The Basic++

User's Manual

THOMAS RICHTER

Contents

1 Overview On Basic++

Basic++ is an implementation of the BASIC programming language for the Atari 8-bit series of home computers. Basic++ is derived from the original Atari Basic, written by Paul Laughton and Kathleen O'Brien for Shepardson Microsystems (SMI).

The design goals of Basic++ where to fix all known bugs if the original Atari Basic revisions, improve the accuracy of the mathematics, and to extend the language in a minimally invasive way given the fairly restricted ROM space available. Compatibility to Atari Basic was a primary goal, and programs running under Atari Basic were required to work under Basic++, too. The reverse is not necessarily so because Basic++ adds a couple of features not present in the original Basic.

If possible, I also tried to increase the execution speed of Basic programs. It is hard to provide an average speedup factor since it depends very much on the individual features an executing Basic program makes use of, though one should expect approximately twice the speed for longer programs, sometimes even more than that. Due to lack of ROM space, Basic++ does not reach the speed of TurboBasic. Which is, by the way, also derived from the original Atari Basic.

Basic++ requires only a standard Atari Os, the revision does not matter. It works best, though, with Os++ by the same author as it was specifically developped for it. Optimal execution speed and computing precision is only possible under Os++. Basic++ and Os++ are both built into the Atari++ emulator.

Given the requirement to stay compatible with Atari Basic, some of its known drawbacks remained in Basic++: Basic++ does not support string arrays, instead one has to emulate them by using substrings of larger strings. Basic++ does not support Player-Missile Graphics, except through `PEEK` and `POKE`, and Basic++ does not include any block I/O commands besides `PRINT` and `INPUT`.

This manual is structured as follows: The next sections discuss the extensions of individual Basic statements and Basic operators. This is followed by a list of bugs that have been fixed in Basic++, plus some known bugs that remained. The last section describes the memory map of Basic.

Thomas Richter, Stuttgart, October 2015

2 Starting Basic++

Basic++ comes as a regular cartridge image that could, in principle, be burned into an 8K ROM. To run it, the computer (emulated or real) simply has to be turned on *without* holding down the Option console key. Atari++ has a menu item for this in the *Keyboard* section.

One extension has been made: If a disk is present in the first disk drive, and a file management system is available — and it always is with Os++ — Basic++ tries to locate a file named `AUTORUN.BAS` on this disk drive. If it is present, this file is loaded and executed, very much in the same way Atari DOS or Fms++ loads the `AUTORUN.SYS` file. Unlike the latter, the former is a regular Basic program that has been saved to disk with the `SAVE` command.

A second extension is that Basic++ writes its version number to the boot screen.

3 Statement Set

Most Basic++ statements work like their Atari Basic counterparts, though some extensions have been made here and there. To keep the list brief, only the changes are listed here. A full list of the Atari Basic command can be found in the Atari Basic manual.

3.1 `INPUT`

This statement reads numeric or string input from either the editor, or an open IOCB channel. Its syntax is similar to the Atari Basic statement, and similar to Atari Basic — but unlike most conventional Basic dialects — `INPUT` does not allow an additional prompt string. However, Basic++ allows to fill array elements, i.e.

```
INPUT A(I)
```

is valid. It reads a number from the editor and fills this number into the `I`th array element of the array `A`. This new syntax can be combined with all other variants of `INPUT`, i.e.

```
INPUT #1,A,B(4)
```

reads two numbers from channel 1 and fills the first into the variable `A` and the second into the array element `B(4)`.

Note, however, that `A$(5)` as in

```
INPUT #1,A$(5)
```

does not indicate a string array, and hence remains *invalid*. Instead, indices on string variables "slice" strings, create substrings of strings. This, of course, makes little sense in the combination with `INPUT` that fills an entire variable.

Additionally, the first revision of Atari Basic also allowed `INPUT` without any parameters. This form did never work in any way, and Basic++ explicitly disallows that. `INPUT` requires at least one single variable to fill.

3.2 `LIST`

The `LIST` statement lists individual lines or line ranges of the currently loaded Basic program to the editor. This works both as a statement entered in direct mode, as well as part of a Basic program. With a single numeric argument, the source code line of the corresponding line number is listed, with two arguments, the source code lines of line numbers between the first and the second argument are printed to the screen. All this is identical to Atari Basic.

Basic++ does also allow a third form of the `LIST` statement where the second numeric argument is omitted, as follows:

```
LIST 100,
```

This form lists all source code lines whose line numbers are larger than the first and only argument, up to the end of the program. Hence, in this particular example, it lists all line numbers of 100 and above. Listing stops either at the end of the program, or if you hit the BREAK key.

If a program is LISTed to a file, failure to open a the target file resulted in the strange side effect that an entry was created on the run-time stack, and the next RETURN statement would return to the line the faulty LIST was executed at. Basic++ checks proper file creation before entering the body of the LIST function, though an error in the middle of an active LIST — for example due to a full disk — still creates an entry on the run-time stack.

3.3 ENTER

This command is the reverse of LIST, it reads a program in ATASCII format and adds it to the program already in memory. Atari Basic does not allow to interrupt ENTER by pressing the BREAK key, unless the device handler that is being read from checks the key itself and generates an I/O error. Basic++ fixes this by allowing interruption of ENTER.

