BC BASIC

Reference manual and tutorial. Version 2.0

*BC BASIC is the quick and easy way to add your custom functionality to the Best Calculator.*

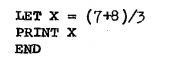
*Best Calculator is available for Windows and Windows Phone platform from the Microsoft Windows Store.*

*Language manual is © 2015 by Peter Smith. All rights reserved.*

*BC BASIC Reference manual and tutorial is © Copyright 2015 by Peter D. Smith.   
All rights reserved.*

# *A brief historical note*

As the first programming language designed for students, BASIC holds a special place in the history of programming languages.



Sample BASIC program from the first Dartmouth BASIC instruction manual, 1964

Calculator BASIC lets you program your own simple functions into the Best Calculator, extending its abilities to exactly match your needs. Everyone who’s a specialist has the same problem with typical calculators: there’s some standard calculation in your profession, but no calculator includes those particular functions on the keyboard. Best

Adding a program to the Best Calculator is easy. The BC BASIC Library button shows you all the programs you’ve written, neatly organized into individual *packages.* The BC BASIC environment gives you access to a wide variety of sample programs. Naturally, Help is just a click away. Within the programming environment you can add or edit your new program, run your program, and bind your program to any of the five programmable keys on the keyboard.

The programs you write will *roam* between your computers and your phone. You can write a program on one computer, and it will automatically roam to your other computers. (This requires that you’ve signed in with a Microsoft Account, of course). You can also *Export* your packages to a file, and then *Import* the package into Best Calculator running on another computer.

[*Introduction* iii](#_Toc430537387)

[*A brief historical note* v](#_Toc430537388)

[1 When should you use Best Calculator BASIC? 1](#_Toc430537389)

[2 Your first program, step by step 2](#_Toc430537390)

[2.1 Add a new package for your program 2](#_Toc430537391)

[2.2 Add a new program to your package 3](#_Toc430537392)

[2.3 Edit the program to do the conversion 4](#_Toc430537393)

[2.4 Run the program to test it 6](#_Toc430537394)

[2.5 Bind the program to the P1 key 6](#_Toc430537395)

[2.6 Next steps 8](#_Toc430537396)

[2.7 The rest of the manual 8](#_Toc430537397)

[3 What all can you do in the BC BASIC environment? 10](#_Toc430537398)

[3.1 All the main edit dialogs 10](#_Toc430537399)

[3.2 Library of Packages 11](#_Toc430537400)

[3.3 Library Properties (Import BC BASIC package) 13](#_Toc430537401)

[3.4 Bind a program to a key 14](#_Toc430537402)

[3.5 About this package 16](#_Toc430537403)

[3.6 List of programs 18](#_Toc430537404)

[3.7 About this program 20](#_Toc430537405)

[3.8 Edit Program 22](#_Toc430537406)

[3.9 Output screen 24](#_Toc430537407)

[4 Reference: Language Basics 26](#_Toc430537408)

[4.1 Packages and Programs 26](#_Toc430537409)

[4.2 Program structure 28](#_Toc430537410)

[4.3 Numbers and Strings and Variables 28](#_Toc430537411)

[4.4 <expression> 31](#_Toc430537412)

[4.5 String functions (LEFT, MID, RIGHT, LEN, CHR$, CODE, VAL) 37](#_Toc430537413)

[4.6 <statement> 40](#_Toc430537414)

[5 Reference: Extensions 41](#_Toc430537415)

[5.1 Calculator.Value and Calculator.Message Extension 41](#_Toc430537416)

[5.2 Memory Extension 42](#_Toc430537417)

[5.3 Math Extension 46](#_Toc430537418)

[5.4 Screen.H and Screen.W Extension 50](#_Toc430537419)

[6 Reference: Statements 51](#_Toc430537420)

[6.1 [CALL] <function> (<expression>, …) 51](#_Toc430537421)

[6.2 CLS [<color>] and PAPER <color> 51](#_Toc430537422)

[6.3 CONSOLE <expression> [, <expression>] 53](#_Toc430537423)

[6.4 DUMP 53](#_Toc430537424)

[6.5 FOR <variable> = <start> TO <end> [STEP <step>] … NEXT <variable> 54](#_Toc430537425)

[6.6 FUNCTION <name> ( <args> ) … RETURN [<value>] 57](#_Toc430537426)

[6.7 GOSUB <linenumber> and RETURN 57](#_Toc430537427)

[6.8 GOTO <linenumber> 58](#_Toc430537428)

[6.9 IF (<expression>) THEN <statement> 58](#_Toc430537429)

[6.10 IMPORT FUNCTIONS FROM “program” 59](#_Toc430537430)

[6.11 INPUT <variable> 60](#_Toc430537431)

[6.12 (LET) <variable> = <expression> 60](#_Toc430537432)

[6.13 PAUSE <frames> 61](#_Toc430537433)

[6.14 PRINT [AT row,col] <expression> [ (, or ;) [AT row,col] <expression]\* 62](#_Toc430537434)

[6.15 RAND <seed> & the RND value 63](#_Toc430537435)

[6.16 REM comment words to the end of the line 64](#_Toc430537436)

[6.17 STOP and END 64](#_Toc430537437)

[7 Graphics and BC BASIC 66](#_Toc430537438)

[7.1 Fixed character screen commands 67](#_Toc430537439)

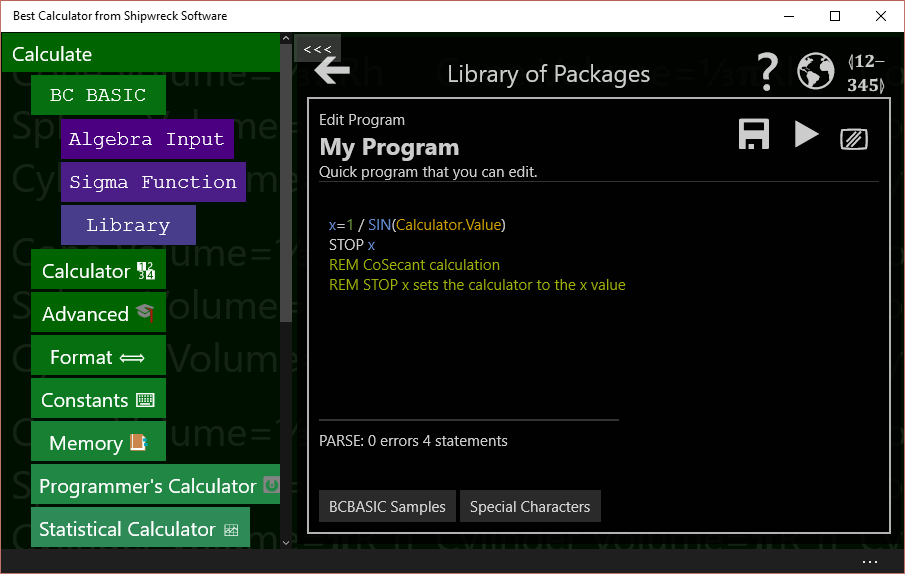
[7.2 Console Commands 69](#_Toc430537440)

[Index to BC BASIC Manual 70](#_Toc430537441)

# Algebra Input: your first program

Tap the BC BASIC and then the Algebra Input buttons. An Edit windows is displayed with a mini program. Your first program is written for you so you can see what a program looks like.

Welcome to BC BASIC Programming!



## What is a program?

A program is a list of *statements*, each of which performs some action. Many useful programs are just a single algebraic equation – for example, to convert feet to acres, or do a financial calculation

## How do I run my program?

Press the [[PLAY]] button to run the program. When you’re in the editor, the final result will be displayed. The final result is either what you’ve retuned in the STOP statement, or is the result of the last assignment statement.

## How can I print some text?

Use the PRINT statement like this:

PRINT “Calculation complete!”

## How can my program use the calculator value?

The best way is the *Input expression*.

