batari Basic - commands

BY FRED "BATARI" QUIMBY (ADAPTED BY DUANE ALAN HAHN)
PDF BY SCORP.IUS

The text below started out as a semi-simplified version of the 0.2 Alpha batari Basic reference page (help file). The commands were put in an order I could understand better. I also added colors to some text, an NTSC color chart, an interactive playfield graph, and other things to help me out. The 0.3 Alpha version of the help page was based on this page you are reading now, so that's why they look similar. You can view the latest version of the official bB help page by clicking here.

If you are like me and love front ends, be sure to check out the front end for batari Basic (2600IDE by attendo). Like most front ends, it turns slow and complicated into quick and easy. It also comes with a built-in sprite editor and playfield editor.

I make frequent changes to this page around the time of a new bB update, and I'm not just talking about once a day either. Sometimes I update this page dozens of times in one day, so you might want to hit Refresh/Reload on return visits to make sure you are seeing the most up-to-date version.

Things You Should Know

Memory

In batari Basic, you are currently limited to about 3k of program space in a 4k ROM. 1k of this is used for the display kernel and other support routines.

The Atari 2600 also only has 128 bytes of RAM. Of this RAM, 26 bytes are available for general use in your programs, as variables a-z.

Timing

Timing is crucial in batari Basic, in that you only have about 2 milliseconds between successive calls to drawscreen. See drawscreen for more information.

White Space

Batari Basic Alpha 0.1 was very picky about whitespace. These problems are pretty much nonexistent in Alpha 0.2. As long as you don't try anything totally goofy, chances are your code will be parsed correctly by the compiler. If you can read it easily, it's likely that the compiler can too.

For example, suppose you want to do "for l = 1 to 10 : t = t + 4 : next"

In Alpha 0.2, the following would be parsed correctly:

for l=1 to 10:t=t+4:next
for   l   =1   to 10:t=t   +4: next
for l=1 to 10 : t= t+4 :next

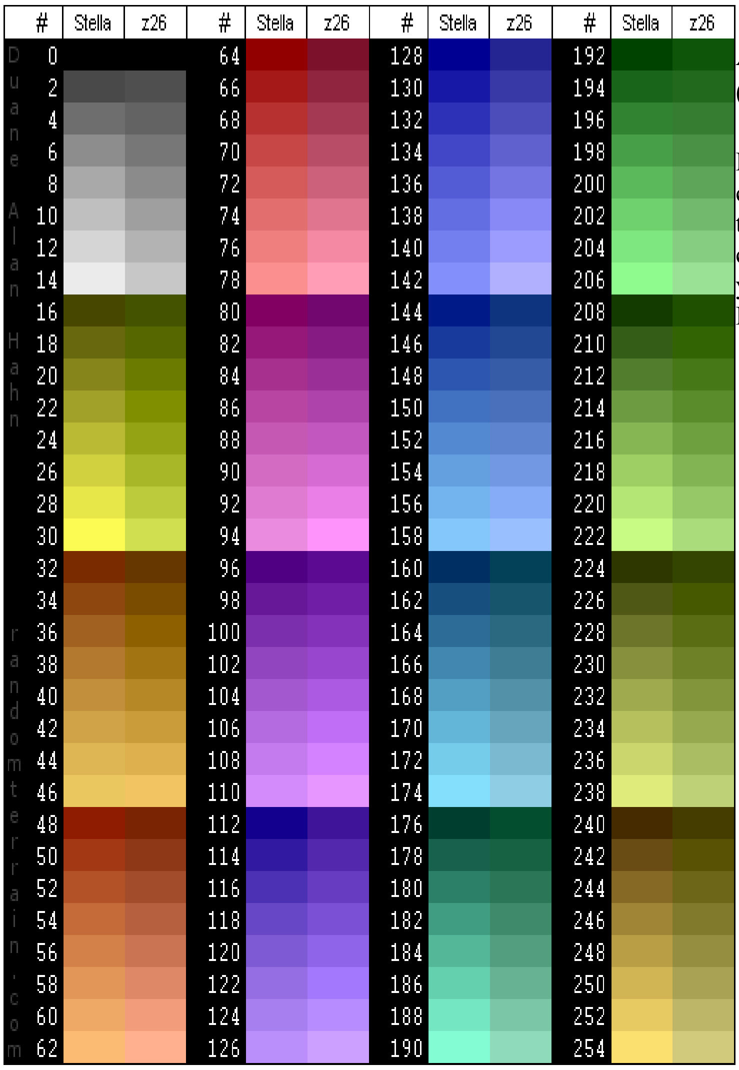
The following would not:

forl=1to10:t=t+4:next
forl=1 to 10:t=t+4:next
for l=1 to10 :t=t+4:next

In other words, any keywords or commands must be spaced properly or batari Basic will think they are variables and compilation will fail, but anything else is fair game. As long as there is a recognizable separator, such as +, -, =, :, *, /, &, &&, |, ||, ^ and possibly others, you can space however you want (or not at all).

Atari 2600 NTSC TIA Color Chart (Decimal Numbers)

Below is a color chart to make it easier to select colors for your games. I included colors from the two most popular emulators. It seems like Stella is closer to the real colors, but you may not agree. If you need PAL colors, check out Glenn Saunder's HTML TIA color chart.