3.4 FOR and NEXT

The FOR and NEXT statements form a program loop and execute all statements between the two as long as the counter remains within the bounds given in the FOR statement. The syntax of FOR and NEXT remains unchanged from that of Atari Basic, however, the internal maintenance of Basic with respect to such loops changed, and FOR-NEXT loops will be executed faster than under Atari Basic.

The reason for faster execution is that Atari Basic stores the offset and line number of the FOR statement, and hence has to search the program area when looping back from NEXT to FOR. Basic++ stores in addition the absolute address of the program line containing the FOR, and hence grants an almost instand access to the target line.

3.5 GOTO, ON x GOTO and GO TO

These statements remain unchanged compared to Atari Basic, but use a smarter algorithm detecting their target line. While Atari Basic always started searching the target line from the start of the program, Basic++ first checks whether the jump target is above or below the current line. If it is below the current line, Basic++ starts the search at the current line and not at the head of the program, and hence proceeds much faster.

The same goes of course also for the implicit GOTO after THEN in an IF-THEN clause when the target line number for the branch follows directly the THEN.

3.6 GOSUB, ON x GOSUB and RETURN

These statements work exactly alike their Atari Basic counterparts, but quite like FOR-NEXT loops, GOSUB also stores its absolute line address such that a RETURN statement does not have to search the program for its return address. Hence, RETURN is much faster under Basic++.

Similar to GOTO, GOSUB also uses an optimized search procedure to detect its target line.

Basic++ also fixes one bug of the first Atari Basic revision: A ON x GOSUB clause still placed a return address on the Basic run time check even if the argument was out of range. So for example, a

```
10 X=3
20 ON X GOSUB 100,110
30 PRINT "HI"
40 RETURN
```

would print HI twice before failing with an error, the first by falling through the ON x GOSUB class because no jump target is found for the case X=3, and the second time because the RETURN erraneously returns to line 20, even though the GOSUB was never executed.

3.7 TRAP

The TRAP statement specifies a program line the program jumps to in case of an error. While the syntax remained unchanged from Atari Basic, there is one subtle difference between the original TRAP and its implementation under Basic++: Basic++ does not execute the TRAP if the statement that failed was entered in direct mode. Hence, if a TRAP is active under Atari Basic, and you enter a LOAD statement for a file that is not present, Atari Basic would — probably very much to your suprise — execute the TRAP and continue the aborted program. Basic++ remains in direct mode and prints the error on the editor.

3.8 CONT

The CONT statement resumes program execution after a STOP or program interruption by the BREAK key. If the line containing the STOP or the line where the program was interrupted got deleted before CONT is executed, Atari Basic would resume program execution not with the next line, but the line after the next line.

Basic++ fixes CONT to resume always with the next available line after the line Basic stopped in, regardless of whether the stopping line was deleted in between or not.

3.9 DIM and COM

These two statements reserve memory for arrays and string variables. Atari Basic, however, never checked properly whether the array dimension or string dimension overruns the available memory, or overruns the computing precision. Thus, Atari Basic allowed to DIMension arrays that would require more than 64K of memory, more than even potentially available.

Basic++ checks array dimensions more carefully and fails when an attempt is made to create an array whose size is larger than the available memory — or larger than 64K.

3.10 END

This statement terminates program execution. Nevertheless, Atari Basic allowed to continue a program beyond the END statement by entering CONT in direct mode. Basic++ does not allow this anymore, CONT has no effect after an END.

3.11 STATUS

The STATUS command receives the current input/output status from an open IOCB channel and places the status value in its argument. Atari Basic only allowed here numerical variables, whereas Basic++ also allows array elements. Hence,

```
STATUS #1,S(2)
```

is valid and fills the numerical I/O status into the array element S(2).

3.12 NOTE

This statement receives the current file position as sector and byte offset from the file management system (or any other handler that implements it) and places the two offsets into its arguments. The first argument receives the absolute sector on the disk, the second the byte address. While Atari Basic documented that array elements were valid, the following statement never executed correctly:

```
NOTE #2,SECTOR(I),BYTE(I)
```

With Basic++, the above will work as intended and will place the sector and byte offset into two array elements indexed by the variable `I`.

3.13 POINT

`POINT` is the counterpart of `NOTE` and places the file pointer of an open file at a specific sector and byte offset. Atari Basic only allowed numeric variables and arrays as arguments whereas Basic++ allows arbitrary expressions here. Hence,

```
POINT #2,SECTOR(I),0
```

is valid. Note, however, that it is usually not possible to compute the byte and sector address of a file on disk since the Fms will not necessarily assign ssubsequent sector numbers to files, and not every sector needs to be filled up to the last byte.

3.14 PRINT

The `PRINT` command and its shortcut, the question mark, showed a strange bug in the first revision of Atari Basic. If a string was printed whose last character had the ATASCII code of the keyboard sequence `CONTROL+U`, then `PRINT` behaved as if a semicolon was appended to the clause, i.e. the line feed was never executed. Later versions of Atari Basic and Basic++ fix this problem. Surprisingly, the same bug reappeared in TurboBasic.

