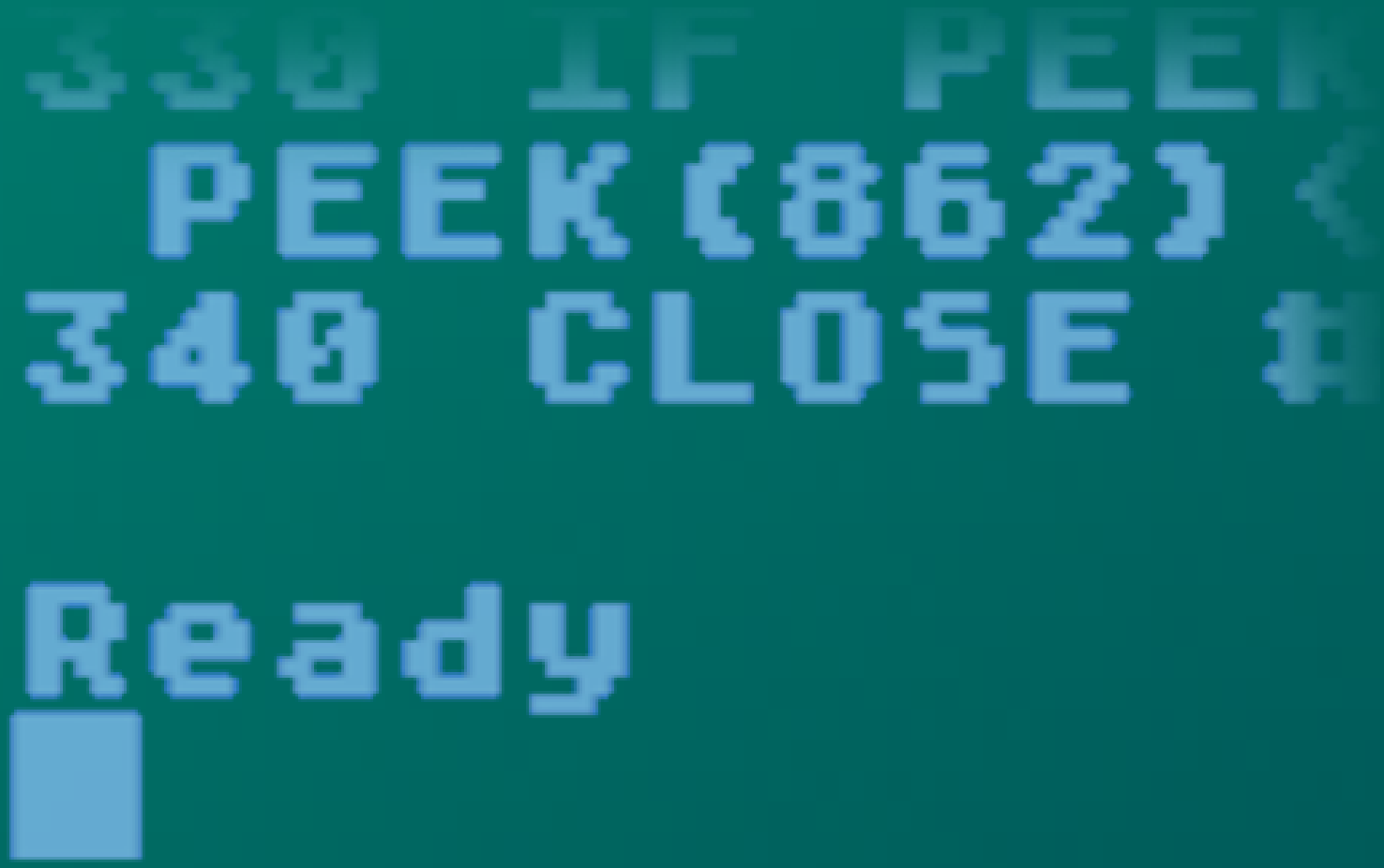

Altirra BASIC Reference Manual

2018-08-12 Edition

Avery Lee

Table of Contents

Introduction

Altirra BASIC is a BASIC interpreter for the Atari 8-bit computer line, including the 400/800, 1200XL, 600/800XL, 800/130XE, and XEGS computers. It is designed to be a drop-in compatible replacement for Atari BASIC, but includes selected extensions to run faster and be easier to use than Atari BASIC.

While compatible, Altirra BASIC is a reimplementation from scratch – it does not share any code with and is not based on Atari BASIC. Unlike the latter, Altirra BASIC is freely distributable in both source and object code form.

Starting Altirra BASIC

Altirra BASIC comes in two forms, an 8KB cartridge ROM image (atbasic.bin) and a DOS executable (atbasicx.xex). Both start the same BASIC interpreter, with minor differences.

Cartridge version

The 8KB cartridge version is made as a drop-in replacement for the Atari BASIC cartridge or system BASIC ROM (XL/XE systems). It can be burned into an EPROM to replace the original BASIC ROM in hardware, or written to a compatible flash memory cartridge. With emulators, it can substitute for the ATARIBAS.ROM file required to supply BASIC.

The cartridge ROM is configured to allow a disk boot on startup so that if a disk drive is attached, DOS boots before BASIC is started.

Executable version

The executable version can be run from any compatible DOS, including Atari DOS 2.0S and SpartaDOS X. It can also be run as AUTORUN.SYS during boot.

48K of memory is required to run the executable version, and additionally, the cartridge area ($A000-BFFF) must be free memory. An error will be displayed if it is occupied by a cartridge or otherwise not available. On XL/XE systems, the loader will automatically attempt to disable on-board BASIC if it is enabled. With SpartaDOS X, the X command must be used to load Altirra BASIC.

The executable version has two other features. By default, it will automatically attempt to load and start D:AUTORUN.BAS on startup. It can also auto-start a different program if a filename is specified on the command line.

On a 400/800, the executable version may run with 4KB less memory available than the cartridge ROM version. The reason is the OS-A/OS-B screen editor has a bug that causes it to overwrite up to a full memory page above MEMTOP when clearing the screen. This is of no consequence when running BASIC from ROM, but this would trash the executable version. Therefore, the executable version tests for whether an OS-A/B level screen editor is in use and whether it has this bug. If so, the memory limit

is lowered by 4KB to avoid the problem.

Immediate (direct) mode

When Altirra BASIC is started, it displays a banner and a Ready prompt:

```
Altirra 8K BASIC 1.55
Ready
```

The Ready prompt indicates that BASIC is in immediate mode and ready to receive a new typed-in command. Immediate mode allows two main actions: editing program code and running direct commands.

Direct commands

BASIC statements can be entered directly in immediate mode. One of the most useful is the PRINT statement, which can be used to evaluate expressions:

```
Ready
PRINT 1+2*3
7

Ready
```

Editing program code

BASIC program code can also entered for later use. This is done by entering a line starting with a *line number*:

```
Ready
10 PRINT "HELLO, WORLD!"
20 PRINT 1+2*3
```

This PRINT statement is not executed immediately, but instead stored for later use. The stored program can in turn be printed out with the LIST statement, and run with the RUN statement:

```
Ready
LIST
10 PRINT "HELLO, WORLD!"
20 PRINT 1+2*3

Ready
```

```
RUN
HELLO, WORLD!
7

Ready
```

This is called *deferred mode*. This is the more usual and powerful way of using BASIC.

A program consists of all numbered lines, in order. If lines are entered out of order, they are still reordered with increasing line numbers.

A line can be replaced by re-entering a new line with the same line number. If a line number is entered alone with no statements after it, the line with that number is deleted.

When adding, replacing, or deleting lines from the program, no Ready prompt is printed after each edit is processed. This usually isn't an issue as BASIC is quick to process each edit.

Syntax

Line numbers

All stored program lines start with a line number, which is an integer from 0 to 32767. Line numbers cannot be negative or greater than 32767. A fractional line number like 1.5 can be entered, but is rounded to an integer before being used.

Common practice is to advance line numbers by a round number like 10, so as to leave room in between if lines have to be inserted.

Statement syntax

Following the line number is a *statement*, which is a command for BASIC to do something. Statements are named by one or two words. Altirra BASIC requires keywords, including statement names, to be typed in uppercase.

Each statement has unique syntax for how its arguments, if any, are specified. Arguments include X and Y positions for drawing commands, and numbers and strings for PRINT commands. In many cases, the arguments supplied to a statement can vary, including possibly having no arguments if they are all optional.

In this manual, the syntax for each statement and its arguments. is shown using this notation:

$$LPRINT \left[\begin{matrix};\\,\end{matrix}\right]\left\{\begin{matrix}aexp\\sexp\end{matrix}\right\}\left[\left\{\begin{matrix};\\,\end{matrix}\right\}\left[\left\{\begin{matrix}aexp\\sexp\end{matrix}\right\}\right]\right]...$$

This shows the elements of the statement that can be included, from left to right. The semicolon and comma are shown vertically stacked, which means that either one can be used. The stack is surrounded

by square brackets [], which means that the stack is optional and can be left out. The curly braces are not intended to be entered, but indicate grouping. Finally, the ellipsis at the end (…) indicates that the last part can be repeated. Therefore, the following statements are all valid according to the above syntax:

```
LPRINT
LPRINT 10
LPRINT ,10
LPRINT ,10,
LPRINT ,A$;B$
```

The names *aexp* and *sexp* also appear in the syntax, which mean *arithmetic expression* and *string expression*. An *aexp* can be a number, a numeric variable, or an expression like 1+2*3, and is any expression that results in a number. Similarly, an *sexp* is a string expression, such as A$(4,7) or CHR$(125). Most statements take *aexp*s or *sexp*s.

avar and *svar* indicate that an arithmetic (numeric) variable or string varable is needed. X, I, and COUNT are valid *avar*s, while NAME$ and "ABC" are valid *svar*s. Expressions like X+1 and A$(4), on the other hand, are not valid *avar*s or *svar*s. Most of the cases where *avar*s and *svar*s are involve *out* parameters, where a value is stored into the variables instead of retrieved from the variable.

An *mvar* is a *matrix variable*, or a numeric array variable. A() and MAP() are array variables.

lineno refers to a line number, or a number from 0 to 32767. It is used in the few cases where a statement must take a line number directly, such as IF. Most statements that take line numbers actually accept an *aexp*, instead.

Finally, an *iocb* is an I/O control block number, from 0-7. It is always preceded by a hash sign (#) and usually a number, but any expression that is valid for an *aexp* can also be used as an *iocb*.

Combining statements on one line

A colon (:) may be used to include more than one statement on the same line.

```
10 PRINT "HELLO, WORLD!":GOTO 10
```

There are a couple of exceptions: REM and DATA statements must be the last statement on the line. Any colon after REM or DATA is interpreted as part of the statement rather than as a statement separator.

Abbreviating statement names

Any statement name can be abbreviated by typing the first part of the name and ending with a period. The first matching statement in the statement with that prefix is used. The statement list is ordered such that more commonly used statements have shorter abbreviations. For instance, G. is GOTO, not GET or GRAPHICS. The statement with the shortest abbreviation is REM, which can be abbreviated with just

a single period.

Abbreviated statement names are always expanded to the full name upon tokenization, so after a line has been entered, there is no difference between using an abbreviated or full statement name – the name is expanded either way. The abbreviations are just a shortcut to save typing.

Variable and function names cannot be abbreviated, only statement names.

Note: There is one quirky case where the single period abbreviation of REM may not work, when it is at the beginning of a line:

```
.10 LOOP ITERATIONS
Error-   .10 LOOP ITERATIONS
```

The reason is that the ".10" is picked up as a line number. Entering a remark as an immediate statement is unusual, but this could be encountered by accident in an ENTER script.

Program control

RUN (RU.)

Syntax: *RUN* [*sexp*]

Begins execution of a program in deferred mode, starting with the first (lowest numbered) line. Each line is executed in turn with the next highest line number, until either the end of the program is reached, an error occurs, or a statement redirects execution elsewhere.

The RUN statement takes an optional string expression with a filename of a program to run, previously saved with the SAVE statement. If present, that file is loaded and run.

```
RUN "D:PROGRAM.BAS"
```

While a program is executing, the [BREAK] key may be pressed at any time to interrupt execution and return to immediate mode.

RUN can also be used in deferred mode. When used without an argument, it restarts execution at the beginning of the program, and when used with a filename, it "chains" to another program, restarting execution at the beginning of the newly loaded program.

NEW

Resets BASIC state for a new program. The current program is cleared out, including all program lines, variables, arrays, and strings. This is used to restart with a clean slate without having to restart the computer.

If used in deferred mode, execution stops after NEW is executed.

END

Ends program execution. Any open I/O channels are closed, and all sound channels are silenced.

Ordinarily this isn't needed, as an implicit END automatically occurs when the end of a program is reached. However, sometimes an END in the middle of the program is useful, when there are subroutines or other lines afterward that shouldn't be executed.

Although unusual, END can also be used in immediate mode, if for nothing else but to quiet a cacophony from prior SOUND statements.

CONT (CON.)

Continues deferred execution from the point where it last stopped.

CONT is used to resume execution after a stop from an error or STOP statement. Variables can be checked with PRINT or fixes made to the program, and then execution continued with CONT. This allows a form of “edit and continue” when working on a BASIC program. CONT cannot continue after END has been executed, however.

Execution resumes at the first **line** after the line number where execution was stopped, not the next statement. Any number of lines can be deleted, and execution will resume at the next line that still exists. However, if the line where the stop occurred still exists, any remaining statements on that line are skipped.

There are a couple of more limitations on edits that can be done before CONT. If there are outstanding FOR or GOSUB statements that BASIC has remembered, those lines should not be edited, or CONT may fail with a GOSUB/FOR gone error (error 15), or the subroutine/loop may malfunction. Also, if the program is in the middle of READing from DATA statements, a RESTORE must be executed, or else the next READ may fail.

STOP (STO.)

Temporarily stops execution, printing a message with line number:

```
Stopped at line 140
```

This can be used to signal a problem that does not raise an actual error. The line number can be used to diagnose the problem, and if necessary, execution can be resumed with CONT or GOTO.

Listing a program

LIST (L.)

Syntax: *LIST* [*sexp*]
LIST *aexp*[,*aexp*[,*sexp*]]

Prints out program lines in readable form.

If a line number is specified, only that line is listed. If two line numbers are specified, only lines with numbers within that range are listed. For instance, LIST 10,50 will list lines starting with line 10 and ending with line 50, inclusive. Neither line has to exist.

An optional filename can be included to send the output to a file or device instead. This can be used to write a listing to disk for later use with the ENTER command, e.g. LIST “D:PROGRAM.TXT”, or to print out a listing, e.g. LIST “P:”.

ENTER (E.)

Syntax: *ENTER sexp*

Executes lines from a text file as immediate mode lines. This is the opposite of LIST, as it converts a text file to program lines instead of the other way around. Each line can either hold direct commands or edits to program code. The LIST statement is one way to generate such a file.

Note that because ENTER processes lines as if they were typed in, using ENTER will by default **merge** the statements from the file into the current program. This is useful for combining programs; if not desired, the NEW statement should be used to clear the current program beforehand.

ENTER automatically stops current program execution. No further commands are executed in the current line, or in deferred mode, in the rest of the program, unless the text file itself contains commands to do so. It is, however, possible for a text file executed by ENTER to contain a GOTO, RUN, or CONT statement.

Load/save

LOAD (LO.)

Syntax: *LOAD sexp*

Loads a program from disk that was previously saved by the SAVE command.

