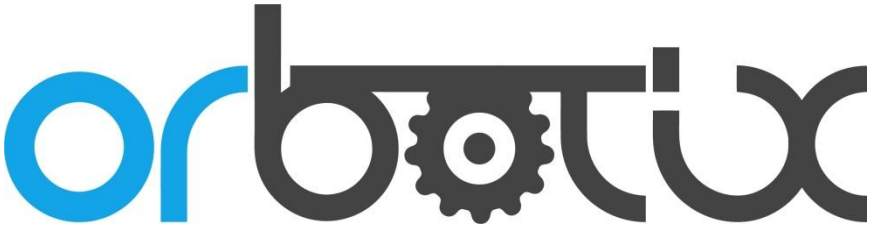
Gen 1 orbBasic Interpreter

By Dan Danknick

document revision 1.05



Introduction

Back in April 2011 when I was laying out the memory map of Sphero, co-founder Adam Wilson asked me to set aside some sort of "overlay area" in which temporary routines could be dynamically dropped in for execution. I considered various options and weighed their risk against implementation in a sealed robot. Eventually I pulled out my old copy of *Microsoft Basic Decoded* book to take a look at how Bill Gates had written his BASIC language on the 8-bit TRS-80.

After a weekend of review and a lot of coffee, I settled on using Adam Dunkel's source code for *µBasic* and bolted it into Sphero. It worked! I mothballed the project until the remainder of the Sphero firmware was well underway and stable, then in February 2012 I began reworking the µBasic code into what I call orbBasic, a stand-alone Basic interpreter suitable for Sphero.

I fixed a number of inherent bugs, extended the language greatly by adding new intrinsic variables, operators and verbs while optimizing the tokenizer, verb dispatcher and adding a line number cache.

orbBasic, like the macro executive, is another powerful resource that Sphero provides for autonomous behavior and integration with smart phone applications. It runs as another thread in the main system in parallel with driving and macros. With execution rates of well over 3000 lines/second and support for both a helper program (resident in RAM) and a permanent one (resident in Flash), orbBasic adds an entirely new dimension to gaming and apps. It is the ultimate helper app.

This document covers the first generation interpreter and its minor revisions. Although I refer to tokens, the program stream itself is not tokenized and still resides as ASCII in the execution area. If the demand for orbBasic takes off and we need more power – or to store longer programs – then I will implement a second generation interpreter that operates on a tokenized stream: Level II orbBasic. My guess is this would more than double the execution speed from 9,000 lines/sec to 18,000.

March 16, 2012
Boulder, CO

Program Format

Ultimately an orbBasic program is a sequence of program lines ending with a NULL (00h) character.

```
<program> := <line(s)> <NULL>
```

Each line begins with a line number, some whitespace, a program statement and a terminating LF (10h) character:

```
<line> := <line number> <space(s)> <statement> <LF>
```

The interpreter is given a starting line number when invoked as programs do not need to start from the first line in the program. Once the statement on a line is complete, execution continues at the next line in the program stream which may not actually be the next line *numerically* in the program. Although this is a side effect of a less than well-formed program, it is entirely legal. Good practice is to send down programs where the line numbers are in ascending order.

Statements are covered in detail below but currently only one statement per line is supported. A future enhancement to the interpreter will be to add the colon ":" token which separates multiple statements on a line.

Variables

User Variables

50 unique user controlled variables are supported, half directly named and the other half indexed:

- Direct variables are named A..Y (or as a..y if you prefer)
- Z is in the index variable and when accessed, the Yth index is dereferenced, implying Z(Y). There are no zero indices in Basic so the valid range of Y when accessing Z is 1..25

All variables are 32-bit signed integers yielding a range of -2,147,483,647 to +2,147,483,647. If you want to use fractions then you will need to learn how to use fixed point math (Appendix C).

There is currently no support for strings, other than their temporary assembly in print statements.

When a program is run, all variables, timers, pending delays and flags are initialized to zero. You can assume a clean slate.

System Variables

A number of special system variables are implemented allowing enhanced functionality and control.

timerA, timerB, timerC

Access	Size	Available
read, write	16-bit positive value	ver 0.9

These are built-in timers that count down to zero by themselves, once per millisecond. They are both readable and writeable. The following demonstrates a 125ms delay between print statements (though it would be smarter to use the `delay` keyword):

```
10 print "Before"
20 timerA = 125
30 if timerA > 0 then goto 30
40 print "After"
```

You can also use them to time events:

```
10 timerB = 32000       ' 32 second window
20 if speed > 0 then goto 20
30 A = (32000-timerB)/1000
40 if A = 0 then print "Never stopped!" else print A " seconds to stop"
```

ctrl

Access	Size	Available
read, write	1-bit value	ver 0.9

When read, this variable returns 0 if the control system if off and 1 if it is on. When assigned, 0 turns off the control system and any other value turns it on.