X = INPUT DEFAULT 3.4 PROMPT “Enter a tax rate”

Input is only for numeric values (you can’t get a person’s name, for example)

## How can I write an equation?

Use LET, the *assignment* statement, and an *expression* (equation). For example, suppose you need to calculate the circumference of a circle and the equation is PI times the diameter. Your equation might be

LET circumference = PI \* diameter

Or suppose you need to calculate the inner diameter of a pipe given the outer circumference and the pipe thickness.

LET thickness = 3  
LET outerCircumference = 30  
LET outerDiameter = outerCircumference / PI  
LET innerDiameter = outerDiameter – 2 \* thickness

## How can I write to the calculator?

You can write to the calculator in three ways.

You can write a little text message to the top of the calculator display with Calculator.Message.

Calculator.Message = “Hello World”

Which then show up on the calculator results windows



Use Calculator.Value to get or set the result value

Calculator.Value = 7337



If you end your program with a STOP <value>, the value will be placed into the calculator display. Or, if there isn’t a stop, then the last assignment statement (LET statement) evaluated is sent to the calculator. The *algebraic* form (starting a line with just an equals sign), is considered an assignment even though the value isn’t assigned to a variable.

## Why do some examples start with an = sign?

Lines that just start with an equals sign are the “algebraic” form. You can put any expression (equation) on the right of the equals, and the value will be computed. If the

## How can I make several different programs?

See the section on using the library. You can write different programs and give them all different names. They will even roam between your different devices.

## What are all the buttons on the screen?

The [[BACK]] button will take you to the Library.

The [[??]] is for help; you can pop up this manual

The [[GLOBE]] takes you to the Best Calculator web site.

The [[12345]] button lets you program one of the pink buttons with your program. That way you can run your program straight from the regular calculator.

The [[SAVE]] will save your program.

The [[PLAY]] will run your program. You can also press the F5 key.

The [[SCREEN]] button will clear the colorful output screen. That screen is only displayed if you PRINT output.

Under the edit area is the “PARSE” output; it tells you if your program “parses” correctly – it tells you if and when you make a mistake. Don’t fret too much about making parse errors; experienced programmer make them all the time

The BC BASIC Samples button show you a few simple samples to get started.

The Special Characters button lets you insert some of the hard-to-type characters that BC BASIC can use.

## Why is it called Algebraic Input?

Because the simplest form lets you enter an algebra-like equation, and can replace other calculator’s “algebra” input mode. However, BC BASIC is much more powerful.

## How can I learn more?

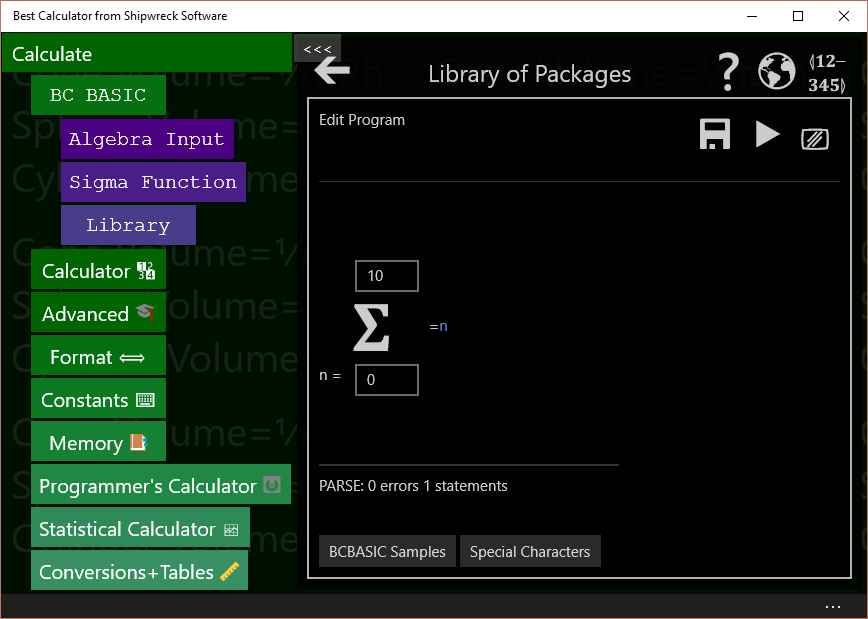
The Reference: Language Basics section tells you everything that makes up a valid BC BASIC program. As a helpful reminder, press the BC BASIC Samples button to get a little pop-up with some common code snippets. Or press the big [[?]] button to see this manual.

A BC BASIC program is a series of statements (lines); each line is an equation. A common statement is *assignment*, which lets you do math. Other common statements are PRINT and INPUT to print results for the user and get numerical values from the user.

# Sigma Function: advanced programming

Press the BC BASIC and then the Sigma Function Input buttons. An Edit windows is displayed with a mini sigma expression program. Your first expression is written for you so you can see what a program looks like.

The Sigma Function page will run an expression that you provide, summing up the response.



Press [[PLAY]] or F5 and see the result



The Sigma function is run 11 times. Each time the variable n is set, first to 0, then to 1, and so on up to and including 10. The value of the expression is summed, and the overall value displayed.

Press the →▣ button to copy the result into the calculator display.

The rest of the Sigma Function page is the same as the Algebra Input page. The Samples include an example of the Taylor expansion to calculate the Sin of a value

REM TAYLOR expansion for SIN  
REM Convert n(0,1,2,3,4) into series (1,3,5,7,9...)  
series = n\*2+ 1  
x = Calculator.Value  
val = x\*\*series / Math.Factorial(series)  
isOdd = (Math.Mod (n,2) = 1)  
IF (isOdd) THEN val = -val  
=val

The Taylor expansion for Sin requires a series of numbers 1, 3, 5, 7, and so on. The Sigma function only produces numbers 0, 1, 2, 3, … .The first step is to convert the **n** value (0, 1, 2, 3, …) into a series value (1, 3, 5). This is done with the line series = n\*2+1.

Next the code calculates the **xseries** value and divides by **series!**. Odd parts of the sequence are supposed to be subtracted from the sequence, so those are negated.

The x value is taken from the **Calculator.Value** number from the calculator display.

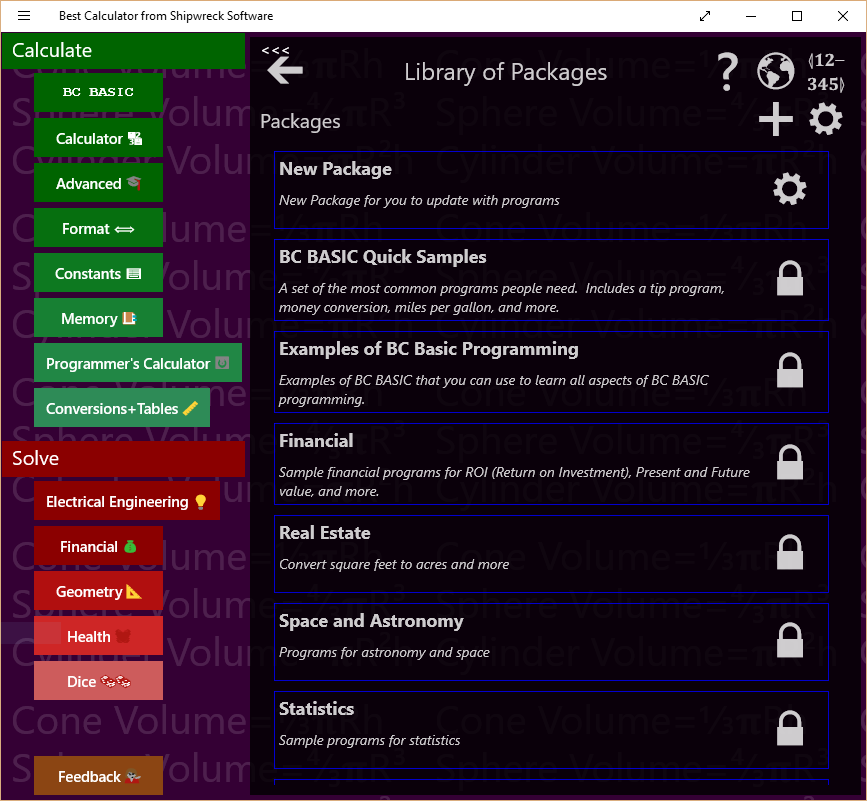
Lastly the resulting value for a single series value is return (=val).