Execution stops when LOAD is executed. No further commands on the current line or the rest of the program are run.

If a problem occurs during a LOAD statement, such as insufficient memory or an I/O error, an implicit NEW is executed.

SAVE (S.)

Syntax: *SAVE sexp*

Saves the current program to disk in binary format, so that it can later be loaded back with LOAD.

CLOAD (CLOA.)

Loads a program from cassette tape. A single beep will be signaled by the operating system so that the user can press PLAY on the tape player, and then press a key. The program will then be loaded from tape.

Note: CLOAD differs from LOAD "C" in that it opens the C: device in short IRG mode, without gaps in between records. Use CLOAD over LOAD "C".

CSAVE (CS.)

Saves the current program to cassette tape. When used, the operating system will sound two beeps to ask the user to press RECORD on the tape player and then press a key. The tape recorder will then start up, the program will be saved to tape, and the tape recorder will automatically stop.

Note: CSAVE differs from SAVE "C" in that it opens the C: device in short IRG mode, without gaps in between records. Use CSAVE over SAVE "C".

Expressions

Many arguments to a statement, in turn, take expressions as arguments. Expressions combine values to produce a derived value. The simplest and most common statement to do so is the LET statement.

LET (LE.)

Syntax: $[LET]\begin{Bmatrix} avar=aexp \\ mvar(aexp[,aexp])=aexp \\ svar[(aexp[,aexp])]=sexp \end{Bmatrix}$

The LET statement assigns the value of an expression to a numeric variable, element of a numeric array, or to part or all of a string array. It is unique among the statements in that the LET statement name is optional and usually omitted, leaving just *var = value* as the statement. This is also called an *assignment*.

After the LET statement executes, the variable, array element, or (sub)string on the left has the same value as the expression on the right. The value is stored and not the expression, so if the result of the expression later changes, the variable doesn't.

```
A=1

Ready
B=A

Ready
A=2
```

```
Ready
PRINT A,B
2         1
```

Here, the assignment B=A sets B equal to A's value at the time of the assignment (1), but afterward changing A doesn't change B.

Values

The simplest form of an expression is a value – a number or variable. An expression always contains at least one value.

Altirra BASIC allows numbers to be entered in three different ways: decimal, exponential, and hexadecimal.

d.ddd (decimal number)

Decimal numbers are entered as leading integer digits, followed by a period, and then following fractional digits. 0, .1, 1.5, 2., and 79.273 are all valid decimal numbers. A + or – may also precede the number to indicate its sign.

d.dddE+dd (exponential notation)

Numbers can also be entered in exponential notation, as a base value (mantissa) and a base-10 exponent. An E followed by a sign (+ or -) is used to separate the two and signify use of exponential notation. For instance, -1.04E+07 means -1.04×10^{7}, while 2.9E-06 means 2.9×10^{-6}.

By default, Altirra BASIC uses exponential notation for numbers smaller than 0.01 or larger than 10^{10} in magnitude. However, nothing prevents use of exponential notation within that range and 1E2 is accepted as 100. Denormalized forms such as 0.01E6 and 107.4E-2 are also valid.

$hhhh (hexadecimal value)

Encodes a hexadecimal (base 16) value from 0-65535. These are most useful for memory addresses and for bit-encoded values. For example, $A9CF = 43471.

About floating point

All numbers are stored in Altirra BASIC in floating point format. This allows storage of both fractional, integer, small, and big numbers. The tradeoff is that big numbers are less accurate than small numbers. However, because *decimal* floating point is used, the rule is simple: at least 9 significant decimal digits of precision are always preserved when storing a number.

For example, the value 0.0012307 has three significant digits (12307), so it can be stored exactly. However, 3.14159265358979 has 15, so it can only be stored as 3.14159265. In some cases, 10 will be stored, depending on the position of the decimal point (even or odd placement).

Nine significant digits are also enough to store any integer numbers up to ten million exactly. Integers are numbers with no fraction, so they are a special case of floating point numbers. This means that, for

instance, memory addresses can always be stored exactly, because all values 0-65535 are well within this range.

The smallest value that can be stored is $\pm 10^{-98}$ (1E-98), while the largest is $\pm 9.99999999 \times 10^{97}$ (9.99999999E+97).

Operators

Two or more values can be combined into a more complex expression with an *operator* in between, which combines the values to produce a new value.

For instance, the addition operator (+) adds two values together and replaces them with their sum. The expression A+1 contains two values, the variable A and the number 1, combined with the addition operator. The resulting expression takes the current value stored in A and adds 1 to it. The notation is meant to be familiar to conventional algebra, with some concessions for symbols not available on the keyboard: 1+2×3 becomes 1+2*3.

Most operators are *binary,* meaning that they combine two values. This leads to an alternation of *value, operator, value...*, always beginning and ending with a value. There are a couple of exceptions that are *unary* operators, which mean that they work on only one value, which follows the operator.

The available operators in Altirra BASIC are:

+, - (unary plus, minus)

Preserves or flips the sign of a real number.

+, - (addition, subtraction)

Computes the arithmetic sum or difference of two real numbers.

**, / (multiplication, division)*

Computes the arithmetic product or ratio of two real numbers.

An attempt to divide by zero results in a value error (error 3).

Caution: Multiplication and division can cause a value error (error 3) when the result is so small that it can no longer be stored as a non-zero value, a case known as underflow. This is an unconventional behavior in the original Atari operating system math routines, which BASIC uses. Some alternate operating systems may give the more conventional behavior of underflowing to zero.

^ (exponentiation)

Computes the first value raised to the power of the second. For integer powers, this is equivalent to repeated multiplication, i.e. $2^6 = 2*2*2*2*2*2$. This is conventionally written as x^y, but due to limitations on the computer, x^y is written instead.

If x is a negative number, y must be an integer or a value error (error 3) results.

Raising 0 to the 0^{th} power gives 1.

& (bitwise AND)

Computes the bitwise AND function of two integer values, where a resulting bit is set only if the bit is set in the same position in both values when represented in binary. The integers can be in the range of 0 to 65535, inclusive.

! (bitwise OR)

Computes the bitwise OR function of two integer values, where a resulting bit is set if the bit in that position is set in either value. The integers can be in the range of 0 to 65535, inclusive.

% (bitwise exclusive OR)

Computes the bitwise exclusive OR function of two integer values, where a resulting bit is set if bit in that position is set in only one, and not both, of the two values. The integers can be in the range of 0 to 65535, inclusive.

AND

Computes the logical AND of two boolean values, where the result is true only if both values are true.

A value is considered to be boolean *true* if it is nonzero, and *false* if it is zero. When producing boolean values, such by AND, OR, or a relational operator, BASIC produces 1 for true and 0 for false.

OR

Computes the logical OR of two boolean values, where the result is true if either value is true.

<, >, <=, >=, =, <> (relational operators)

Compares two numbers or string values, returning boolean false or true depending on whether the first value is less than, greater than, less than or equal to, greater than or equal to, equal to, or not equal to the second. <=, >=, and <> are how the mathematical operators ≤ (less than or equal to), ≥ (greater than or equal to), and ≠ (not equal to) are entered.

For strings, the strings are compared character-by-character starting at the first character, using the number of each character in ATASCII. If the character values in the two strings don't match at some point, the result is the comparison of the first character that doesn't match between the two strings. If no mismatch is found but one of the strings is shorter than the other, the shorter string is considered to be less than the longer string.

String comparisons are case *sensitive*: lowercase “a” is not equal to uppercase “A”.

Operator precedence

When multiple operators are present in the same expression, BASIC evaluates the operators in order according to the *precedence* of each operator, similarly to how arithmetic formulas are used. For instance, with 1+2*3, the multiplication operation is performed first, giving 1+6, and then the addition operation, giving 7.

The precedence of each operator is listed in the following table:

Evaluated first	< > <= >= <> = (strings)
	+ - (unary)
	^ (power)
	& ! % (bitwise operations)
	* / (multiplicative)
	+ - (additive)
	< > <= >= <> = (numbers)
	NOT (logical)
	AND (logical)
	OR (logical)
Evaluated last	= (assignment)

Table 1: Operator precedence

Whenever two operators are involved in the same expression at the same precedence level, the leftmost one is evaluated first. Therefore, 10/2/2 = 2.5.

When the order of operations as determined by operator precedence is not as desired, parentheses can be used to group subexpressions to force evaluation order: (1+2)*3 = 9, because the addition is performed first, giving 3*3 as an intermediate result.

Note that there are a couple of differences in evaluation order versus other BASIC dialects or other programming languages:

- There is no short-circuit evaluation for logical AND and OR operators. Both arguments are always evaluated, even if the first argument would make the second moot. 0 AND 1/0, for instance, gives error 11.
- Unary minus is higher precedence than exponentation, so -2^4 = 16 instead of -16.
- String comparisons have higher precedence than all other operations. This means that X+A$<B$ is evaluated as X+(A$<B$), while X+A<B is evaluated as (X+A)<B.

Control flow

BASIC provides a number of *control flow* structures to direct program execution from one place to another, so that the program is not simply just run from top to bottom. This can include repeating portions (loops) and reusing the same lines from multiple places (subroutine calls).

To provide loop and subroutine support, BASIC maintains a *runtime stack* which contains all of the loops and subroutine calls that the interpreter is currently tracking. This is not ordinarily something to be minded as long as the beginning and end of loops and subroutines are matched, but it can occasionally become an issue if such control flow structures are left open.

REM (.)

Syntax: *REM ...*

Includes a remark (comment) in the program. Any text can be entered after the REM statement, which is ignored by the interpreter. Nothing happens when REM is encountered during execution. REM must be the last statement on a line.

Besides adding documentation to a program, another use for REM is to temporarily hold a line in place when there is nothing else to put in that line. This can happen if all statements on a line need to be removed while a program is being modified, but there are other parts of the program left that still do a GOTO or GOSUB to that line. A line cannot exist with no statements on it, so REM can be used to hold the line in place.

GOTO (G.) / GO TO

Syntax: *GOTO aexp*

Jumps to the beginning of the line with the supplied line number. If the line cannot be found, a line not found error (error 12) occurs.

In Altirra BASIC, GOTO can be used with expressions as well as line numbers. This is known as a *computed GOTO*, and is a convenient way to switch between multiple jump targets:

```
10 FOR I=1 TO 3
20 GOTO 30+I
31 PRINT "ONE":GOTO 40
32 PRINT "TWO":GOTO 40
33 PRINT "THREE":GOTO 40
40 NEXT I
RUN
ONE
TWO
THREE
```

GO TO is a synonym for GOTO.

FOR...NEXT (F. .. N.)

Syntax: *FOR avar = aexp* TO *aexp* [*STEP aexp*]
NEXT avar

Creates a counting loop, where statements between a FOR statement and a NEXT statement with a matching variable are executed with a variable counting within the given range, starting from the beginning value and through and including the ending value.

The step value controls the rate at which the counting variable increases or decreases for each loop iteration. It defaults to 1 if omitted, so FOR I=1 TO 5 … NEXT I will execute the enclosed statements

with I=1, 2, 3, 4, and finally 5. FOR I=5 TO 0 STEP -2, on the other hand, will execute the loop body with I=5, 3, and finally 1. If the initial value and step value cause the loop to miss the ending value, the loop will stop before the ending value and not go past it.

```
10 FOR I=0 TO 10 STEP 2
20 PRINT I
30 NEXT I
RUN
0
2
4
6
8
10

Ready
```

A FOR...NEXT loop is always executed at least once, no matter how weird the values. FOR I=2 TO 0...NEXT I, for instance, will execute the loop body once with I=2.

FOR...NEXT loops can be nested. When the NEXT statement is encountered, it is matched against the most recent FOR statement with the same variable, and any FOR frames in between are forgotten. NEXT will not match a FOR outside of a GOSUB, though:

```
10 FOR I=1 TO 2
20 FOR J=1 TO 10
30 PRINT I,J
40 NEXT I
50 NEXT J
RUN
1          1
2          1

Error-   13 at line 50
15 GOSUB 20
RUN
1          1
Error-   13 at line 40
```

In this case, the NEXT I statement on line 40 causes the FOR J statement at line 20 to be forgotten, so the NEXT J on line 50 fails. However, when GOSUB is added in between at line 15, the NEXT I statement no longer finds the FOR I loop. This can get confusing, so it's best to match FOR and NEXT statements so they cleanly nest.

The body of a FOR...NEXT loop does not have to be a single block of lines, and it is valid to jump in

and out of the loop using statements such as GOSUB and GOTO:

```
10 FOR I=1 TO 3
20 GOSUB 1000
30 NEXT I
40 END
1000 PRINT "At subroutine!",I
1010 RETURN
RUN
AT SUBROUTINE!      1
AT SUBROUTINE!      2
AT SUBROUTINE!      3

Ready
```

Regardless of how program flow proceeds within the loop, the loop is formed by the matching FOR and NEXT statements.

Occasionally it is useful to exit a FOR...NEXT loop early, without waiting for the loop variable to pass the ending value condition. In that case, the POP statement should be used to remove the FOR...NEXT entry on the runtime stack. This can be done either before or after GOTO or another control flow statement is used to exit the loop.

IF..THEN (inline form)

Syntax: $IF\ aexp\ THEN \left\{ \begin{matrix} lineno \\ statement \end{matrix} \right\}$

Tests a boolean condition and optionally executes a statement or jumps to a new line if the condition is true. If the condition is false, execution continues with the next **line**, not the next statement. Thus, all remaining statements on the line are controlled by the IF.

```
IF 0=1 THEN PRINT "0 EQUALS 1"

Ready
IF 1=1 THEN PRINT "1 EQUALS 1"
1 EQUALS 1

Ready
```

If a line number is used directly, i.e. IF A<>1 THEN 20, the line number must be a number – it cannot be an expression or variable like for the GOTO statement. If a computed line number is needed, a GOTO statement should be used instead: IF A<>1 THEN GOTO 10*I.

IF..ELSE..ENDIF (block form)

Syntax:
IF aexp
statements
[*ELSE*
statements]
ENDIF

Tests a boolean condition and optionally executes one block of statements or another. If the condition is true, the statements between the IF and the matching ELSE/ENDIF are executed; if it is false, the statements between ELSE and ENDIF are executed instead.

This form differs from IF..THEN in that the controlled statements are delimited by ELSE/ENDIF instead of the end of the line and may span multiple lines. THEN is not used for IF blocks.

It is also possible to nest IF blocks:

```
10 FOR I=1 TO 4
20 IF I<3
30 IF I=1:PRINT "ONE":ELSE:PRINT "TWO":ENDIF
40 ELSE
50 IF I=3:PRINT "THREE":ELSE:PRINT "FOUR":ENDIF
60 ENDIF
70 NEXT I
RUN
ONE
TWO
THREE
FOUR
```

Note that there is no ELSEIF or ELSE IF form.

Care should be taken when jumping in and out of block IF statements. When skipping over code within an IF block statement, the matching is done *statically*, which means in order seen in the listed program. This ignores any other control flow statements such as GOTO and RETURN inside of the skipped code, since it is not executed. It also means that POP has no effect on block IF constructs.

GOSUB (GOS.)

Syntax: *GOSUB aexp*

Calls a subroutine starting at the supplied line number. If the line cannot be found, a line not found error (error 12) occurs.