Example:

```
10 ctrl = 1       'change to 0 and re-run
20 print "Control system is",ctrl,;
30 if ctrl = 1 then ?"(on)" else ?"(off)"
```

speed

Access	Size	Available
read	8-bit positive value	ver 0.9

Returns the approximate speed of the robot in real time. This value is the filtered average speed of both motors so there is some delay between what you see and what this returns. It is read-only and ranges from 0..255.

yaw

Access	Size	Available
read	16-bit positive value	ver 0.9

Returns the current heading of the robot as reported by the IMU. It is read-only and ranges from 0..359 in the usual convention of how Sphero manages headings. This program turns the back LED on and off in 45 degree sectors; download it and spin Sphero like a top.

```
10 ctrl = 0
20 LEDC 8
30 X = 255 * ((yaw/45) & 1)
40 backLED X
50 goto 30
```

pitch

Access	Size	Available
read	16-bit positive value	ver 0.9

Returns the current pitch angle of the robot as reported by the IMU. It is read-only and ranges from -90 when the front of Sphero is pointing straight down to +90 when it is pointing straight up.

roll

Access	Size	Available
read	16-bit positive value	ver 0.9

Returns the current roll angle of the robot as reported by the IMU. It is read-only and ranges from -180 as Sphero rolls to the left and +180 when it rolls completely over to the right.

Here is a good program to display the relationship of all three of the above variables.

```
10 ctrl = 0
20 print pitch, roll, yaw
30 delay 100
40 goto 20
```

accelX, accelY, accelZ

Access	Size	Available
read	16-bit signed value	ver 0.9

Returns the current filtered reading from the accelerometer. The range is -32768 to +32767 for full-scale, which defaults to ±8Gs. The convention is that X is the pitch axis, Y is roll and Z is yaw.

gyroX, gyroY, gyroZ

Access	Size	Available
read	16-bit signed value	ver 0.9

Returns the current filtered reading from the rate gyro in units of 0.1 degrees/second. The range is -20000 to +20000 for full-scale. The convention is that X is the pitch axis, Y is roll and Z is yaw.

Vbatt

Access	Size	Available
read	16-bit unsigned value	ver 0.9

Returns the current battery voltage in 100ths of a volt, so a return value of 756 indicates 7.56V.

Sbatt

Access	Size	Available
read	16-bit unsigned value	ver 0.9

Returns the current state of the power system, as follows:

Value	State
1	Battery is charging
2	Battery is OK
3	Battery voltage low
4	Battery voltage critical

cmdval

Access	Size	Available
read	Boolean	ver 0.9

Returns a value of 1 when a new roll command has been sent by a Bluetooth client, essentially making this a "fresh data" flag. Reading this system variable automatically sets it to zero.

Roll commands usually don't arrive faster than 10Hz so there isn't much use in checking this flag faster than every 100ms. When this variable reads as one, the other two system variables `spdval` and `hdgval` will have fresh values in them.

spdval

Access	Size	Available
read	8-bit positive value	ver 0.9

Returns the last commanded speed portion of a roll command sent from the smartphone client, ranging from 0 (stop) to 255 (full speed).

hdgval

Access	Size	Available
read	16-bit positive value	ver 0.9

Returns the last command heading portion of a roll command sent from the smartphone client ranging from 0 to 359 degrees.

```
10 if cmdval > 0 then ?"New heading, speed:" hdgval, spdval
20 delay 100
30 goto 10
```

isconn

Access	Size	Available
read	Boolean	ver 0.9

This variable provides real time access to the current state of a connected smartphone over Bluetooth.

```
10 if isconn > 0 then LEDC 1 else LEDC 2
20 delay 100
30 goto 10
```

dshake

Access	Size	Available
read	Boolean	ver 0.9

Returns a value of 1 when double-shake event has been detected. Reading this system variable automatically sets it to zero.

```
10 if dshake > 0 then LEDC 1 else LEDC 2
20 delay 100
30 goto 10
```

