
Deletes the program in memory, clears all variables, releases all string and array space. Example: NEW

RUN [<line descriptor>]
If the line descriptor is included, starts execution of the BASIC program at the line specified. If the line descriptor is omitted, execution begins at the first line in memory. In either case, a CLEAR is automatically executed first.
Examples: RUN   RUN COMD   RUN SUBROUTINE+1

* SYMBOL
Types a table of variable names, line descriptors, their types and their locations. Example: SYMBOL

* EDIT <line descriptor>
Causes the interpreter to enter the edit mode on the line described. The line will be printed. Once in edit mode, single letter commands of the form nXs are used, where n is the iteration constant (1 if ommitted), X is the command (detailed below), and s is the search string (if required). Note that if it is desired to use a command without the search string, a carriage-return should be entered immediately after the command letter.

Commands:

U   Prints the line up to the present pointer position.
nD   Deletes n characters starting with the present pointer position.
K   Kills (deletes) a whole line, then enters insert mode.
nSs   Moves the pointer to the n'th occurance of string s.
Is   Performs command S first, then inserts characters at the pointer position until a carriage return is entered.
nCs   Changes the nth occurance of string following C to the string inserted from keyboard.
Q   Returns to command level.
P   Prints the line in the edit line buffer.
A   Appends characters to the end of the line until a carriage return is entered.
nL   Lists n lines starting with the present line, and enters edit mode on the last line listed.
T   Type the rest of the line past the pointer, then the line up to the pointer.
R   Replace the edited line in source with the current edit buffer, and print out new line.   ***** IMPORTANT NOTE *****
Until this command is executed, the source line is not changed.
nF   Move forward n lines and enter edit mode there.
nB   Move backward n lines and enter edit mode there.
Ms   Move to (line descriptor) and enter edit mode there (search string s is used for line descriptor).
nX   Move pointer back n characters.
n<space>   Move pointer forward n characters.

A rubout during command entry will cause it to start all over with the command entry.

Commands are not echoed -- this makes it much easier to see what you are editing--if in doubt about what you typed, hit rubout and start over.

The rubout key functions on insert/change commands as a delete key.

Statements: (can be used from either command or RUN mode)

* APPEND [DISK(<0-3>),]<string expression>[,<string expression>]
Appends a section of program from the ASCII LOAD device (logical unit #2) to the end of the program presently in memory. If the optional line descriptor string is present, execution will begin there. If optional DISK function is not used, last one or 0 is assumed.
Examples: APPEND DISK(3),"FOURIER"   APPEND SUBNAM$

* ASSIGN <logical device number>,<physical device number>
Assigns a physical device to a logical device. Internally, this command sets a bit in the MODES table. See Appendix E for a list of the logical and physical devices.
Examples: ASSIGN 3,1   ASSIGN PR,CRT   ASSIGN LOGICAL,PHYSICAL

* BGET [FILE(<0-63>),]<variable list>
Reads from Binary Input logical device into named variables. If optional file number is not used, the file accessed will be the one used in the last executed FILE function. An OPEN statement with a matching file number must have been used already. If the FILE function has not been used, file 0 is assumed.
Examples: BGET FILE(63),QTY,COST   BGET X,Y(N)

* BLOAD [DISK(<0-3>),]<string expression>[,<string expression>]
Reads a program named <string expression> into memory from the binary input logical device. All characters of string are used. The program must have been saved with BSAVE statement.
If optional DISK function not used, last value or 0 is assumed.
If optional line descriptor string used, starts execution at that location, and can be used to chain programs in this way.
Examples: BLOAD "PAYROLL"   BLOAD NEXTPROG$,"BEGIN"

* BPUT [FILE(<0-63>),]<variable list>
Writes the named variables onto the binary output logical device. The same rules about the FILE function apply as with BGET.
Examples: BPUT FILE(N),QTY   BPUT A,B,C(N)   BPUT X

* BSAVE [DISK(<0-3>),]<string expression>
Writes a program named <string expression> onto the binary output logical device. All characters of the string are used. Programs saved using this command will save and load considerably faster than those saved with the SAVE command.
Examples: BSAVE DISK(0),"PAYROLL"   BSAVE PROGNAME$

CHANNEL
Prints a table of the assignments of physical to logical devices. See Appendix E for the default assignments.

* CLOSE [FILE(<0-63>),]<numeric expression>
Discontinues use (closes) a file which was previously opened under the logical unit <integer expression>.
The optional FILE function need only be used to name the file being closed when it is different than the last used.
Examples: CLOSE FILE(35),3   CLOSE 5   CLOSE BINARY

* DATA <expression list>
Specifies data to be read by a "READ" statement. Expressions are allowed. String constants must be enclosed by quotes.
Example: DATA 1,3,5,7,X+Y,Z^2,"DON"+"TARBELL"

DEF FN<function name>(<variable list>)=<expression>
Defines a function. The function name can be any legal variable name. The variable list is a list of dummy variables representing the variables in the function call. The value of the function is determined by substituting the values of the variables into the expression. Functions may be nested to any depth, and string functions are legal. Any number of variables can be used.

```
Examples:   DEF FNCUBE(X)=X*X*X   DEF FNL3(S$)=LEFT$(S$,3)
            DEF FNRMS(X,Y) = SQR(X^2+Y^2)
```

DIM <array name>(integer)[,<array name>(integer)]...
Allocates space for array variables. Any number of dimensions per array are allowed. The value of each expression gives the maximum subscript permissible. The smallest is 0. If no "DIM" statement is encountered before a reference to an array, an error message is given. Multiple arrays may be dimensioned. Arrays are cleared to zero (numeric) or null (string). Real subscripts are allowed in array references, and if used, the integer part of the subscript will be used for the reference.

```
Examples:  DIM PARTNO$(100),X(3,10),VERYLONGNAMESAREALLOWED(5)
```

* DROP <numeric expression>,<numeric expression>
Drops the assignment of the logical device selected by the first expression to the physical device selected by the second expression.

```
Examples:   DROP 1,1     DROP LOGICAL,PHYSICAL    DROP PRINTD,TTY
```

END
Puts BASIC back into command mode without a message. Normally the last statement in a program, but not required. Example: END

FOR <variable name> = <expr1> TO <expr2> [STEP <expr3>]
Execution sets <variable name> = <expr1>. The program then proceeds until a "NEXT" statement is encountered. <expr3> (or 1 if STEP <expr3> is omitted) is then added to <variable name>. If <expr3> < 0 and <variable name> >= <expr2>, or if <expr3> >0 and <variable name> <= <expr3>, then the program continues with the statement following the "FOR" statement. Otherwise, the program continues after the "NEXT" statement.

```
Examples:  FOR N=1 TO 5          FOR IND=START TO FINISH STEP INCR
```