The GOSUB statement is used with the RETURN statement to create *subroutines*, blocks of code that can be reused more than once in a program:

```
10 PRINT "FIRST LINE":GOSUB 1000
```

```
20 PRINT "SECOND LINE":GOSUB 1000
30 PRINT "THIRD LINE":GOSUB 1000
40 END
1000 REM DELAY FOR A BIT
1010 FOR I=1 TO 100:NEXT I
1020 RETURN
```

When GOSUB is executed, an entry is created on the runtime stack to remember the current location, and then execution proceeds to the supplied line. The RETURN statement is then used to return to after the GOSUB statement.

If afterward it turns out that RETURNing back to the previous location is no longer necessary, the POP statement can be used to remove the GOSUB entry on the runtime stack, causing BASIC to forget the last GOSUB encountered. Each execution of a GOSUB statement should be matched to a corresponding POP or RETURN; otherwise, BASIC may end up remembering an endless number of GOSUBs, and eventually run out of memory with the runtime stack:

```
10 GOSUB 10
RUN

Error-   2 at line 10
```

There is, however, no requirement that a particular GOSUB always be matched with the same POP or RETURN.

RETURN (RET.)

Returns execution to immediately after the last GOSUB executed (and not removed by POP). Normally, a subroutine called by GOSUB ends in a RETURN statement to continue execution after the GOSUB. Any FOR statements still open on the runtime stack since the last GOSUB are forgotten.

If no GOSUB statements are left on the runtime stack, thus indicating a RETURN with no matching GOSUB, a Bad RETURN error (error 16) occurs.

ON..GOTO/GOSUB

Syntax: $ON\ aexp \begin{Bmatrix} GOTO \\ GOSUB \end{Bmatrix} aexp[,aexp]...$

Jumps to or calls a subroutine at one of a list of line numbers.

The integer portion of the supplied number is used to select the line number from the list, where 1 selects the first line number, 2 selects the second, etc. If the number is 0 or greater than the number of supplied line numbers, execution proceeds with the next statement instead. The value must be between 0 and 255 regardless; a value error (error 3) results otherwise.

The GOTO and GOSUB keywords in an ON..GOTO or ON..GOSUB statement cannot be abbreviated

and must be fully typed out.

POP

Syntax: *POP*

Forgets the most recently found FOR or GOSUB statement, removing its entry from the runtime stack. If there aren't any such statements and the runtime stack is empty, nothing happens.

POP is useful when a subroutine no longer needs to return to where it was called from, or when a FOR...NEXT loop needs to be exited early:

```
10 GOSUB 1000
20 END
1000 FOR I=1 TO 5
1010 GOSUB 2000
1020 NEXT I
1030 RETURN
2000 IF I<>3 THEN PRINT "I DON'T LIKE ";I:RETURN
2010 POP:REM REMOVE GOSUB
2020 POP:REM REMOVE FOR
2030 PRINT "I LIKE ";I
2040 RETURN
RUN
I DON'T LIKE 1
I DON'T LIKE 2
I LIKE 3

Ready
```

It prevents BASIC from remembering stale subroutine calls or loops, which can cause either an out of memory error or a NEXT/RETURN mismatch error.

Note that since IF statements are not remembered by BASIC on the runtime stack, POP has no effect on an IF...ENDIF or IF...ELSE...ENDIF block, and is not needed when jumping out of one.

Mathematical functions

ABS()

Computes the absolute value of a number, giving a value that is non-negative (zero or positive). This returns the argument for zero or positive numbers, and -X for ABS(X) where X is negative.

CLOG()

Computes the common (base 10) logarithm log(x) of a number. A value error results if the argument is negative. X=CLOG(Y) is the opposite (inverse) operation of Y=10^X.

Note: Depending on the operating system installed in your computer, CLOG(0) may either return zero or a large negative number. The older 400/800 operating system software, known as OS-B, will give the negative number, while the XL/XE operating system will give error 3.

EXP()

Computes the exponential function of a number, or e^x.

INT()

Computes floor function of a number. This gives the nearest integer less than or equal to the number. The same value is returned if already an integer.

```
PRINT INT(1),INT(1.5),INT(-1.5)
1         1         -2
```

LOG()

Computes the natural (base e) logarithm ln(x) of a number. A value error results if the argument is negative.

Note: Depending on the operating system installed in your computer, LOG(0) may either return zero or a large negative number. The older 400/800 operating system software, known as OS-B, will give the negative number, while the XL/XE operating system will give error 3.

RND()

Computes a random value from 0 to 1. The argument is ignored. A new random value is produced every time the function is called.

```
10 GRAPHICS 8
20 COLOR 1
30 PLOT RND(0)*319,RND(0)*159
40 GOTO 30
```

Warning: The value of the RND() function depends on the time that has passed since the sound chip (POKEY) was last reset. Over a million machine cycles occur per second, so normally this is fine. However, in rare cases the delay between the last reset and the next call to RND() may be too deterministic, resulting in rather un-random results. One solution is to ask the user to press a key or button. This introduces a random delay to re-seed the random generator.

SGN()

Extracts the sign of a number, giving +1 for positive numbers, -1 for negative numbers, and 0 for zero.

SQR()

Computes the square root sqrt(x) of a number. This is a slightly faster and more accurate version of

X^0.5. A value error results if the argument is negative.

Arrays

DIM (DI.) / COM

Syntax: $DIM \left\{ \begin{matrix} mvar(N) \\ mvar(M,N) \\ svar(N) \end{matrix} \right\} \left[, \left\{ \begin{matrix} mvar(N) \\ mvar(M,N) \\ svar(N) \end{matrix} \right\} \right] ...$

Sets dimensions for numeric or string arrays, allocating memory for their storage. An array must be DIMensioned before it can be used. Once allocated, an array cannot be re-dimensioned without being reset by CLR.

For numeric arrays, the value(s) specified are upper bounds, not counts. For instance, DIM A(4,3) creates an array where the first subscript is in the range 0..4 and the second in 0..3, giving an array with 5×4 = 20 elements.

For string arrays, the value is a character count, which must be greater than zero. Unlike some other BASICs, Altirra BASIC allows strings larger than 255 characters, limited only by available memory. String arrays cannot be two-dimensional.

COM is a synonym for DIM.

Array subscripting

An array subscript expression is used to select an element within an array. For one-dimensional arrays, a single subscript is used, i.e. A(1), and for two-dimensional arrays, two subscripts are used: A(1,2). This can be used on the left side of an assignment to set an array element:

```
DIM A(5)

Ready
A(1)=10

Ready
PRINT A(1)
10
```

CLR

Syntax: *CLR*

Clears all runtime variables. Numeric variables are set to 0 and arrays are returned to undimensioned status, freeing all associated memory. This is the only way to clear an array to free its memory or resize it while a program is running.

Note that CLR does not remove the variables themselves.

Strings

String literals

A string literal is a string specified directly within an expression, with quotes. The most common use of a string literal is with the PRINT statement, to print text to the screen:

```
PRINT "HELLO, WORLD!"
```

String literals begin and end with a double quote character ("). The ending quote character is optional, if the string literal is at the end of a line, and is automatically added if omitted.

The Screen Editor provided by the OS has an escaping feature that allows some special characters to be put into programs. For instance, the Move Left command can be embedded in a string by pressing ESC, followed by Ctrl+Left Arrow. This allows cursor movements, character insert/delete, line insert/delete, and clear screen commands to be used in PRINT statements.

Because the double quote is used to end the string, it can't be typed into a string literal. CHR$(34) can be used to obtain this character instead.

Allocating a string array

While string literals are useful, in order to modify or manipulate strings, a string array must be used.

Before a string array can be used, it must be allocated with the DIM statement to set its maximum length and allocate memory for it:

```
DIM A$(100)
```

The maximum length (size) is a tradeoff. Larger string arrays can hold more text, but require more memory. Once DIMensioned, the size of a string array is set. Some BASICs allow string variables to dynamically change size as needed, but then either accumulate unusable portions of memory called garbage or have to run an expensive periodic garbage collection pass to compact memory. This doesn't occur in Altirra BASIC, where all string memory is preallocated.

At any point, a string has two sizes associated with it: a length and a capacity. The length is how many characters long the string currently in the string array is, while the capacity is the maximum number of characters that the string array can hold, as set in the DIM statement. Newly DIMensioned string arrays are empty and have length 0. The length always ranges from 0 up to the capacity. This means that while a maximum length is set with DIM, the string array can hold shorter strings at different times and the current length of the string is tracked separately.

Assigning to a string array

Once allocated, a string array is set by assigning to it, similarly as to a numeric variable:

```
A$="HELLO, WORLD!"
```

Afterward, the string array now contains the string from the right side of the assignment. If the string is too long to fit in the string array, as much as can fit is copied and the rest is dropped.

A string assignment creates a copy of a string. When a string is copied from one string array to another, the original string array can then be modified without affecting the copy.

```
10 DIM A$(100),B$(100)
20 A$="ABC"
30 B$=A$
40 A$="DEF"
50 PRINT A$,B$
RUN
DEF       ABC
```

String subscripting

A string array may be accessed with subscripts to select a portion of the string array. This can be used to extract portions of a string, or to insert a substring into a larger string.

There are two forms that can be used. If one subscript is specified, i.e. A$(4), the remainder of the string array is selected starting at the given position, where 1 is the first character of the string array. If two subscripts are used, i.e. A$(4, 7), the portion is chosen starting at the character position selected by the first subscript, and ending at the second subscript (inclusive). This means that A$(4, 7) selects the subscript with four characters starting at the fourth character – positions 4, 5, 6, and 7. The second subscript must be equal to or greater than the first.

A string subscript expression can only select from the valid portion of a string, regardless of how long the string array is. It can't extend beyond the current end of the string. If the string array only has three characters in it, any subscript expression must end at the third character, or a bad string length error (error 5) results.

```
DIM A$(10):A$="ABCDEF":PRINT A$(2,5):PRINT A$(3)
BCDE
CDE

Ready
PRINT A$(7)

Error-   5
```

String subscripting can also be used on the left side of an assignment to assign to only part of a string array. When string subscripting is used on the left side of an assignment, the requirements are relaxed: the selected substring may extend beyond the current length of the string, but must still be within the DIMensioned length of the string array. An assignment to a substring that extends beyond the current string automatically extends the string.

```
A$="ABCDEF":A$(5)="123456":PRINT A$,LEN(A$)
ABCD123456          10
```

Subscripting creates a view of a string array without copying the string. This is useful for extracting or inserting a substring. Usually this is not an issue, except for an assignment to the same string array. In that case, the result is as if the string is copied one character at a time, starting from the beginning. This can be exploited to quickly initialize a string.

```
DIM A$(10):A$="0":A$(10)="0":A$(2)=A$:PRINT LEN(A$),A$
10        0000000000
```

Warning: For compatibility reasons, Altirra BASIC does not pre-clear string arrays. This can lead to old data or garbage appearing in strings when it is extended via subscripting:

```
A$="ABCDEF":A$="":A$(5)="":PRINT A$
ABCD
```

To avoid this, any such portions should be manually cleared.

LEN()

Syntax: $aexp = LEN\ (sexp)$

Returns the current length of a string in a string array, in characters. This always ranges from 0 up to the length originally declared in the DIM statement for the string array.

```
PRINT LEN("ABC")
3
```

Note that for a string array, this returns the current length of the string in the string array, not the maximum length:

```
DIM A$(10):A$="A":PRINT LEN(A$)
1
```

One use for LEN() is string concatenation:

```
DIM A$(30):A$="ABC":A$(LEN(A$)+1)="DEF":PRINT A$
ABCDEF
```

STR$()

Syntax: $sexp = STR\$(sexp)$

Converts a number to string form. This can be used to insert a number into a string.

```
DIM A$(30):A$="   <- COUNT":A$(1,3)=STR$(10):PRINT A$
10 <- COUNT
```

Note: Only one instance of STR$() or HEX$() can be used in an expression. Using it more than once, i.e. STR$(A)<HEX$(B), can result in incorrect results. It can be used more than once in the same statement, however.

See also: VAL()

VAL()

Parses a decimal number out of a string. Any leading spaces are skipped, and parsing stops at the first character not part of a valid number. An invalid string error (error 18) occurs if no number is found at all.

```
PRINT VAL(" 42 ITEMS")
42
```

VAL() and STR$() are opposites: VAL() converts a string to a number, and STR$() converts a number to a string.

See also: ASC(), STR$()

ASC()

Extracts the first character of a string and returns its character value as a number.

```
PRINT ASC("A")
65
```

Note: The result of applying ASC() to an empty string is an undefined value between 0-255.

See also: VAL()

CHR$()

Converts a number to a string containing a single character with that number, as represented in ATASCII. This can be used to obtain characters that cannot be directly entered into a string literal, such as the end of line character (155).

```
PRINT CHR$(65)
A
```

Note: Only one instance of CHR$() can be used in an expression. Using it more than once, i.e. CHR$(A)<CHR$(B), can result in incorrect results. It can be used more than once in the same statement, however.

HEX$()

Converts a number to hexadecimal form (base 16) in a string. Four hexadecimal digits are always printed.

```
PRINT HEX$(49152)
C000
```

Note: Only one instance of HEX$() or STR$() can be used in an expression. Using it more than once, i.e. STR$(A)<HEX$(B), can result in incorrect results. It can be used more than once in the same statement, however.

Input/output

Altirra BASIC uses the computer operating system's character I/O (CIO) facility to do I/O. CIO provides a common interface for devices, including the screen editor, disk drives, printers, serial ports, and real-time clocks. Programs that use CIO, including BASIC, can make use of any device that has a CIO device driver available for it.

I/O channels

All operations on devices are done on I/O channels, which are in turn controlled by I/O control blocks (IOCBs). There are eight I/O channels, numbered from #0 to #7, with eight corresponding IOCBs. Some of these I/O channels have defined uses by BASIC:

- #0 is reserved for the Screen Editor (E:) device.
- #6 is used by the GRAPHICS statement to talk to the Screen (S:) device, which controls the graphical screen (top half in split modes).
- #7 is used by the CLOAD, CSAVE, LOAD, SAVE, ENTER, DIR, RENAME, PROTECT, UNPROTECT, and ERASE statements, and by LIST when a filename is supplied.

IOCBs #1-5 are not used by BASIC and are available for program use. In addition, #6 and #7 may be used when not conflicting with the above statements.

Only the PRINT and INPUT statements allow use of IOCB #0. Other statements will trigger an invalid device number error (error 20) on an attempt to use IOCB #0.

File/device specifications

CIO uses a standard format to refer to both files and devices. The device is specified by a single letter, followed by an optional unit number (unit 1 is assumed otherwise), and then a colon. For instance, the

editor is E:, while the second printer is P2:.

Some devices also take a filename afterward, particularly the disk (D:) device managed by DOS. A file on disk 2 would be accessed with a filename like D2:PROGRAM.BAS.

Five devices are built into the OS and are always present. They are the editor (E:), screen (S:), printer (P:), cassette tape recorder (C:), and keyboard (K:). For many of these, it is more convenient to use appropriate BASIC statements, such as LPRINT to access P:, but sometimes it is useful to access them directly with OPEN.

Additional devices that are often accessed from BASIC include R: and T: for serial devices and Z: for a real-time clock. Device drivers have to be loaded beforehand for these devices to be available.

OPEN (O.)

Syntax: *OPEN #iocb,aexp,aexp,sexp*

Opens an I/O channel to the specified device and file.

The first *aexp* is the mode for the channel. It is typically 4 for read, 8 for write, and 12 for both read/write. Other values may be allowed by the device. For instance, DOS also allows 6 for reading a directory listing and 9 for appending to a file.