#	Stella	z26	#	Stella	z26	#	Stella	z26	#	Stella	z26
0	⬛	⬛	64	🟥	🟥	128	🟦	🟦	192	🟩	🟩
2	⬛	⬛	66	🟥	🟥	130	🟦	🟦	194	🟩	🟩
4	⬛	⬛	68	🟥	🟥	132	🟦	🟦	196	🟩	🟩
6	⬛	⬛	70	🟥	🟥	134	🟦	🟦	198	🟩	🟩
8	⬜	⬛	72	🟥	🟥	136	🟦	🟦	200	🟩	🟩
10	⬜	⬜	74	🟥	🟥	138	🟦	🟦	202	🟩	🟩
12	⬜	⬜	76	🟥	🟥	140	🟦	🟦	204	🟩	🟩
14	⬜	⬜	78	🟥	🟥	142	🟦	🟦	206	🟩	🟩
16	🟫	🟫	80	🟪	🟪	144	🟦	🟦	208	🟩	🟩
18	🟫	🟩	82	🟪	🟪	146	🟦	🟦	210	🟩	🟩
20	🟨	🟩	84	🟪	🟪	148	🟦	🟦	212	🟩	🟩
22	🟨	🟨	86	🟪	🟪	150	🟦	🟦	214	🟩	🟩
24	🟨	🟨	88	🟪	🟪	152	🟦	🟦	216	🟩	🟩
26	🟨	🟨	90	🟪	🟪	154	🟦	🟦	218	🟩	🟩
28	🟨	🟨	92	🟪	🟪	156	🟦	🟦	220	🟩	🟩
30	🟨	🟨	94	🟪	🟪	158	🟦	🟦	222	🟩	🟩
32	🟫	🟫	96	🟪	🟪	160	🟦	🟦	224	🟩	🟩
34	🟫	🟫	98	🟪	🟪	162	🟦	🟦	226	🟩	🟩
36	🟧	🟫	100	🟪	🟪	164	🟦	🟦	228	🟩	🟩
38	🟧	🟧	102	🟪	🟪	166	🟦	🟦	230	🟩	🟩
40	🟧	🟧	104	🟪	🟪	168	🟦	🟦	232	🟨	🟩
42	🟧	🟧	106	🟪	🟪	170	🟦	🟦	234	🟨	🟩
44	🟧	🟧	108	🟪	🟪	172	🟦	🟦	236	🟨	🟨
46	🟧	🟧	110	🟪	🟪	174	🟦	🟦	238	🟨	🟨
48	🟥	🟥	112	🟦	🟦	176	🟩	🟩	240	🟫	🟫
50	🟥	🟥	114	🟦	🟦	178	🟩	🟩	242	🟫	🟫
52	🟥	🟥	116	🟦	🟪	180	🟩	🟩	244	🟫	🟫
54	🟧	🟥	118	🟪	🟪	182	🟩	🟩	246	🟫	🟫
56	🟧	🟧	120	🟪	🟪	184	🟩	🟩	248	🟨	🟫
58	🟧	🟧	122	🟪	🟪	186	🟩	🟩	250	🟨	🟨
60	🟧	🟧	124	🟪	🟪	188	🟩	🟩	252	🟨	🟨
62	🟧	🟧	126	🟪	🟪	190	🟩	🟩	254	🟨	🟨

Duane Alan Hahn
randomterrain.com

COLUBK (Color-Luminosity Background)
Sets the background color. Example: COLUBK = 112

COLUPF (Color-Luminosity Playfield, Ball)
Sets the playfield color and ball color. Example: COLUPF = 154

COLUP0 (Color-Luminosity Player 0, Missile 0)
Sets the color for player 0 and missile 0. Example: COLUP0 = 42

COLUP1 (Color-Luminosity Player 1, Missile 1)
Sets the color for player 1 and missile 1. Example: COLUP1 = 222

rem (short for Remark)

The rem statement is used for in-program comments. These comments are very helpful not only to other programmers trying to make sense of your code, but to yourself if your memory is anything like mine :)
Note that, unlike old interpreted Basics, you can use rem as much as you want and it will not affect the length of your compiled program.

Constants

To declare a constant in batari Basic, use the const command. const declares a constant value for use within a program. This improves readability in a program in the case where a value is used several times but will not change, or you want to try different values in a program but don't want to change your code in several places first.

An example: the following is near the beginning of your program:

```
const myconst=200
const monsterheight=$12
```

Anytime myconst or monsterheight is used thereafter, the compiler will substitute 200 or $12 respectively.

DOS Compatibility

Although batari Basic is a command-line program, you are expected to run it under Windows 95 or later because it requires a DPMI (DOS protected mode interface) and uses long filenames. If you wish to run under pure DOS, however, you can, but you will need to:

- obtain a DPMI program, such as cwsdpmi.exe, and run this before running batari Basic.
- rename all filenames to 8.3 format
- edit includes files to point to renamed files above
- use the command-line parameter -r to specify an alternate variable redefinition file that conforms to 8.3.

Functions

Alpha 0.3 provides a simple interface for you to define your own functions. These functions can be defined within your program itself or compiled to their own separate .asm file then included with the include command. Functions can be written in batari Basic or assembly language.

To call a function, you assign it to a variable. This is currently the only way to call a function. Functions can have up to six input arguments, but they only have one explicit return value (that which is passed to the variable you assigned to the function.) You can have multiple return values but they will be implicit, meaning that the function will modify a variable and then you can access that variable after you call the function.

A function should have input arguments. In bB, a function can be called with no input arguments if you want, but you might as well use a subroutine instead, as it will save space.

To declare a function, use the function command, and specify a name for the function. Then place your bB code below and end it by specifying end. To return a value, use the return keyword.

Note that in bB, all variables are global, and arguments are passed to the function by use of the temp variables, temp1-temp6. Therefore it is recommended that you use the same temp variables for calculations within your function wherever possible so that the normal variables are not affected.

Example of declaring a function in bB:

```
function sgn
rem this function returns the sign of a number
rem if 0 to 127, it returns 1
rem if -128 to 1 (or 128 to 255), it returns -1 (or 255)
rem if 0, it returns 0
if temp1=0 then return 0
if temp1 <128 then return 1 else return 255 end>
```

To call the above function, assign to a variable, as follows:

```
a=sgn(f)
```

Note that there is no checking to see if the number of arguments is correct. If you specify too many, the additional arguments will be ignored. If you specify too few, you will likely get incorrect results.

To specify more arguments in a function, you can separate them by a comma. Supposing you called a function called max that determined the largest of three values you passed to it:

```
 function max
 if temp1>temp2 then temp1bigger else temp2bigger
temp1bigger
 if temp1>temp3 then return temp1 else return temp3
temp2bigger
 if temp2>temp3 then return temp2 else return temp3
end
```

To call this function, you might do:

```
f=max(d, a[3], 7)
```

Special consideration for functions in assembly language:

To write an asm function for use in bB, many of the considerations are the same - you can pass up to six values to the function and return one. The difference is that the first two arguments are not copied to temp1 and temp2, but instead, the accumulator and the Y register, respectively. Additional arguments are copied to temp3-temp6. To return a value, load it into the accumulator and call an RTS.