3.15 READ

This statement reads the variables given as arguments from the `DATA` lines in the program. `READ` works similar to `INPUT`, except that the latter reads from the editor or an IOCB channel, but the former reads from the `DATA` lines.

Basic++ extends `READ` in the same way it extends `INPUT`, namely array elements are allowed as arguments. So for example, the following program

```
10 DIM A(10)
20 FOR I=1 TO 10
30 READ A(I)
40 NEXT I
50 END
60 DATA 1,4,9,16,25,36,49,64,81,100
```

fills the array `A()` with the square numbers from 1 to 100. The clause `READ A(I)` would not be valid in Atari Basic.

Basic++ also fixes a bug concerning `READ`. If a program using `READ` is interrupted, and the user deletes the line containing the `DATA` statement `READ` was reading from, the next `READ` does not continue with the start of the following `DATA` statement, but somewhere in its middle. Basic++ always restores the `READ` location to the start of the next available `DATA` statement if the current statement is lost.

3.16 GET

This statement reads a single byte from an I/O channel and places the value read into its argument. Unlike Atari Basic, Basic++ also allows array elements here, i.e.

```
GET #3,A(5)
```

is valid and reads a single byte into an array. In particular, a loop like the following

```
FOR I = 1 to 10:GET #1,A(I):NEXT I
```

fills an entire array without requiring a temporary variable.

3.17 GRAPHICS

This statement changes the graphics mode of the screen. The original Atari Basic closed the IOCB channel going to the screen before evaluating its argument; if the expression failed, the screen was left unusable. This happens for example with the clause

```
GRAPHICS 1/0
```

If a windowed screen mode was open before, it looks as if the screen is still available after this (failing) clause. However, any attempt to plot or draw on the screen failed now because the screen channel is no longer open.

Basic++ fixes this by first evaluating the expression and then executing the statement.

3.18 LOCATE

`LOCATE` reads the pixel color of a pixel from the screen and places the color into its third argument. Unlike Atari Basic, Basic++ also allows array elements as third argument, such that the statement

```
LOCATE 2,3,COLOR(6)
```

becomes valid. It reads the color of the pixel at position $2, 3$ into the array element `COLOR(6)`.

3.19 SOUND

This command generates sound by channel, frequency, distortion and volume given by its four arguments. The original Atari Basic implementation always overwrites one hardware register that not only controls sound, but also configures the keyboard and paddle operation, namely `SkStat` at position `$d20a`.

Basic++ only resets the bits related to audio output and leaves all other bits alone.

3.20 CLOAD and CSAVE

These two Basic++ commands work like their Atari Basic counterparts and load or save a program from the tape. Note, however, that Os++ has no ROM resident tape handler and the two commands will thus usually fail. To access the tape, the disk-based tape handler needs to be loaded first.

For this, first save your program to disk under a unique name, unless you want to replace it anyhow. Then go to the DOS command line with

```
DOS
```

and insert the Os++ system disk. Now load the `TAPE.EXE` file from the disk by entering

```
TAPE.EXE
```

on the DOS prompt. Since the resident tape handler requires parts of the Basic program space, you now need to clear the Basic program area from within DOS. For that, enter

```
NEW
```

at the DOS command prompt. Finally, go back to Basic++ with

```
CAR
```

In case you had a basic program loaded, it will now be lost since part of the program space was required for the tape handler. If you need it back, use the `LOAD` command.

However, due to the now present tape handler, `CLOAD` and `CSAVE` will now execute correctly. If you need the tape handler permantently, make a backup of the system disk and rename the `TAPE.EXE` file as `HANDLERS.SYS`. Booting from this disk will now make the tape handler resident permanently without having to go through the DOS command line every time.

3.21 `DIR`

This is the only new statement Basic++ introduces. It takes one optional argument, namely a file name or a wild card. Without any further arguments, this statement lists the contents of the directory in the first drive. With additional arguments, it lists all files in the directory of the given device that match the wild card. So specifically,

```
DIR "D2:*.*"
```

lists all files on the second drive. Under Os++, the same can be abbreviated as

```
DIR "D2:-"
```

Even the last double quote may be omitted.

The Basic++ `DIR` command is identical to the one implemented in TurboBasic, and TurboBasic programs using `DIR` as *only* extension will execute correctly under Basic++ (and vice versa).

4 Operators

Basic++ also improves some of the Atari Basic operators. These improvements are listed in the following section.

4.1 The `NOT` Operator

This operator returns 1 if its argument is 0, and returns 0 otherwise. It is most useful to invert the condition of an `IF` clause. The first revision of Atari Basic had a bug that caused a crash if two `NOT` operators were following each other, forming a double inversion. All later revisions did not allow this construction at all. Basic++ fixes `NOT` and also two or more `NOT` operators following each other. In specific, the expression

```
NOT NOT A
```

is not quite as absurd as it sounds: If `A` is zero, it evaluates to zero. All other values of `A` evaluate to one. Thus, it is an alternative way of writing

```
A <> 0
```

Basic++ also changes the priority of this operator compared to the first revision of Atari Basic. Similar to all later editions, `NOT` binds now as strong as the unary + and − operators. In case of doubt, place brackets around subexpressions to make the priority explicit.