The second *aexp* is an auxiliary mode byte, which depends on the device being opened. It is often zero for files and most devices.

The *sexp* argument is the filename to open.

Warning: Using the OPEN statement on an I/O channel that is already open will result in an IOCB in use error (error 129), but can also corrupt the state of the existing IOCB and result in a crash if that IOCB is subsequently used.

Warning: If an OPEN statement fails with an error, the I/O channel is left in a half-open state and must be CLOSEd. This is the behavior of the OS's CIO facility.

CLOSE (CL.)

Syntax: *CLOSE #iocb*

Closes an I/O channel. If the I/O channel is already closed, no error results.

It is good practice to close I/O channels that are no longer needed, particularly those related to files on disk (D:), or communications (R:). This frees up resources involved with the I/O channel and can help avoid conflicts. In the case of a disk file opened for write (mode 8), it also finishes the file.

Warning: Closing an I/O channel can trigger an error, such as if the device driver attempts and fails to flush remaining buffered data to the device.

PRINT (PR.) / ?

Syntax: $PRINT\ [\#iocb]\left[\begin{Bmatrix};\\,\end{Bmatrix}\right]\begin{Bmatrix}aexp\\sexp\end{Bmatrix}\left[\left[\begin{Bmatrix};\\,\end{Bmatrix}\right]\begin{Bmatrix}aexp\\sexp\end{Bmatrix}\right]...$

Formats numbers or string values and writes them to either the screen or an I/O channel.

Numbers are written out as either decimal form (1.23) or exponential form (1.23E+04) depending on how big the number is. Strings are written out as-is. Comma or semicolon symbols are used to separate multiple values and may also precede or follow the list of values. Semicolons allow values to print back-to-back, while commas separate values into columns 10 characters wide.

If #iocb is present, it specifies the I/O channel to use. Unlike most other commands, the screen editor (#0) may be used with PRINT. A comma or semicolon must separate #iocb from following values. In most cases, a semicolon should be used to avoid padding to the next column before the first value.

By default, a PRINT statement ends its output with an end-of-line (EOL) value, so that any further output begins on the next line. The PRINT statement can be ended with a semicolon to prevent this.

The columns established by commas are by default 10 characters apart starting from the beginning of the PRINT statement. If a line is created with more than one PRINT statement, the columns from the second PRINT may not line up with the first. Column widths other than 10 characters can be used by POKEing the desired width into PTABW at location 201 ($C9).

? is a synonym for PRINT.

INPUT (I.)

Syntax: $INPUT\ \left[\#iocb\begin{Bmatrix};\\,\end{Bmatrix}\right]\begin{Bmatrix}avar\\svar\end{Bmatrix}\left[\left[\begin{Bmatrix};\\,\end{Bmatrix}\right]\begin{Bmatrix}avar\\svar\end{Bmatrix}\right]...$

Reads numeric or string values from either the screen editor or an I/O channel, and places the result into one or more variables. When reading from the screen editor (#0), a ? prompt is printed before each line of input is accepted.

For numeric variables, the number may be in either decimal form (1.23) or exponential form (1.23E+04). For string variables, the string is any combination of characters. Both numeric and string values may be separated by commas, so that INPUT A$,B will take “ABC,123” and put “ABC” into A$ and 123 into B. If there are more values on a line than needed, the extra items are ignored.

If a string read is longer than the string array it is being read into, the whole string is read, but only the characters that can fit are put into the string array.

Either commas or semicolons can be used to separate variables. The INPUT statement acts the same way regardless of which one is used.

Lines read by the INPUT statement cannot exceed 255 bytes. If a longer line is encountered, a record truncated error (error 137) results.

See also: READ

LPRINT (LP.)

Syntax: $LPRINT \left[\begin{matrix};\\,\end{matrix}\right]\left\{\begin{matrix}aexp\\sexp\end{matrix}\right\}\left[\left\{\begin{matrix};\\,\end{matrix}\right\}\left[\left\{\begin{matrix}aexp\\sexp\end{matrix}\right\}\right]\right]...$

Formats number or string values and writes them to the printer (P:).

The syntax is the same as for the PRINT statement, except that an I/O channel cannot be specified. I/O channel #7 is temporarily used to write to the printer and must not already be OPEN.

GET (GE.)

Syntax: *GET #iocb,avar*

Reads a character from an I/O channel and places the character value into the given numeric variable.

If the I/O device is an interactive device like E: or K: and no input is available, GET will wait for new input and then return the next available character. For a non-interactive source like a file on disk, an end of file (error 136) occurs instead.

PUT (PU.)

Syntax: *PUT #iocb,aexp*

Writes a single character to an I/O channel with the character value specified by the expression, 0-255.

NOTE (NO.)

Syntax: *NOTE #iocb,avar,avar*

Records the current position of an I/O channel into a pair of variables. This saves the current position of the file so that it can be restored later with the POINT statement, seeking back to the same position. There is no limit to the number of positions that can be temporarily saved this way.

The meaning of the two values depends on the device. For files on disk (D:), this depends on the version of DOS used. Some versions of DOS use a sector number and byte offset pair, whereas others use the two values together as a single byte offset within the file. The most portable use of the NOTE and POINT statements is therefore only to use POINT with values that were originally retrieved from NOTE, while the file is still open.

Under the hood, the first value is derived from AUX3+256*AUX4 on the IOCB, and the second value from AUX5.

See also: POINT

POINT (PO.)

Syntax: *POINT #iocb,avar,avar*

Sets the current position of an I/O channel to that specified by a pair of variables, typically retrieved earlier by NOTE. The values used by POINT are specific to the device and in particular can vary between different versions of DOS or on different disks.

Note that unusually, POINT requires an *avar* for both arguments instead of an *aexp*, even though both are input arguments. This is for compatibility reasons.

See also: NOTE

XIO (X.)

Syntax: *XIO aexp,#iocb,aexp,aexp,sexp*

Executes a special device-specific command on an I/O channel.

The first argument is the command to execute, while the third and fourth arguments are the two mode bytes. All three are device-specific.

XIO can be used either with an open or closed I/O channel, depending on the command. For commands that are used with open I/O channels, the last argument is a string parameter for the command; it is often ignored or set to the device name, i.e. “R:”. For commands that are used with closed I/O channels, like DOS lock/unlock commands, it is a filename used to temporarily open the channel.

Caution: Use of XIO commands on an open I/O channel can cause subsequent GET, PUT, INPUT, PRINT, BGET, and BPUT commands on that channel to fail. This is because the mode bytes overwrite the ones originally used to open the channel, and the first one in particular determines the get/put permissions on the channel. Check the documentation for the device for details; some device drivers, particularly the R: serial device, are written to handle this. Commands that are designed to run on a temporary I/O channel, such as the DOS command to rename a file (XIO 33), don't have this problem.

STATUS (ST.)

Syntax: *STATUS #iocb,avar*

Retrieves the CIO status code from the last command executed on an I/O channel. This will be an error code if the last operation on that channel failed, or a value below 128 if the last operation succeeded.

STATUS won't cause an error if used with an unopened I/O channel. Instead, the error 130 that results is put into the variable.

BGET

Syntax: *BGET #iocb,aexp,aexp*

Reads binary data from an I/O channel to a location in memory.

The first number is the starting address in memory, and the second number is the number of bytes to read.

BPUT

Syntax: *BPUT #iocb,aexp,aexp*

Writes binary data from a location in memory to an I/O channel.

The first number is the starting address in memory, and the second number is the number of bytes to write.

Warning: Some versions of DOS cannot handle a BPUT from read-only memory (ROM), such as BPUT #1,$E000,$0800, because they attempt to temporarily modify the memory to include sector headers to speed up the transfer. This also means that doing a BPUT from screen memory can result in glitches on screen, and trying to save ROM contents directly to disk can result in a corrupted disk. To avoid these problems, use MOVE to copy the memory to a temporary area and BPUT from that area.

Data

DATA (D.)

Syntax: *DATA ...*

Stores data to be read by the READ statement. The DATA statement occupies the rest of the line and cannot be followed by another statement on the same line.

The contents of a DATA statement are either strings or numbers. Strings in DATA statements do not require or use quotes. Multiple values can be placed in the same DATA statement, separated by commas.

DATA statements are skipped when encountered during execution, and can be placed anywhere in the program.

READ (REA.)

Syntax: $READ \left\{ \begin{matrix} avar \\ svar \end{matrix} \right\} \left[, \left\{ \begin{matrix} avar \\ svar \end{matrix} \right\} \right] ...$

Reads values from one or more DATA statements into variables, starting with the beginning of the program, or the last line number supplied with RESTORE.

READ is similar to the INPUT statement, except that it reads from DATA statements in the program instead of from the screen editor or an I/O channel. There is also one other important difference in string handling: the sequence “ABC,DEF” is read as one string by INPUT and two strings by READ.

Note that for strings, READ only allows an *svar*, referring to an entire string array. Reading into a portion of a string array directly is not possible.

RESTORE (RES.)

Syntax: *RESTORE* [*aexp*]

Resets the current read location for the READ statement to a new line number. The next READ statement will begin reading from the first line whose line number is ≥ the specified line number. If no line number is specified, the beginning of the program is used (0).

It is valid to RESTORE to a line number that doesn't currently exist. If it is later added before the next READ, the scan will begin at that line, and if the line still doesn't exist, the scan will start with the next nearest line instead.

Trigonometric functions

The trigonometric functions involve relations between lengths and angles within a right-angle triangle.

One function not provided by Altirra BASIC that is sometimes useful is the tangent function or tan(x), the slope of a line with the given angle. This can be computed as SIN(x)/COS(x).

ATN()

Computes atan(x), the arctangent of a number.

COS()

Computes the mathematical cos(x) function.

DEG

Switches the SIN(), COS(), and ATN() functions to use degrees.

See also: RAD

RAD

Switches the SIN(), COS(), and ATN() functions to use radians for angles.

See also: DEG

SIN()

Computes the mathematical sin(x) function.

Controllers

Altirra BASIC has several functions to read inputs from joystick and paddle controllers. Joystick controllers are numbered for 0-3, while paddle controllers are numbered 0-7.

The later XL and XE computer models only have two controller ports and thus only joysticks 0-1 and paddles 0-3 are valid.

Joystick and paddle controllers share some of the same signals in hardware, and therefore the functions can return erroneous inputs if used with the wrong type of controller.

HSTICK()

Syntax: *aexp*=*HSTICK* (*aexp*)

Returns the current horizontal direction state of a joystick controller: -1 if the joystick is being pushed left, +1 if it is being pushed right, and 0 if neither left nor right.

PADDLE()

Syntax: *aexp*=*PADDLE* (*aexp*)

Returns the current rotational position of a paddle controller, from 0 to 228. The value increases when the paddle knob is rotated to the left (counterclockwise) and decreases when rotated to the right (clockwise).

It is normal for the output value of a paddle to be somewhat noisy, especially when the paddle knob is rotated.

PTRIG()

Syntax: *aexp*=*PTRIG* (*aexp*)

Returns the current trigger state of a paddle controller: 0 if the paddle button is pressed, 1 if it is not pressed.

STICK()

Syntax: *aexp*=*STICK* (*aexp*)

Returns the current direction state of a joystick controller, from 0-15. These are the standard values returned by STICK() for all 9 possible joystick directions:

10	14	6
11	15	7
9	13	5

Note that any value from 0-15 can be returned, even those not in the above chart. This can happen either due to a noisy joystick or due to another type of controller being plugged in to the port.

STRIG()

Syntax: *aexp*=*STRIG* (*aexp*)

Returns the current trigger (fire) button state of a joystick controller: 0 if the paddle button is pressed, 1 if it is not pressed.

VSTICK()

Syntax: *aexp=VSTICK(aexp)*

Returns the current vertical direction state of a joystick controller: -1 if the joystick is pushed down, +1 if pushed up, and 0 if neither down nor up.

Graphics

GRAPHICS (GR.)

Syntax: *GRAPHICS aexp*

Switches the screen to a new text or graphics mode. The following modes are defined:

Type	Mode	Full screen resolution	Split screen resolution	Colors	Notes
Text	0	40x24	N/A	1*	One color at two brightness levels
	1	40x24	40x20	5	Background + four text colors
	2	40x12	40x10	5	Background + four text colors
Tile graphics	12	40x24	40x20	5	Four colors + PF2/3 switch
	13	40x12	40x10	5	Four colors + PF2/3 switch
Graphics	3	40x24	40x20	4	
	4	80x48	80x40	2	
	5	80x48	80x40	4	
	6	160x96	160x80	2	
	7	160x96	160x80	4	
	8	320x192	320x160	1*	One color at two brightness levels
	9	80x192	N/A	1*	16 brightness levels of one color
	10	80x192	N/A	9	
	11	80x192	N/A	16*	Black + 15 colors at one brightness
	14	160x192	160x160	2	
	15	160x192	160x160	4	

Table 2: Graphics modes

Modes 9-15 are only available on the 600XL, 800XL, 800XE, 65XE, 130XE, and XEGS. The 400, 800, and 1200XL only support modes 0-8 with the standard OS.

In addition, two values can be added to the mode number for additional effects.

Adding 16 to the mode number suppresses the split screen at the bottom, giving a full screen text or graphics mode instead. However, writing any text to the screen editor will end full screen mode and revert the screen to the regular text screen (GRAPHICS 0). Also, modes 9-11 are always full screen regardless of this flag.

Adding 32 to the mode number suppresses the screen clear, preserving whatever is in the memory used for the new screen mode.

Note: Do not rely on screen contents being preserved when switching to another screen mode and back with the +32 modifier. The operating system places some additional screen data called the display list before the screen memory, and this can corrupt whatever you were trying to preserve.

Setting a GRAPHICS mode can require extra memory, up to about 7K more than a standard GR.0 screen. The more colorful and higher resolution modes take more memory. While the new mode is active, this memory is not available to your program for arrays or the runtime stack, although it will be returned when a different mode requiring less memory is used. The amount of memory needed can be seen by the difference in FRE(0). If there isn't enough memory free for the requested mode, error 147 results and a mode 0 screen is automatically reopened.

After a GRAPHICS statement is successfully executed, I/O channel #6 is open to the graphics portion of the screen. PRINT #6 can be used to write to the graphics screen, which is most useful with the text modes 1 and 2. Only end of line (155) and clear screen (125) characters work on the graphics screen. However, if I/O channel #6 is closed, either by the CLOSE statement or implicitly by END, graphics commands will stop working.

COLOR (C.)

Syntax: COLOR *aexp*

Sets the color to use for PLOT and DRAWTO statements, and for fill commands. No default color is defined by default, so COLOR must always be used before the first PLOT or DRAWTO to set a color other than the background.

For most graphics modes, zero is the background color, and 1-3 are the playfield colors. A few of the graphics modes allow 0-15.

In text modes, the value is a character value (0-255). In graphics 0, 0-127 are regular ATASCII characters, and 128-255 are inverted versions of the characters. In graphics 1 and 2, only 64 of the characters are available, but in four colors depending on whether an offset of +0, +64, +128, or +192 is used. Two character sets are available for these modes, called the uppercase set and the lowercase set; the lowercase set is selected by POKE 756, 226, while the default uppercase set is restored with POKE 756, 224.

In tiled modes (12 and 13), 128 tiles are available, 0-127. 128-255 are the same tiles, but with the third color (SETCOLOR 2) replaced by the fourth color (SETCOLOR 3).