Also, the function is entered with S and Z flags set as per the current value of the accumulator.
For example, here is the sgn function rewritten in asm:

```
sgn
 bmi minus
 lda #$FF
 rts
 beq zero
 lda #1
zero
 rts
```

To use the above function in your bB program, you can either use inline asm in your bB program, or compile it separately and include its asm file using the include command, then you just call it as normal.

Compiling a Function as a Module

You can create modules that are written in batari Basic or assembly language. To make an asm module, just write the module and include the file using the include command in bB. To create your own bB module, you must first compile it separately into asm, but not into a full binary. To do this, you can use the following commands. incidentally, the commands in DOS or Unix are the same:

```
preprocess < myfile.bas | 2600basic > myfile.asm
```

Note that you may need to add a ./ before preprocess or 2600basic in Unix if your path isn't set to look in the current directory.

include

include is used to include modules in the final binary that are not normally available. You may also include additional modules using the includesfile command, but you may prefer to use include if you just want an extra module or two to be compiled in addition to what is already in the default includes file.

An example is for fixed point math. Although you do not need to include anything to use many of the fixed point functions, for a few, you will need the fixed_point_math.asm module. You may find it easier to use

```
include fixed_point_math.asm
```

at the beginning of your program instead of creating a new includes file, and this will allow you to share your source without also needing to attach your includes file as well.

includes file

An includes file contains all modules that will be included in the final binary. Modules may be general routines, functions or display kernels. The includes file also specifies the order in which they will appear (this is crucial.) The default includes file (default.inc) contains the standard kernel (which is currently the only kernel) and some of the more commonly used modules.

To create your own includes file, you can use the default.inc file as a template. The default.inc file itself contains comments that should guide you. Save it to a new name and use the .inc extension. To specify a new includes file in your program, supposing your includes file is called myincludes.inc, use:

```
includesfile myincludes.inc
```

You do not need to use the includesfile command to use the default.inc file.

One reason you may want to use your own includes file, however, is in the case where you need space in your program more than you need a standard module, such as the one for playfield scrolling.

Note that if you are using an includes file and you wish to share your Basic code with others, you should also share the includes file so that they can compile your code.

Set Command

set (smartbranching/romsize/optimization)

set allows you to specify certain compiler parameters in your program. Currently, the following is supported:

set smartbranching on
set smartbranching off
set romsize 2k
set romsize 4k
set optimization speed
set optimization size
set optimization none

smartbranching

The smartbranching directive tells the compiler to generate optimal assembly code when it encounters an if-then statement with a line number or label after the **then**.

It is set to "off" by default, because this makes the generated assembly code easier for a human to read and understand. This will sometimes cause the compilation to fail, however. In this case, you will need to use "then goto" instead of just "then" for the if-then statement that caused the problem.

if you "set smartbranching on" then you just use "then" and the compiler will figure out whether to use "then" or "then goto" for you. The drawback to doing this is that the generated assembly file will be much harder for a human to read. If you do not care to see the assembly language that bB generates, however, you should set smartbranching on.

You can also use "then goto" all the time and not have to worry about smartbranching at all, but "then goto" uses more code space than just "then" so you should take this into consideration as well.

romsize

romsize allows you to specify the size of the generated binary. Currently you may generate 2k or 4k. In the future, you might be allowed to specify an 8k bankswitched ROM, but I make no promises.

optimization

optimization can tell the compiler to try to generate code that runs faster or has a smaller size. Currently, however, the only code that is optimized is division and multiplication.

set may be used anywhere in your program, and may be used more than once, except for romsize. The parameter is active for any code below the set. For example, you may turn optimization to speed for a certain portion of your program, then set it back to none.

Fixed Point Variables

Fixed point variables can be assigned to fractional values, similar to floating point variables in other languages. bB provides two fixed point types:

- a two-byte variable, with one byte as the integer portion and one as the fractional portion. This is referred to as an 8.8 type. These can range from 0 to 255, with a fractional portion that is accurate to within 1/256 or 0.00390625. This type can also be considered as signed in the same way an integer variable is.

- a one-byte variable, with four bits as the integer and four bits as the fraction. This is referred to as a 4.4 type. This type ranges from -8 to 7.9375, and is accurate to within 1/16 or 0.0625. In other words, this type is always signed.

To declare these special types, you must use the dim statement. To declare an 8.8 type, you must specify the integer and fractional portion by the following:

```
dim myvar=a.b
dim monsterx=d.r
```

The first will use the variable a as the integer and b as the fraction, the second will use d as the integer and r as the fraction. Then anytime you use myvar or monsterx in a variable assignment, the compiler will know to use the 8.8 fixed point type.

To declare a 4.4 type, you use a similar syntax as the above, but you use the same variable name for the integer and fraction:

```
dim xvelocity=c.c
dim yvelocity=d.d
```

After this dim, using xvelocity will tell the compiler to use the 4.4 type.

You may use some fixed-point operations without any changes to your program, but some will require you to include the fixed point module. If you need to include the fixed point module, place this somewhere near the beginning of your program:

```
include fixed_point_math.asm
```

Alternatively, you may modify the includes file to include it automatically. See *include* or *includes file* for more information.

In a 2600 game, the 8.8 types are particularly useful for specifying coordinates. The 4.4 types are useful for specifying velocity without using the extra bytes needed in 8.8 types.

This subpixel movement, or movement of a non-integer number of pixels per frame, allows more flexibility in gameplay mechanics than ordinary integer variables provide, or at least it simplifies the calculations that would otherwise be required is only using integers.

The 8.8 type can be used interchangeably anywhere an integer would normally be used. If this is done, for example by assigning an 8.8 to player0x, the fractional portion of the number will be ignored, just like the int() function in other BASIC dialects. The 4.4 types, however, cannot be used anywhere - they can only be added, subtracted or assigned to/from themselves, integers or 8.8 types. Well, that's not totally true. If you use a 4.4 type in some way other than an assignment, addition or subtraction, its value will be multiplied by 16.