# What all can you do in the BC BASIC environment?

It’s time for a introduction to all of the different dialogs you will use to create and run your programs.

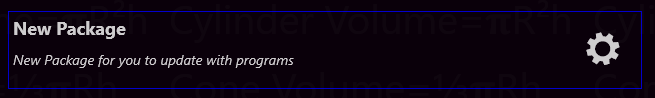
## All the main edit dialogs

## Library of Packages

Library of Package is the first library dialog you see. It lists all of the packages that you can run, examine, or change.

In the Library of Packages dialog there are many important controls.

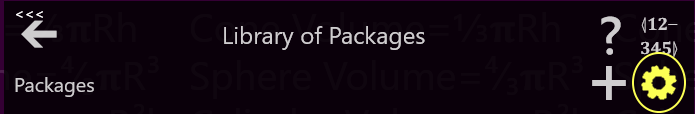
|  |  |
| --- | --- |
| **Control** | **How and when to use it** |
| Back arrow | The big back arrow is present on all of the dialogs. Press it to return to the parent dialog. If you are in the Library of Packages dialog, the BC BASIC environment will be hidden, and you can see the Best Calculator display.  Don’t worry! None of your changes are lost. Just press the BC BASIC button to see the Library of Packages dialog again  The little “chevron” arrow is a Best Calculator arrow; it hides the Best Calculator menu of calculators. |
| Help | Displays the main HELP PDF page using the default PDF reader (often a web browser). |
| 📓  Buy the manual | Goes to Amazon.com. You can buy a copy of the Best Calculator manual from Amazon. |
| Web site | Goes to the Best Calculator web site |
| Bind Key | Displays the Bind a program to a key dialog. This lets you bind one of your programs to one of the P1, P2 and so on key in the Best Calculator keyboard. When you press a key, the program you selected will be run. |
| Create new package | Creates a new package. The new package gets a name which you should change. Use the GEAR icon () to change the name. |
| Properties | Displays up the Library Properties dialog. From this dialog you can Import a BC BASIC package. |

You can also tap on individual packages in the list of packages. Each package entry shows its name and description and includes a GEAR icon to display the package details.

|  |  |
| --- | --- |
| **Control** | **When and how to use it** |
| Tap on a package | Brings up the List of programs for that package. From the list you can add new programs and edit and run your programs. |
| Properties | Bring up the About this package dialog. From that dialog you can change the name and description for a package. |

## Library Properties (Import BC BASIC package)

Bring up the Library Properties dialog by tapping the Library of Packages GEAR button (, highlighted). The Library of Packages header lets you go to the help file, bind a program to a key, add a package for your programs or display the library properties screen.

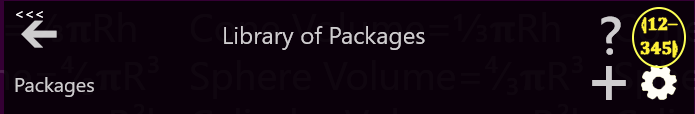


In the Library properties there is a single control

|  |  |
| --- | --- |
| **Control** | **When and how to use it** |
| Import | Brings up the Import dialog. Use this to import (read in) new BC BASIC packages from your job, your school, your friends, or even the internet. |

## Bind a program to a key

Bring up the Bind a program to a key screen by tapping the Library of Packages BIND button (highlighted below)



There are several programmable key which you can bind a program to. When you tap one of those keys, the program that has been bound will be run. When you first get Best Calculator, a selection of programs has already been bound.

To pick a program to run when a programmable key is pressed, select an answer to the three questions in the Bind a program to a key screen and tap SAVE.

The first question is *What button do you want to bind to?* Tap one of the buttons in the button list (labeled P1, P2, P3 and so on) to pick a button to bind to. People often just pick button P1. The button list tells you what package and program the button is currently bound to. This helps you pick the right button to use.

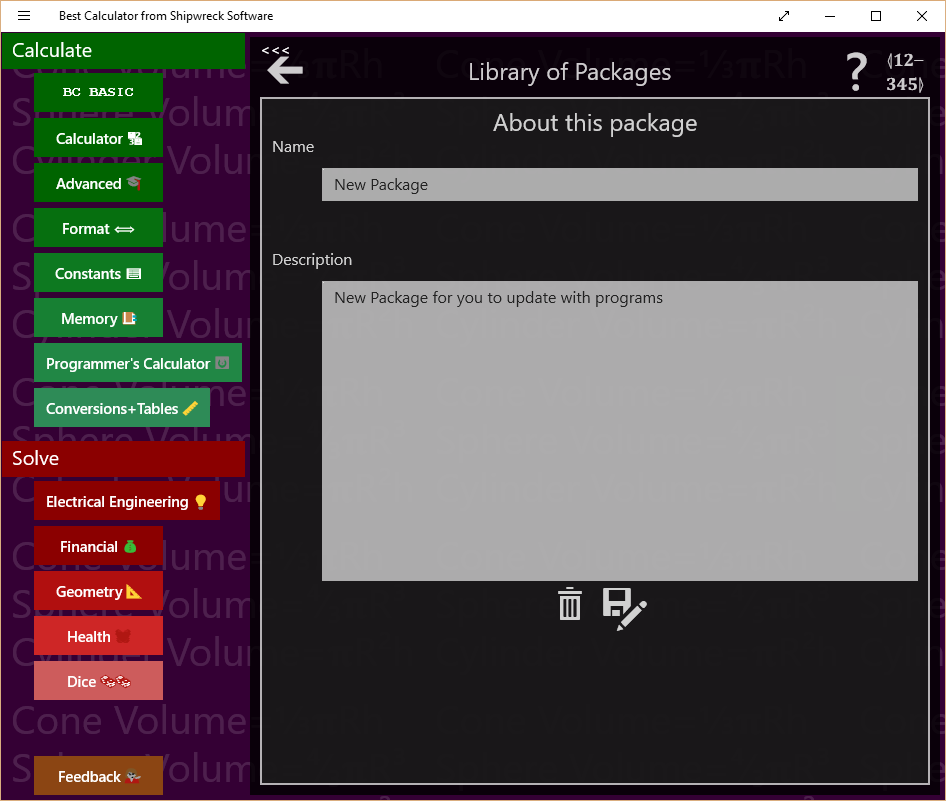
The second question is *What package is the program in?* All the possible packages are listed. As you tap on a package, the next list changes to show the programs in that package. Tap on the package with the program you want to run.

The third question is *What program do you want to run?* Tap the program to run.

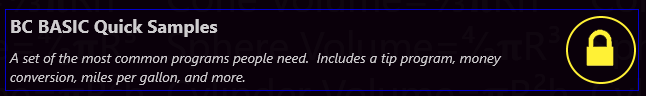
Lastly, **be sure to tap the SAVE button** (). Your selection isn’t saved until you press save.

You can keep on binding programs to more keys, or press the BACK ARROW ()to go back to your last dialog box.

## About this package



Bring up the About this package by tapping either the GEAR () or LOCK () in the package listing (see highlighted) in the Library of Packages screen.