4.2 Unary + and − Operators

The unary minus operator inverts the sign of its argument, whereas the first operator does not perform any operation and is only present for symmetry. The first revision of Atari Basic crashed if it found a pair of plus or minus signs, so for example

```
A=--A
```

did not work at all. Later versions of Atari Basic simply disallowed such expressions.

Basic++ allows them and handles them correctly. A double inversion does nothing, and the overall effect of a chain of plus and minus signs is either inversion or the identity, depending on whether the number of minus signs is odd or even. For example,

```
PRINT +--+-+--+-+-3
```

prints `-3`. (Count!)

4.3 The Power ^ Operator

This operator takes its left argument and raises it to the power of its right argument. Atari Basic had a couple of problems with this operator, namely first not getting integer results for integer arguments in its first revision, or returning entirely wrong results in its second and later revision. Particularly, expressions such as 1^{44} returned nonsensical results such as 2.

Basic++ improves the power operator by always returning the correct result for integer arguments, and by improving the precision and execution speed for non-integer arguments.

4.4 Array Accesses

Atari Basic allowed to access one-dimensional arrays with two subscripts if the second subscript is zero, and it also allowed to access two-dimensional arrays with only one subscript, assuming then that the second missing subscript is zero. Hence, the following program passed, surprisingly without an error:

```
10 DIM A(10),B(2,3)
20 A(10,0)=3
30 B(2)=A(10)
```

Basic++ no longer allows this and checks whether the dimensionality of the array when accessed is identical to the dimensionality when it was created. The above program would hence fail under Basic++.

4.5 String Slicing

A unique feature of Atari Basic is its string handling. Creating substrings from string variables is an operation called *slicing*, and Atari basic supports it by the string index operator, the round brackets.

The brackets can either contain a single argument, or two arguments separated by comma. In the first case, the result of the bracket operator is the substring of the given string that starts at the given index of the string, and that extends to the end of the string, i.e.

```
DIM A$(10)
A$="HELLO"
PRINT A$(4)
```

prints `LO`. Unlike Atari Basic, Basic++ also allows the index value to be equal to the size of the string plus one, a case that would fail under Atari Basic. The result of this operation is the emty string:

```
DIM A$(10)
A$="HELLO"
PRINT A$(6)
```

This prints nothing, but does neither generate an error. The index is, however, not allowed to be larger than one plus the string size, this would still generate an error. This extension avoids typical bounds checks for string handling and covers a common case.

The string bracket operator may also take two arguments; this form takes from the given substring a string that starts at the index given by the first expression in the brackets, and extends to the second expression in the bracket. Both first and last index are *inclusive*, i.e. the characters at the given start and the end index are both included in the resulting substring. For example,

```
DIM A$(10)
A$="HELLO"
PRINT A$(2,3)
```

prints `EL`. It is the substring starting at the second and extending to the third character within the variable `A$`.

As an extension, Basic++ also allows the second index to be one smaller than the first, in which case the resulting substring is empty. For example,

```
DIM A$(10)
A$="HELLO"
PRINT A$(2,1)
```

prints nothing, but neither returns an error. Again, this avoids some bounds checking for typical string handling algorithms, and only this single special case is allowed. The second index is not allowed to be smaller than one minus the first index, otherwise an error is generated.

Both extensions, the handling of indices beyond the end of the string and the second index being smaller than the first are only valid if the string is used as an argument, or at the right side of an assignment, that is, if the string that is sliced by the bracket operator is to be read from. The extensions are *not* valid on the left hand side of an assigment, or when the string is to be written to. If the first index of the bracket operator on the left side of a string assignment extends beyond the reserved string size — not the actual string size — the assignment fails:

```
DIM A$(10)
A$="HELLO"
A$(11)=A$
```

because the string can only hold ten characters, and there is no eleventh character. However,

```
DIM A$(10)
A$="HELLO"
A$(6)=A$
```

is valid, and places the value `HELLOHELLO` into `A$`. Even though the sixth character of `A$` is currently not used, it can still be assigned to.

The same holds for the second extension:

```
DIM A$(10)
A$="HELLO"
A$(6,5)=A$
```

is *not* valid, since the left hand side leaves no room at all the right hand side could be assigned to.

4.6 The STR$ Function

The STR$ operator takes a numeric argument and converts it into a string. For example,

```
DIM A$(10)
A$=STR$(5)
```

sets the string variable A$ to the string containing the single character 5.

Atari Basic had a strange bug, though, namely STR$ could not be used twice in the same expression without causing strange side effects. For example, the following

```
IF STR$(5) = STR$(6) THEN PRINT "THE NUMBERS ARE EQUAL"
```

prints THE NUMBERS ARE EQUAL even though they are clearly not. This bug has been fixed in Basic++.

4.7 The CHR$ Function

A related operator is the CHR$ operator. It also converts a number into a string, but this time the result is a string consisting of a single character whose ATASCII code is equal to the argument. For example,

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

prints A on the screen because the ATASCII code of the letter A is 65.

Similar to STR$, CHR$ had a bug that caused strange effects if the operator is used twice in the same expression. Basic++ fixes this bug.

4.8 The ASC Function

This operator is the reverse of the ASC$ operator: It takes a string, and returns the ATASCII code of its first element. For example,

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

prints the ATASCII code of the letter A, hence 65.