* GET [FILE(<0-63>),]<variable list>
Read from the ASCII mass storage device into the variables on <variable list>. The data should have been previously saved by a PUT statement. An OPEN statement using the same FILE number should have previously been executed. No FILE number is required if it is the same file as last accessed.

```
Examples:  GET FILE(2),DES$(N)   GET X   GET X,Y$,Z
```

* GOPROC <line descriptor>[,<variable list>]
Calls the statement <line descriptor>, passing the variables on the list. Similar to GOSUB, except it allows the subroutine to have local variables, which are not affected by assignments outside the procedure. Also allows passing variables to the subroutine. The subroutine need not contain a PROCEDURE statement, which is only used to declare local (not global) variables.
Examples: GOPROC SEARCH,STR1$,STR2$,POSITION   GOPROC SORT

GOSUB <line descriptor>[,<variable list>]
A subroutine call is made to the line indicated. That is, execution continues at <line descriptor> until a RETURN statement is encountered, at which time execution is continued at the statement following the GOSUB statement. Variables on the optional <variable list> are passed to the subroutine on the control stack, and may be picked up by a RECEIVE statement, in the same way that they are in a GOPROC operation.
Examples: GOSUB CALC,X,A$ GOSUB 10570   GOSUB GET+1

GOTO <line descriptor>
An unconditional branch is made to the line indicated. That is, execution continues at <line descriptor> instead of the next statement. Examples: GOTO 100   GOTO LOOP+2   GOTO LAST-5

* IF <logical expression> GOTO <line descriptor>
If the value of <logical expression> = -1, then execution continues at the line indicated. Otherwise, execution continues with the line following the IF statement. The logical connectives allowed in <logical expression> are: AND, OR, NOT, >, <, = . See Appendix B for explanation of logical expressions.
Examples: IF X<128 AND X>31 GOTO EXTRA   IF STR$<>"NO" GOTO 100

* IF <logical expression> THEN <statement> [ELSE <statement>]
If the value of <logical expression> = -1 (true), then the first <statement> is executed. Otherwise, it is not. If the ELSE option is used, the second statement is executed if the value of <logical expression> is false. See Appendix B for def. of logical expression.
Examples: IF ANS$="YES" THEN GOSUB INSTR   IF 3*Y=4 THEN PRINT "OK"
           IF ARRAY(N)=0 THEN GOTO LOOP ELSE STOP

INPUT ["<string>"];<variable list>
Assigns entries from the INPUT (logical unit #0) device to the variables on the list. Prompts may be included by enclosing a string in quotes, separated from the variables by semicolons. With no prompt, a "?" is printed. A carriage return must be used to terminate string input. If a carriage return alone is entered, the variable is set to a null for strings or to a zero for numbers. If a number is entered in "e" format, be sure to put a sign or a space after the E, then two digits.
Examples: INPUT A,B$ INPUT "FILENAME";NAM$

[LET] <variable name>=<expression>
Assigns the value of <expression> to <variable name>. The word "LET" is optional. Examples: LET X$=Y$+Z$ LET INDEX=5 X=2+2

* LOAD [DISK(<0-3>),]<string expression>[,<string expression>]
Loads a program from the ASCII LOAD (logical unit #2) device into memory. A NEW command is automatically issued before the program is loaded. If the optional line descriptor string is used, execution begins automatically at that line. This makes it possible to chain ASCII programs the same way that the BLOAD can chain binary programs. If the optional DISK function is not used, the last use of it or 0 is used.
Examples: LOAD DISK(3),"CHESS","START"   LOAD "STARTREK"

NEXT [<variable list]
Terminates a "FOR" loop. Without the optional variable list, it terminates the most recent loop. See the "FOR" statement. After leaving the loop, the index variable remains the last value.
Examples: NEXT   NEXT N   NEXT I,J,K

ON <numeric expression> GOSUB <line descriptor list>
Calls a subroutine at the line in the list corresponding to the value of <numeric expression>. If <numeric expression> equals zero, or if it's greater than the number of line descriptors, execution continues with the next statement. If it's less than zero, an error results.
Examples: ON I GOSUB 20,5,100,10   ON 2*I GOSUB TEST+2,SUBR5

ON <numeric expression> GOTO <line descriptor list>
Transfers execution (branches) to the line in the list corresponding to the value of INT(<numeric expression>). If <numeric expression>=0 or if it's greater than the number of line descriptors, execution continues with the next statement. If it's <0 an error results.
Examples: ON N GOTO 10,20,30,40   ON N-2 GOTO FIRST,CALC,LAST

OPEN [<special function>,]<numeric expression>[,<string expression>]
Makes a file available for use through the logical device specified by the numeric expression. Normally, this would be logical devices 2, 3, 4, or 5. BLOAD, BSAVE, LOAD, SAVE, and APPEND do not require an OPEN or a CLOSE statement, only BPUT, BGET, PUT, and GET. See DISK, FILE, RECORD, and TYPE special functions. The optional <string expression> need only be used to name a file when the file name is different than the last file accessed.
Examples: OPEN 3   OPEN 3,"DATA"   OPEN FILE(7),DISK(1),3,"DATA"

OUT <numeric expression #1>,<numeric expression #2>
Sends byte resulting from the first expression to the port determined by the second expression.
Examples: OUT 1,7   OUT PORT,DATA   OUT X-5,Z+2

POKE <numeric expression #1>,<numeric expression #2>
Stores byte from second expression into memory location of the first.
Examples: POKE 4096,255   POKE ADDRESS,BYTE   POKE A+256,48+N

PRINT <expression list> or ?<expression list>
Prints the value of each expression on the expression list onto the PRINT device (logical unit #1). Spacing between elements is defined by punctuation. A comma starts the following element at the next 14 column field. A semicolon starts the following element immediately after the preceeding element. If the last character of the list is a comma or a semicolon, no carriage return will be printed at the end of the statement. Otherwise, a carriage return will be printed at the end of the statement.
Examples: PRINT "X=",X   PRINT 33*X,A$,CHR$(7)   ?FRE(0)

* PROCEDURE <variable list>
Used to declare local variables. The variables on the list can be used without disturbing their original values in the main program. The original value of each variable will be restored by the next RETURN statement. (See GOPROC, RECEIVE, RETURN)
Examples: PROCEDURE ANS$,X   PROCEDURE A,B,RESULT

* PUT [FILE(<0-63>),]<variable list>
Write from variables on <variable list> to the ASCII SAVE device. The FILE used should have been previously OPEN'd.
Examples: PUT FILE(7),PART$,COST   PUT X,Y

READ <variable list>
Assigns the value of each expression of a "DATA" statement to a variable on the variable list, starting with the first element of the first "DATA" statement. Expressions of the "DATA" statement(s) are evaluated when the first element of the "DATA" statement is read. (See DATA and RESTORE statements.)
Examples: READ X,Y   READ X,Y,Z$ READ TABLE(N)