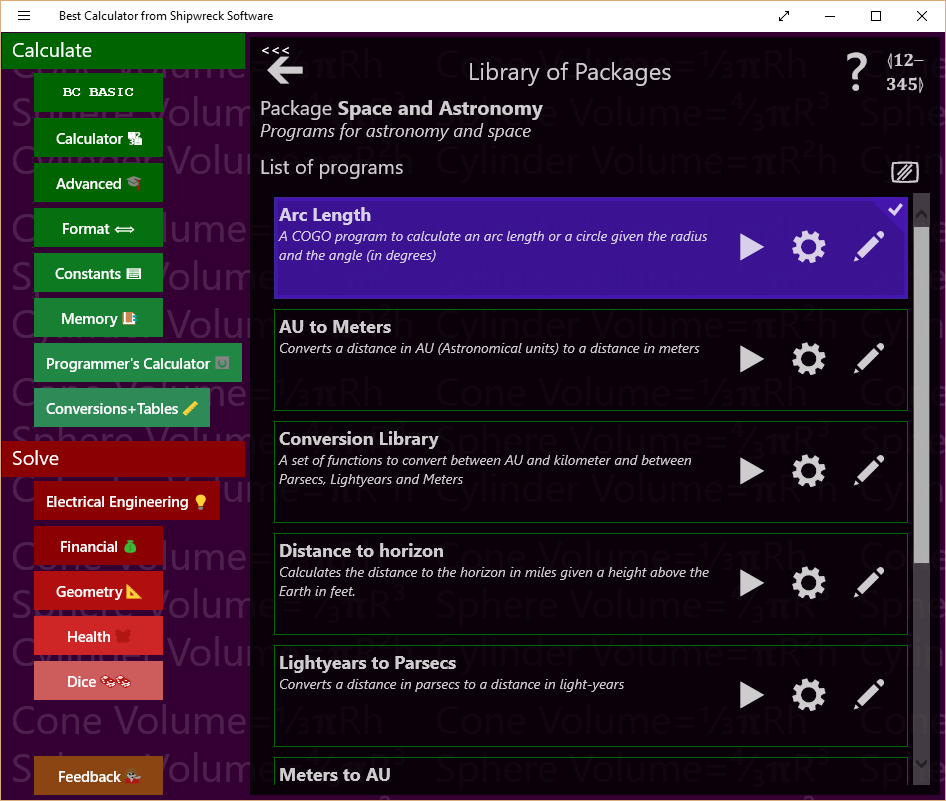


The About this package screen lets you change the name and description of a package, delete it, or export it (save it to an external file).

You can only modify your own packages (the ones with a GEAR icon). Packages that came with Best Calculator (with a LOCK icon) cannot be modified.

|  |  |
| --- | --- |
| **Control** | **When and how to use it** |
| Name | You can change the name of the package here. Just enter a new name. The name is automatically saved. |
| Description | You can change the description of the package here. Just enter or modify the description. It will be automatically saved. |
| Delete | Deletes the package. Once deleted, you will not be able to bring the package back. You will be prompted to confirm the delete. |
| Save As | Saves (exports) the package (and all the programs in it). You will be prompted for a file name to save to.  Once you Save (Export) a package, you can store it in OneDrive, save to a web page, or email to a friend or coworker. They can *Import* the package from the Library Properties page. |

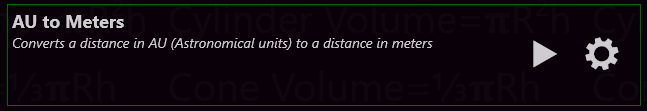
## List of programs

Bring up the list of programs by tapping on a package in the Library of Packages screen.

Packages contain multiple programs which you can run. The List of programs screen lets you see all and run the programs in a package, add new programs, edit programs, and more.

|  |  |
| --- | --- |
| **Control** | **When and how to use it** |
| Add Program | Adds a new program to the package. Once created, you will need to tap the GEAR () to set the program’s name and description.  You can only add programs to your own packages. Packages that came with Best Calculator cannot be modified. |
| Clear Screen | Clears the output screen. If the output screen is not visible, you won’t see a change (but it will be cleared nonetheless) |

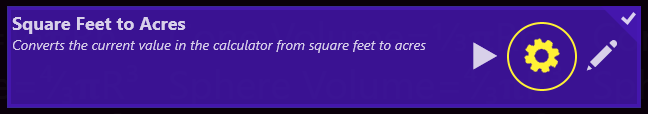
You can also tap on individual programs in the list of programs. Each program entry shows the name and description of the program and lets you run and examine the properties of the program. If the program is one you wrote, you can also edit it.



|  |  |
| --- | --- |
| **Control** | **When and how to use it** |
| Tap on a program | Does nothing in particular ☺ |
| Run program | Runs the program from the start |
| Properties | Displays the About this program dialog. From that dialog you can edit the program name and description. |
| Edit program | Display the Edit Program dialog. From that dialog you can edit and run the program.  Packages that come with Best Calculator cannot be modified. For those program, the Edit button will show you the source code for the program but will not let you change it. |

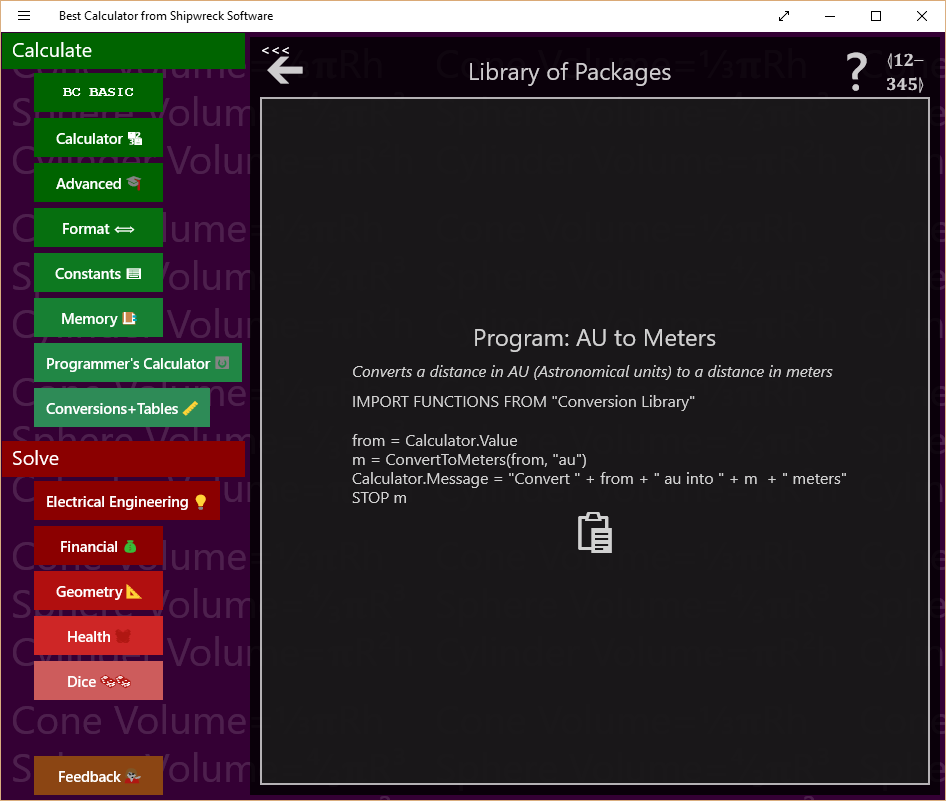
## About this program

Bring up the About this program screen by clicking the GEAR icon in the program entry in the List of programs (see highlighted). The About this program screen lets you modify the name and description of a program, go straight to the edit program screen, or delete the program.