Atari Basic does not, however, check whether the string argument does actually *contain* a character at all, and returns a nonsense result for the empty string. This has been fixed, and

```
PRINT ASC("")
```

creates an error.

4.9 The ATN Function

This function computes the arcus tangens, i.e. the inverse tangens of its argument. That is, it finds the angle whose tangens is equal to the argument of the function. Basic++ improves the precision of this function, for example,

```
DEG
PRINT ATN(1)
```

prints 45. This is because the tangens of 45 degrees is equal to 1. Atari Basic returned a result that was slightly off the correct value here.

4.10 The LOG and CLOG Functions

These two functions compute the logarithm of its argument. LOG returns the natural logarithm to the base of e, CLOG the logarithm to the base of ten. Basic++ improves the precision of these two functions, especially LOG(1) and CLOG(1) are exactly 0.

4.11 The SQR Function

This function returns the square root of its arguments. The Basic++ implementation is completely new and more than four times faster than the original. It is also more precise, the result is rounded correctly up to the last digit.

5 Additional Bug Fixes

Despite the modifications listed above, Basic++ fixes a couple of bugs that are hard to classify and that are listed one by one.

5.1 Potential Lock-ups: Memory Allocation and Release

The first revision of Atari Basic had a bug in the memory release function: If an exact multiple of 256 bytes was released, parts of the program could have been corrupted, potentially resulting in a complete lock-up of the Basic interpreter. The second revision fixed this bug, but introduced a second bug that could lock-up the interpreter when an exact multiple of 256 bytes was allocated. It also introduced another bug that lost 16 bytes of memory every time a program was saved to disk and loaded back. The last revision fixed the memory allocation bug, but not the 16 byte loss.

Basic++ fixes all these bugs.

5.2 Parser Errors

The Atari Basic parser confused the ESC-character (ATASCII code 27) with the End-Of-Line code (ATASCII code 155) and hence treated ESC as command separator, with sometimes bizarre side effects. Basic++ does not accept ESC as a command separator and errors in case it is found outside of a string constant.

The parser also allowed in some placed — but not in all — variable names that are similar to operator names. So for example,

```
LET GOSUB=200
```

was accepted, but

```
GOSUB=200
```

was not. The places in which such odd-named variable were acceptable depends pretty much on the internal structure of the parser and is hence hard to predict. Basic++ does not allow variable names that conflict with operator names. So for example, a string variable such as ABS$ is legitimate because it cannot possibly be confused with ABS which has a numeric value, but GOSUB as variable name is inacceptable. Similarly, a string variable cannot be named CHR$ because there is a function of the same name, but CHR as variable name is fine.

5.3 Error Handling

Atari Basic resets the `DirectFlag` variable that prohibits execution of control sequences on errors. If this variable is set, the Os editor rather prints the symbolic version of ATASCII codes instead of executing them. Any type of error, even completely unrelated, cleared this variable, potentially causing ill side effects. Basic++ only resets this Os variable when input is expected from the user, hence allowing free usage of the keyboard.

When `RESET` was pressed in the middle of a longer memory allocation or release function, the results could have been unpredictable with Atari Basic. The currently edited program will certainly be damaged, potentially resulting in a completely unresponsive system when contuing with program edits. Basic++ erases in such a case the program, giving you at least the editor back — your program would be corrupted in either case, but at least you have now the chance to load an earlier version from disk.

Atari Basic created file names by temporarily modifying the string variable or string constant containing the file name by appending an EOF to the string, then restoring the original string value afterwards or in case of error. However, this procedure is quite fragile and could have created invalidated strings or even invalidated programs if the restauration procedure did not succeed. Basic++ uses an alternative and more robust approach by first copying the file name into a buffer and modifying it there.

5.4 Numerics

Atari Basic implements the unary minus operator by inverting the sign of the number, creating signed zeros. While this is an acceptable solution, it did not tread the signed zero equivalently to the unsigned zero, resulting in some strange failure cases, for example in `IF` clauses, comparisons or printing of such numbers. Basic++ uses a much more careful numeric model that, together with the improved math pack of Os++, provides more speed and higher precision.

5.5 Speed Improvements

As already mentioned above, `FOR-NEXT` loops and `RETURN` from subroutines are much faster now as Basic++ stores the absolute address of the start of the loop or of the line to return to. In addition, `READ` and `LIST` have also been speed up as they also use the run-time stack to temporarily store the current program line. Basic++ is careful enough to be permissive in case the program is edited between `FOR` and `NEXT` or between `GOSUB` and `RETURN`. In such cases, the absolute address of the line to return to could have been moved. Basic++ detects such cases and then falls back to the full search Atari Basic used to use.

Additionally, one-dimensional array accesses have been speed up by avoiding an unnecessary multiplication.

Some numerical functions such as `SQR` or the power-operator ^ have been largely improved; both implementations have been completely revised and made faster and more precise.

6 Memory Map

This section contains the memory map of Basic++. Unlike the Os++ memory map, none of the Basic++ variables are supposed to be modified externally, and most of the variables are for internal use only. Basic programs can rarely take advantage of any of them.