Also, note that fixed point types are subject to the same limitations in if-then statements. Although you may compare two 4.4 types in an if-then, if you compare a 4.4 with a number or another type, the 4.4 will be multiplied by 16. If you use an 8.8 type in an if-then, the fractional portion will be ignored.

If you want to use just the fractional portion in an if-then, this can be done by accessing the variable to assigned to the fraction... In the examples above this would be b or r. Note that if you access b or r directly, the fractional portion will be multiplied by 256.

Multiplication and division of fixed point types is subject to the same limitations above.

Some valid operations using fixed point types, that is, ones that are not subject to limitations are listed below. This is not a complete list. Those denoted with a (*) require that the fixed_point_math.asm module is included, those without a (*) do not.

Note: assume that my44 is a 4.4 type, myint is an integer, and my88 is an 8.8:

```
my88=12.662
 my44=4.67
 my88=-12.662
 my44=-4.67
 my88=myint
 my44=myint
 myint=my44
 myint=my88
 my88=my44 (*)
 my44=my88 (*)
 my88=my88+1.45
 my44=my44+2.55
 my88=my88-1.45
 my44=my44-2.55
 my88=my44+6.45 (*)
 my44=my88+3.45 (*)
 my88=my44-6.45 (*)
 my44=my88-3.45 (*)
 my88=my88+my88
 my44=my44+my44
 my88=my88-my88
 my44=my44-my44
 my88=my44+my88 (*)
 my44=my88+my44 (*)
 my88=my44-my88 (*)
 my44=my88-my44 (*)
```

In other words, if you mix 4.4 and 8.8 types in a statement, you need to include the fixed_point_math.asm module.

Variables

You have 26 general purpose variables in batari Basic, fixed as a-z. Although they are fixed, you can use the *dim* command to map an alias to any of these variables.

26 variables is not a lot, so you will use them up quickly. Therefore, it's recommended that you use the *bit operations* to access single bits when using variables for flags or game state information wherever possible.

If you do run out of variables, you can use four bytes from the playfield if you're not scrolling it. You can also use temp1-temp6 as temporary storage, but these are obliterated when drawscreen is run, and some are used for playfield operations as well, so use these at your own risk.

Although there might be unused bytes in the stack space, it is not recommended that you use these in your program since later versions of batari Basic will probably use these for something else.

dim (create descriptive names for variables / assign names to fixed point types)

dim is used to create more descriptive names for the 26 variables, or to create a fixed-point type and assign it to some of these variables.

Using dim to Create More Descriptive Names

Unlike other Basics, the most common use of the dim statement is not for arrays in batari Basic, but rather for creating an alias, a more descriptive name for each variable than a-z. The statement simply maps a descriptive name to any of the original 26 variables.

Although dim is typically called at the beginning of the program, for creating a variable alias, it can actually be called at any time, and is applicable to the entire program no matter where it is placed (this is not true for creating fixed point variables.) The first character of an alias must be one of the 26 letters of the alphabet.

Examples:

```
dim monsterxpos=a
dim monsterypos=b
```

Note that more than one alias may be mapped to the same variable. This is useful for when you will inevitably need to reuse variables in multiple places.

dim to map variables to fixed point types: Although you can use the variables a-z as integers without ever using dim, you cannot use a-z as fixed point variables without using dim. See *fixed point variables* for more information.

Bit Operations (squeeze more out of your variables)

On modern systems, one may not think twice of using an entire byte or even a word for every flag. For example, to determine whether a game is in progress or it is over, often an entire byte is used even though its only value is 0 or 1.

Since the Atari 2600 only has 128 bytes of RAM, and batari Basic only has 26 bytes available for variables, it is very likely that you will need to use individual bits for game state information. For example, a common flag is to determine whether a game is over or it is still in progress.

The bits of a byte are numbered from 0 to 7, with 0 being the least significant bit (LSB) or smallest.

For example, to access the LSB in a variable or register:

```
a{0} = 1
a{0} = 0
if a{0} = 0 then gameover
```

You may also assign one bit to another bit. For example:

```
d{3}=r{4}
f{5}=!f{5}
```

Accessing the bits of a variable is almost like turning it into 8 separate variables. Instead of having only 26 variables, you potentially have 208. You just have to remember that these itty-bitty variables can only hold 0 or 1.

Bitwise (Logical) Operators (&, |, ^)

Batari Basic has three operators for logical operations. They are tokenized as:

& = AND
| = OR (Note: the "|" key is usually above the backslash: "\")
^ = XOR (exclusive OR)

These can be used to change the state of individual bits or to mask multiple bits.

Examples:

```
 a = a & $0F
 a = b ^ %00110000
 a = a | 1
```

let (optional)

The let statement is optional, and is used for variable assignment. It was left in because an early unreleased version of batari Basic required it. If you wish to use it, it will not affect program length—it will simply be ignored by the compiler.

Example:

```
 let x = x + 1
```

Labels and Line Numbers

Batari Basic supports alphanumeric labels. You may use line numbers if you prefer. Some old-school programmers like line numbers, or at least use them as a matter of comfort since they were necessary in early Basics. In any case, labels and line numbers are optional. Typically you will only need them when you want to specify a goto or gosub target.

Labels must not be indented, and all program statements must be. You may use labels with or without program statements after them. A label can have any combination of letters or numbers, even as the first character. You can also use underscores. A label must not match a known keyword or any labels internal to bB (like start, kernel, and so on). For example, you cannot name a label next or pfpixel.

Example:

```
10 pfpixel 2 3 on
20 drawscreen
 player0x=player0x+1:goto mylocation
 player0y=29:goto mylocation2
mylocation
 x=x+1
mylocation2 x=x-1
```

Jumping Around

goto

The goto statement is used to jump to a line number or label anywhere in your program.

Examples:

```
goto 100
goto mysubroutine
```

on … goto

This works similar to a case statement in other languages. It is useful for replacing multiple if-then statements when conditions happen in an ordinal fashion.

For example:

```
on x goto 100 200 300 400
```

is the same as:

```
if x=0 then 100
if x=1 then 200
if x=2 then 300
if x=3 then 400
```

gosub

The gosub statement is often used for a subroutine that is called by multiple locations throughout your program.