* RECEIVE <variable list>
Transfers values of variables from "GOPROC" or "RETURN" statement to <variable list>. The variables on the list are filled in the same order that the variables appear on the GOPROC or RETURN statement. Examples: RECEIVE X   RECEIVE RES$,ANSWER

REM[anything]
Allows insertion of remarks in the program text. The remarks must follow the REM statement on the same line.
Examples: REM   THIS PROGRAM CALCULATES TRIG TABLES

RESTORE [<line identifier>]
Sets the READ DATA pointer to the first data statement, or, optionally, to the statement <line identifier>. This allows reading seperate tables or seperate portions of a table without having to read through all DATA statements.
Examples: RESTORE   RESTORE TABLE2   RESTORE START+5

RETURN [<variable list>]
Causes execution to continue at the statement following the last GOSUB or GOPROC statement executed. If the optional variable list is included, passes the values of the variables on the list to the variables on the list of a "RECEIVE" statement.
Examples: RETURN   RETURN N   RETURN X$,ANSWER,RESULT$

* SAVE [DISK(<0-3>),]<string expression>
Writes the BASIC program from memory onto the ASCII save device.
Examples: SAVE DISK(2),"STARTREK"   SAVE PROGNAME$

STOP
Terminates execution of the BASIC program, and returns back to the command mode with the message: STOP IN <line descriptor> .
Example: STOP

WAIT <num. expr. #1>,<num. expr. #2>[,<num. expr. #3>]
An input from port <expr1> is performed. The byte received is XOR'd with <expr3> if included, then AND'ed with <expr2>. The above operation is repeated until a non-zero result is obtained, upon which the next statement is executed.
Example: WAIT 0,1   WAIT PORT,MASK,INVERT

INTRINSIC (BUILT-IN) FUNCTIONS

A FUNCTION, built-in or otherwise, can be used anywhere that an expression can be used. It can be a part of an expression, and it can have an expression as it's argument. It returns a single value, which is defined by the descriptions below. Some functions return string values, and some return numeric ones.

ABS(<numeric expression>)
Returns the absolute value of <numeric expression>. In other words, the expression is evaluated; if the result is minus, the minus sign is removed to make it positive.
Examples: ABS(X-5*SIN(Y)) ABS(B^2-4*A*C)

ASC(<string expression>)
Returns the ASCII code of the first character of <string expression>. (i.e. the number which corresponds with the ASCII character)
Examples: ASC("A") ASC(ANS$) ASC(A$(N))

ATN(<numeric expression>)
Returns the arctangent of <numeric expression>, which is in radians.
Examples: ATN(RADIANS) ATN(DEGREES*PI/180) ATN(.053)

* CALL(<numeric expression>,<numeric expression>)
Calls a machine language subroutine at the address indicated by the first <numeric expression>, with the value of the second <numeric expression> in registers D&E. The CALL function evaluates to the number which is returned in registers D&E. The returned value in D&E is in the range -32768 to +32767.
Example: PRINT CALL(PLOT,X)

CHR$(<numeric expression>)
Returns a single character string whose ASCII code is <numeric expression>.
Examples: CHR$(7) CHR$(48+NUM) CHR$(CONTROL)

COS(<numeric expression>)
Returns the cosine of the angle <numeric expression>, which is in radians. Example: COS(DEG*3.14159/180)

EOF(<numeric expression>)
Returns false (0) if an end-of-file has not been encountered, or true (-1) if an end-of-file has been encountered, during the last read operation from a file through the logical unit specified by <numeric expression>.
Examples: IF EOF(2) THEN GOTO QUIT LET CASFLG=EOF(4)

EXP(<numeric expression>)
Returns the constant e (2.718282) to the <numeric expression> power.
Examples: EXP(1) EXP(0) EXP(X+Y*2)

FRE(<expression>)
Returns the amount of free (unused) space in memory. Because the null string ("") takes less space in memory, this form will return a slightly larger number than a numeric argument.
Examples: FRE("") FRE(0) FRE(1)

* HEX(<hexadecimal string>)
Returns the decimal equivalent of the <hexadecimal string>.
Examples: LET ADDR=HEX(HADDR$) FOR N=0 TO HEX("A")

* HEX$(<numeric expression>)
Returns the hexadecimal string representation of the decimal value of <numeric expression> with no leading zeroes.
Examples: HEX$(ADR+OFFSET)   HEX$(N)   HEX$(99)

INP(<numeric expression>)
Performs a read from the machine input port <numeric expression>. Returns the value of the machine input port <numeric expression>.

INT(<numeric expression>)
Returns the largest integer which is less than or equal to <numeric expression>. Examples: INT(-3.5) INT(0) INT(3.14159)

LEFT$(<string expression>,<numeric expression>)
Returns the leftmost <numeric expression> characters of <string expression>. Examples: LEFT$(ANS$,3)   LEFT$(A$+B$,N-M)

LEN(<string expression>)
Returns the length of <string expression>.
Examples: LEN(A$+B$)   LEN(ALPHABET$)   LEN("ABC"+STRING$)

* LOC(<variable name>)
Returns the decimal address of the location in memory of the variable's value. Useful for passing addresses to routines which are accessed via the CALL function.
Examples: LOC(ARRAY$(N)) LOC(N) LOC(A$)

LOG(<numeric expression>)
Returns the natural logarithm (base e) of <numeric expression>.
Examples: LOG(1)   LOG(X^2 +Y/5)   LOG(.5*SIN(X+Y))

* MATCH(<string expression>,<string expression>,numeric expression>)
Returns the position of the first occurence of the first string expression in the second string expression, starting with the character position indicated by the numeric expression. A zero will be returned if no match is found. The following pattern matching features are implemented:

1) A pound sign(#) will match any digit (0-9).
2) An exclamation mark (!) will match any upper or lower case letter.
3) A question mark (?) will match any character.