PLOT (PL.)

Syntax: *PLOT aexp,aexp*

Draws a pixel at the given position in the current color as set by the last COLOR statement, and makes that position the last drawn position.

Positions are given as column, row where 0 is the leftmost column the topmost row. Increasing coordinates indicate positions to further to the right or down.

DRAWTO (DR.)

Syntax: *DRAWTO aexp,aexp*

Draws a line from the last drawn position to the given position (column,row) in the current color. That position then becomes the last drawn position.

The starting pixel at the last drawn position is not drawn by DRAWTO, but the ending pixel is.

DRAWTO acts erratically when used in a GRAPHICS 0 screen, due to the screen (S:) device not handling cursor updates very well when the editor (E:) shares the same screen. In particular, the last character can sometimes be corrupted. It's recommended to avoid drawing lines on a GR.0 screen.

POSITION (POS.)

Syntax: *POSITION aexp,aexp*

In GRAPHICS 0, moves the text cursor to the given (column, row) position.

In any other GRAPHICS mode, moves the drawing cursor to a new position. Nothing is actually drawn at that location, however. Note that this is different than the last drawn cursor used by DRAWTO; the new position is not copied to the last drawn position unless a write to IOCB #6 occurs.

LOCATE (LOC.)

Syntax: *LOCATE aexp,aexp,avar*

Reads the color of the pixel at the given location, specified by column and then row, and places its value into a variable. The drawing cursor is moved to the new position.

In GRAPHICS 0, this also moves the cursor to one past the given location.

SETCOLOR (SE.)

Syntax: *SETCOLOR aexp,aexp,aexp*

Changes one of the four playfield colors or the background color.

The first argument is the color to set, 0-3 for the playfield colors and 4 for the background color. The values are different than for the COLOR statement, where 0 is the background color and 1-3 are the first three playfield colors.

The second argument is the hue of the color, from 0-15. Zero is no hue (grayscale).

The third argument is the brightness (luminance) of the color, from 0-15. There are eight brightnesses at even values from 0-14. An odd value gives the same brightness as the next lower even value.

The playfield color values drawn with the PLOT and DRAWTO statements always match the corresponding colors set by SETCOLOR, and changing one of the playfield colors will immediately change all pixels that were drawn with that color. In other words, any pixel drawn with COLOR 1 will always have the color set by SETCOLOR 0, even if those pixels were drawn before SETCOLOR was executed. Trying to use SETCOLOR between PLOT/DRAWTO statements to put more colors on screen won't work.

In high-resolution modes – GRAPHICS 0 and 8 – the background color for the text area is color 2, but the foreground color has the hue of color 2 with the brightness of color 1. The hue of color 1 is ignored, and the foreground and background colors cannot have different hues.

All colors set by SETCOLOR are reset to defaults whenever a GRAPHICS statement is executed, equivalent to the following statements:

```
SETCOLOR 0,2,8
SETCOLOR 1,12,10
SETCOLOR 2,9,4
SETCOLOR 3,4,6
SETCOLOR 4,0,0
```

Sound

SOUND (SO.)

Syntax: *SOUND aexp,aexp,aexp,aexp*

Plays sound on one of the four sound channels. The parameters are, in order: channel, period, distortion, and volume.

The sound channel ranges from 0-3. Four independent sounds can be played simultaneously on each of the four sound channels.

The period of the sound ranges from 0-255. Period is the inverse of pitch, so lower period values result in higher pitched tones. For pure notes, this is related to the pitch of the sound as follows:

$$pitch = \frac{31960.4}{period+1} Hz \quad \text{(NTSC)}$$

$$pitch = \frac{31668.7}{period+1} Hz \quad \text{(PAL/SECAM)}$$

The formula to use depends on the television standard supported by your computer (or enabled in an

emulator). The following is one set of period values to produce a musical scale:

C (lower octave)	254
C#	240
D	227
D#	214
E	202
F	190
F#	180
G	170
G#	160
A	151
A#	143
B	134
C (upper octave)	127

Table 3: Sample pitch values for music

The distortion value selects whether pure tones or noises are played:

1, 3, 5, 7, 9, 11, 13, 15	Silent
0	17-bit + 5-bit noise
2, 6	5-bit noise
4	4-bit + 5-bit noise
8	17-bit noise
10, 14	Pure tone (square wave)
12	4-bit noise

Table 4: Distortion values for SOUND statement

Three different noise generators can be combined via the distortion setting. For the noise generators, the resulting sound depends both on the types of noise selected and the period.

Volume ranges from 0-15, where 0 is silent and 15 is loudest. Use volume 0 to silence a channel.

A sound continues to play until replaced by a new sound on the same channel, or stopped by END. Playing sounds can also be interrupted by disk access or any other access over the SIO bus, such as using the printer or a serial (RS-232) device.

For more details on sound generation, see the POKEY chapter of the *Altirra Hardware Reference Manual*. The SOUND statement resets AUDCTL to 17-bit noise polynomial and 64KHz clock, clears asynchronous receive mode to unlock channels 3+4, writes the period value to AUDFn, and combines

the distortion and volume to write to AUDCn.

Player/missile graphics

Altirra BASIC also supports *player/missile graphics*. These are graphics objects that lie on top of the playfield. They are limited in color depth and size, but because they are overlaid in hardware, they can be moved quickly without having to redraw the playfield underneath. These objects are also sometimes called *hardware sprites*.

Atari BASIC provides no direct support for player/missile graphics, so it is common practice to use POKEs to set up the P/M graphics, mapping the P/M graphics onto a string array for fast manipulation. This technique is still effective in Altirra BASIC, but direct P/M graphics statements make players and missiles easier to set up and use.

PMGRAPHICS

Syntax: *PMGRAPHICS aexp*

Initializes player/missile graphics.

The PMGRAPHICS statement is the first step needed to use P/M graphics. The single argument specifies the mode: 1 specifies single-line graphics, 2 specifies double-line graphics, and 0 turns off P/M graphics. About 1K must be free for double-line graphics and 2K for single-line graphics.

There are several tradeoffs between single-line graphics and double-line graphics. Single-line graphics have higher resolution, but take more memory and are slower to update. Double-line graphics have half the vertical resolution and look chunkier, but take less memory and are faster to update.

P/M graphics are automatically disabled whenever the GRAPHICS statement is executed.

PMADR()

Syntax: *PMADR(aexp)*

Returns the starting address of a player or missile. Values 0-3 retrieve the base address of players 0-3, and values 4-7 retrieve the base address of missiles.

PMADR() is the most direct way to load player graphics. It can be written using POKE, or block copied into via the MOVE statement, for updates that can't be done directly with BASIC statements. The P/M address can even be passed off to a machine-language routine called via USR() for maximum speed.

Each player or missile occupies 256 bytes in single-line mode and 128 bytes in double-line mode. The address given is for scan line 0, which is off-screen at the top, and each successive address is one pixel lower. For a standard screen set up with GRAPHICS, the range PMADR(n)+32 to PMADR(n)+223 matches the playfield in single-line mode; in double-line mode, the offsets are halved, so it is PMADR(n)+16 to PMADR(n)+111.

All missiles are stored in the same strip of memory, just in different bits. Missile 0 is stored in bits 0-1, missile 1 in bits 2-3, missile 2 in 4-5, and missile 3 in bits 6-7. Thus, PMADR(4) through PMADR(7) all return the same address.

PMCOLOR

Syntax: *PMCOLOR aexp,aexp,aexp*

Sets one of the four player/missile colors. Unlike the playfield colors, which are set to defaults after a GRAPHICS statement, there are no defaults for the P/M colors. The PMCOLOR statement is used to set those colors.

The first argument is the player/missile index (0-3), the second is the hue, and the third is the brightness. Hue and brightness values are the same as for the SETCOLOR statement. Each player/missile pair shares the same color, so player 0 has the same color as missile 0, player 1 the same as missile 1, etc.

Setting up a player/missile

Once P/M graphics have been set up with PMGRAPHICS, it is possible to load the image for a player or missile with PMADR().

Colors also need to be set up for a player or missile. There are four colors available, assigned to each player/missile pair: player 0 and missile 0, player 1 and missile 1, etc. There is unfortunately no statement to set player/missile colors, so this needs to be done with POKE 704+N,C, where N is the player/missile number (0-3) and the color value is hue and brightness combined (H*16+B).

The width of a player or missile also can be set. This is also not supported by a statement, but can be set using POKE 53256+N,SIZE for players, where N=0-3 and SIZE=0, 1, or 3 for 1x, 2x, and 4x width, respectively. Missile widths are set all at once by POKE 53260,M3*64+M2*16+M1*4+M0.

PMMOVE

Syntax: *PMMOVE aexp*[*,aexp*][*;aexp*]

Moves a player or missile horizontally, vertically, or both.

The first argument selects the player or missile to move: 0-3 for players 0-3, and 4-7 for missiles 0-3.

A value specified after a comma gives the new horizontal position for the player or missile. For a standard screen set up with GRAPHICS, positions 48-207 ($30-$DF) cover the playfield, with lower values being farther left. The player/missile is moved so that the left pixel of the object covers the specified pixel position. Position 0 is off-screen.

A value specified after a semicolon causes a relative vertical motion, from -255 to +255 pixels. It must come after the horizontal position if both are specified. Positive values move the object upward, while negative values move the object downward. When the player/missile is moved, blank data is scrolled in to fill the gap at the end.

Note that the horizontal value is *absolute* whereas the vertical value is *relative*. The horizontal value is the position to move *to*, while the vertical value is the amount to move *by.*

Performance-wise, horizontal movement is faster than vertical movement. Moving a player or missile horizontally simply involves changing a hardware register, but moving vertically requires BASIC to move up to 256 bytes of memory.

MISSILE

Syntax: *MISSILE aexp , aexp , aexp*

Creates or removes part of a missile.

The first argument is the missile number, 0-3 for missiles 0-3. Values 4-7 can also be used for convenience, to match other statements that handle both players and missiles.

The second and third arguments are the top pixel of the missile to change and the number of pixels to change, respectively. Both pixels in each row in this range are *toggled*, so that pixels that were off turn on, and vice versa. This allows MISSILE to be used both to set or clear a missile.

PMCLR

Syntax: *PMCLR aexp*

Clears the strip of memory for a player or missile. Values 0-3 clear players 0-3, and values 4-7 clear *all* missiles (not just the selected one). PMCLR is faster than using either MOVE or PMMOVE to clear memory.

BUMP()

Syntax: *BUMP (aexp , aexp)*

Checks if a P/M graphics collision has occurred.

The first value specifies a player/missile, 0-3 for players 0-3 and 4-7 for missiles 0-3, while the second value specifies either a player (0-3) or a playfield color (8-11). There are thus four types of collisions that BUMP() can track: player-player, player-missile, player-playfield, and missile-playfield.

A collision occurs if any pixel from one object overlaps any pixel from another. This happens even if the two objects are the same color and even if both are hidden by another object. If more than two objects are involved, all possible pairs are flagged. On the other hand, a player never collides with itself. Once a collision is detected, it stays flagged until cleared in hardware with a write to HITCLR (POKE 53278,0).

For playfield collisions, playfields 0-3 correspond to the colors set by SETCOLOR 0-3, and playfields 0-2 match pixels drawn with COLOR 0-2 in graphics modes. Playfield 3 is rarer and only appears in the colored text modes (GR.1/2) or tiled modes (GR.12/13). For hi-res modes (GR.0/8), lit pixels are flagged as collisions with playfield 2.

Missile-missile collisions are not supported.

Reliably detecting collisions

Hardware P/M collisions are useful because they require little work and are pixel-accurate, but they have one very big caveat: collisions are only detected while the object is being displayed. This happens 50 or 60 times a second, so it doesn't take long, but even BASIC is fast enough that it is possible to move an object with PMMOVE and then check for collisions with BUMP() before the spot on the screen where the collision occurs has been displayed, resulting in a missed collision. Care must therefore be taken to ensure that the hardware has time to detect the collision.

One way to ensure that collisions are detected is to watch for a change in PEEK(20), which counts up at the end of every frame. Therefore, a reliable process for detecting collisions is as follows:

- Update and move players and missiles.
- Wait for a change in PEEK(20).
- Clear collisions with POKE 53278,0.
- Do some quick work.
- Wait for another change in PEEK(20).
- Check collisions.

Note that it isn't enough to just check PEEK(20) once, because that might happen right before it updates.

This procedure ensures that all objects are stable for a full frame to reliably detect collisions. However, it takes two frames per iteration, limiting the smoothness of animation to half-rate (25Hz or 30Hz). Faster processes are possible that permit 50/60Hz operation, but the timing of each phase has to be more carefully managed. The vertical region for the playfield takes about three-quarters of the frame time to display on an NTSC computer and three-fifths of the frame for PAL.

Error handling

TRAP (TR.)

Syntax: *TRAP aexp*

Sets a program line to jump to when an error occurs, or clears any such line if ≥32768. If a error trap has already been set, the new setting replaces the previous one.

Ordinarily, whenever an error occurs, BASIC prints an error message with the error number and line number where the error occurred, and then drops to immediate mode with a Ready prompt. Often, it is useful for the program itself to be able to handle errors more gracefully. TRAP allows the program to jump to an error handling routine instead. For instance, when trying to open a file, TRAP can be used to

display a readable error message to the user and continue the program.

When an error occurs and execution jumps to the TRAP line, three memory locations hold information about the error. PEEK(195) or ERR(0) retrieves the error that occurred, and PEEK(186)+256*PEEK(187), DPEEK(186), or ERR(1) retrieves the line number of the line where the error occurred. The TRAP setting is automatically disabled when the trap handler is executed, so TRAP must be used to reset it if error trapping is still desired.

An existing TRAP is also reset by NEW, but not by LOAD, RUN, or CLR.

The [BREAK] key cannot be trapped by TRAP and always drops to immediate mode with a stopped message.

ERR()

Syntax: *ERR*(0)
ERR(1)

Returns either the last error number (ERR(0)) or the line at which the error occurred (ERR(1)). These are useful in TRAP routines to more gracefully handle errors that occur.

ERR(0) is equivalent to PEEK(195), and ERR(1) is equivalent to DPEEK(186).

File management

Several statements are included in Altirra BASIC to allow manipulation of files without having to exit temporarily to the DOS menu. These are mainly intended to be used from immediate mode, but can also be used from a program in deferred mode.

DIR

Syntax: *DIR* [*sexp*]

Lists files on disk or a device. If no filename is specified, all files on D1: are listed (D1:*.*). Otherwise, files on the given device and pattern are listed. Note that a filename pattern must be included, so to list files on D2:, the string “D2:*.*” must be supplied and not just “D2:”.

ERASE

Syntax: *ERASE sexp*

Deletes a file.

This is a synonym for XIO 33,#7,0,0,filename.

PROTECT

Syntax: *PROTECT sexp*

Marks a file as protected (read-only). This is also called locking a file.

This is a synonym for XIO 35,#7,0,0,filename.

RENAME

Syntax: *RENAME sexp*

Renames a file. The string argument consists of the device name, original filename, and new filename, i.e.: “D1:OLD.BAS,NEW.BAS”.

This is a synonym for XIO 32,#7,0,0,filenames.

UNPROTECT

Syntax: *UNPROTECT sexp*

Marks a file as unprotected (read-write).

This is a synonym for XIO 36,#7,0,0,filename.

Exiting BASIC

BYE (B.)

Exits BASIC.