While the usage of most locations is identical to that of Atari Basic, they might have been renamed. In rare cases, memory locations moved or their meaning changed. If two locations are given, then the two variables together form a 16 bit value whose low-value is in the first byte and whose high-value is in the second byte. Memory addresses indicated by * are new to Basic++ or have changed their meaning compared to Atari Basic. Addresses indicated by ^{t} are system internal temporaries that cannot be modified to obtain an

observable effect and that do not have any controlled value outside of the Basic internal functions that require them.

$80,$81 `LoMem` Points to the start of the Basic++ RAM area and to a 256 byte buffer used for various purposes described below.

$80,$81 `TokenBuffer` This buffer is used as output buffer of the tokenizer before the tokenized line is moved to its final location.

$80,$81 `OperatorStack` The same 256 byte buffer is also used as stack for storing operators and arguments or values when evaluating expressions. The operator stack fills from the end of the buffer, the argument stack from its beginning.

$82,$83 `VarNamePtr` Points to the table containing all variable names. Variable names are stored consecutively, with the MSB of the last character of the variable name inverted. The table is terminated by a single zero byte.

$84,$85 `VarNameEndPtr` Points to the end of the variable name table. This pointer is used to add new variable names to the table quicky without having to scan through the full table.

$86,$87 `VarValuePtr` Points to the table containing the variable values. The meaning of the entries is identical to the zero page variables `VariableType` and following and is described there.

$88,$89 `StatementPtr` Pointer to the tokenized basic program. This pointer points to the first tokenized line of the currently loaded program.

$8a,$8b `CurrentLinePtr` Pointer to the current line that is being executed.

$8c,$8d `ArrayTablePtr` The area starting at this memory location contains the values of arrays and string variables.

$8e,$8f `RunStack` This is the start of the Basic++ run time stack. Basic stores return addresses for `GOSUB`, `FOR`, `READ` and `LIST` here. The stack grows upwards, i.e. towards higher addresses.

$90,$91 `HiMem` Pointer behind the last byte used by Basic++. It copies this pointer also to `AppMemHi` to inform the Os where it may start allocating memory for its screen buffer.

$92 `GenerationCounter`* This counter is incremented every time Basic++ changes to the interactive mode. It is used to potentially invalidate the address field of Basic run-time stack entries when edits are applied to the program.

$93 `PoppedGenerationCounter`^{t*} This is the copy of the `GenerationCounter` that was found on the run-time stack. If this value does not co-incide with the value found in $92, the absolute address on the run-time stack is considered invalid.

$94 `cox`^{t} Pointer into the `TokenBuffer` when tokenizing a Basic line.

$94 `PrintCounter`^{t} Counts the characters printed by `PRINT` and is used to implement tabulation by the comma operator of the `PRINT` statement.

$95,$96 `ScanPtr`^{t} Used as a temporary when scanning the variable name list, the statement list or operator list when looking for a statement, operator or variable name.

$95,$96 `PrintPtr`^{t} Points the character buffer currently printed, either by `PRINT` or by any other output operation performed by Basic.

$95,$96 `PoppedLinePtr`^{t*} Absolute address of the line as removed from the run-time stack.

$97,$98 `AllocMemPtr`^{t} This pointer points to the memory that has been allocated by increasing one of the Basic++ tables. It is filled by the Basic++ memory manager and used to access the allocated memory region.

$97,$98 `EndIndex`^{t} When slicing strings, this is the index of the last character in the string slice.

$97,$98 `Index2`^{t} When accessing two-dimensional arrays, this is the value of the second index.

$99,$9a `MoveSourcePtr`^{t} Temporary pointer of the memory manager to move memory regions around. This is the pointer to the source of the block move.

$99,$9a `IntegerExponent`^{t*} Value of the exponent of the currently computed power ^ expression, rounded down to the next integer.

$9b,$9c `MoveDestPtr`^{t} Destination pointer for a block-move operation of the memory manager.

$9b `PowError`^{t*} This flag is set in case the currently executed power ^ expression overflows. In case the exponent is negative, the result will be set to zero, otherwise an overflow is generated.

$9d,$9e `VarTableEntryPtr`^{t} When accessing variables, this pointer points to the entry in the variable table where the variable currently being accessed is stored.

$9f `MaxValidcix`^{t} Offset into the current line being tokenized up to which the Basic parser could make sense of the input. In case the line could not be parsed successfully, the Basic parser will mark this location to indicate the potential position of the error.

$9f `LineLength`^{t} Size of the current line being executed.

$a0,$a1 `TargetLineNumber`^{t} Used as input variables to the line-search function, this pair contains the line number to search for. It contains the target line number `GOTO`, `GOSUB` and related statements branch to.

$a2,$a3 `MoveSize`^{t} Size of the memory block that is moved by the memory manager on a block move.

$a4,$a5 `AllocSize`^{t} This pair temporarily stores the size of the memory to be allocated or released by the memory manager.

$a6 `DirectModeFlag`^{t} This flag of the basic parser determines what to do with the line parsed off recently. If bit 7 is set, the line was entered in direct mode and is to be executed immediately; otherwise, it is merged into the program. I bit 6 is set, the line contained an error and the erraneous line is to be listed on the screen.

$a6 `InputMode`^{t*} Identifies whether the current statement executed is a `READ` or `INPUT` to a common subroutine. `READ` aborts reading strings at commas, `INPUT` treats them as part of the string.