accelone

Access	Size	Available
read	16-bit positive value	ver 0.9

Returns the effective acceleration vector that Sphero is experiencing. The IMU computes this as the vector sum of the X, Y and Z axis components so it is expressed here multiplied by 1,000. A perfectly calibrated Sphero will return 1,000 when still, 0 when in free fall and up to 8,000 when being shaken. But since few things are perfect, you will need to apply some soft ranges to convert this hard value to a state.

```
10 if accelone < 200 then ? "Free fall! (" accelone ")"
20 if accelone > 900 then ?"Still"
30 delay 200
40 goto 10
```

xpos, ypos

Access	Size	Available
read	16-bit signed value	ver 0.9

Returns the current x and y position of Sphero as determined by the internal locator. The approximate scale is in centimeters. Use the function `locate` to assign these values.

Run this program in the background while using the Drive App to move Sphero in and out of the 40cm circle.

```
10 locate 0,0
20 delay 100
30 R = sqrt (xpos*xpos + ypos*ypos)
40 if R < 20 then LEDC 1 else LEDC 2
50 goto 20
```

Expressions

Simple mathematical operators are supported up to a reasonable expression depth.

Operator	Description	Example	Precedence
(	Begin a new subexpression	2 + 3 * 5 yields 17	high
)	End a subexpression	(2 + 3) *5 yields 25	high
*	Integer multiplication	2 *3 yields 6	medium
/	Integer division	10 / 3 yields 3	medium
%	Integer remainder	10 % 3 yields 1	medium
{	Binary left shift	3 { 2 yields 12	medium
}	Binary right shift	20 } 2 yields 5	medium
+	Addition	1 + 2 yields 3	low
-	Subtraction	1 - 2 yields -1	low
&	Bitwise AND	45 & 85 yields 5	low
|	Bitwise OR	45 | 85 yields 125	low

My goal was to allow an expression anywhere a variable or a numeric literal was permitted, offering the most flexibility. But this may not always be the case (testing will reveal!)

Functions

Few of these are supported but if demanded, I will expand the list. I started with the most useful ones.

sqrt *x*

Access	Size	Available
read	32-bit positive value	ver 0.9

Returns the square root of any expression that follows. A negative number halts on error.

Since orbBasic only supports integer math you need to understand the following identity to really use this for smaller numbers:

$\sqrt{a \cdot b} = \sqrt{a} \cdot \sqrt{b}$ where a is the number you want to take the square root of and b is a multiplier to increase the precision. Since the maximum positive value of any expression in orbBasic is around 2.1 million, choosing b as 100^2 or 10,000 is a good choice. Consider the following example where you want to take the square root of 42:

Let a = 42 and b = 10000, so `sqrt 420000` yields 648. Since $\sqrt{100000}$ equals 100, this answer is 100 times too large. Which lines up with the real answer of $\sqrt{42}$ which is about 6.48

```
10 for X = 10 to 100 step 10
20 Q = sqrt X*10000
30 print "sqrt(" X ") = " Q/100 "." Q % 100
40 next X
```

outputs

```
sqrt(10) = 3.16
sqrt(20) = 4.47
sqrt(30) = 5.47
sqrt(40) = 6.32
sqrt(50) = 7.7
sqrt(60) = 7.74
sqrt(70) = 8.36
sqrt(80) = 8.94
sqrt(90) = 9.48
sqrt(100) = 10.0
```

Using a multiplier of 10,000 to extract two extra decimal places of precision in your answer will allow you to take the square root of up to about 214,000 or 462^2.

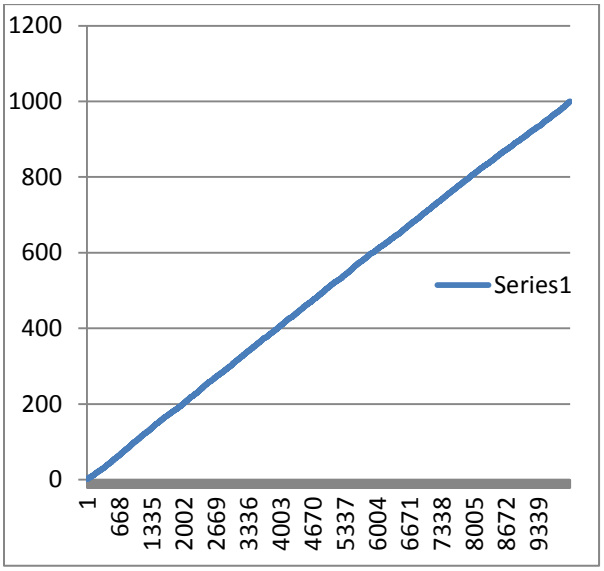
rnd *x*

Access	Size	Available
read	32-bit positive value	ver 0.9

This returns a pseudo random number between 1 and the value provided, which can be either a literal number or an expression. Use the `random` statement to reseed the number generator. For example:

```
10 for X = 1 to 50
20 RGB rnd 255,rnd 255,rnd 255
30 delay rnd 250
40 next X
```