On the 400/800 OS (OS-A or OS-B), this will start the Memo Pad. On the XL/XE OS, this invokes the self-test.

Typically, there is not much use for this command. However, one use for it is to force a cold restart to boot a disk. Normally, pressing the RESET button on an XL/XE will simply restart BASIC, but if BYE is used to exit to the self test, RESET will then start the disk boot process. This avoids having to power off and on the computer.

DOS (DO.) / CP

Exits to DOS. CP is a synonym for DOS, named after versions of DOS that have a line-oriented command processor (CP) instead of a DUP.SYS-like menu.

DOS 2 provides a facility called MEM.SAV to preserve the current BASIC program while the DOS menu is active. If this file is present, the contents of the BASIC program are saved to the MEM.SAV file when the menu is loaded, and restored when BASIC is restarted. This allows the menu to be used without disturbing the current BASIC program. However, writing and reading MEM.SAV takes time, and Altirra BASIC provides utility commands like DIR and ERASE that can avoid the need to enter the DOS menu.

Some alternative DOSes don't have a MEM.SAV-like facility. If a MEM.SAV-like facility is not used, entering DOS via the DOS statement will typically cause the current program to be lost. Others can use

a ramdisk on a 130XE or other system with extended memory to save and restore the BASIC program much more quickly.

Memory access

Several statements and functions are provided to directly access memory in the system. This allows access to both software and hardware functionality not otherwise directly accessible from BASIC.

Memory addresses range from 0-65535, and each address corresponds to a memory location that holds a byte value from 0-255. PEEK and POKE are the classic methods to access memory directly, but Altirra BASIC supports more powerful memory access commands, such as DPEEK() and MOVE.

While direct memory access is a powerful technique, there are **no** checks on any addresses used. Improper memory access can corrupt the running program and its data, hang the system, or even cause corruption to floppy disks on subsequent disk accesses. Care should be taken when using these facilities.

One problem with direct memory access is finding an area of memory that is free for use. The base address of BASIC depends on whatever DOS is loaded, so the BASIC program may be at varying addresses. String arrays can be used to allocate memory, but move whenever variables are added or program lines are altered, making them unsafe for some uses. The upper half of page six ($0680-06FF in hex, or 1664-1791 in decimal) is unused by BASIC and frequently used as an area for machine language programs. In cases where INPUT or ENTER statements are not used with long lines, the lower half of page six ($0600-067F hex, 1536-1663 decimal) can also be used.

Another way to reserve memory is to use the LOMEM command to move up the BASIC program, reserving memory between DOS or the operating system and BASIC. However, because LOMEM clears the current program, it must be used before the program is loaded.

POKE

Syntax: *POKE aexp,aexp*

Changes the byte value at the given memory location. The first argument is the memory address and the second is the value to write.

PEEK()

Syntax: *aexp=PEEK(aexp)*

Returns the byte value at the given memory location.

DPEEK()

Syntax: *aexp=DPEEK(aexp)*

Returns the 16-bit integer stored at the given location.

DPEEK(addr) is equivalent to PEEK(addr)+256*PEEK(addr+1), but faster and easier to use.

DPOKE

Syntax: *DPOKE aexp,aexp*

Stores a 16-bit integer at the given location.

DPOKE addr, value is equivalent to POKE addr, value&$FF: POKE addr+1, (value&$FF00)/256, but faster and easier to use.

ADR()

Syntax: *aexp=ADR(sexp)*

Returns the starting address in memory of a string. This can be used with a string expression, literal, or variable. If a substring is used, i.e. A$(4) or A$(4,7), the address points to the start of the substring.

The string address may become invalid after it has been retrieved by ADR(). Addresses of string arrays or substrings of string arrays can be invalidated by CLR or adding/removing/changing lines in the program. Addresses of string literals will be invalidated by changes to program lines.

FRE()

Syntax: *aexp=FRE(aexp)*

Returns the amount of free memory, in bytes. The argument is ignored.

Free memory is defined as the amount of memory remaining for additional program storage, arrays, strings, the runtime stack, and text/graphics.

MOVE

Syntax: *MOVE aexp,aexp,aexp*

Moves (copies) a block of memory from one location to another. The three arguments are, in order: starting source address, starting destination address, and number of bytes to move. This is much faster than moving memory with PEEK and POKE statements.

By default, MOVE does an *ascending copy,* which means that the copy starts at the addresses given and advances to higher addresses. This is fine if the source and destination blocks do not overlap, or if the source address is higher than the destination address. It fails if the blocks overlap and the source address is lower than the destination address, though, because part of the source is overwritten before it can be moved. This results in part of the source block being repeated.

To solve this, MOVE also supports a *descending copy* by use of a negative length. In that case, the copy starts from the end of the source and destination blocks, and proceeds downward in memory, allowing the third case to work. Note that in this case, the addresses are still for the *start* of the source and destination blocks.

In summary:

- If the blocks do not overlap, either ascending or descending copy will work.
- If the blocks overlap and source > dest, use positive length (ascending copy).
- If the blocks overlap and source < dest, use negative length (descending copy).

Note that it is also possible to deliberately use the “wrong” method in order to fill memory with a pattern. For instance, MOVE adr, adr+1, len will fill memory with the value of PEEK(adr).

LOMEM

Syntax: *LOMEM aexp*

By default, BASIC uses all memory available, starting at the bottom of memory just above the OS or DOS, and all the way up to just before the screen. The LOMEM command allows the bottom to be raised up so that memory can be reserved between the OS/DOS and BASIC. Once LOMEM is executed, BASIC is reset to use the given address as the new bottom of memory.

For instance, when booting DOS 2.0S with BASIC, the bottom of BASIC memory is usually at 7420 ($1CFC). Executing LOMEM $2000 causes BASIC to only use memory starting at 8192 ($2000), reserving 772 bytes of memory from 7420-8191 ($1CFC-1FFF). This memory is then free for other uses, such as P/M graphics or USR() routines. The current value of LOMEM can be retrieved with DPEEK(128).

LOMEM can also be used to lower the bottom of BASIC memory, if it was previously raised. However, there are **no** checks on the value passed to LOMEM. Executing LOMEM 0, for instance, will crash the computer in short order.

USR()

Syntax: $aexp = USR\left(aexp[,aexp]...\right)$

Invokes a machine language routine by address. This is a powerful facility, as machine code can run much faster and do more than BASIC code can.

The first argument to USR() is the starting address of the routine to invoke, followed by zero or more optional arguments. On entry, the byte on the top of the 6502 stack will contain the number of arguments passed to the function, followed by any 16-bit integer arguments in high-then-low order, starting with the leftmost argument. On exit, the routine should place the desired return value as a 16-bit integer at FR0 (212 / $D4) before returning to BASIC with RTS.

Because USR() always pushes a minimum of one byte onto the stack that must be removed (argument count), it is not generally possible or safe to use it to invoke routines that were not originally written to be called with USR(). There are no such routines by default, so one must be provided by the BASIC program, either in a string or written to memory with POKE, DPOKE, or BGET statements.

Here is an example of an assembly language routine compatible with USR() to compute the bitwise left

shift of an integer number, $a \times 2^b$ mod 65536.

Assembly language:

```
      ;pop argument count (not used)
      pla

      ;pop the value to shift into FR0
      pla
      sta   fr0+1
      pla
      sta   fr0

      ;pop the shift count
      pla
      tay
      pla
      tax

      ;return 0 if we're shifting by 16 bits or more
      tya
      bne   is_zero
      cpx   #16
      bcs   is_zero

      ;check if shifting by zero bits
      txa
      beq   no_shift

shift_loop:
      asl   fr0
      rol   fr0+1
      dex
      bne   shift_loop
no_shift:
      rts

is_zero:
      ;zero FR0
      jsr   zfr0
      rts
```

Example BASIC program using it:

```
10 FOR I=0 TO 28:READ X:POKE 1664+I,X:NEXT I
20 PRINT USR(1664,1000,10)
30 DATA 104,104,133,213,104,133,212,104,168
40 DATA 104,170,152,208,15,224,16,176,11,138
50 DATA 240,7,6,212,38,213,202,208,249,96
RUN
40960
```

Error codes

BASIC errors

2 – Out of memory

Not enough free memory is left to load a program, add a program line, allocate an array, or record the location of a FOR or GOSUB statement. The FRE() function can be used to monitor the amount of free memory.

Note that if Error 2 results from a runtime stack overflow (FOR/GOSUB), a CLR statement may be needed to clear the runtime stack before you can do anything. Otherwise, system memory may be so full that there is no room for the statements you are trying to enter.

The GRAPHICS statement can also trigger an out of memory error, but in that case the system error 147 is triggered instead.

3 – Value error

An invalid value was passed to a statement or function, such as LOG(-1).

4 – Too many variables

The maximum of 128 program variables has been reached and there is no room for another variable. Every unique variable name occupies one of these variable slots. A, A(), and A$() are all distinct variable names.

Removing all references to a variable does not remove the variable itself. The only way to remove references to unused variables is to LIST the program to a text listing, reset the program and variable tables with NEW, and re-enter it with the ENTER statement.

5 – Bad string length

An attempt was made to DIMension a string array to length 0, or to reference a substring with invalid indices. This includes a starting index of 0, starting index beyond the string length, or ending index before starting index.

6 – Out of data

A READ statement attempted to read data beyond the end of the last DATA statement.

7 – Value greater than 32K

A statement or function that accepts integers was given an unsupported value between 32768 and 65535. This includes string indices, line number expressions, and arguments to the game controller functions.

8 – Input error

A READ or INPUT statement encountered input that failed to parse, such as string input when a number was expected.

9 – DIM error

A DIM statement attempted to initialize an array to an invalid size of 0, or larger than 64KB.

10 – Argument stack overflow

Currently, this error is not used by Altirra BASIC.

11 – Math error

A numeric expression resulted in overflow, underflow, or another error such as division by zero or invalid input (LOG(-1)). An overflow occurs when a result exceeds $\pm 10^{98}$, while an underflow results when the magnitude is below $\pm 10^{98}$. Most operators will signal overflows, but not all will signal underflows.

12 – Line not found

A GOTO, GOSUB, ON...GOTO/GOSUB, IF, TRAP, or CONT statement encountered a line number reference to a line that doesn't exist.

13 – No matching FOR

A NEXT statement was executed that could not be matched to a FOR statement with the same variable.

14 – Line too long

An entered line either produced too long of a tokenized line (exceeds 255 bytes). Some tokens, particularly small numbers, take many more bytes than in the original text, and so it is possible for a syntactically valid line to take too many bytes when parsed.

Note: Atari BASIC can issue this error in some cases when its internal parsing stack overflows while parsing a complex expression. Altirra BASIC can generally parse more complex expressions than Atari BASIC.

15 – GOSUB or FOR gone

The line that originally held a remembered GOSUB or FOR statement has been deleted.

In Altirra BASIC, this error is issued when CONT is executed, rather than later when the NEXT or RETURN statement is executed.

16 – Bad RETURN

A RETURN statement was executed when there is no matching GOSUB statement to return to.

17 – Syntax error

Program execution reached a line that contains a syntax error.

Altirra BASIC does not store syntax errors, so this cannot normally occur unless the program was originally saved from Atari BASIC with a syntax error in it.

18 – Invalid string

The VAL() function was invoked with a string that did not parse to a valid number.

19 – Load error

A LOAD or CLOAD statement was executed and the program was read, but was either too long to fit in memory or was not a binary saved BASIC program.

20 – Bad device number

An invalid I/O channel number was used, such as #8, or #0 for a statement other than PRINT or INPUT.

28 – Invalid structure

A broken IF...ENDIF or IF...ELSE...ENDIF block structure was found during program execution, such as an IF that has no matching ENDIF.

System errors

The most common system errors are listed below. Some DOSes may report their own specific error codes not listed here.

129 – IOCB in use

An attempt was made to OPEN an I/O channel that is already open.

130 – unknown device

A statement was executed with a filename that references an unknown device, such as A:FILE when no A: device has been registered.

131 – write only

A get operation was attempted on an I/O channel that is open only for read-only access.

132 – invalid command

An I/O command was attempted on an I/O channel that the device does not support, such as an unknown XIO command.

133 – not open

An I/O operation was attempted on an I/O channel that isn't open.

134 – invalid IOCB number

An I/O operation was attempted with an invalid I/O channel number, not corresponding to one of the eight I/O channels.

This error ordinarily should not occur with BASIC statements, as BASIC validates the I/O channel number first.

135 – read only

A put operation was attempted on an I/O channel that is open only for write access.

136 – end of file

A get operation, such as INPUT, GET, or BGET, reached the end of the stream before the operation completed. Note that an operation that completes exactly at the end of the file does not trigger this error.

137 – truncated record

A record-based get command encountered a line that was too long. In BASIC, this occurs if the INPUT statement reads a line that is longer than 255 characters, including the EOL. Note that this does not occur if INPUT reads into a string array that is too short; in that case, the extra data is discarded.

138 – device timeout

No response was encountered when attempting a serial bus operation. This can happen when referencing a disk drive that doesn't exist – D3: with only two disk drives – or trying to use a disk drive that is in the middle of formatting a disk.

139 – device NAK

A serial device returned a NAK response to a command, which means that the command was unsupported or invalid.

140 – serial framing error

A framing error was encountered while sending or receiving data from a serial device. This typically means an unstable connection between the computer and the serial device. With disk drives and a high-speed OS, it can mean that the transfer rate is too high.

141 – cursor out of range

The cursor was moved out of valid (X,Y) coordinates for the current screen on either the E: device (#0) or S: device (#6).

142 – serial overrun

The computer did not read data quickly enough from a serial device and some data was lost. With disk drives, this typically means that a high-speed transfer is too fast for the display, especially if custom vertical blank (VBI) or display list (DLI) routines are used.

143 – serial checksum error

A checksum error was encountered while receiving data from a serial device, which means that the data transferred with errors. This probably means an unstable connection between the computer and the serial device.

144 – device error

A serial device reported an error during an operation. This can happen with disk operations if the disk drive has difficulty reading or writing to a disk. Attempting to write to a write-protected disk can also cause this error.

145 – bad screen mode

A GRAPHICS command was executed with a screen mode that the computer's OS doesn't support. For the old OS-A or OS-B ROMs on a 400/800, this happens if modes 12-15 are requested.

146 – not supported

An XIO command was issued on a device that doesn't support the command.

147 – out of memory

Not enough free memory to complete the operation. This can happen if the GRAPHICS statement is executed with a screen mode that requires more memory than is free, e.g. GRAPHICS 8 with FRE(0) = 4000.

DOS errors

160 – bad disk drive number

Attempted to use a disk drive number higher than supported by the loaded DOS, such as D4: when only D1: and D2: are supported.

161 – too many open files

Attempt to open a file on disk when the limit of open files in DOS has already been reached. For Atari DOS 2.x, the limit is usually three simultaneously open files.

162 – disk full

No space left on the disk.

164 – internal file number mismatch

DOS encountered broken file structures on the disk during an I/O operation, specifically a file linking to sectors that don't belong to the file. This can indicate a damaged disk that requires repair. It can also happen if POINT is executed with an invalid position.

165 – filename error

A filename was used that doesn't conform to DOS naming rules, such as a name with characters that are not allowed ($ with DOS 2.0).

166 – point data length error

One of the parameters to the POINT command is invalid and doesn't correspond to a valid file position.

167 – file locked

Attempted a write operation to a file that has been locked with PROTECT or XIO 35.