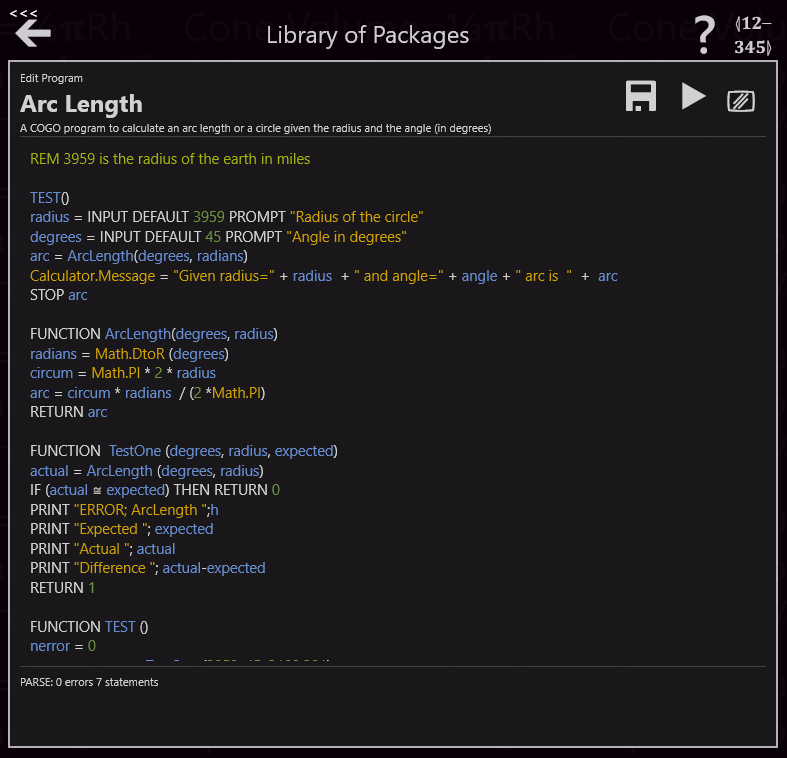
|  |  |
| --- | --- |
| **Control** | **When and how to use it** |
| Name | You can change the name of the program here. Just enter a new name. The name is automatically saved when the program is saved. |
| Description | You can change the description of the program here. Just enter or modify the description. It will be automatically saved. When the program is saved. |
| Edit program | Display the Edit Program dialog. From that dialog you can edit and run the program.  You cannot edit programs that come with Best Calculator |
| Delete | Deletes the program. Once deleted, you will not be able to bring the program back. You will be prompted to confirm the delete.` |

Programs that come with Best Calculator are locked and cannot be changed. Their About screen is a little different.



|  |  |
| --- | --- |
| **Control** | **When and how to use it** |
| Copy to clipboard | Copies the source code for the program to the clipboard. |

## Edit Program



The Edit program dialog is where you enter your BC BASIC code.

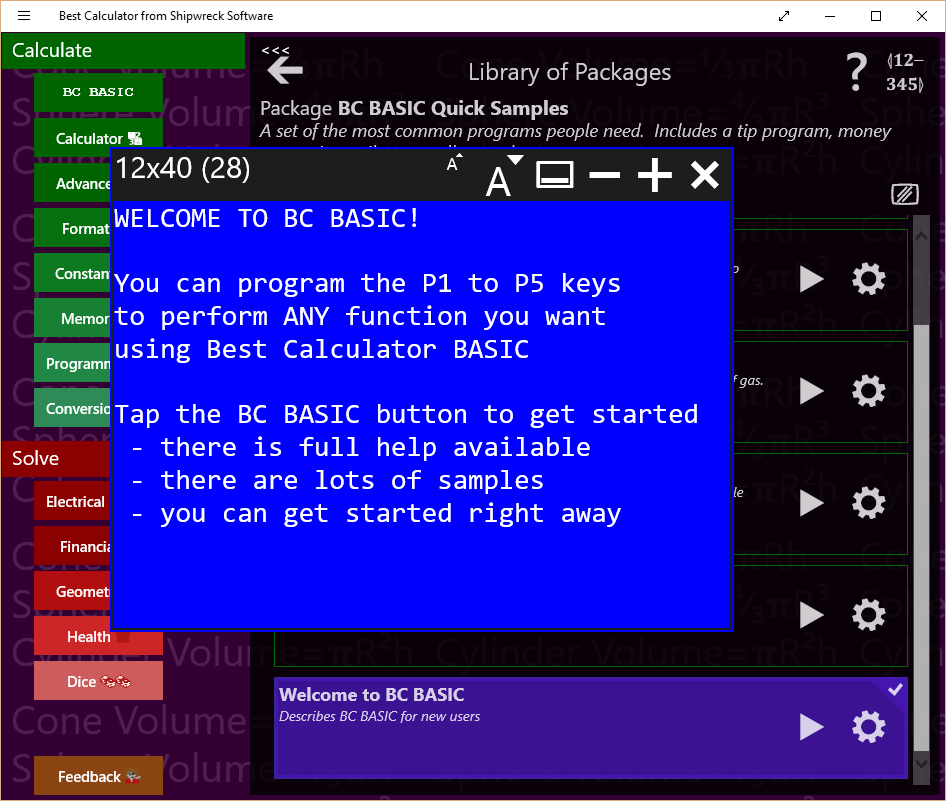
Bring up the Edit program dialog by tapping the EDIT button on either the program list or by tapping the EDIT () button in the About this program screen.

|  |  |
| --- | --- |
| **Control** | **How and when to use it** |
| Save | Saves your changes. Your changes are automatically saved when you press RUN. Your program is saved as part of the package file; this is managed for you. |
| Run program | Runs the program from the start. Some common errors when you press RUN but your program does not appear to run are:   1. Your program might have a syntax or other error that prevents it from running. You will can tell because the parse indicator will show an error 2. You program might have run, but didn’t display anything to the screen. The output is often displayed on the calculator display. Press the Calculator button to see the calculator screen. |
| Clear screen | Clears the output screen. If the output screen is not visible, you won’t see a change (but it will be cleared nonetheless) |
| Parse indicator | This indicates if you program compiles. BC BASIC automatically compiles your program as you type it, and tells you of any syntax errors.  The editor uses *syntax coloring* on your code; different parts of the code will be displayed in different colors. When a syntax error is discovered, only the program up to the syntax error is colored; the rest shows up in white. |

There are two special keyboard keys while you are editing a program

|  |  |
| --- | --- |
| **Key** | **How and when to use it** |
| Escape | Stops the program that’s currently running. This is useful when you’ve written an infinite loop (a program that doesn’t end) |
| F5 | Acts like the Run program button. Press F5 to start the program running. |

## Output screen



The output screen is where the results of the PRINT, CONSOLE and DUMP commands are written. It is the screen that’s cleared or whose color changes when CLS or PAPER is run. You can only write to the screen; you can’t read back from it.

The output screen contains both the fixed-character size screen (which is the normal output screen) and the scrolling console output. The console output is mostly intended for debugging.

The controls at the top of the screen let you modify the screen’s looks.

|  |  |
| --- | --- |
| **Control** | **How and when to use it** |
| Screen Size | The output screen always tells you how large the screen is and the font size. Screen sizes include 12x20, 12x40, 16x60 and 24x80 |
| Smaller font | Reduces the font size, automatically making the output screen smaller. |
| Larger font | Increases the font size, automatically making the output screen larger |
| Display/hide console | Toggles the console portion of the output screen on and off. Unlike the output screen, the console is a long scrolling list of output. |
| Fewer characters on screen | Reduces the number of characters on the screen. |
| More characters on screen | Increases the number of characters on the screen. |
| Close | Closes the screen. The screen contents are not lost; when the screen is re-shown, the original contents will still be present. When you PRINT to the screen, it will be displayed automatically. |

# BASIC Language Reference

## Program structure

A BC BASIC program is a list of *statements* and *functions.* Each statement can optionally start with a *line number*; line numbers are simple integers.

**Examples of statements**

FOR I = 1 TO 4  
PRINT “Hello World”  
NEXT I  
100 REM this statement has a line number

Statements are normally one line long. They can be extended with various “return” type characters:

|  |  |  |
| --- | --- | --- |
| ↲ | U+21B2 | DOWNWARDS ARROW WITH TIP LEFTWARDS |
| ↵ | U+21B5 | DOWNWARDS ARROW WITH CORNER LEFTWARDS |
| ⤶ | U+2936 | ARROW POINTING DOWNWARDS THEN CURVING LEFTWARDS |

Statements do not have a statement terminator (for example, the “C” language terminates statements with semicolons). Statements start with a command name like PRINT except that the CALL and LET command names are optional. Examples of statements include CLS BLUE and LET a=10. The CALL and LET commands are optional; LET a=10 is the same as plain a=10, and CALL PrintName() is the same as PrintName().

Lines are technically a sequence of characters that ends with a \n, \r or \v. The \v (vertical tab) is sometimes used by Microsoft Word when cut-and-pasting text.