$a7 `StatementLen`^{t} Offset to the length byte of the statement currently being parsed. As soon as the length is known, it will be filled in here.

$a7 `StatementEnd`^{t} Offset of the end of the current statement in the current line being executed.

$a8 `StatementStart`^{t} Offset to the start of the statement currently being parsed.

$a8 `StatementOffset`^{t} Offset to the currently executed statement in the currently executed line.

$a9 `SyntaxStackPtr`^{t} Pointer to the execution stack of the Atari Basic Meta Language interpreter parsing the current command. The stack itself is located at `SyntaxStack`.

$a9 `OperatorStackPtr`^{t} This is the operator stack pointer; the operator stack stores the priority of the operators when evaluating an expression. The stack itself is stored in the 256 byte buffer pointed to by `OperatorStack`; the stack starts at the last byte of the buffer and grows downwards.

$aa `TableSkip`^{t} Number of bytes to skip when scanning a table containing variable, operator or statement names. Only the statement table contains two additional bytes upfront that point to the ABML syntax description of the statement.

$aa `ArgumentStackPtr`^{t} The argument stack pointer; the argument stack stores numerical values when evaluating expressions. Each expression is eight bytes long and uses a layout identical to those of variables. The stack itself is located in the buffer pointed to by `OperatorStack`; the stack starts at the first byte of the buffer and grows upwards. If the argument stack grows into the operator stack, the stack is full and Basic++ generates an error.

$ab `ParseStartcox`^{t} Backup of the output offset `cox` of the Basic parser. In case parsing a subexpression fails, the Basic parser rewinds its input and output and tries an alternative scan.

$ab `OperatorToken`^{t} The token of the operator currently under evaluation.

$ac `ParseStartcix`^{t} Backup of the input offset `CIX, $f2` (see the Os++ manual) pointer used for parsing. This is kept here so the parser can rewind the input parsing position in case parsing of a subexpression failed and another interpretation needs to be tried.

$ac `OperatorPriority`^{t} Priority of the operator. The upper nibble defines how strong the operator binds to the left, the lower nibble how strong it binds to the right. The higher the value, the stronger the binding. Stronger binding operators are evaluated first.

$ac,$ad `ListEndLineNumber`^{t*} Contains the upper bound for the list range of the `LIST` statement.

$ad,$ae `PreviousVarNameEndPtr` This pointer contains a backup of the `VarNameEndPtr` before the Basic parser started. The value is stored here to allow the Basic parser to remove variables that have been temporarily created by parsing a line that contained errors, and that hence have entered the variable name pointer only by error. These variables are removed as soon as Basic detects the parsing error.

**$af,** `ScanOffset`^{t} Offset into the operator or statement name when scanning through the table of all operators or statements.

**$af,** `PrintOffset`^{t} Offset into the buffer pointed to by `PrintPtr`.

**$b0,** `ParsedOperator`^{t} Recently parsed off operator token. This is stored here so the same operator needs not to be parsed over and over again. Keeping a backup speeds up parsing.

$b0 `CommaCount`^{t} Counts the number of commas Basic++ finds when going through the arguments of a statement as part of its execution. This is used to determine the dimensionality of arrays or the argument count of functions.

$b1 `AdditionalVariables` Counts the number of variables created by the Basic parser, and hence the number of entries in the variable value table that were created by the parser. In case the Basic parser detects an error while parsing a line, such erraneously created entries in the variable value table are removed again.

$b1 `AssignFlag`^{t} Determines the nature of an assignment; if bit 7 is set, the left-hand side of an array or string assignment is evaluted and the assignment takes place as soon as the indices on the assignment target have been parsed. If bit 6 is set, an array or string is used as the target for `INPUT`, `READ`, `GET`, `LOCATE`, `NOTE` or `STATUS`, and hence the variable value is not extracted, but its location is recorded. The LSBs contain a copy of the `CommaCount` variable above for string assignments.

$b2 `ParsedOperatorcix`^{t} Backup of `CIX`, (see the Os++ manual) that points behind the last operator scanned. The token of the operator found at this position is in $b0. Keeping this offset here avoids scanning the same operator multiple times and speeds up the parser.

$b2 `PoppedStatementOffset`^{t} Offset of the statement in the line to return to, as removed from the run-time stack.

$b2 `OnValue`^{t*} Keeps the value of the argument to `ON-GOTO` or `ON-GOSUB` and is used there to find the target line number in the line number arguments.

$b3 `LastOperatorcix`^{t} Backup of `CIX` that points at the last parsed operator. This together with the above stores the state of the parser to avoid unnecessary parsing steps.

$b3 `SavedStatementOffset`^{t} Copy of $a8 before the statement offset is placed on the run-time stack for keeping the return location for loops or sub-programs.

$b3 `ChannelOffset`^{t} Audio channel that is addressed by the currently executed `SOUND` statement.

$b4 `EnterIOCB` IOCB number (not offset) of the channel currently used for parsing commands or receiving input. This could also be understood as the stdin for the Basic parser.

$b5 `ListIOCB` IOCB number (not offset) of the channel currently used for generating output, e.g. listing. This could also be called the stdout of the Basic interpreter.

