## Running ✓ Tests in BC BASIC

By writing a good set of tests, you can be more confident that your program works both when you first write it and later when you update it. BC BASIC has a simple system for running tests: you simply write functions in your programs whose name starts out as TEST, and when you tap on the Run Test (✓) button, the tests will be automatically run.

Here is the test function for the Arc Length program (part of the Space and Astronomy package). It verifies that the function works for a variety of inputs where the results are already known.

FUNCTION TEST () ASSERT (ArcLength (3959, 45) ≅ 3109.391) ASSERT (ArcLength (10, 90) ≅ 15.707963) ASSERT (ArcLength (0, 90) = 0) ASSERT (ArcLength (10, 0) = 0)RETURN

The simplest way to write a test is to make a function whose name starts with TEST and uses the ASSERT function. The ASSERT function takes an equality expression (an expression that uses any of < <= = >= > <> ≅ or ≇) The tests will finish and will tell you which tests passed and which failed.

The Run Test (✓) button is available on each program, in the settings for each package, and in the overall Library setting. If you’re editing a program, you can also just press the F7 key to run the test for just that program. Each one will run more tests: the one on each program will run only the tests in that program; the one in the Package settings will run all of the test in all of the programs in the one package. The one in the Library setting will run all the tests in all of the packages and programs.

A different way to write a TEST function without using ASSERT is to return the number of errors that the test found. If it doesn’t return a number that is zero (all the tests passed) or more than zero (the number of failures), then that is an error, too.

## Release Notes (3.14, November 2017)

It’s super easy to write and run TEST functions with ASSERT()s in your code. Add TEST functions to validate the functions that you make.

Added support for READ and DATA statements.

READ m will read data from the DATA statements into variable m. DATA 1,2,3,4 will create a set of data. Values in a data statement can be numbers or strings but not variables or expressions. READ and DATA are global; when a program starts all DATA statements will be found and added to the overall program state; as each READ happens (either in the global area or in a function) one item is taken from the global DATA statement.

Added support for the INKEY$ variable. Example: a=INKEY$ will either return a blank string (“”) or a keyboard key. The Screen.Inkey value will return the current keyboard input that hasn’t been read so far.

To help with compatibility, the ASC function has been added as a synonym of CODE.

Added graphic.Arc (cx, cy, innerR, outerR, ang1, ang2) which draws an arc with an inner and outer radius from ang1 to ang2 (both in radians)

Added poly = graphic.Polygon() and the poly.SetPoints(x1, y1, x2, y2…) method. Call SetPoints() with the list of points that the polygon is for.

There is much better support for infinity. The ∞ symbol is recognized as infinity and ∞ has been added to the list of special symbols. Unlike Math.NaN, ∞ can be compared in an expression. Lastly, Math.Infinity has been added for people who prefer to not type in non-ASCII characters.

The ≇ (NEITHER APPROXIMATELY EQUAL NOR ACTUALLY EQUAL TO) symbol has been added as a counterpoint to the approximately equal to sign.

Added support for the Math.Sigma (x) function. This function returns 1 / (1 + e-x); mathematically, it maps an input which ranges anywhere from –∞ to +∞ into the range 0 to 1. This is often used with AI functions in order to squeeze data with an arbitrary range into a common range.

Added support for g.Border so you can control the color of a graphics window border. Also updated the code so that when you make a graphics window, it starts off as the color of the main fixed-size screen.

TODO: Graphics supports an Tapped event. When you call g.SetTapped (“functionName”, “arg”) and then tap the screen, the function you specified will be called arguments (x, y, “arg”) where “arg” is whatever value you passed in originally.

TODO: can set CX or CY on any rectangular object, and rotate with obj.Rotate (radians) where 0 is to the right. Can also set CXD and CYD which are the CX and CY Offsets. The offsets are (ready for this?) multiplied by the font size