Examples: MATCH("DEF","ABCDEFGHIJ",1)   (returns 4)
MATCH(PATTERN$,OBJECT$,START)

MID$(<string expression>,<numeric expression>[,<numeric expression>])
Without the optional second numeric expression, returns rightmost characters of <string expression> starting with the first <numeric expression>. With the second numeric expression, returns a string whose length is determined by the second numeric expression, starting with the character of <string expression> whose position is determined by the first numeric expression.
Examples: MID$(A$,5)   MID$(STRING$,POSITION,LENGTH)

OCT(<string expression>)
Returns the decimal equivalent of the string expression, which should be a valid octal number.
Examples: OCT("377") OCT(OCTADR$)

* OCT$(<numeric expression>)
Returns a string which represents the octal value of the numeric expression. Examples: OCT$(10) OCT$(X+Y) OCT$(DECIMAL)

PEEK(<numeric expression>)
Returns the value of the byte in memory address <numeric expression>.
Examples: PEEK(0) PEEK(1024+OFFSET) PEEK(DECIMALADDRESS)

POS(<expression>)
Returns the current position of the PRINT device. If used within a PRINT statement, zero will always be returned, since the function is evaluated before the line is printed. This function is normally used after a PRINT statement ending with a semicolon.
Examples: POS(0) POS("") POS(anything)

RIGHT$(<string expression>,<numeric expression>)
Returns the rightmost <numeric expression> characters of <string expression>. Examples: RIGHT$(SENT$,1) RIGHT$(S$,NUM)

RND(<numeric expression>)
If <numeric expression> is less than zero, starts a new sequence of random numbers. If it's equal to zero, returns the same number as the last RND returned. If it's greater than zero, returns the next random number in the sequence.
Examples: RND(-1) RND(0) RND(1) RND(X)

SGN(<numeric expression>)
If <numeric expression> is greater than zero, returns 1.
If it's equal to zero, returns 0; if less than zero, returns -1.
Examples: SGN(-2.57) SGN(0) SGN(353.2) SGN(X^3+Z)

SIN(<numeric expression>)
Returns the sine of angle <numeric expression>, which is in radians.
Examples: SIN(DEG*PI/180) SIN(.256) SIN(X/Y)

SPACE$(<numeric expression>)
Returns a string of <numeric expression> spaces.
Examples: SPACES$(BUFFERSIZE) SPACES$(4+LEN(LINE$))

SPC(<numeric expression>)
Prints <numeric expression> spaces on the PRINT device.
Examples: SPC(20) SPC(N/3) SPC(INT(X*2))

SQR(<numeric expression>)
Returns the square root of <numeric expression>. An error message will result if <numeric expression> evaluates to a negative number.
Examples: SQR(B*B-4*A*C) SQR(2) SQR(X)

STR$(<numeric expression>)
Returns the string representation of <numeric expression>, without leading or trailing spaces.
Examples: STR$(3052.67) STR$(NUMBER) STR$(X*Y/Z)

TAB(<numeric expression>)

Spaces to column <numeric expression> on the PRINT device. If tabbed column is less than the present position, the next output from PRINT will go on the next line in the correct position.

Examples: TAB(20)   TAB(30)   TAB(N*2)   TAB(POSITION)

TAN(<numeric expression>)

Returns the tangent of angle <numeric expression>, which is in radians.

Examples: TAN(DEGREES*3.14/180)   TAN(.25)   TAN(X^2/Y)

USR(<expression>)

Calls a user (machine language) subroutine at the address in location USER. The address of location USER is in the 11th and 12th bytes after the start of BASIC (see appendix C). The <numeric expression> is evaluated and placed in registers D&E. The USR function returns with the value that is returned in registers D&E. For example, if the machine language subroutine decremented D&E by 5, the value of the USR function would be 5 less than it's argument. Of course, anything may be done in a USR subroutine, but it is recommended that all registers that are changed besides D&E should be saved and restored on a stack.

Example: USR(0)   USR(N)   USR(N*M)   USR(ARG)

VAL(<string expression>)

Returns the numerical value of the string <string expression>. Leading spaces are ignored. If the string expression is not a valid number, zero is returned.

Examples: VAL(FIELD4$)   VAL(COST$)   VAL(A$)   VAL("3.14")

Special Functions:

The purpose of the special functions is to set values into memory locations, so that these values can be used by external subroutines. Invocation of any of these functions does not cause control to leave BASIC, as it does with the USR and CALL functions. Thus, the functions may be used without having any routine that actually uses their results. One thing peculiar about these functions is that they have no value. That is, no value is returned when they are used. They may be used nearly anywhere, except that they must be separated from all other elements of an expression by commas, and if they are used in an assignment statement (LET or FOR), they must be the last elements of the expression. Normally, they are used in disk input/output statements, such as LOAD, SAVE, OPEN, CLOSE, GET, PUT, etc. to pass useful parameters. Another thing common to all these functions is that once the function is used, the associated parameter(s) remain that way until it is used again. That way, the functions only need to be used when it a change is required from the current values. The parameters are all initialized to zero by the I/O section when BASIC is first entered.

DISK(<numeric expression between 0 and 3>)
This function is normally used to specify the number of the disk drive which you wish to select. If the function is not used, the last drive selected will be used. The value of <numeric expression> is placed in the location DISK. See appendix C for the address of the pointer. The I/O section normally initializes the location of DISK to zero (disk A).

```
Examples:  OPEN DISK(1),FILE(8),3,"DATATEST"
           LOAD DISK(0),"STARTREK"
```

DO(<expression>,<expression>)
Each expression can be either a numeric expression or a string expression. If numeric, the 2-byte number is passed. If string, the string's address is passed. The first expression is passed to location DO. The second expression is passed into location DOPARA. The addresses of these locations are in the address table at the beginning of BASIC. See appendix C. The main idea of the DO function is to use the first expression to decide on the type of function, and to use the second expression to pass the argument. The present I/O section does not support any particular DO operation.

FILE(<numeric expression from 0 to 63>)
This function is used to specify the number of the file being used. It is not used when loading or saving programs, only data. The initial file number is zero. Since the file number stays the same until the FILE function is used, it is not necessary to use this function until it is required to access a data file different than the last one accessed. The file number can be any arbitrary number from 0 to 63. It should be used in the OPEN statement for the file, and any time a different file number is required.

```
Examples:  OPEN FILE(34),DISK(1),3,"DATAFILE"
           PUT FILE(34),TESTDATA$
           GET FILE(34),TESTDATA$
           CLOSE FILE(34),3
```

* TYPE(<numeric expression>)
The value of the expression is placed in location TYPE, described in appendix C. This function is normally used to specify the type of file which is being accessed. The following conventions will be used by the I/O sections provided by Tarbell Electronics: 0 for sequential, 1 for random. TYPE is initialized to 0 by the I/O section.

```
Examples:       SEQ=0:RAN=1
        OPEN DISK(1),FILE(7),TYPE(RAN),RECORD(80),2,"RANFILE"
        GET FILE(7),RECORD(N),X,Y$,Z
```

RECORD(<numeric expression>)
The value of the expression is placed in location RECORD, described in appendix C. This function is normally used to specify the record number of a random file, as part of a GET or PUT operation, or to specify the number of bytes per record, as part of an OPEN operation.
See examples above.

NOTE: As of October 23, 1978, the TYPE and RECORD functions had not yet been implemented in the input/output section.