Here is a graph showing the distribution of 10,000 sequentially generated random numbers. It's uniform enough for government work.



abs *x*

Access	Size	Available
read	32-bit positive value	ver 0.9

This returns the absolute value of an expression, which is always positive.

Statements

These are a mix of the traditional ones supported in Basic plus extensions that allow interaction with Sphero's other systems. Case is regarded so `goto` is not the same as `GOTO`. I chose statements in lower case so that a) programs don't look like shouting in email and b) capitalizing user variables helps them stand out.

Conditional

This is the classic if…then…else construct. In some variants of Basic the `then` keyword was implied and therefor optional; here it is required. The `else` clause is optional. You can use `and/or` to glue two relations together if necessary.

```
<conditional> := if <relation> { and, or } <relation> then <statement>
      { else <statement> }

<relation> := <expression> <operator> <expression>
```

Currently the only operators supported are:

Expression Operator	Description
=	Equality
!	Inequality
<	Less than
>	Greater than

Examples:

- `if A > 5 then A = 5`
- `if A*B = B*C then goto 60`
- `if timer1 = 0 then print "Out of time!"`
- `if A ! B then print "Not equal"`
- `if A>1 and A<10 then ?"In range" else return`

It isn't possible to accommodate every possible nested statement in the true/false clause processing but the following are supported: `print`, `data`, `RGB`, `LEDC`, `read`, `rstr`, `goto`, `gosub`, `on X goto/gosub`, `return`, `delay`, `backLED`, `tron`, `troff` and `goroll`.

Looping

```
<loop> := for <loop variable> = <start value> to <end value>
      { step <step value> }
{ statements }
next <loop variable>
```

The standard method for looping over a number range in orbBasic is the for..next loop. The loop variable can be any user variable. The start and end values can be literal numbers or expressions – even involving other variables (both user and system). The step value is optional and if excluded, 1 is assumed. Of course this won't work if your ending value is lower than your starting one.

There is a maximum nesting level and when exceeded an appropriate error is generated. It is acceptable to change the loop variable in the middle of the loop, for example, if you wish to terminate before completing the original numbers of loops. (If you choose to do this just remember that the loop terminates when the loop variable falls outside the end value.)

Examples:

- `for X = 1 to 5 step 2  …  next x`
  - `yields 3 loops with X = 1,3,5`
- `for Y = 10 to 3 step -3  … next x`
  - `yields 3 loops with Y = 10,7,4`

Branching

```
<branch> := goto <line number> | gosub <line number>
```

This is pretty straight forward. In both cases the target line number must be a numeric literal (i.e. it cannot be an expression). Not too surprisingly the way to return from a subroutine is to use the `return` statement. There is a maximum nesting level and when exceeded, an appropriate error is generated.

Indexed Branching

```
<indexed branch> := on <expression> goto, gosub <line number list >
```

This statement implements an N-way transfer of control to one of N line numbers. The expression must evaluate to a number between 1 and N, when there are N line numbers in the list.

Example:

- on X*Y goto 100,200,300,400

print

orbBasic wouldn't be too useful if you couldn't export information from your running program back to the smartphone client. The `print` statement takes care of this.

The final length of each print statement is limited to 32 bytes. (If you exceed this, a warning message is printed instead.) The `print` statement is a sequence of string literals and expressions. String literals get copied exactly as written while expressions are evaluated and their numeric value is expressed.

if the $ symbol precedes an expression, it is evaluated but instead of the number itself a single character, the ASCII value of the expression, is emitted instead.

Commas between print elements insert a space in the output. If the final character of the print string is terminated with a semicolon then no linefeed is appended to the message sent back to the client.

The output of `print` is enqueued for transmit so it is possible to fill up that queue faster than it can be emptied over the Bluetooth link. If this happens, data is not lost but program execution is paused until the `print` statement can complete normally.