Examples:

```
gosub 100
gosub mysubroutine
if x > 10 then gosub sinkship
```

To return control back to the main program, issue a return in your subroutine. Example:

```
mysubroutine
 a = a - 1
 x = x + 10
 return
```

Note that each gosub will use two bytes of stack space, which will be recovered after a return. Only 6 bytes of stack space are reserved for this, so do not use too many nested subroutines, especially since this may be changed in later versions.

return

The return statement is used in a subroutine to return to the part of the program right after a gosub which called the subroutine.

Be careful when using return. If a running program encounters one without a gosub that called it, the program will crash or strange things may happen.

Decision Making (Brains of a Game)

if-then

Perhaps the most important statement is the if-then statement. These can divert the flow of your program based on a condition.

The basic syntax is:

if condition then action

Action can be a statement, label or line number if you prefer. If the **condition** is true, then the statement will be executed. Specifying a line number or label will jump there if the condition is true. Put into numerical terms, the result of any comparison that equals a zero is false, with all other numbers being true.

Note that in specific cases, assembly of if-then statements with a line number or label as the target will fail. If this happens, DASM will report a "branch out of range." One way to fix this is to change the offending if-then statement to if-then goto, or you can let the compiler fix the problem for you by turning on smart branching.

To do this, use the following:

```
rem smartbranching on
```

Place it near the beginning of your program. Be aware that turning *smartbranching* on will slightly obfuscate the generated assembly file, so do not use it if you plan to examine the assembly later to see how it works. See smartbranching for more information.

In batari Basic, there are three types of if-then statements. The first is a simple check where the condition is a single statement.

For example:

```
if a then 20
```

diverts program flow to line 20 if **a** is anything except zero.

This type of if-then statement is more often used for checking the state of various read registers. For example, the joysticks, console switches and hardware collisions are all checked this way (they can't be read any other way.)

For example:

```
if joy0up then x = x + 1
```

That will add 1 to x if the left joystick is pushed up.

```
if switchreset then 200
```

Jumps to line 200 if the reset switch on the console is set.

```
if collision(player1,playfield) then t=1
```

Sets t to 1 if player1 collides with the playfield.

A second type of statement includes a simple comparison. Valid comparisons are = , < , >, <=, >=, and <>.

Examples:

```
if a < 2 then 50
if f = g then f = f + 1
if r <> e then r = e
```

The third type of if-then is a complex of compound statements, that is, one containing a boolean && (AND) or || (OR) operator. You are allowed one boolean for each if-then.

For example:

```
if x < 10 && x > 2 then b = b - 1
if !joy0up && gameover = 0 then 200
if x = 5 || x = 6 then x = x - 4
```

Boolean Operators

Boolean operators are used as conditions in if-then statements. They are tokenized as:

&& = AND

|| = OR

! = NOT

The current version of batari Basic supports at most ONE boolean operator for each if-then statement. The NOT (!) operator may only be used with statements that do not include a comparison token (such as =, <, >, or <>).

For example:

```
if a < 31 && a > 0 then 50
if a = 2 || a = 4 then a = a + 1
if !joy0up then 200
```

else keyword

In Alpha 0.3, you may use "else" after any if-then statement. An if-then will check if a condition is true and divert program flow, but an else allows you to also divert program flow in a different way if the condition turns out to be false.

An else must be on the same line as the if-then that it belongs to. You can include statements separated by colons before the else, but the else must not come after a colon itself.

For example:

```
if r=2 then 20 else 30
 if a>b then r=2:pfpixel 3 5 on: d=d-1 else d=d+1:r=3
 if a>b then 20 else c[4]=12
```

Loops

A common form of a loop is a for-next loop, but a loop in general is any program that repeats. In this sense, all batari Basic programs must be loops, in that the programs never exit.

In batari Basic, you should limit your use of loops that do not include the drawscreen function somewhere. Too many loops take time which is somewhat limited. See *drawscreen* for more information.

for-next

For-next loops work similar to the way they work in other Basics.

The syntax is:

```
for variable = value1 to value2 [step value3]
```

Variable is any variable, and value1, 2, and 3 can be variables or numbers. You may also specify a negative step for value3.

The step keyword is optional. Omitting it will default the step to 1.

Examples:

```
for x = 1 to 10
for a = b to c step d
for l = player0y to 0 step -1
```

next

Normally, one would place a variable after the next keyword, but batari Basic ignores the keyword and instead finds the nearest for and jumps back there. Therefore, the usual way to call next is without a variable. If any variable is specified after a next, it will be ignored.

Example:

```
for x = 1 to 10: a[x] = x : next
```

It is also important to note that the next doesn't care about the program flow—it will instead find the nearest for based on distance.

For example:

```
for x = 1 to 20
goto 100
for g = 2 to 49
100 next
```

The next above WILL NOT jump back to the first for, instead it will jump to the nearest one, even if this statement has never been executed. Therefore, you should be very careful when using next.

Random Numbers

rand

The rand function returns a random number between 1 and 255 every time it is called. You typically call this function by something like this:

```
a = rand
```

However, you can also use it in an if-then statement:

```
if rand < 32 then r = r + 1
```

You can also assign the value of rand to something else, at least until it is accessed again. The only reason you would ever want to do this is to seed the randomizer. If you do this, pay careful attention to the value you store there, since storing a zero in rand will "break" it such that all subsequent reads will also be zero!

Data Statements and Arrays (read-only)

For convenience, you may specify a list of values that will essentially create a read-only array in ROM. You create these lists of values as data tables using the data statement. Although the data statement is similar in its method of operation as in other Basic languages, there are some important differences. Most notably, access to the data does not need to be linear, as with the read function in other Basics.

In batari Basic, any element of the data statement can be accessed at any time. In this vein, it operates like an array. To declare a set of data, use data at the beginning of a line, then include an identifier after the statement. The actual data is included after a linefeed and can continue for a number of lines before being terminated by end. Suppose you declare a data statement as follows, with array name mydata:

```
data mydata
200, 43, 33, 93, 255, 54, 22
end
```

To access the elements of the data table, simply index it like you would an array in RAM. For example, mydata[0] is 200, mydata[1] is 43, ... and mydata[6] is 22. The maximum size for a data table is 256 elements. Note that you may access values beyond the table, but the values you get there probably won't be very useful.