OPERATORS

Arithmetic Operators (in order of precedence)

1.	expressions enclosed in parenthesis	
2.	^	exponentiation
3.	-	negation
4.	* /	multiplication and division
5.	+ -	addition and subtraction
6.	relational operators (same for all)	
	=	equal
	<>	not equal
	<	less than
	>	greater than
	<=	less than or equal
	>=	greater than or equal
7.	NOT	inversion of all bits
8.	AND	logical multiplication of each bit
9.	OR	logical addition of each bit

String Operators

1.	+	Concatenates (hooks together) two strings end-to-end.

2. Comparison Operators

=	equals
>	greater than
<	less than
<=	less than or equal
>=	greater than or equal
<>	not equal

Comparison is made by comparing the ASCII codes of each character of each string, starting with the first character of each string. The comparison continues with each set of corresponding characters until there is a mismatch, at which time the string with the code having the higher ASCII value is declared the greater. If there is no mismatch, the strings are of equal value. If one string is shorter than the other, the longer string is considered greater.

Logical Operators:

OR, AND, and NOT are used as logical operators in IF statements. OR and AND operate on the logical expressions between which they are placed, while NOT operates on the logical expression following it. Remember that the value of a logical expression must be either -1 (true) or 0 (false).

OR
When OR is placed between two logical expressions, the total expression is true if either or both of the two logical expressions are true.

AND
When AND is placed between two logical expressions, the total expression is true if and only if both of the two logical expressions are true.

NOT
When NOT is placed before a logical expression, the total expression is true if the logical expression is false, and the total expression is false if the logical expression is true.

Below are truth tables for the three logical operators, where T stands for TRUE (-1), F stands for FALSE (0), and A and B are logical expressions:

A	B	A OR B	A AND B	NOT A
F	F	F	F	T
F	T	T	F	T
T	F	T	F	F
T	T	T	T	F

Examples:

0 AND 1 equals 0
1 AND 1 equals 1
2 AND 1 equals 0
2 AND 3 equals 2

0 OR 1 equals 1
1 OR 1 equals 1
2 OR 1 equals 3
2 OR 3 equals 3

NOT 0 equals -1
NOT -1 equals 0
NOT 1 equals -2

ERROR CODE EXPLANATIONS

The system of programming error detection and reporting in TARBELL BASIC is a compromise between the need for clear error reporting, and the memory required for error detection and messages. Some systems use error code numbers, or 1 or 2 code letters. These usually have to be looked up in the reference manual, so they waste time. Some use long english explanations, which are nice, but take up a lot of memory space. TARBELL BASIC uses abbreviated messages, which are hopefully easy to remember after they're looked up the first time.

No.	Mnemonic	Description
1	OVRFLW	Arithmetic Overflow (too large a number).
2	UNDRFLW	Arithmetic Underflow (too small a number).
3	/0	A division by zero was attempted.
4	EX>>	Exponent was too large (EXP function).
5	BIN CON >>	Number too large to convert to binary.
6	-LOG	Attempted to take log of a minus number.
7	LINE DES	Illegal line descriptor.
8	COMM	Illegal command.
9	VRBL AS STATE	Variable name used as statement name.
10	SYNTAX	The statement was not properly formed.
11	VRBL NM	Illegal variable name.
12	>>)	Too many right parenthesis.
13	>>(	Too many left parenthesis.
14	2 OPERS	Two operators in a row.
15	2 OPANDS	Two operands in a row.
16	ILGL FUNC	Illegal user defined function.
17	STATE AS VRBL	Statement name used as variable.
18	NEW SYMB	New symbol when in command mode.
19	NO TO	No "TO" in "FOR" statement.
22	CAN'T CONT	Can't continue 'cause program was modified.
23	READ	An error was detected on a tape or disk read.
24	STRING	Illegal string usage.
25	COMMA	Illegal comma or semicolon.
26	OPRND	Illegal Operand.
27	<*mem*>	Out of memory.
28	UNDIM	Undimensioned array referenced.
29	SUBSCPT>>	An array subscript was too large.
30	SUBSCPT OVFLW	Subscript overflow.
31	ASSIGN	An assignment to a non-variable (4=4).
32	STR AS NUM	A string is used where a number is needed.
33	NUM AS STR	A number is used where a string is needed.
34	CNTRL STCK	The control stack is where the following items are placed: return location for subroutines & procedures, arguments for subroutines & procedures, index variables for FOR-NEXT loops.
35	ON GOTO	ON...GOTO, GOSUB index out of limits.
36	<<DATA	Out of Data.
37	RCV DATA	Receive data error.
39	-SQR	The square root of a minus number is illegal.
40	LOGICAL	A true (-1) or false (0) was expected.

The message BASIC IS CRASHED indicates that the BASIC interpreter has be written into, thus making the interpreter unreliable.

DEFINITIONS OF TERMS USED IN THIS MANUAL

<numeral>
Any of the following: 0 1 2 3 4 5 6 7 8 9

<upper case letter>
Any of the following: ABCDEFGHIJKLMNOPQRSTUVWXYZ

<lower case letter>
Any of the following: abcdefghijklmnopqrstuvwxyz

<letter>
Any <upper case letter> or <lower case letter>.

<alphanumeric character>
A <numeral>, a <letter>, or a dollar sign ($).

<special character>
Any of the following: !"#$%&'()=-^\{}[]+;*:<>,.?/
or a space.

<control character>
Control characters are bytes that do not normally print a visible character on a terminal, but instead, may perform some particular function in the terminal or terminal driver. Examples of common control characters are listed below:

00	null	03	quit	07	bell	08	backspace
09	horz tab	0A	line feed	0B	vert tab	0C	form feed
0D	carriage-return			13	stop temporarily		
15	cancel line			1A	end-of-file		
1B	escape			7F	rub out		

<character>
A <alphanumeric character>, <special character>, or a <control character>.

<numeric constant>
A number, represented by a series of numerals, preceeded by an optional plus (+) or minus (-) sign, including an optional decimal point (.), and ending with an optional "E", followed by a +, =, or a space, followed by a power of ten. Three characters must follow the "E". A space may be used instead of a plus sign (+). The range of a floating point number (one with a decimal point) is from 9.9999999E+99 to 9.9999999E-99, plus and minus. The range of a integer number is from 0 to 9999999999, plus and minus. Expressions evaluate to integers if and only if every element of the expression evaluates to an integer.

<string constant>

A string constant is a sequence of any characters, represented literally, either <alphanumeric character>s or <special character>s enclosed in quotes ("), or CHR$ functions with a constant argument. There is no limit to the length of a string constant. Quotes may be represented by a double quote (""), or by CHR$(34). Control characters may be represented by using the CHR$ function.

Examples: "ABCDEFGHIJKLMNOPQRSTUVWXYZ 0123456789 !" "#$%&'()"

<constant>

A value which is named as such explicity in the program. May be either a <string constant> or a <numeric constant>.