169 – directory full

Attempted to create a file on a disk that cannot hold any more. With DOS 2.x, only 64 files can be stored on a disk. This can happen even if there are free sectors on the disk.

170 – file not found

A disk operation specified a file that doesn't exist on the disk.

Defined memory addresses

Below is a list of memory locations that are defined in Altirra BASIC and can be read with the PEEK() or DPEEK() functions, or where noted, modified with POKE/DPOKE.

186-187 / $BA-BB (STOPLN)

Contains the line number where the last error occurred. This can also be retrieved with ERR(1).

195 / $C3 (ERRSAV)

Contains the last error number. This can also be retrieved with ERR(0).

201 / $C9 (PTABW)

Width of a tab stop, as used by the PRINT statement when comma separators are used. Reset to 10 on boot and after a NEW.

203-211 / $CB-D1

These memory locations are guaranteed not to be used by Altirra BASIC. You can freely use them with PEEK/POKE and in USR() routines.

212-217 / $D4-D9 (FR0)

Floating point register 0, used as the primary accumulator for the floating-point library. Machine language routines invoked by USR() should place the return value as a 16-bit unsigned integer in locations $D4 (low byte) and $D5 (high byte) before returning.

The entire FR0 register, as well as all other locations used with the floating-point library, may be used temporarily and overwritten by the USR() program until control returns to BASIC.

756 / $2F4 (CHBAS)

Sets the character set used in text modes; normally set to 224 ($E0). In graphics modes 1 and 2, it can be changed to 226 ($E2) to select the lowercase character set.

765 / $2FD (FILDAT)

POKE the desired color for fill operations (XIO 18) into this location.

1408-1663 / $580-67F (LBUFF)

Used by BASIC for various purposes, including line buffering for INPUT statements and temporary storage during trancendental functions. The second half of this buffer (1536-1663 / $600-67F) can be

used for persistent storage by the user program if INPUT or ENTER is not used with lines longer than 127 characters. The entire buffer may be used for temporary storage by a USR() routine until control returns to BASIC.

1664-1791 / $680-6FF

Unused by OS, DOS, and the BASIC interpreter; available for user program use. This is a popular area to place small machine-language programs.

Note: The lower half of page six at 1536-1663 ($0600-067F) is also commonly used. This area is not entirely safe as the BASIC interpreter will use it whenever the INPUT or ENTER statement reads lines longer than 128 characters long, but can be used otherwise.

Implementation limits

Expression nesting

Expressions may contain up to 32 levels of precedence nesting.

FOR/NEXT and GOSUB nesting

Limited only by available memory, as long as there is room for the runtime stack.

Line length

A line may contain up to 255 characters before tokenization and up to 255 bytes after tokenization.

Numeric array size

Limited only by available memory. Normally, the largest numeric array that can be created is 6315 elements, with no program loaded and the highest possible FRE(0) value of 37902 bytes.

Program length

Limited only by available memory.

String array size

The longest string array that can be created is 32767 characters.

Variable count

Programs may contain up to 128 uniquely named variables, including numeric variables, numeric arrays, and string arrays.

Variable name length

Variable names may be up to 128 characters long.

Special techniques

Optimizing for size

Numeric constants

Constant numbers in a program, i.e. 12, are big – they take one byte for the token and six bytes for the number. Constant numbers can therefore contribute a large amount to the overall size of a tokenized program. Since variable references only take a single byte, a common trick is to declare variables for commonly used constants, such as 0, 1, and the line numbers of frequently used subroutines.

Numeric arrays

Numeric arrays require 6 bytes per element. This can add up quickly for large numeric arrays. An alternative to using arrays is strings, which require one byte per character. When only values 0-255 are required, the values can be stored as characters in a string array.

Also, remember that arrays in Altirra BASIC start with subscript 0, not 1. DIM A(5) allocates *six* elements.

Variables

Each variable takes 8 bytes for the value of the variable, along with one byte for each character in the variable's name. Shortening variable names and reusing variables can therefore save memory.

Note that these costs don't scale with the number of times a variable is used. Each use of a variable only takes one byte, no matter how long or short the name of the variable is.

Cleaning out unused variables by LISTing a program out to disk and re-ENTERing it can also recover some memory.

DATA statements

DATA statements are a reasonably compact way to store numbers, as the contents of a DATA statement are stored as text. Small numbers therefore can be stored in 2-4 bytes instead of the 7+ bytes that constants would require. However, it is still more compact to use strings, if each value can be encoded into a character.

Chaining program execution

One program can chain to another by means of the RUN statement with a filename. This can remove seldom-used portions of a program from memory, requiring only the core to be resident at the most memory hungry point. Variables and other data must be passed between the programs by other means, such as a static area of memory (upper half of page six), a file, or even the screen.

Some programs use the ENTER and CONT commands to create overlays, where *portions* of a program are deleted and inserted during execution. This can be a more flexible way to save memory, but it is much slower than RUN as program text must be parsed. Also, it is important to note a limitation in Altirra BASIC's CONT statement, which is that any line on the runtime stack must exist when CONT is

executed. RESTORE should also be used before a READ statement after such edits, or the READ statement may read garbage data.

Combining program lines

Each program line occupies three bytes in memory, and thus splicing together lines with : can save a bit of memory.

Optimizing for speed

Abbreviated statement names

Abbreviating statement names, i.e. GR. instead of GRAPHICS, does not affect program speed. All abbreviated statement names are always expanded to the full name anyway and stored as the same token in the tokenized program.

Variable lookups

All variables are referenced by index in the tokenized text. This means that, unlike some other BASIC interpreters, the length of variables names and the order in which they are declared or first used does not matter. All variables are accessed in constant time in Altirra BASIC without any searching.

Line number lookups

Whenever a line number is referenced, such as in a GOTO or RESTORE statement, BASIC must search for a line with that line number. This can take a lot of time, particularly in big programs with a lot of lines.

Altirra BASIC contains some optimizations to reduce the amount of time spent searching for lines: it caches the return locations for FOR and GOSUB statements, and starts searching from the current line instead of the beginning of the program when appropriate. However, line searches can still take a noticeable amount of time.

Two optimizations can be applied to speed up line searches. First, lines can be combined by placing multiple statements on the same line. The search time is determined by the number of lines checked and not their length, so reducing the number of lines speeds up the search even though the lines are longer.

Second, moving target lines earlier in the program can also speed up the search. A commonly used technique is to place frequently executed code at the lowest line numbers and seldomly used code at higher line numbers.

Caution: Some code movements that are advantageous under Atari BASIC may not be under Altirra BASIC due to the starting point optimization. In particular, changing 1000 GOTO 1010 to 1000 GOTO 990 can be faster under Atari BASIC, but slower under Altirra BASIC. This is because in the former case, Altirra BASIC can start the line search at 1000, whereas in the latter it must start from the beginning of the program as Atari BASIC does.

Computed GOTOs

Altirra BASIC allows expressions for all line numbers except for the IF statement, which is trivially worked around with IF...THEN GOTO. This can often be exploited to avoid a large amount of conditional logic, such as moving a character in response to joystick movement by means of GOTO 10+STICK(0). This incurs a line lookup, but the line lookup is still much faster than executing multiple BASIC statements.

IF..THEN vs. block IF

When multiple statements are controlled and a condition is false, IF..THEN is faster than block IF. This is because the interpreter must scan a statement at a time for IF blocks, whereas for IF..THEN the target is specified by line number and therefore proceeds a line at a time. The block IF is particularly slower when scanning past nested IF instructions as it must also parse the condition.

Strength reduction

Often an expression can be simplified to use faster operators: x*2 = x+x, x^2 = x*x. The latter is especially lucrative as x^y is computed internally as 10^(clog(x)*y), which is rather slow. However, be aware that the results may not be exactly the same (although usually more accurate).

Moving and filling memory

While PEEK and POKE can be used in a loop to copy memory around, doing so is very slow due to loop overhead in the interpreter.

The classic way to move memory around quickly is to use strings. String manipulation is much faster in BASIC than moving bytes or characters manually. There is no requirement that a string array actually hold text, as it can hold arbitrary binary data, including null (zero) bytes and end-of-line (EOL) characters. One use for string manipulation is to quickly move player/missile (P/M) graphics.

In Altirra BASIC, the MOVE command can also be used to quickly move and fill memory.

Recovering a protected program

Occasionally there is a need to recover a BASIC program that has been protected to prevent LISTing. Such programs cause the BASIC interpreter to lock up when any command is entered in immediate mode after the program is loaded, even with a warm reset. Typically, such programs can be recovered with the following steps:

```
TRAP 32750
LOAD "D:PROGRAM.BAS"
32750 DPOKE 140,DPEEK(138)+3:DPOKE DPEEK(138),32768:POKE
DPEEK(138)+2,3:END
-1
```

The TRAP statement **must** be executed before the protected program is loaded. The added line 32750 transforms itself into the new immediate mode line, removing any deliberately damaged lines after it.

The final -1 invokes line 32750 by causing an Error 3 without triggering a line lookup for line 32768, which is what happens when commands are entered in immediate mode and what normally causes the lockup.

This trick can also be done in Atari BASIC, but is more difficult without adding a variable to the program and without DPEEK().

Alternatively, if only a readable listing is required, the injected line can simply be used to trigger the LIST command:

```
TRAP 32750
LOAD "D:PROGRAM.BAS"
32750 LIST "D:PROGRAM.LST",0,32749:NEW
-1
```

Compatibility with other BASICs

Compatibility with Atari BASIC

Saved program compatibility

A program saved from Atari BASIC can be loaded into Altirra BASIC, without issues.

Programs saved from Altirra BASIC can be loaded into Atari BASIC, as long as only features compatible with Atari BASIC are used. Features specific to Altirra BASIC, such as hexadecimal constants ($12AF) and new statements (MOVE) cannot be used in a program to be loaded back into Atari BASIC. Attempting to do so may produce unpredictable results, including the Atari BASIC interpreter crashing.

The following is a list of tokens (statements, operators, and functions) that are compatible with Atari BASIC:

+, -	CONT	GRAPHICS	PADDLE()	SAVE
*, /	COS()	IF	PEEK()	SETCOLOR
<, >, <=, >=	CSAVE	INPUT	PLOT	SGN()
<>, =	DATA	INT()	POINT	SIN()
?	DEG	LEN()	POKE	SOUND
ABS()	DIM	LET	POSITION	SQR()
ADR()	DOS	LIST	POP	STATUS
ASC()	DRAWTO	LOAD	PRINT	STICK()
ATN()	END	LOCATE	PTRIG()	STOP
BYE	ENTER	LOG()	PUT	STR$()
CHR$()	EXP()	LPRINT	RAD	STRIG()
CLOAD	FRE()	NEW	READ	TRAP
CLOG()	FOR	NEXT	REM	USR()
CLOSE	GET	NOTE	RESTORE	VAL()
CLR	GOSUB	ON..GOSUB	RETURN	XIO
COLOR	GO TO	ON..GOTO	RND()	
COM	GOTO	OPEN	RUN	

The following tokens are **not** compatible with Atari BASIC:

```
%, !, &          DIR             ERASE           MISSILE         PMMOVE
$hhhh (hex)      DPEEK()         ERR()           MOVE            PROTECT
BGET             DPOKE           HEX$()          PMCLR           RENAME
BPUT             ELSE            HSTICK()        PMCOLOR         UNPROTECT
BUMP()           ENDIF           LOMEM           PMGRAPHICS      VSTICK()
```

The IF statement needs special mention as it has two forms that share the same token. The IF...THEN form is compatible with Atari BASIC, while the IF block form is not.

PRINT/LPRINT token checking bug (Revision A only)

Atari BASIC rev. A has a bug in its code to check whether a PRINT or LPRINT statement ends in a dangling semicolon or comma, causing it to erroneously also check the last byte of string and numeric literals as if it were an expression token. As a result, printing a literal string ending in Control-R or Control-U, or constant numbers with the digits 12 or 15 in the last mantissa byte (e.g. 1.00000012), will suppress the ending newline as if a semicolon was used. This bug was fixed in rev. B and Altirra BASIC doesn't have this bug.

Save bug (Revision B only)

Revision B – and *only* rev. B – of Atari BASIC has a bug where it extends the BASIC program by 16 bytes every time it is saved. This is reflected as growth in the first segment of the program that holds the argument stack area. Altirra BASIC does not have this bug, but it detects and corrects this problem when the program is re-saved. The error is kept on load to preserve memory layout.

Array sizes

Atari BASIC limits the size of a numeric array to 32,767 bytes or less. Altirra BASIC does not have this limitation. However, it does still limit the size of a string array to 32,767 bytes.

FRE() bug

The FRE() function in Atari BASIC actually reports one byte less than is actually available between the top of the runtime stack and the bottom of the screen. However, Atari BASIC doesn't actually allow that last byte to be used, so this is consistent. Altirra BASIC replicates this bug.

CHR$() / STR$() bug

Due to a lack of a dynamic string allocation system, Atari BASIC uses fixed buffers for the results of the CHR$() and STR$() functions. The result is incorrect results if either function is used more than once in the same expression, as the second invocation will overwrite the buffer used by the first, and then the second string is used twice. Because of implementation constraints, Altirra BASIC has the same behavior.

Unary operator parsing

Atari BASIC only allows one unary operator preceding a value. For instance, -X is allowed, but --X is

not. When a constant is used, one of the leading signs can be absorbed by the number parser, so --1 is allowed, but ---1 is not.

Altirra BASIC can both parse and execute multiple leading unary operators. A program containing such constructs, however, is not guaranteed to execute properly under Atari BASIC. Parentheses can be used if necessary to separate the unary operators.

Negative numbers are generally encoded more compactly by Altirra BASIC, which encodes them directly as a negative number instead of a unary minus operator followed by a positive number.

CONT limitations

Atari BASIC caches the program location for the next DATA statement after a READ and does not update this cache when the program is edited. If a subsequent READ occurs without a RESTORE, garbage can be read as data. RESTORE is therefore required to reset the DATA location. This is also true in Altirra BASIC.

The CONT statement in Atari BASIC can continue program execution even if some of the FOR or GOSUB statements being tracked in the runtime stack no longer exist. This produces an error if those lines still don't exist when a matching NEXT or RETURN is executed for those entries. Altirra BASIC checks this immediately when CONT is executed and will fire an error immediately if one of the referenced lines is missing.

Invalid IOCB numbers

For compatibility, Altirra BASIC replicates some of the I/O channel number validation bugs in Atari BASIC. A notable case is that most commands prohibit use of IOCB #0, but due to overflow will allow its use via IOCB #16. INPUT #16 is particularly used as it bypasses the ? prompt normally emitted.

Memory layout differences

Like Atari BASIC, Altirra BASIC is an 8K left-slot cartridge with no cartridge initialization routine and entry point at the cartridge base address of 40960 ($A000).

Zero page usage for Altirra BASIC is different than for Atari BASIC, except for specific locations matched for compatibility. Programs that attempt to reuse portions of the BASIC interpreter's page zero space or read/modify undocumented variables may fail. Altirra BASIC is compatible with usage of memory locations documented in the Atari BASIC manual.

Program memory layout is mostly the same between Altirra BASIC and Atari BASIC. The variable name table, variable value table, statement table, and string/array tables have the same format and order. However, the argument stack and runtime stacks have different formats.

Compatibility with OSS Basic XL/XE

Saved program compatibility

The Altirra BASIC language is a subset of that supported by OSS Basic XL and intentionally uses the same token values for compatibility. An Altirra BASIC program can therefore be loaded and run by

Basic XL or Basic XE.