Comments start with the REM command; the rest of the line is the comment.

## Numbers and Strings and Variables

Numbers and strings are the two most common things people want to manipulate. Inside a BC BASIC program, you can write number constants (like, 4 or 3.14) and string constants (lie, “apple”).

### Numbers

Number can be any of

* Integers like 3 or -5
* Integers using hexadecimal (base 16) notation like 0x65
* Floats (actually stored as double-precision) like 1.2, or -7.8
* Numbers in exponential notation like 45E3 (which is equal to 45000)

Doubles which are smaller than 1 can start with just a decimal place (0.3 and .3 are both OK). To help improve the look of your programs, BC BASIC allows for either a computer-style minus or a wider Unicode minus and dash characters. Example: computer-style is - and wider Unicode is – or –. Microsoft’s Word program sometimes converts one into the other.

**Example of using different types of minus signs:**

REM Constant numbers

V1 = -12.34E-6

V2 = –12.34E–6

V3 = −12.34E−6

REM Expressions

E1 = V1 - V2

E2 = V1 – V2

E3 = V1 − V2

REM Negated Values

N1 = - E1

N2 = – E2

N3 = − E3

Technically, these are Unicode characters HYPHEN MINUS (U+2D), EN DASH (U+2013) and MINUS SIGN (U+2013). The special Unicode plus signs are not accepted or the other Unicode minus signs.

### Strings

String constants can use either computer-style quotes like "" or “smart” quotes. A string that starts with an opening smart must be ended with a smart end quote. You can’t nest smart quotes: the string “hello “special” world” is incorrect. The string will be ended at the end of the word special; the rest of what you think is the string will be misinterpreted.

BC BASIC includes functions for manipulating strings. You can concatenate strings together with the + operator, get the length of a string with the LEN function, and extract parts of a string with the LEFT, MID and RIGHT functions.

### Variables

BC BASIC allows you to create and use variables at any time. Variables are given names; names must start with a letter (a to z and A to Z) and from then on can contain letters, numbers, and underscores. Variables can optionally end with a dollar sign ($). Variables are case sensitive; the variable “name” is different from the variable “NAME”.

In BC BASIC the “$” at the end of a variable has no special meaning. The $ is allowed for compatibility with other versions of BASIC. In some versions of BASIC, a “$” indicates that a variable is a string. In BC BASIC, any variable can be any kind of variable.

Variables created inside of functions are *scoped* to the function; they cannot be used out of the function.

**Example of using variables:**

LET a = 10

LET b = a + 3

LET c = 7.89 / b

LET bignum = -1.3E23

LET smallnum = 2.78E-23

LET name = “Person of interest”

LET information$ = “You can use smart quotes”

LET dog\_name = "Sumi"

LET check9 = 99

CLS GREEN

PRINT "Sample variables"

DUMP

## <expression>

### Quick introduction to expressions

“1 + 1” is one of the simplest expressions; it adds two *constants* (the two 1 values). Best Calculator BASIC has the normal set of operators and precedence rules for modern computer languages, plus a few extra convenience functions.

### Expression Rules

BC BASIC is designed to make most expressions work normally. 1+2\*3, for example, will multiple the 2\*3 first, and then add the 1. You can put parenthesis around your expressions. You can use either curved parenthesis or square brackets.

|  |  |
| --- | --- |
| **Expression type** | **Explanation and Sample** |
| Variable | When evaluated, is the value of the variable.  Examples:  A  B$ |
| Constant | A numeric or string constant  Examples:  1  3.14  0xFF  “Hello World”  “She said, &QUOT;Hello&QUOT;” |
| Named values PI and RND | PI is always equal to PI (3.1415…). It’s more common in BC BASIC to use the Math.PI value and not PI by itself.  RND is a new random number between 0 and 1. |
| ( *expression* )  [ *expression* ] | You can place expression inside of parenthesis () or square braces [] to show which operations should happen first. |
| *expression* OPERATOR *expression* | Any of the standard math operators like + and - . See the table below for a full list. BC BASIC also includes a variety of comparison and logical operators, and the INPUT operator. |
| PREFIX *expression*  *expression* POSTFIX | Use minus (-) to negate a number.  Use power and root operators to take a square root, or raise a value to a power. |
| Function ( argument, argument ) | The value of the given function. There are a set of built-in functions (SGN ABS SIN COS TAN ASN ACS ATN LN EXP SQR INT LEN CODE CHR$), or you can define your own.  The parentheses are normally required. Unlick some other versions of BASIC, you cannot omit the parenthesis for the built-in functions. |

Constants are always handled as doubles (1.2, or 0.1, or -56.7, or just plain 4 or -2

Example of expressions are

|  |  |
| --- | --- |
| **Example** | **Explanation** |
| 1 | A constant |
| apple | The value of the variable “apple”. The variable should have already been defined; otherwise it’s assumed to be a “NaN” (Not a Number) |
| SIN (PI / 4) | The SIN of ¼ PI radians. The trigonometry functions all take their values in radians. Use the Math.DtoR() function to convert degrees to radians. |
| 1 + 2 \* 3 | Is the value 7 (and not 9); multiplication is higher priority than addition so the 2\*3 is done first and then the 1 is added to it. |
| (1+2) \* 3 | Is the value 9; the parenthesis force the addition to be done first |
| “A” < “B” | Is the value 1 (for ‘TRUE’) because A is sorted before B. |

### negate, power, root prefix and postfix operators

|  |  |
| --- | --- |
| **Operator** | **Explanation and Sample** |
| ² ³ ⁴ | Square, cube and fourth power operators. BC BASIC is special among most programming languages for allowing these superscripts to be used. You can also use the Math.Pow() function or the \*\* operator  Example:  5² is 25 because 5\*5 is 25  Math.Pow(5, 2) is also 25  5\*\*2 is also 25 |
| √ ∛ ∜ | Square root, cube root and fourth root operators. You can also use the Math.Pow () function or the \*\* operator  Example:  √64 is 8 because 8\*8 is 64  Math.Pow(64, 1/2) is also 8  64\*\*0.5 is also 8 |
| - | Unary minus converts any number to its negative  Example:  12 + -3 is 9 because 12 – 3 is 9 |

### Operators + - \* / and more

Each operator has a precedence value; operators with a higher precedence will be done before operators with a lower precedence.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Precedence** | **Explanation and Sample** |
| \*\* | 10 | \*\* is the “raise to the power” operator  Example:  2 \*\* 6 is 64 because 2\*2\*2\*2\*2\*2 is 64  Historical note: early versions of BC Basic includes a “root finding” operator. This proved to be confusing in practice, and has been removed. |
| \* / | 9 | \* is the computer sign for multiply  / is the standard computer sign for divide  Examples:  3 \* 4 is 12  10 / 2 is 5 |
| + - | 6 | + is the standard computer sign for addition or string concatenation  - is the standard computer sign for subtraction  Example:  1+2 is 3  10 - 2 is 8  “1” + 2 is “12”, but 1 + “2” is 12. When the left side of a + is a string, both sides are treated as string and concatenated together; when the left side is a number, both sides are treated as numbers. Strings that cannot be converted are handled as a NaN  Extra bonus: The Unicode character set designates three characters that are commonly used for minus signs:  HYPHEN-MINUS (U+2D), the normal minus sign  MINUS SIGN (U+2212) EN DASH (U+2013)  Any of these can be used for a minus sign. Among other things, this makes it easier to copy your programs back and forth between BC BASIC and Microsoft Word. |
| < <= = >= > <> ≅ | 5 | The normal set of operators for less than, less-than-or-equal, and so on. The <> operator is for “not equals”.  The ≅ is for “approximately equals”; for numbers it means that the two numbers are within 5 significant figures of each others, and for strings that they compare equal using the CurrentCultureIgnoreCase flag.  Note that two numbers which are different signs are never considered approximately equal even if they are both really, really close to zero. |
| NOT | 4 | Inverts the logical value of its argument.  Example  IF NOT A=B THEN <statement> will do the statement if A is NOT equal to B.  **Note on logical values:**  The logical operators (NOT, AND, and OR) can take any numerical value and treat it as a logical value; anything that’s zero is treated as FALSE and all other numeric values are TRUE. The operators will only ever produce a 0 or 1. |
| AND | 3 | A logical AND operation; A AND B will be 1 if both A and B are TRUE (non-zero). |
| OR | 2 | A logical OR operation. A OR B will be 1 if either A or B are TRUE (or if both of them are). |
| INPUT |  | INPUT is a complex operator with two optional values, a PROMPT and a DEFAULT value. When your program runs, a dialog box will pop up with the prompt you specific and with a starting value of whatever the default value was (it can be an expression). The user then enters a value and presses the OK button; the resulting value is the value of the INPUT expression.  Example:  birth\_year = INPUT DEFAULT 1967 PROMPT “Enter your birth year: ” |