To help prevent this from happening, Alpha 0.3 introduces a new parameter - the data statement length. When you define a data statement, a constant is automatically defined for you - this constant contains the length, or the number of elements, in the data. The constant will have the same name as the name of the data statement, but it will have _length appended to it.

For example, if you declare:

```
 data mydata
 1,2,3,4,5,6,7,8,9
end
```

you can then access the length of the data with mydata_length. You can assign this to variables or use anywhere else you would use a number.

Example:

```
a = mydata_length
```

Note again that these data tables are in ROM. Attempting to write values to data tables with commands like mydata[1]=200 will compile but will perform no function.

The Playfield

In batari Basic, you get a 32 x 11 bitmapped, asymmetric playfield (32 x 12 if you count the hidden row that is only seen if scrolled). This takes the full vertical length of the screen, except for the area reserved for the score, but takes only the center 80% of the screen due to timing constraints. You may use the left or right 10% of the screen, but you can only draw vertical lines there, and they take the full length of the screen. For example, you can put a vertical border around the drawable portion of the playfield by PF0=128. You can use **COLUBK** to set the background color and **COLUPF** to set the playfield color. See the *color chart* for available colors.

Please see *pfpixel, pfvline, pfhline,* and *pfscroll* for more information about drawing to the playfield. Click here if you want to play with a simple interactive playfield graph.

drawscreen

The drawscreen command displays the screen. Any objects with changed colors, positions or height will be updated. Internally, this command runs the display kernel.

Normally, drawscreen is called once within the normal game loop, but it can be called anywhere. The drawscreen operation takes about 12 milliseconds to complete, since it needs to render the entire television display, one scanline at a time. Control will be returned to batari Basic once the visible portion of the screen has been rendered.

It is important to note that the drawscreen command must be run at least 60 times a second. Aside from rendering the visible screen, drawscreen also sends synchronization signals to the television. Failure to run drawscreen quickly enough will result in a display that shakes, jitters, or at worst, rolls.

Therefore, it is possible that your game loop will take up too much time and cause the television to lose sync. Note that your game loop cannot execute for around 2 milliseconds, so you should keep the number of loops and calls to playfield scrolling routines to a minimum. This works out to about 2,700 machine cycles, which can get used up pretty fast if you are doing many complicated operations.

If your screen rolls, jitters or shakes, the only solution is to simplify your operations or to try and spread your operations across two or more television frames by calling drawscreen at strategic times. Note also that doing so may slow your game down if you do not also move your sprites or other objects between calls to drawscreen.

However, at the time of this writing, nobody has complained of batari Basic using too many cycles.

pfpixel (Playfield Pixel) •

This draws a single pixel with playfield blocks. Uses 80 processor cycles every frame. The syntax is:

pfpixel xpos ypos function

Xpos can be 0-31, ypos can be 0-11 (11 is hidden off of the screen and only seen if scrolled.)

Function is any of on, off, or flip. On turns the block on, off turns it off, and flip turns it off if it was on or on if it was off.

Note that there is no checking if the bounds of the function are exceeded. If you do so, strange things may happen, including crashing your program.

pfhline (Playfield Horizontal Line) —

This draws a horizontal line with playfield blocks. Uses 250 to1500 processor cycles every frame depending on length (Approx 210+42*length). The syntax is:

pfhline xpos ypos endxpos function

Xpos can be 0-31, ypos can be 0-11 (11 is hidden off of the screen and only seen if scrolled).

Endxpos should be greater than xpos or the command will not work properly, and strange things may happen.

Function is any of on, off, or flip. On turns the block on, off turns it off, and flip turns it off if it was on or on if it was off.

Note that there is no checking if the bounds of the function are exceeded. If you do so, strange things may happen, including crashing your program.

pfvline (Playfield Vertical Line)

This draws a vertical line with playfield blocks. Uses 230 to 600 processor cycles every frame depending on length (Approx 200+34*length). The syntax is:

pfvline xpos ypos endypos function

Xpos can be 0-31, ypos can be 0-11 (11 is hidden off of the screen and only seen if scrolled).

Endypos should be greater than ypos or the command will not work properly, and strange things may happen.

Function is any of on, off, or flip. On turns the block on, off turns it off, and flip turns it off if it was on or on if it was off.

Note that there is no checking if the bounds of the function are exceeded. If you do so, strange things may happen, including crashing your program.

pfscroll (Playfield Scroll)

This command scrolls the playfield. It is useful for a moving background or other purposes.

Valid values are:

```
pfscroll left
pfscroll right
pfscroll up
pfscroll down
```

Using pfscroll left or right will use quite a few processor cycles every frame (500 cycles), so use it sparingly. Using pfscroll up or down uses 650 cycles every 8th time it's called, 30 cycles otherwise.

When using pfscroll up or down, the hidden blocks at y position 11 are useful. Although these blocks are never seen, they are "scrolled in" to the visible screen by the commands. This invisible area can therefore be used to simulate a changing background, instead of showing the same background over and over again.

Note that if you are not using pfscroll in your program, you can use these hidden 4 blocks as general variable storage. Be careful you don't overwrite them with wayward playfield commands!

One way to use these extra bytes is with dim:

```
dim var1=playfield+44
dim var2=playfield+45
dim var3=playfield+46
dim var4=playfield+47
```

Again, if you choose to do the above, be careful!

pfread function

pfread is used to determine whether an existing playfield pixel is on or off. It can only be used in an if-then statement at this time. You may use numbers, variables or arrays as arguments.

you access it as follows:

```
if pfread(10,8) then 20
if !pfread (a[x], b) then 40
```

score

The score keyword is used to change the score. The score is fixed at 6 digits, and it currently resides permanently at the bottom of the screen. Unlike other variables, batari Basic accepts values from 0-999999 when dealing with the score.

Before the score will appear, you should set its color. Use scorecolor = value, where value is 0 to 255.

To change the score, some examples of valid operations are:

```
score = 1000
score = score + 2000
score = score – 10
```

score = score + a

Be careful when using the last one. It will compile, but upon execution, "a" must always be a BCD compliant number. If a non-BCD number is in "a," part of your score may end up garbled.

What is a BCD Compliant Number?