The single `?` character is shorthand for `print` and can help save program space.

Examples:

```
10 print "The answer is: " A*44

10 ? "Line " L ": the speed is: " speed;

10 for X = 65 to 67
20 print X " as a number, " $X " as a char"
30 next X
```

outputs

```
65 as a number, A as a char
66 as a number, B as a char
67 as a number, C as a char
```

input

Access	Size	Available
see below	see below	ver 1.5

Just like `print`, the absence of an input statement would severely limit the interaction of orbBasic with the user (or a smartphone app). Unlike most forms of BASIC, our `input` statement takes two forms because Sphero is a multitasking robot and running an orbBasic program is just ONE of the things it can do.

The first form is the traditional blocking type you've seen before:

```
10 print "Enter a number:"
20 input X
30 print "You entered " X ";
40 print " and half of that is " X/2
```

In this form the input statement will wait forever until the API command to deliver a number is sent to Sphero. (Note that the API command is DID 02h, CID 64h) The only other way out is by sending an abort command to the orbBasic interpreter.

The second form helps manage the infinite wait:

```
10 print "Enter a number:"
20 input X,3000,0
30 if X = 0 then ?"No answer!" else ?"You entered " X
```

The two extra parameters added to the input statement are respectively how long to wait for an answer (in milliseconds) and what value to assign to the variable if that time limit expires. In this example the program will wait for 3 seconds and if no API command delivers a new value for X, it is assigned a zero.

delay *x*

Access	Size	Available
parameterized	16-bit positive value	ver 0.9

This is a built-in delay statement that doesn't use the three timers. It takes one parameter, an expression resolving to the number of milliseconds in the delay. Here is an interesting use of this statement:

```
10 for X = 1 to 74
20 RGB X,X,X
30 delay X*4
40 RGB 0,0,0
50 delay X{1     'same as X*2
60 next X
```

end

Programs terminate in the following ways: the final line number is executed, an error occurs, an abort command is sent or an `end` statement is encountered. This statement allows you to, well, end in the middle of a program.

RGB *r,g,b*

Access	Size	Available
parameterized	8-bit positive values	ver 0.9

This commands the RGB LED with the three provided parameters, each of which is evaluated as an expression. Values range from 0,0,0 (off) to 255,255,255 (white). Note that this statement is in all-caps since its still an abbreviation. The RGB LED is updated every 1ms so changing its value faster than this with orbBasic is ineffective.

When orbBasic is running, the RGB LED assumes the color of any programmatic RGB commands *unless* a macro is also running – where it will take the value of the macro commands. You can override this precedence with an orbBasic flag, however.

LEDC *x*

Access	Size	Available
parameterized	8-bit positive value	ver 0.9

This is like the RGB command but sets the LED to one of eight predefined colors, without requiring you to know the three component values.

Value	Color
0	<off>
1	red
2	green
3	blue
4	orange
5	purple
6	white
7	yellow
8	<off>

Example:

```
10 for X = 1 to 8
20 LEDC X
30 delay 100
40 next X
```

Advanced example:

```
10 for X = 1 to 8
20 LEDC (X % 2) * 5  'modulo 2 operation returns 0 or 1
30 delay 100
40 next X
```

backLED *x*

Access	Size	Available
parameterized	8-bit positive value	ver 0.9

This commands the aiming LED with an intensity from 0..255. For example:

```
10 X = 1
20 backLED X
30 delay 50
40 X = X { 1
50 if X < 256 then goto 20
60 goto 10
```

random

This reseeds the pseudorandom number generator based on the jitter of the sensor network. Since this operation is performed prior to the execution of an orbBasic program, you shouldn't normally need to call this – but hey, you can if you wish.

goroll *h,s,g*

Access	Size	Available
parameterized	three values, see below	ver 1.2

This executes a roll command to the control system with the provided values of heading and speed. Heading is over the usual range of 0..359 and speed over 0..255. The last parameter, a flag for "go" tells the control system whether to force fast rotate to on (g=2), attempt to drive normally (g = 1) or perform a controlled stop (g=0).

If you haven't turned on exclusive drive ability from orbBasic via the `basflg` statement, roll commands from macros or the Bluetooth client can overwrite this and cause unexpected behavior.

heading *h*

Access	Size	Available
parameterized	16-bit positive value	ver 1.2

This assigns the current yaw heading to the provided value, in the range of 0..359.

raw *Lmode,Lspeed,Rmode,Rspeed*

Access	Size	Available
parameterized	four values, see below	ver 1.3

This allows you to take over one or both of the motor output values, instead of having the stabilization system control them. Each motor (left and right) requires a mode (see below) and a power value from 0-255. This command will disable stabilization if both modes aren't "ignore" so you'll need to re-enable it once you're done.