$b6 `DataOffset` Offset into a line containing `DATA` statements where the next `READ` will take its data from.

$b7,$b8 `DataLine` Line number where the next `READ` will start searching for `DATA` statements to be read from. This variable can be set by `RESTORE`, which also resets `DataOffset` to zero.

$bc,$bd `TrapLine` Line number where Basic++ will continue execution from in case a program error is detected. If this line number is larger or equal to 32768, then program execution will abort in case of an error.

$ba,$bb `StopLine` Line number of the line where Basic++ stopped execution, or line number where an error has been found before branching to `TrapLine`. A `CONT` statement will resume execution *on the following line*.

$be,$bf `SyntaxPtr`^{t*} Pointer to the ABML syntax that is currently being used to parse off the next token.

$be,$bf `JumpOriginLinePtr`^{t} Copy of $8a,$8b before a jump or a sub-program call is executed. In case the target line of a branch or sub-program cannot be found, the current program line is restored before executing a `TRAP` or aborting the program, ensuring that the error points to the correct line.

$c0 `IOCommand`^{t} CIO command to be issued next.

$c1 `IOCB`^{t} IOCB number to be used for the next input/output operation.

$c3 `TrappedErrorNumber` This variable contains the error number of the last error created. Basic programs may take advantage of this variable to find out why the program branched to the `TRAP` error line.

$c4,$c5 `RunStackPtr`^{t} Keeps a copy of the current pointer to the top of the run-stack. This is used to locate elements on the stack, most notably for `FOR-NEXT` loops.

$c6 `StepSign`^{t} Contains the direction of the `FOR-NEXT` loop currently executed. Bit 7 is set for a decreasing loop, otherwise the loop is increasing.

$c6 `USRArgCount`^{t} This temporary counts the arguments of the `USR` function.

$c7 `ForVariableIdx`^{t} Contains the index of the variable that is used as the loop counter of the currently executed `FOR-NEXT` loop.

$c8 `Color` The previously selected color for plotting and line-drawing. This variable is set by `COLOR`.

$c9 `PrintColonWidth` Separation between two formatting stops generated by the comma operator of a `PRINT` statement. This variable may be set by Basic programs.

$ca `LoadFlag` If this flag is set, the Basic++ statement list is inconsistent and the Basic program area is erased when returning back to direct mode. This flag is set while loading programs, or while extending the program area.

$d2 `VariableType`^{t} This flag contains the type of the variable currently worked on, and is a copy of the first byte of the eight-byte variables values stored in the table pointed to by `VarValuePtr`. If bit 7 is set, the variable is a string, if bit 6 is set, it is an array. If bit 1 is set, then the offset of a string into the table at `ArrayTablePtr` has already been converted to an absolute address, or is a string constant in first place. If bit 0 is set, the array or string is dimensioned, if bit 1 is set, the array is two-dimensional instead of one-dimensional.

$d3 `VariableIndex` The index of the variable being accessed. This counts from zero up. Up to 128 variables can be accessed.

$d4 to $da `fr0` This is the floating-point register zero, it contains the numerical value of the currently accessed variable for numerical variables.

$d4,$d5 `VariablePointer` For array or string variables, this stores the offset of the array or string within `ArrayTablePtr`, or absolute address of the string if bit 1 of `VariableType` is set.

$d6,$d7 `StringLength` For strings, this stores the current size of the string variable currently accessed.

$d6,$d7 `FirstArrayDimension` For arrays, this stores the size of the first or only dimension of the array plus one.

$d8,$d9 `StringDimension` For strings, this stores the maximum capacity of the string being accessed.

$d8,$d9 `SecondArrayDimension` For two-dimensinal arrays, this stores the size of the second dimension plus one. For one-dimensinal arrays, this is zero.

$f0 `SignFlag`^{t} This flag is set case the argument of `ATN` is negative.

$f1 `TransformFlag`^{t} This flag is set in case the argument of `ATN` is larger than one and has been reflected back into the unit interval.

$f5,$f6 `StartIndex`^{t} When slicing strings, this is the index where the string slice starts at. The end index is at $97,$98.

$f5,$f6 `Index1`^{t} When accessing arrays, this is the first or only index into the array. If the array is two-dimensional, the second index is at $97,$98.

$fd `RadFlag` This switch is zero in case trigonometric functions are evaluated in radiants, or six in case they are evaluated in degrees. The default is zero, i.e. radiants.

$480 `ArgumentStack`^{t*} Even though currently not used as such, this area is reserved as an alternative argument stack for future extensions.

$480 `SyntaxStack`^{t} This is a 256 byte buffer that holds the stack of the ABML syntax parser that is used to parse off Basic statements. Each entry is four bytes large: The input offset at which the subexpression starts parsing, i.e. a copy of `CIX`, the output offset into the `TokenBuffer` where the subexpression will go to, and a copy of `SyntaxPtr` that points to the ABML syntax of the subexpression.

$5e6 `FPSrc`^{t} A temporary buffer for a floating point value.

$5ec `FPSrc2`^{t*} Another temporary buffer for a floating point value.

$5f2 `FPSrc3`^{t*} A third temporary buffer for a floating point value.

$5f8 `VariableTemp`^{t*} A temporary holding space for a single variable.