Similarly, a Basic XL/XE program that uses only the subset of tokens supported by Altirra BASIC can be loaded and run by the latter. Attempting to load a Basic XL/XE program with unsupported functionality, however, will lead to undefined results.

IOCB #0

Basic XL allows IOCB #0 to be used with some commands where prohibited by Atari BASIC. Altirra BASIC follows the latter, still prohibiting use of IOCB #0.

TRAP and the [Break] key

Basic XL/XE allows TRAP to intercept the [Break] key, if enabled with the SET statement. In Altirra BASIC, [Break] is always untrappable.

Compatibility with Turbo-BASIC XL

Saved program compatibility

Programs can be loaded and saved between Altirra BASIC and Turbo-BASIC XL, but in general, only the common Atari BASIC subset can be exchanged. This is because the two languages differ in extensions and even when the same extensions are present, such as MOVE, the token values are different. Attempting to LOAD a program saved in one interpreter that was SAVEd from the other with an incompatible feature will likely lead to an interpreter crash.

However, in cases where the same extended statement is available in both languages with the same syntax, i.e. MOVE, the program can be converted by LISTing it out from one and ENTERing it into the other.

Program format

Memory layout and binary format

Altirra BASIC splits main memory into eight different regions:

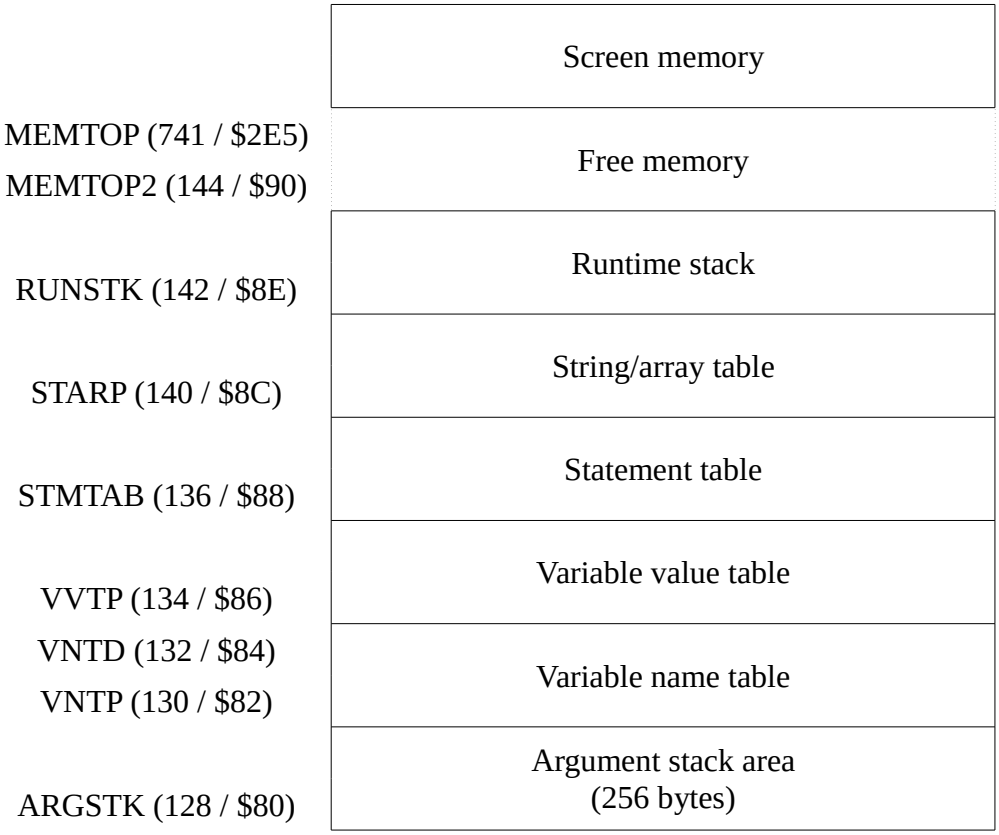

Table 5: Altirra BASIC memory layout

All pointers are 16-bit. Screen memory grows down from the top (typically 49151 / $9FFF), and BASIC memory grows up from the bottom.

Typically the argument stack area will start at 1792 / $0700 if running without DOS and around 7420 / $1CFC when Atari DOS 2.x is loaded. On a 48K system, the top of free memory (MEMTOP) will be around 39667 / $9C1F when the regular GR.0 text screen is active.

The binary format used by SAVE and LOAD is a memory image of the vector table (14 bytes) followed by the memory from VNTP to STARP. Thus, the variable name table, variable value table, and statement table are saved, while the argument stack area, string/array table, and runtime stack are not. Also, while the variable value table is saved, only the type and token information is used – the variable values are cleared on load.

The vector table is stored as relative to ARGSTK, such that all values stored as offsets from the bottom of the argument stack area. This means that the first four bytes of the saved vector table should always be $00 00 00 01. (The VNTP offset in the third and fourth bytes may be greater for some programs saved by Atari BASIC revision B.) The sixth pointer value between STMTAB and STARP is STMCUR ($8A), which points to the current statement. It is saved out but not used on load.

Variable name table

The variable name table (VNT) consists of the names of each variable, in order that they were added to the program. The base of the VNT is pointed to by VNTP (130 / $82) and the end by VNTD (132 /

$84). Each variable name is stored as ATASCII except for the last byte of each name, which is stored with bit 7 set. A $00 byte terminates the VNT at the VNTD address.

For numeric arrays, the variable name ends with a (. For string arrays, it ends in $.

The VNT is only used during LISTing and parsing, but not during execution. Some commercial programs exploit this by storing invalid data in the VNT.

Variable value table

The variable value table (VVT) holds the current value of each variable, and has an 8-byte entry for each variable in the VNT. The VVT extends from the base at VVTP (134 / $86) to just before STMTAB (136 / $88).

The first byte of each VVT entry holds type information about the variable:

$00	Numeric
$40	Numeric array, undimensioned
$41	Numeric array, dimensioned
$80	String array, undimensioned
$81	String array, dimensioned

Table 6: Variable types

The second byte holds the variable index for the variable. It is always $00 for the first entry, $01 for the second, etc.

The remaining six bytes are interpreted differently depending on the type of variable:

Type	**Bytes**					
	0	**1**	**2**	**3**	**4**	**5**
Numeric	Floating-point number					
Array	Storage offset		First dimension limit		Second dimension limit	
String array	Storage offset		Length		Capacity	

Table 7: Variable entry format

The storage offset for arrays and string arrays is a relative byte offset from STARP (140 / $8C), where 0 is assigned for the first array dimensioned and thus placed at the beginning of the string/array area. This relative offset is converted to an absolute address at runtime by adding STARP to the relative offset.

For numeric arrays, the original dimensions are stored as limits, which are one greater than the values that were passed in the DIM statement. Therefore, the limits stored are counts, rather than maximum subscript values. For one-dimensional arrays, the second dimension has a limit of 1.

Statement table

The statement table, pointed to by STMTAB (136 / $88) and ending before STARP (140 / $8C), contains all lines in the program. It consists of each program line stored back-to-back, in order of increasing line number.

Each line begins with a two-byte line number and a length byte, where the length byte contains the number of bytes in the entire statement, including the line number and length byte. Thus, it is always at least three for a valid program. Adding the length to the starting address of the line gives the starting address of the next line.

A valid line always has at least $03 in the length byte. If the length byte is changed to $00, it can cause BASIC to lock up when it encounters this line during a line search. This is a common technique for protecting BASIC programs, as it prevents any statement from executing in immediate mode once the program is LOADed.

The length byte is followed by one or more statements. Each statement starts with an end position byte containing the byte offset from the start of the line to the beginning of the next statement. After that is a statement token, and finally zero or more expression tokens.

The following is a sample statement table:

```
;10 FOR I=1 TO 100:PRINT I:NEXT I
1E24: 0A 00                      Line 10
      1F                         Line length
      17                         Offset of next statement
         08                      FOR
         83                      I
         2D                      =
         0E 40 01 00 00 00 00    1
         19                      TO
         0E 41 01 00 00 00 00    100
         14                      End of statement (:)
      1B                         Offset of next statement
         20                      PRINT
         83                      I
         14                      End of statement (:)
      1F                         Offset of next statement
         09                      NEXT
         83                      I
         16                      End of line
;32768 LIST
1E43: 00 80                      Line 32768
      06                         Line length
      06                         Offset of next statement
         04                      LIST
         16                      End of line
```

The immediate mode line is stored as line 32768, at the end of the statement table. This line is always present and is even SAVEd along with the rest of the program.

Floating-point (real) numbers

Floating point numbers are stored in the standard 6-byte format used by the OS math pack, consisting of a biased-integer exponent byte and a five byte, ten-digit binary coded decimal (BCD) mantissa.

The exponent byte is first and indicates a power of 100, biased by 64 so that 64 means an exponent of 100^0 and 65 = 100^1. The exponent is constrained to be within ±49, encoded as 15-113 ($0F-71). Additionally, bit 7 contains a sign bit and is set if the value is negative.

The mantissa is stored as BCD digit pairs ($00-99), with the most significant digit pair first. The first pair is to the left of the decimal point (×1), and the successive bytes are the ×0.01, ×0.0001, ×0.000001, and ×0.00000001 positions. The number is always normalized so that the most significant digit pair is non-zero, unless the value is zero in which case **both** the exponent byte and the first mantissa byte are $00.

Examples:

```
00 00 00 00 00 00 = 0
40 01 00 00 00 00 = 1.0
40 03 14 15 92 65 = 3.14159265
44 12 34 56 78 91 = 1.234567891E+09
BF 01 00 00 00 00 = -0.01
```

Statement tokens

$00 REM	$0F CONT	$1E ON	$2D POSITION	$43 BPUT
$01 DATA	$10 COM	$1F POKE	$2E DOS	$44 BGET
$02 INPUT	$11 CLOSE	$20 PRINT	$2F DRAWTO	$46 CP
$03 COLOR	$12 CLR	$21 RAD	$30 SETCOLOR	$47 ERASE
$04 LIST	$13 DEG	$22 READ	$31 LOCATE	$48 PROTECT
$05 ENTER	$14 DIM	$23 RESTORE	$32 SOUND	$49 UNPROTECT
$06 LET	$15 END	$24 RETURN	$33 LPRINT	$4A DIR
$07 IF	$16 NEW	$25 RUN	$34 CSAVE	$4B RENAME
$08 FOR	$17 OPEN	$26 STOP	$35 CLOAD	$4C MOVE
$09 NEXT	$18 LOAD	$27 POP	$36 [let]	$4D MISSILE
$0A GOTO	$19 SAVE	$28 ?	$37 ERROR-	$4E PMCLR
$0B GO TO	$1A STATUS	$29 GET	$3C ELSE	$4F PMCOLOR
$0C GOSUB	$1B NOTE	$2A PUT	$3D ENDIF	$50 PMGRAPHICS
$0D TRAP	$1C POINT	$2B GRAPHICS	$3E DPOKE	$51 PMMOVE
$0E BYE	$1D XIO	$2C PLOT	$3F LOMEM	

Statement tokens $00-37 are compatible with Atari BASIC. Tokens $38-51 are compatible with BASIC XL/XE.

Most statement tokens are followed by one or more expression tokens. The REM ($00), DATA ($01), and syntax error ($37) tokens are exceptions as they are followed by raw text. Altirra BASIC lists and executes the syntax error ($37) token if found in a loaded program, but never generates it during parsing.

Except for the text after the above three tokens, spaces are not stored in the token stream. They are implicit based on the token pattern and deduced by BASIC when LISTing the tokenized program.

$36 is the implicit LET token, used for bare assignments with no preceding statement name.

Expression tokens

```
$0D hex number   $20 <= [numeric] $30 <> [string]  $40 ASC          $50 INT
$0E dec number   $21 >  [numeric] $31 >= [string]  $41 VAL          $51 PADDLE
$0F str literal  $22 =  [numeric] $32 <  [string]  $42 LEN          $52 STICK
$12 ,  [args]    $23 ^            $33 >  [string]  $43 ADR          $53 PTRIG
$14 stmt. end    $24 *            $34 =  [string]  $44 ATN          $54 STRIG
$15 ;            $25 +  [binary]  $35 +  [unary]   $45 COS          $56 %
$16 line end     $26 -  [binary]  $36 –  [unary]   $46 PEEK         $57 !
$17 GOTO         $27 /            $37 (  [string]  $47 SIN          $58 &
$18 GOSUB        $28 NOT          $38 (  [array]   $48 RND          $5A BUMP(
$19 TO           $29 OR           $39 ( [arry DIM] $49 FRE          $5C HEX$
$1A STEP         $2A AND          $3A ( [function] $4A EXP          $5E DPEEK
$1B THEN         $2B (            $3B ( [str. DIM] $4B LOG          $60 VSTICK
$1C #            $2C )            $3C , [arry/str] $4C CLOG         $61 HSTICK
$1D <= [numeric] $2D =  [num asn] $3D STR$         $4D SQR          $62 PMADR
$1E <> [numeric] $2E =  [str asn] $3E CHR$         $4E SGN          $63 ERR
$1F >= [numeric] $2F <= [string]  $3F USR          $4F ABS
```

Expression tokens $0E-54 are compatible with Atari BASIC. Tokens $0D and $56-63 are compatible with BASIC XL/XE.

The expression tokens overlap the statement token values, so the proper context must be tracked as to whether tokens should be decoded as statement or expression tokens. This is straightforward as statements always begin with a statement token and are then followed by expression tokens, even in the case of an implicit LET, i.e. X=1. Expression tokens are stored in the order that they are parsed and listed (infix) and not how they should be executed (postfix), so operator evaluation order has to be worked out at runtime by precedence. There is always at least one expression token after a statement token, except for the three special tokens that use raw text ($00, $01, $37).

The $0D and $0E tokens are followed by a six-byte floating point number and are treated the same during execution, but simply LISTed differently. The $0F string literal token is followed by a length byte and then the string literal value. The length byte is the number of bytes in the string literal, not counting the length byte itself.

Several tokens have the same textual representation but different meanings. The relational operators have dual token forms for string and numeric comparisons. $2D and $2E are used for the = in an assignment; $2D is also used for the = in a FOR statement. $12 is the comma used for separating statement arguments, while $3C is used to separate function arguments and array subscripts.

The open parenthesis tokens are the most numerous. $3A starts a function's argument list; $39 and $3B precede the subscripts for numeric and string arrays in a DIM statement, respectively; $38 and $37 precede subscripts for numeric and string arrays outside a DIM statement (or in a subexpression within one!). $2B is the plain open-parens for grouping expressions. All share the one $2C token for the close parens.

The $5A token for BUMP(is unique as it doubles as both the function and the open parens token. The

$3A token that normally follows a function token and supplies the (character does not follow the $5A BUMP(token.

The end of statement token ($14) and end of line token ($16) generally occur at the end of each statement, where $14 is used except for the last statement in a line, where it is replaced by $16. The $14 token represents the : used to connect statements. Therefore, it is absent in the special case of an IF...THEN statement followed by another controlled statement, which ends immediately after the THEN ($1B) token with no $14 token afterward. It is present in the case of an IF...THEN followed by a line number and then a statement afterward, a consistent but useless case as the statement after the line number is never executed.

In addition to the above tokens, all tokens with bit 7 set ($80-FF) correspond to variable references. $80 is the first variable, $81 is the second variable, etc.