BCD stands for Binary-coded decimal. In essence, it is a hexadecimal number represented as a decimal number.

For instance, $99 is the BCD number for decimal 99. $23 is the BCD number for decimal 23. There is no BCD number for $3E, for instance, since it contains a non-decimal value (the E.) For example, if "a" contained $3E, your score could end up garbled.

TIA Registers

There are a few TIA registers that may be useful in batari Basic. This is not a complete list. I'm only mentioning the registers and functions therein that you will most likely find useful. You can learn more by visiting the *Stella Programmer's Guide.*

Registers: **NUSIZ0, NUSIZ1**

Changes the size and/or other properties of player0/1 and missile0/1.

Value:	**Effect:**
$0x (x means don't care)	*missile = 1 pixel wide*
$1x	*missile = 2 pixels wide*
$2x	*missile = 4 pixels wide*
$3x	*missile = 8 pixels wide*
$x0	*one copy of player and missile*
$x1	*two close copies of player and missile*
$x2	*two medium copies of player and missile*
$x3	three close copies of player and missile
$x4	two wide copies of player and missile
$x5	double-sized player
$x6	three medium copies of player and missile
$x7	quad-sized player

Note that missile and player properties may be combined in a single write.

Example: NUSIZ0=$13 will make missile0 8 wide, plus make three close copies of player0 and missile0.

Register: CTRLPF
Changes properties of the playfield and/or ball.

Value:	**Effect:**
$0x (x means don't care)	ball = 1 pixel wide
$1x	ball = 2 pixels wide
$2x	ball = 4 pixels wide
$3x	ball = 8 pixels wide
$x1	None of the below
$x3	left half of playfield gets color of player0, right half gets color of player1
$x5	players move behind playfield
$x7	Both of the above

Note that ball and playfield properties may be combined in a single write.

Registers: REFP0, REFP1

Reflects player sprites. (Flip it horizontally.)

Value:	Effect:
0	Do not reflect
8	Reflect

This is useful for asymmetric sprites so that they can give the appearance of changing direction without needing to redefine their graphics.

Registers: PF0

Set or clear the left and right 10% of the playfield.

Value:	Effect:
$0x through $Fx	Set vertical lines covering entire height of playfield

PF0 is useful for creating a border in batari Basic. In other kernels or in assembly, it has other uses.

Registers: AUDC0, AUDC1, AUDF0, AUDF1, AUDV0, AUDV1

See *sound* for more information about these.

Objects

Player Graphics

The Atari 2600 can display two player sprites, which are 8 pixels wide and any height. In batari Basic, you access these sprites by using various reserved values and commands. To define a sprite, you use **player0:** and **player1:**

Example:

```
player0:
 %00100010
 %01110111
%01111111
end
```

This will define a player0 sprite which is 3 blocks in height.

Note that the bytes that make up a sprite are upside down in your code. Sprites will be flipped in the game. If you have trouble visualizing, try the built-in Sprite Editor in attendo's batari Basic IDE.

To display the objects, you must first set the colors with **COLUP0** and **COLUP1**. They are black by default, which will not display against a black background.

To set the coordinates, you set player0x, player0y, player1x, or player1y. On the screen, player0x and player1x values between 0 and around 164 are useful. You can specify numbers larger than 164 but you may see some jumping at the edges of the screen. Values of player0y and player1y between 0 and about 88 are useful. Others will simply cause the player to move off of the screen.

Missiles

Batari Basic can display two missiles on the screen. These are simply vertical lines of a height you specify, and at coordinates you specify. The missiles are the same color as their respective players.

To access missiles, you can set missile0x, missile0y, and missile0height for missile 0, and missile1x, missile1y, and missile1height for missile 1.

There are more things you can do with missiles by modifying the TIA registers. See *TIA Registers* for more information.

Ball

The ball is one of the objects that the Atari 2600 can display in the screen.
The ball is the same color as the playfield. It is accessed by ballx, bally, and ballheight, much like accessing the missiles.

Collision Detection

if collision (object,object)

This function is used for checking if two objects have collided on the screen. Valid arguments are **playfield, ball, player0, player1, missile0, missile1**. The two objects can be specified in any order.

The collision() function is only valid when used in an if-then statement.

Examples:

```
if collision(playfield,player0) then a = a + 1
 if !collision(player0,missile1) then 400
```

Sound

There are no special functions for accessing sound in batari Basic. Instead, you must access the TIA registers for sound directly. Don't panic, the TIA registers for sound are quite friendly, at least as far as that damn TIA goes.

AUDV0
Channel 0 Volume (valid values are 0 to 15)

AUDC0
Channel 0 Tone [Distortion] (valid values are 0 to 15)

AUDF0
Channel 0 Frequency (valid values are 0 to 31)

AUDV1
Channel 1 Volume (valid values are 0 to 15)

AUDC1
Channel 1 Tone [Distortion] (valid values are 0 to 15)

AUDF1
Channel 1 Frequency (valid values are 0 to 31)

Setting the values, for instance, by **AUDV0 = 10 : AUDC0 = 12 : AUDF0 = 4** will produce a tone, and it will stay on until you set AUDV0 = 0. Typically, a frame counter is set up that keeps sounds on for a specified number of frames (which occur 60 times a second).

The following is adapted from Atari 2600 VCS Precise Sound Values and Distortion Breakdown by Glenn Saunders:

Tone	What it Sounds Like
0	No sound (silent).
1	Buzzy tones.
2	Carries distortion 1 downward into a rumble.
3	Flangy wavering tones, like a UFO.
4	Pure tone.
5	Same as 4.
6	Between pure tone and buzzy tone (Adventure death uses this).
7	Reedy tones, much brighter, down to Enduro car rumble.
8	White noise/explosions/lightning, jet/spacecraft engine.
9	Same as 7.
10	Same as 6.
11	Same as 0.
12	Pure tone, goes much lower in pitch than 4 & 5.
13	Same as 12.
14	Electronic tones, mostly lows, extends to rumble.
15	Electronic tones, mostly lows, extends to rumble.