**Examples of expressions:**

REM Multiplication binds more than addition

REM a will be 7 (1 + (2\*3)) and not 9 ((1+2) \* 3)

a = 1 + 2 \* 3

b = 10 - 4 / 2

REM Demonstrate cube root and raise to 4th power

c = 3 √ 10

d = 6 \*\* 4

REM Comparing values. PI is not about 22/7  
REM but it is about 355/113

e = PI ≅ 22/7

f = PI ≅ 355/113

REM These are all false (except j)

g =PI > 22.7

h = PI >= 22.7

i = PI = 22.7

j = PI <> 22.7

REM Logical operators

k = c >2 AND c < 4

l = c > 2 OR d < 10

m = NOT (c > 2 OR d < 10)

REM You can ask for input from the user

n = INPUT DEFAULT 5 PROMPT ↲

"Please enter a number"

CLS BLUE

PRINT "All the variables"

DUMP

**Example of using the RND and PAUSE statement to make a little random animated display:**

CLS BLUE

COUNT = 0

REM You can also use a FOR..NEXT loop

10 A = DrawDot()

COUNT = COUNT + 1

IF (COUNT > 100) THEN GOTO 20

PAUSE 1

GOTO 10

20 PRINT AT 1,1 "DONE"

FUNCTION DrawDot()

col = INT (RND \* Screen.W) + 1

row = INT (RND \* Screen.H) + 1

ch = “\*”

PRINT AT row,col ch

RETURN

### The INPUT expression

The INPUT expression lets you prompt your user to enter a value. You can use DEFAULT <value> to supply a default value and a PROMPT <string> to specify a prompt string. You have to specify the DEFAULT before the PROMPT.

There is also an INPUT statement that is less powerful than the INPUT expression. It’s provided for compatibility with other versions of BASIC.

**Example of the INPUT expression:**

LET interest = INPUT DEFAULT 3.5 ↲

PROMPT “Interest rate”

The user will enter a value which will be interpreted as a number. You cannot get a string (like a name) from the user; you can only enter numbers.

## Math Functions

BC BASIC includes a small but powerful set of math functions. These are designed to be compatible with other versions of BASIC.

Some version of BASIC let you use these functions without parenthesis; BC BASIC does not.

There are even more functions available in the Math extension.

### Trigonometry functions SIN COS TAN ASN ACS ATN

|  |  |
| --- | --- |
| **Function** | **Notes** |
| ACS(value) | Calculates the inverse of COS; given a value will compute the corresponding angle in radians. |
| ASN(value) | Calculates the inverse of SIN; given a value will compute the corresponding angle in radians. |
| ATN(value) | Calculates the inverse of TAN; given a value will compute the corresponding angle in radians. Note that TAN of 90° is infinite. |
| COS (radians) | Calculates the cosine of an angle given in radians |
| SIN (radians) | Calculates the sin of an angle given in radians |
| TAN (radians) | Calculates the tangent of an angle given radians |

### Logarithm and Power functions LN EXP SQR

|  |  |
| --- | --- |
| **Function** | **Notes** |
| EXP (value) | Calculates the value *e*value for any given value. This is the inverse of the LN function  This is the same as Math.Exp(value) |
| LN (value) | Calculates the natural (base *e*) logarithm of the given value. This is the inverse of LN.  This is the same as Math.Log(value) |
| SQR (value) | Calculates the square root of a value.  You can also use the √ square root operator, or use the \*\* power operator.  This is the same as Math.Sqrt(value). |

### Rounding and sign functions SGN ABS INT

|  |  |
| --- | --- |
| **Function** | **Notes** |
| ABS (value) | Calculate the absolute value of a number.  This is the same as Math.Abs(value) |
| INT (value) | Calculates the floor of a number. The floor is the number rounded down to the nearest integer. For example, INT (12.8) is 12 and INT (-12.8) is -13.  This is the same as Math.Floor (value) |
| SGN (value) | Return the sign of a number. The sign is 1 for positive values, -1 for negative values, and 0 for zero.  This is the same as Math.Sign(value) |

## String functions (LEFT, MID, RIGHT, LEN, CHR$, CODE, VAL)

### LEFT (string, count)., MID (string, index, count) and RIGHT (string, count)

The LEFT, MID and RIGHT functions get data from the start (LEFT), middle (MID) or end (RIGHT) ends of a string. The return value is always a string.

For the MID function, you don’t need to provide the count value; it will be assumed to be 1. The index for the MID function starts at 1; MID (string, 1, count) is exactly the same as LEFT (string, count).

**Examples of the string functions:**

REM LEFT(string, count) returns string 'count' long

REM from the left part of the input string

REM The example will print AB

PRINT “LEFT("ABCDE", 2) = ”; LEFT("ABCDE", 2)

REM MID(string, index, count) returns

REM a string 'count' long

REM from the middle part of the input string

REM starting at 'index'. The first letter is index 1.

REM The example will print BCD

PRINT “MID("ABCDE", 2, 3) = ”; MID("ABCDE", 2, 3)

REM RIGHT(string, count) returns

REM a string 'count' long

REM from the right part of the input string

REM The example will print DE

PRINT “RIGHT("ABCDE", 2) = ”; RIGHT("ABCDE", 2)

Each of the functions has similar special cases.

1. The count is always truncated so that the return value doesn’t go past the size of the string
2. If the count value is less than one, then a blank (zero length) string is returned
3. If the index for MID is more than the length of the string, then a blank string is returned

### LEN string

The LEN function (which has optional parenthesis) returns the length of a string. Blank strings have a length of zero.

**Examples of the LEN function:**

REM The length is always 3

PRINT “LEN of string ABC is 3”

PRINT LEN “ABC”

PRINT LEN (“ABC”)

REM Length of a blank string is zero

PRINT LEN “”

REM LEN PI is the length of the string

REM representing the number PI

REM (the answer is 16 printed digits)

PRINT LEN Math.PI

Special cases include:

1. When given a number, returns the length of the string that represents the number
2. Blank strings have zero length
3. Unicode strings that use *surrogate pairs* will count each surrogate pair as two characters. For example, the Unicode U+1F60B character point (FACE SAVORING DELICIOUS FOOD) is presented as two characters, U+D83D and U+DE0B. These two characters are called a surrogate pair, and represent the Unicode character.

### CHR, CODE

CHR converts a series of numerical Unicode values into a string. There is also, for compatibility, a CHR$() that takes a single argument. The CHR$ function is technically an expression operator and does not require parenthesis. That is, CHR$ (65) can be written as CHR$ 65.

The CODE function returns the Unicode code point of the first character in a string. The CODE function is technically an expression operator and does not require parenthesis. That is, CODE (“A”) can be written as CODE “A”.

In Windows, Unicode code points outside the Basic Multilingual Plane are encoded as surrogate pairs and are handled as two characters.

**Example of the CHR, CHR$ and CODE functions:**

REM CHR converts a Unicode character number

REM to a string

REM 65 is the ASCII A

REM Unicode U+1F60B is

REM "FACE SAVOURING DELICIOUS FOOD".

REM It is converted into two chars

REM (a surrogate pair).