Examples: "DON TARBELL" 3.14159 2 "ABCDEFG"

<expression>

A sequence of constants and/or variables, separated by operators according to certain rules (see pages 14&15) and optionally grouped by parenthesis.

Examples: 1 X "ABC"+REST$ 3*(X/Y) SQR(B^2-4*A*C)

<numeric expression>

An expression which evaluates to a number.

Examples: 1+1 2*(3+5) N/2 4*LEN(STRING$) SIN(X)

<string expression>

An expression which evaluates to a string.

Examples: LEFT$("ABCDEFG",3) "123"+"ABC"+A$ CHR$(N+64)

<expression list>

A sequence of expressions normally separated by commas or semicolons.

Examples: "THE COST IS ";COST;" DOLLARS.",TOTAL,X*5/Y X,Y X

<variable>

An entity which can assume different values, either string or numeric.

<variable name>

A sequence of <alphanumeric character>s, beginning with a letter, which is used to identify a particular variable. If a variable name ends with a dollar sign ($), it is forced to a string.

<variable list>
A sequence of variable names, seperated by commas or semicolons.

<logical constant>
A constant which has the value of either -1 (true) or 0 (false). Notice that in some systems, any integer other than zero is considered true. This can produce an ambiguity, however, in that a NOT TRUE operation could produce a TRUE value.

<logical operator>
AND, OR, and NOT are the logical operators. When the AND operator is between two logical constants, the combination is true if both values are true. When the OR operator is between two logical expressions, the combination is true if either value is true. When the NOT operator is before a logical expression, the combination produces a TRUE value if the expression were FALSE, and a FALSE value if the expression were TRUE. The logical operators all perform as if they were operating on each bit of a 16-bit binary number, with all bits operated on in parallel.

```
Examples:  2 AND 3 produces 3
           1 OR 4  produces 5
           NOT 0 produces -1
```

<logical variable>
A variable whose value is either -1 (true) or 0 (false).

<logical expression>
An expression which evaluates to either a -1 (true) or 0 (false).

Examples:

```
        LET TRUE=-1:LET FALSE=0
        TRUE AND TRUE  produces TRUE
        TRUE AND FALSE produces FALSE
        FALSE AND TRUE produces FALSE
        FALSE AND FALSE produces FALSE
        TRUE OR TRUE produces TRUE
        TRUE OR FALSE produces TRUE
        FALSE OR TRUE produces TRUE
        FALSE OR FALSE produces FALSE
        NOT TRUE produces FALSE
        NOT FALSE produces TRUE
```

<line descriptor>
A sequence of <alphanumeric character>s, which starts with the first character position (left-hand margin) in a TARBELL BASIC statement line, and which is terminated by either a space or a tab (ctl-I), and which is not one of the reservea words in Appendix H. If the descriptor is in a statement referencing another statement, a + or - offset may be included.

<line descriptor list>
A sequence of line descriptors, seperated by commas.

MEMORY USAGE - version 7

Address Range (hexadecimal)	Description
0000 - 00FF	Unused, available space for your stuff, except in CP/M systems, where it is used by CP/M.
0100 - 04FF	Standard Input/Output Routines (Listing Included). There may be extra room here. See the listing.
0500 - 0502	A jump instruction into TARBELL BASIC.
0503 - 0562	A table of addresses, each of which point to a useful table, subroutine, or parameter in BASIC. These addresses may be used from outside the main body of the interpreter. Examples of this are shown in the Tarbell BASIC I/O system listing. See page C-2 for a list of these addresses.
0563 - 53C5	The TARBELL BASIC interpreter, which may be in ROM. Note that these addresses may change with versions.
53C6 - XXXX	Interpreter Workspace, must be in RAM. (fixed length) This can be seen on the last page of the source listing as a series of DS's.
XXXX - XXXX	Program Source, in internal form. Fixed at RUN time. Defined by pointers FSRC and ESRC.
XXXX - XXXX	Variables and Array Pointers. Fixed at RUN time.
XXXX - XXXX	FOR/NEXT and local variable stack. Dynamic.
XXXX - XXXX	Input Line Space. Dynamic.
XXXX - XXXX	Array and String Space. Dynamic.
XXXX - XXXX	Symbol Directory. Fixed at RUN time.
XXXX - XXXX	Symbol Table. Fixed at RUN time. End of Specified Memory.

Allocation Notes:
Before runtime, will consist only of moving the symbol directory as the symbol table grows. At runtime, variable and array pointers fished out of symbol directory and space assigned. As local variables are encountered, they are assigned on the stack. Arrays and strings are assigned by sequential assignment-random release- clean up garbage when full.

TABLE OF ADDRESSES - version 7

Address Range (hexadecimal)	Description
0503 - 0504	CHANL - Contains the address of the Channel Table.
0505 - 0506	TRMNL - Contains the address of the Terminal Table.
0507 - 0508	SSSS - Defines the end of useable memory. If zero, causes BASIC to use all available memory.
0509 - 050A	CNVRA - Defines the number of digits that will be printed in normal (as opposed to scientific) notation.
050B - 050C	USER - Contains the address of a location which contains the address of a user routine accessed by the USR function.
050D - 050E	MODES - Contains the address of the MODES Table.
050F - 0510	FSRC - Address of pointer to start of source.
0511 - 0512	ESRC - Address of pointer to end of source.
0513 - 0514	ERROR - Pointer to error routine.
0515 - 0516	TSCN - Points to token just scanned.
0517 - 0518	NSCN - Points to token to be scanned next.
0519 - 051A	CHCK - Points to checksum routine.
051B - 051C	INFL - Integer to Floating, (HL) to (DE).
051D - 051E	FLIN - Floating to Integer, (HL) to (DE).
051F - 0520	STNM - String at (HL) to number at (DE).
0521 - 0522	NMST - Number at (HL) to string at (DE).
0523 - 0524	CMPR - Zero and carry set as for (HL)-(DE).
0525 - 0526	SINE - Sine(HL) to (DE).
0527 - 0528	SICO - Cosine(HL) to (DE).
0529 - 052A	TANG - tangent(HL) to (DE).
052B - 052C	ATAN - Arctangent(HL) to (DE).
052D - 052E	BCDB - Number at (HL) to binary in HL.
052F - 0530	BBCD - Binary number in HL to number at (DE).
0531 - 0532	ETOX - E to the (HL) power to (DE).
0533 - 0534	LOGX - Log base E (HL) to (DE).
0535 - 0536	SQUR - (HL) to 1/2 to (DE).
0537 - 0538	PWRS - (HL) to the (DE) power to (BC).
0539 - 053A	ADDER - (HL)+(DE) to (BC)
053B - 053C	SUBER - (HL)-(DE) to (BC)
053D - 053E	MULER - (HL)*(DE) to (BC)
053F - 0540	DIVER - (HL)/(DE) to (BC)
0541 - 0542	KILL - Kill allocated dynamic RAM block.
0543 - 0544	AMBL - Allocate a dynamic RAM block.
0545 - 0546	EOF - End-of-file flag byte address.
0547 - 0548	RECORD - Address of random file record number.
0549 - 054A	FILE - Address of (file # or adr of name).
054B - 054C	TYPE - Address of file type number.
054D - 054E	NAME - Address of address of file name.
054F - 0550	CMP16 - Address of 16-bit compare routine.
0551 - 0552	SUB16 - Address of 16-bit subtract routine.
0553 - 0554	MOVE - Address of block move routine.
0555 - 0556	MULT - 8 by 8 multiply, DE=D*E.
0557 - 0558	ZERO - Zeroes A bytes starting at HL.
0559 - 055A	DIV - L=HL/E. unrounded, h=remainder.
055B - 055C	DO - Address of first parameter of DO function.
055D - 055E	DOPARA - Adr of 2nd parameter of DO function.
055F - 0560	DISK - Address of disk number.
0561 - 0562	KIND - Adr of Kind (of transfer) byte.