Mode	Description
0	Off (motor is open circuit)
1	Forward
2	Reverse
3	Brake (motor is shorted)
4	Ignore (motor mode and power is left unchanged)

locate *x,y*

Access	Size	Available
parameterized	two values, see below	ver 0.9

This assigns the XY position of the internal locator to the provided values.

```
10 locate 0,0
20 goroll 0,100
30 delay 3000
40 goroll 0,0
50 delay 1000
60 print xpos,ypos
```

basflg *x*

Access	Size	Available
write only	16-bit positive value	ver 0.9

This statement accepts a decimal value to assign state flags that are effective during execution of the program. If you want to set multiple flags then add up their values.

At the end of program execution these effect of these flags is removed. I will be adding more as the need arises.

Value	Symbolic Name	Description
1	BF_EXCLUSIVE_DRV	Gives the program exclusive control of driving and excludes those commands from the Bluetooth client.
2	BF_STEALTH	Execution of an orbBasic program will NOT reset the inactivity timeout
4	BF_KEEP_LED	Normally macros own the RGB LED even when orbBasic is executing; setting this flag will keep orbBasic as the owner.

data *d1{,d2,d3…}*

Access	Size	Available
parameterized	32-bit signed values	ver 0.9

Defines a set of constant data values that can be sequentially read via the `read` statement. You can specify more than the maximum number of values (currently set to 25) and they will be ignored. When this statement is encountered, previously stored values are overwritten so it is possible to re-assign the constants during program execution.

rstr

Forces the next `read` statement to start back at the beginning of the current constant data set.

read *X{,Y…}*

Access	Size	Available
parameterized	One or more 32-bit signed values	ver 1.2

Reads the next value(s) from the current data set into the specified variable(s). If you attempt to read past the end, you will get an error.

Example:

```
10 data 10,20,30      'data table size is 3
20 read A,B
30 rstr
40 read C
50 print A,B,C
60 data 100           'data table size is now 1
70 read A
80 print A,B,C
```

displays:

```
10 20 10
100 20 10
```

tron

This turns line number tracing on to aid in debugging your program. Line numbers are emitted through the standard print channel prior to the execution of that line. This statement takes effect at the next line number.

For example:

```
10 print "at line 10"
20 tron
30 ? "at line 30"
40 ? "and now, 40"
```

displays:

```
at line 10
<30>at line 30
<40>and now, 40
```

troff

This turns line number tracing off.

reset

This resets the orbBasic environment and restarts the program freshly from the initial starting line number. All gosub and for/next frames are cleared and all variables set to zero.

sleep *duration, macro, line_number*

Access	Size	Available
parameterized	see below	ver 0.9

This command immediately puts Sphero to sleep with a few modifiers:

parameter	Description
duration	If non-zero, the number of seconds in the future to automatically reawaken from 1 to 65535. Zero causes Sphero to sleep forever (well, until double-shaken awake).
macro	The macro ID to run upon reawakening; zero for none. This ID cannot be the temporary or stream macro ID (254 or 255).
line_number	The line number of an orbBasic program to run on reawakening; zero for none. Note that this line number relates to programs stored in the Flash area.

Since macros and orbBasic programs can run simultaneously you can specify both actions to occur.

macrun *x*

Access	Size	Available
parameterized	8-bit unsigned value	ver 0.9

Attempts to run the macro number specified. If a macro is currently running, this will have no effect (and will not cause an error).

mackill

Kills any currently executing macro, with the exception of system macros that have the "unkillable" flag bit set.

macstat

Access	Size	Available
read only	32-bit unsigned value	ver 0.9

This returns information about the currently executing macro. In the upper 16 bits is the ID of the currently executing macro, or zero if none is running. In the lower 16 bits is the command number currently being executed.

```
10 X = macstat
20 M = X } 16
30 if M = 0 then ?"None running" else ?"Macro:" M, "Cmd:" X & 65535
```

Appendix A: Errors Reference

Since orbBasic isn't tokenized during download both syntactic and runtime errors can occur. ASCII versions are sent via asynchronous message ID 09h in the format:

Line xx: <error message><LF>

or as a 4-byte binary message through ID 0Ah in the format:

Line #	Line #	Error #	Error #
<msb>	<lsb>	<msb>	<lsb>

The following error messages are currently defined:

Error Message	Code	Description
Syntax error	01h	This is somewhat of a catch all, indicating something unexpected was encountered during interpretation of a statement or expression(versus during execution of the said element).
Unknown statement	02h	An unknown keyword was encountered, perhaps due to a capitalization error (i.e. "rgb" was entered instead of "RGB").
GOSUB depth exceeded	03h	Too many nested gosub statements were encountered.
RETURN without GOSUB	04h	A return statement was encountered but there was no previous gosub command that matched it.
NEXT without FOR	05h	A next statement couldn't be matched with an appropriate for.
FOR depth exceeded	06h	Too many nested for statements.
Bad STEP value	07h	This really only comes up if the step value in a for…next statement is evaluated to zero. You're never going to get from A to B if the step is zero.
Divide by zero	08h	Bad math.
Bad line number	09h	The target of a goto or gosub wasn't found.
Illegal index	0Ah	Either the value of Y is outside the boundary of 1..25 when dereferencing the Z variable or there aren't enough target line numbers in an indexed goto/gosub.
Expression too complex	0Bh	The expression evaluator is recursive and it must be bounded for safety. To get around this error either use fewer levels of parenthesis or break up the expression across a few lines.
Bad numeric parameter	0Ch	This is like the Illegal index error but when a parameter would like to be used as something non-indexing (like for the rnd() function)
User abort	0Dh	This isn't really an error message but an alert if an abort command terminated execution.
Out of data	0Eh	The sequential data set wasn't large enough for the number of reads performed upon it.

Appendix B: Specifications and Limitations

As of version 0.9

Constraint	Value
Program size, RAM	1K
Program size, Flash	4K
User variables, direct access	25
User variables, indexed access	25
User data, sequentially accessed	25
Variable data type	32-bit signed integer
Maximum For..Next nesting depth	4
Maximum Gosub nesting depth	4
Maximum expression complexity	4 levels
Print string length	32 bytes

Appendix C: Using Fixed Point Math

Yes, place all of the tips and tricks here.

Appendix D: Known Issues

As of version 0.9

- There is no handling of negative literal numbers, like "A = -9". The workaround is to form an expression, "A = 0 - 9".
- There is no support for an "input" statement. I'm working on how best to abstract this.
- You can send down programs with unsorted line numbers and it will run top to bottom. But I don't recommend this.
- There is no support for REM; it would burn up important program space anyway. Just write self-documenting code and you'll be fine.
- I suggest coding simple expressions until orbBasic 1.4 is released. It's still easy to blow up.

Appendix E: License Statement

Retrieved from http://dunkels.com/adam/ubasic/license.html on 8/20/2012.

Document Revision History

Revision	Date	Who	Description
1.05	5 Dec 2012	DD	Added the input command to orbBasic ver 1.5
1.04	18 Sept 2012	DD	Documented extra "go" parameter to goroll command, added raw motor command, cleaned up basflg.
1.03	8 August 2012	DD	Added locate statement to assign XY position. Enhanced read statement to process more than one variable. Added heading statement to assign yaw and added the "go" bit to the end of goroll.
1.02	3 July 2012	DD	Clarified meaning of accelerometer system variables.
1.01	20 June 2012	DD	Added xpos, ypos.
1.00	25 May 2012	DD	Added accelone.
0.99	18 May 2012	DD	Added gyro, accelerometer and battery access variables.
0.98	11 May 2012	DD	Renamed timer1-3 to timerA-C. Added abs, dshake, isconn, sleep.
0.96	26 April 2012	DD	Added reset, goroll, cmdval, hdgval, spdval, macrun, mackill and macstat.
0.95	16 April 2012	DD	Added conjoined relation evaluation with and/or.
0.94	13 April 2012	DD	Added tron and troff.
0.93	12 April 2012	DD	Added the sqrt function and a short tutorial on working with fixed point math. Changed rnd to take a range parameter. Added data, read and rstr and ? for print. Added $ modifier inside of print.
0.92	23 March 2012	DD	Added more commands, reformatted errors, etc. Generally just a lot more love. Still on ver 0.9.
0.91	22 March 2012	Jim Atwell	Minor bug fixes.
0.90	16 March 2012	Dan Danknick	Initial revision.