Sound and Music Resources

Atari 2600 VCS Precise Sound Values and Distortion Breakdown

Atari 2600 VCS Sound Frequency and Waveform Guide (http://home.arcor.de/estolberg/texts/freqform.txt)

the BASICs of batari music (http://alienbill.com/2600/basic/music/)

Joysticks

Joysticks are read by using an if-then statement. There are four directional functions and one fire function for each joystick:

if joy0up
True if left joystick is pushed up.

if joy0down
True if left joystick is pushed down.

if joy0left
True if left joystick is pushed left.

if joy0right
True if left joystick is pushed right.

if joy0fire
True if left joystick's fire button is pushed.

if joy1up
True if right joystick is pushed up.

if joy1down
True if right joystick is pushed down.

if joy1left
True if right joystick is pushed left.

if joy1right
True if right joystick is pushed right.

if joy1fire
True if right joystick's fire button is pushed.

Example:

if joy0up then x = x + 1

These can also be inverted using the NOT (!) token. For example:

if !joyup then 230

Console Switches

Reading the console switches is done by using an if-then statement.

if switchreset
True if Reset is pressed.

if switchbw
True if the COLOR/BW switch is set to BW, false if set to COLOR.

if switchselect
True if Select is pressed.

if switchleftb
True if left difficulty is set to B (amateur), false if A (pro).

if switchrightb
True if right difficulty is set to B (amateur), false if A (pro).

These are accessed by, for example:

if switchreset then 200

These can all be inverted by the NOT (!) token:

if !switchreset then 200

Numbers

Decimal Numbers

Numbers in batari Basic are assumed to be in decimal unless otherwise specified by either the **$** (for hexadecimal) or the **%** (for binary).

One exception is signed numbers with the negative bit set, when expressed as a negative. See **negative numbers** for more information.

Hexadecimal Numbers

Often it is handy to express hexadecimal numbers in your Basic program. Simply place the $ before a number to use hexadecimal.

Hex	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Decimal	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Examples:

COLUPF = $2E
a[$12] = $F5

Binary Numbers

Sometimes it is convenient to express numbers as binary instead of decimal or hexadecimal. To express numbers as binary, place the % before a number. Make sure that you define all 8 bits in the byte. The % operator is particularly useful for accessing certain TIA registers that expect individual bits to be set or cleared, without needing to first convert the numbers to hexadecimal or decimal first. The % operator is also useful for defining player sprites.

```
Binary     1   1   1   1   1   1   1   1
Decimal  128  64  32  16   8   4   2   1
```

Examples:

CTRLPF = %00100010
player0:
%00100010
%11100111
end

Negative Numbers

Negative numbers are somewhat supported by batari Basic. Although variable values can contain 0-255, the numbers will wrap if 255 is exceeded. Therefore, one can think of numbers from 128-255 as being functionally equal to -128 to -1. This is called "two's compliment" form because the high bit is set from 128-255, so this high bit can also be called the "sign."

In other words, adding 255 to a variable has exactly the same effect as subtracting 1.

Math

Addition

The + operator is used for addition. For most variables and registers, batari Basic supports simple expressions only, and accepts any combination of registers, variables, unsigned values from 0-255 or signed values from -128 to 127 (see also negative numbers).

If the addition causes the result to equal or exceed 256, it will be wrapped around at 0. For instance, 255+1=0, 255+2=1, ... 255+255=254.

An exception is the score, which can work with values from 0 - 999999.

Examples:

a = a + 1
COLUPF = r + $5F
player0y = player1y + 6

Subtraction

The - operator is used for subtraction. For most variables and registers, batari Basic supports simple expressions only, and accepts any combination of registers, variables, unsigned values from 0-255 or signed values from -128 to 127 (see also negative numbers).

If the subtraction causes the result to be less than 0, it will be wrapped around to 255. For instance, 0-1=255, 1-2=255, ... 0-255=1.

An exception is the score, which can work with values from 0 - 999999.

Examples:

```
a = a - 1
COLUPF = r - $5F
player0y = player1y - 6
```

Multiplication

bB now supports full multiplication of two integer numbers instead of just multiplying by two. However, some multiplication operations require you to include a module. In particular, multiplying two variables together requires the module, as does multiplication of a variable by a number greater than 10 (see exception). Multiplication of a variable to a number 10 or less does not require you to include a module. The exception to this rule is any number that is a power of two, i.e. 16, 32, 64, or 128.

If the multiplication causes the result to exceed 255, the result will probably be bogus. There is a way around this - if you use ** as the multiplication operator instead of *, the result will be stored in 16 bits and the value will wrap properly. In this case, the lower byte of the result will be assigned to the result, and the variable temp1 will contain the upper byte of the result. You should use ** only when you need it, as it takes up additional space and cycles in your program.

The division and multiplication are packaged as a single module. If you need to include the module, place this line near the beginning of your program:

```
include div_mul.asm
```

If you are using **, however, you should include the 16-bit module, by:

```
include div_mul16.asm
```

Or you may place it in an includes file to include it automatically. See include or includes file for more information.

Division

bB now supports full division of two integer numbers instead of just dividing by two. However, some division operations require you to include a module. In particular, dividing two variables requires the module, as does division by any number except a power of two, i.e. dividing by 2, 4, 8, 16, 32, 64, or 128 can be done without the module. You can divide by 1 as well, but I can't imagine why you would want to. If you try to divide by zero, no operation will occur and your program will continue to run.

The division operation will return an integer result, meaning that any fractional portion or remainder will be lost. If you need the remainder, however, you can use the // operator instead. The remainder will then be stored in temp1.

The division and multiplication are packaged as a single module. If you need to include the module, place this line near the beginning of your program:

```
include div_mul.asm
```

If you are using //, however, you should include the 16-bit module, by:

```
include div_mul16.asm
```

Or you may place it in an includes file to include it automatically. See include or includes file for more information.

Assembly Language

asm

Use the asm statement to insert inline assembly language into your program. You do not need to preserve any register values when using this feature, except the stack pointer. Mnemonics should be indented by at least one space, and labels should not be indented.

Example (clears the playfield)

```
asm
ldx #47
lda #0
playfieldclear
sta
playfield,x
dex
bne playfieldclear
end
```

You may also access any variables from assembly that are defined in batari Basic. For example, another way to express the statement a=20 is:

```
asm
lda #20
sta a
end
```