INTERNAL FORMATS

Symbol Table Format: ASCII, last character has bit 7 set=1.

Symbol Directory Format:
Bytes 0&1 are pointer to location (0 if inactive dummy).
Byte 2 bits have meanings as follows:

```
  0-statement name     1-variable      2-function
  3-channel name       4-array         5-unused
  6-has been stored to 7-trace on
```

Numeric Array Format:
bytes n,n+1 = back pointer
bytes 2+n to n+x+1 = number of elements per dimension
where n=(table pointer), and x=number of dimensions
bytes 2+n+x to 1+n+x+(6*E) = number storage
where E=total number of elements
To locate an element within an array, location=base+offset,
where base=2+n+x, and offset computed by:

```
        N=1
        OFFSET=S(N)
LOOP    N=N+1
        OFFSET=(OFFSET)(D(N))+S(N)
        IF N<>LAST DIMENSION GOTO LOOP
        OFFSET=OFFSET*6
        END
```

Where S is subscript, D cements in a dimension, () mean contents of.
Example: Array dimensioned 3,4,5; Get element 2,1,4.

N	Offset
1	2
2	2*4+1=9
3	9*5+4=49
3	49*6=294

String Locator:
  bytes n,n+1=back pointer.
  bytes n+2 to n+1+m=number of elements per dimension.
  bytes m+n+2 to n+1+(2E)+m=string pointers.
  Where m=number of dimensions, and E=number of elements.
  2 Bytes per pointer, same organization as elements of numeric arrays. If (pointer)=0, string is (null). Otherwise, points to first address of (string).

String Format:
  bytes n,n+1=back pointer.
  n+2 to n+1+m=ASCII data.
  Where m=number of characters. Last character as bit 7=1.
  All other characters have bit 7=0.

INTERNAL FORMATS (continued)

String Array Pointer Format:

- Byte 0: bit 0-not used, bit 1=0, bit 2=0, bit 3=1, bit 4=0, bit 5=0, bit 6=0 if not array.=1 if array, bit 7 not used.
- Byte 1 = number of dimensions.
- Bytes 2&3 is a pointer to string locator or string.
- Bytes 4&5 are not used.

String variables are treated internally as single dimension arrays.

Numeric Array Pointer Format:

- Byte 0: bit 0=0 if integer, 1 if floating point.
  bit 1=0, bit 2=1, bit 3=0, bit 4=0, bit 5=0,
  bits 6&7 are unused.
- Byte 1 = number of dimensions.
- Bytes 2&3 is a pointer to table location.
- Bytes 4&5 are not used.

Numeric Format (constants and variables)

- Byte 0: bit 0=0 if integer, 1 if floating point.
  bit 1=1, bit 2=0, bit 3=0, bit 4=0, bit 5=0,
  bit 6 is sign of exponent, bit 7 sign of mantissa.
- Byte 1 = BCD exponent if floating point, MSD, MSD-1 if integer.
- Byte 2 = LSD+7, LSD+6
- Byte 3 = LSD+5, LSD+4
- Byte 4 = LSD+3, LSD+2
- Byte 5 = LSD+1, LSD

INPUT/OUTPUT

The input and output facilities of TARBELL BASIC were designed to create a new standard of flexibility. Essentially, commands are provided to allow any output statement to transfer data to most output devices, and any input statement to transfer data from most input devices. In order to do this, devices are grouped into logical devices and physical devices. Logical devices are those that are activated by the input and output commands, and are listed in the table on the left. Physical devices are actual pieces of hardware, such as a CRT, printer, cassette, and disk. There is a table, called the MODES table, which remembers the assignment of physical devices to logical devices. The MODES table has ten bytes, numbered from 0 to 9. Each byte represents a corresponding I/O device driver in the I/O section. Each bit in each byte corresponds to one of the eight possible logical devices, numbered from 0 to 7. The table below shows the logical and physical devices, and their default assignments for TARBELL CASSETTE BASIC:

Logical Device	Number	Physical Device	Number
INPUT	0	Console Keyboard	0
PRINT	1	Console Printer	1
LOAD	2	Cassette Input	2
SAVE	3	Cassette Output	3
BGET & BLOAD	4	Cassette Input	2
BPUT & BSAVE	5	Cassette Output	3
Spare	6	Spare Input/Output	4
Spare	7	Listing Device Output	5
		Reader Input	6
		Punch Output	7
		Disk Input	8
		Disk Output	9

The current assignments may be viewed by entering the CHANNEL statement. Every place an X occurs, an assignment exists between the physical device to the left and the logical device above. The ASSIGN and DROP statements can be used to set and reset bits in the table, respectively.

To get an idea of how this works, just type DROP 1,1. This will drop the console output device as the PRINT device. Don't worry! Nothing's wrong. Your keyboard is still feeding commands to the console INPUT device, you just can't see the echo. Now simply say ASSIGN 1,1 and you'll be back in business.

Note that the I/O section (see seperate listing) creates the default assignments by transfering ten bytes to the MODES table. If you wish to change the default assignments, just change these ten bytes (at IMODES).

INPUT/OUTPUT (continued)

Mass Storage (cassette or disk) flag useage:

When a file is opened, the zero flag in the 8080 CPU is set upon entering the mass storage output routine. When a file is closed, the carry flag in the 8080 CPU is set upon entering the mass storage output routine. If the carry flag is set upon returning from a mass storage input routine, it is an indication to the BASIC interpreter than an error has occured on a read operation.

Console (CRT, teletype, etc) flag usage:

When an input routine is entered with the zero flag set, it is a check for control-C or control-S, rather than an actual keyboard read operation. If a control-C was pressed on the keyboard, a return is made with the zero flag set.

The Terminal (TRMNL) Table:

This is a table located in the scratch area above BASIC. There are ten entries, with three bytes per entry, each entry corresponding to one of the ten I/O channels defined by the CHANL table. The first byte of each entry is the terminal width, that is, the number of characters after which there is a carriage-return issued. The second byte is the current terminal position. The third byte is used to determine the rubout. The low 7 bits of the byte is the code which is sent to the terminal when a 7F(hex) is received from the keyboard. If the upper bit is 0, the internal pointer is not decremented. If it is 1, it is.

The KIND byte:

This is a byte located in the scratch area above BASIC. It is set every time any mass storage (cassette or disk) operation is invoked. It's purpose is to make available to the I/O section information about the kind of transfer being made. Only the low 5 bits are currently used.

Content	Statement
0	LOAD
1	SAVE
2	BLOAD
3	BSAVE
4	GET
5	PUT
6	BGET
7	BPUT
8	APPEND
20	OPEN
21	CLOSE

Bit	If 0	If 1
0	input	output
1	ASCII	binary
2	program	data
3	not append	append
4	not opn/cls	open/close

Reserved Word List

These words should not be used as line descriptors or variable names:

ABS AND ASC ASSIGN ATN

BGET BLOAD BPUT BSAVE BYE

CADD CALL CHANNEL CHR$ CLEAR CLOSE CONT COS

DATA DEF DELETE DIM DISK DO DROP

EDIT ELSE END ENTER EOF EXP

FILE FOR FRE

GET GOPROC GOSUB GOTO

HEX HEX$

IF INP INPUT INT

LEFT$ LEN LET LIST LOAD LOC LOG

MATCH MID$

NEW NEXT NOT

OCT OCT$ ON OPEN OR OUT

PEEK POKE POS PRINT PROCEDURE PUT

READ RECEIVE RECORD REM RESTORE RETURN RIGHT$ RND RUN

SAVE SGN SIN SPACE$ SPC SQR STEP STOP STR$ SYMBOL

TAB TAN THEN TO TYPE

USR

VAL

WAIT

Known Bugs, Limitations, and Peculiarities

Hopefully this section will remain small. We have, however, decided to not ignore the fact, like some manufacturers do, that there will be forever bugs and other strange things in the system. To expect us to be perfect is asking too much, but we will at least work toward that objective. In that direction, we have already spent several months searching for these vermin, and exterminating them as quickly as possible. But we know that our customers will find some for us, so we'd appreciate it if you would let us know, preferably in writing, when you see any of these creatures creeping about. This page of the manual will change from one release to the next, with an effort to make the page match the release.

CTL STK ERROR message is somewhat obscure.

A space is required after all statements.

Assignments of values to variables are not allowed in command mode unless the variable has been previously defined in a program.

The expression 1/2 will evaluate to 0, since integer mode is retained until a floating point value is seen. Use the expression 1./2 or 1/2. to get the correct answer of .5 .

The LET statement name gets put in if you don't use it.

Parentheses may get rearranged to an equivalent sequence. This is a product of the way expressions are represented internally.

Tabs are not allowed in the middle of a statement.

Random numbers evidently always end in the digit 5.

When entering a number in exponential (E) format, always put either a space, minus sign, or plus sign after the E, then two digits.

Sometimes goes into ENTRY mode at the wrong time.

The expression X/Y*Z is evaluated in the wrong order.

How to Load Tarbell BASIC

If you have TARBELL BASIC on a CP/M disk, simply put the disk into the drive, and type TBASIC. You can ignore the rest of this page.

If you have TARBELL BASIC on cassette (Tarbell, of course), first examine the listing of the I/O section that came with the TARBELL BASIC.

Compare the console and cassette I/O routines to the ones you normally use in your system, to determine if there are any differences. If there are, mark the necessary changes on the listing.

Using either the bootstrap program or input program in the Tarbell cassette interface manual, or the Read-Only-Memory Program, or other monitor, read the TARBELL BASIC interpreter from the cassette into your main memory, using the starting address and length which is specified on the cassette.

NOTE: TARBELL BASIC is stored on tape at a rate of 1500 bits per second, or 800 bits per inch. A several-second leader of clock cycles is followed by the start-byte (3C), then the sync-byte (E6), then the number of bytes of program indicated on the cassette label under "length", then the checksum, all in one big block. The start-byte and sync-byte are detected by the hardware, and it is up to the software to read the proper number of bytes after that, and to check the checksum for errors, if desired.

If you need to make changes in the I/O section, now is the time to do it, using either your front panel DEPOSIT button or suitable monitor in ROM. Note that the top of memory address which is put into location SSSS is done automatically in CP/M systems, but may need changing for other systems. The default in cassette versions is to search for end of memory.

Start your computer running at the starting address specified on the cassette, by doing an examine and run at that location, or by using your ROM monitor to jump to it.

You should now get the opening message.

Comparisons With Other BASIC's

Speed:

TARBELL BASIC will generally run slower than ALTAIR BASIC by a factor up to three, in most tests involving numbers. This is because TARBELL BASIC uses 10 digits of BCD instead of 8 digits of binary. This precludes penny roundoff errors.

One place where TARBELL BASIC is faster, however, is in variable and label (line number/descriptor) references. This advantage in speed will not be significant on small benchmark programs, but only on the larger programs, with many variables and labels. The reason for the higher speed in this area is that TARBELL BASIC substitutes pointers for variable and label references, so instead of having to make a lengthy search through a table or program, the item is found immediately by a vectoring method.

If you purchase the source, you may notice that several of the subroutines are equivalent to Z-80 instructions. One good way for Z-80 users to drastically improve the speed of their TARBELL BASIC, is by patching in Z-80 instructions for these subroutines.

Readability:

This is where TARBELL BASIC really shines. Since most other BASIC's use line numbers, and are restricted to a few significant characters in the variable names, TARBELL BASIC allows line descriptors and long variable names.

Formatted PRINT output (PRINT USING):

Although PRINT USING is not currently part of TARBELL BASIC, it is easier to implement in a subroutine than in most other BASIC's. This is because arguments are allowed for the GOSUB and GOPROC statements, and local variables are allowed by using the PROCEDURE statement.

Interpreter vs Compiler:

The current implementation of TARBELL BASIC is as an interpreter. This allows the programmer to debug a program online, instead of continually going back and forth between edit, compile, and run operations. It does, however, take up more memory than a compiler. For example, whereas TARBELL BASIC requires about 22k of memory, CBASIC requires only 15k. There is one ray of hope, though. Since we make the source available at low cost, it is quite feasible to remove all those portions of the interpreter that a user doesn't need for a particular situation.