REM CHR$ is the same function, but takes in

REM only one parameter

PRINT “CHR (65) = ”; CHR (65)

PRINT "CHR Unicode: "; ↲

CHR(65, 66, 0x1F60B, 67, 68)

REM CODE converts the first character of

REM a string to a code

PRINT “CODE "ABC" = ”; CODE "ABC"

### VAL(string)

The VAL function evaluates the string as a BC BASIC expression and return the value. For example, VAL(“1 + 2”) will return 3. The expression can use variables that you have set.

Note that VAL can be slow.

**Example of the VAL function:**

REM VAL will evaluate an expression

a = 1

b = 2

PRINT “VAL ("a + b") = ”; VAL ("a + b")

## <statement>

Statements are the building blocks of a BC BASIC program. They let you perform calculations, assign variables, loop until a condition is true, define functions and more.

**Examples of statements:**

10 CLS  
A = 3  
B = 4  
20 C = A + B

PRINT “C is ”; C

BC BASIC does not allow for multiple statements on a single line.

Statements can include an optional line number. They are the targets for GOTO and GOSUB statements. They do not need to be in any particular order. Other versions of BASIC always require line numbers and automatically order all lines by line number. BC BASIC instead lets you use any line numbers you want in any order.

Good line number practices:

1. Only number a statement when you have to.
2. Keep your line numbers in numerical order
3. Make your line numbers divisible by 10 or 100; that way you can add new line numbers in between existing values.

Line numbers in a function are *scoped* to that function; two functions can use each other’s line number. You cannot GOTO or GOSUB into or out of a function.

Statements are separated by new-lines; a newline is one or more of carriage-return, new-line, or vertical-tab. (Pressing ALT-newline in Word can separate lines with vertical tabs). You can continue a statement from one line to another by placing a visible enter symbol (↲, U+21B2) at the end of the line where white space would fit.

## Packages and Programs

The BC BASIC *Package* and *Program* concepts are special to the BC BASIC environment.

If you’re just getting started, you should feel free to just place the programs you write into the single package that you made when you wrote your first program. You don’t even need to rename it; you can continue to use the New Package name.

But if you are going to write more than a few programs, you should spend a few minutes understanding the BC BASIC *Package* and *Program* concepts. These are explained more fully later on.

The BC BASIC environment tracks your programs for you. You do not have to deal with their file names or try to remember where the packages are or what programs they have. When you make a new package, the BC BASIC environment makes a file for you in the app roaming directory, and picks the name for you with a file extension of *.bcbasic*. Best Calculator will automatically read in all of the packages you’ve created (and they roam, too, so you can make or edit a package on one computer and it will be automatically sent to your other computers).

### Inside a package file

There is one file per package (and therefore multiple programs are placed into a single file). When Best Calculator starts, it reads in all the *.bcbasic* files both in the BC BASIC directory (these are the samples shipped with Best Calculator) and all the *.bcbasic* files in the app data directory (these are your packages).

### Picking a package for your program

When you make a new program, you should add it to a package where it will logically fit. For your first few package, that will probably be the “New Package” that you created. You will often find that you make a number of programs that you will be using together. For example, you’ve made one program that converts square feet to acres. The reverse program (acres to square feet) would naturally fit into the same package.

### Creating common functions for several programs

The programs in a package are normally independent of each other. They don’t share variables or functions, and you can GOTO or GOSUB from one to the other. There are exceptions to this general rule. The IMPORT statement can read in the functions of another program in the same package. This lets you make a “library” of functions that many programs can use.

### Exporting packages

You can export (write) packages out to a file and then later import (read) the files back in. When you export, you create a JSON file that lists all the programs along with their names and descriptions and BC BASIC code. The export and import mechanisms mean that you can share code with your colleagues, coworkers, fellow students, or friends. You might even find useful packages on the internet. Be careful though: although a BC BASIC program cannot damage your computer, it’s possible for one package to delete your memory variables and conceptually possible to use up excessive disk space.

## All of the Special Symbols

Best Calculator supports a number of special symbols both to help create good-looking program and so that your programs can be copy and pasted into word processors like Microsoft Word.

|  |  |  |
| --- | --- | --- |
| **Symbol** | **Name** | **How it’s used** |
| √ | Square Root U+221A | C = √ (A² + B²) |
| ∛ | Cube Root U+221B | X = ∛Y |
| ∜ | Fourth Root U+221C | W = ∜ (X+Y) |
| ² | Superscript 2 U+B2 | C=X² |
| ³ | Superscript 3 U+B3 | C=X³ |
| ⁴ | Superscript 4 U+2074 | C=X⁴ |
| - | Hypen-Minus U+2D | The minus sign on a keyboard |
| − | Minus Sign U+2212 | Alternate minus sign |
| – | En dash U+2013 | Alternate minus sign |
| ≅ | Approximately equal to  U+2245 | Helpful to compare floating point numbers. |
| “ ” | Left and right double quotation marks  U+201C and U+201D | Smart Quotes |
| \v | Line Tabulation U+B | Can be used just like a normal carriage return (or Enter, Return or Line feed). Word sometimes converts those into a line tabulation |
| ↲ | Downwards arrow with tip leftwards U+21B2 | Line continuation: use at the end of a line to continue onto the next line. |
| ↲ | Downwards arrow with corner leftwards  U+21B5 | Line continuation |
| ⤶ | Arrow pointing downwards then curving leftwards U+2936 | Line continuation |

# BASIC Statements Reference

## [CALL] <function> (<expression>, …)

Calls the function by name, passing in the given parameters. The word CALL is optional, but the parentheses are required.

**Example of using CALL and defining a FUNCTION:**

CALL Hello(“Mom”)  
Hello(“Dad”)  
  
FUNCTION Hello (name)  
PRINT “Hello ”;name  
RETURN

The example will print Hello Mom and Hello Dad.

## CLS [<color>] and PAPER <color>

CLS will clear the scrolling console and screen and potentially change the screen color. PAPER will change the screen color without clearing the screen.

Normally after you press ‘Run’ the scrolling console will have the results of previous runs and will also have the results of evaluating your program. You can change the background color of the screen by specifying either a color name or a color index from the table below.

Pick the color for CLS and PAPER with either a color number (0 to 7) or a color name.

|  |  |
| --- | --- |
| Color  Number | Color  Name |
| 0 | BLACK |
| 1 | BLUE |
| 2 | RED |
| 3 | MAGENTA |
| 4 | GREEN |
| 5 | CYAN |
| 6 | YELLOW |
| 7 | WHITE |

**Example of using CLS to clear the screen:**

CLS BLUE

**This example is like a color stroboscope:**

REM The screen only shows up if you PRINT

REM something to it.

REM PAUSE delay is in "frames"; there are

REM 50 frames per second.

PRINT " "

delay = 25

FOR i=1 TO 3

FOR color = 0 TO 7

CLS color

PAUSE delay

NEXT color

REM You can specify colors with names

CLS BLACK

PAUSE delay

CLS BLUE

PAUSE delay

CLS RED

PAUSE delay

CLS MAGENTA

PAUSE delay

CLS GREEN

PAUSE delay

CLS CYAN

PAUSE delay

CLS YELLOW

PAUSE delay

CLS WHITE

PAUSE delay

NEXT i

CLS BLACK

## CONSOLE <expression> [, <expression>]

Writes the expressions out to the scrolling console. Any number of expressions can be given (including none)

**Example of writing to the console:**

CLS BLUE

ANGLE = 45

RADIANS = Math.DtoR (ANGLE)

REM PRINT to the screen to see the console.

PRINT " "

CONSOLE "SIN(45 degrees)", SIN(RADIANS)

The CONSOLE command prints out about 0.707. You might need to tap the console button () to see the console. See ‘[Graphics and Best Calculator BASIC](#_Graphics_and_Best)’ for a description of the console screen.

## DUMP

Prints all the variables to the scrolling console. This is a common mechanism to see what your program is doing. DUMP will also print out all of the memory values.

**Example of using DUMP:**

CLS BLUE

ANGLE = 45

RADIANS = Math.DtoR (ANGLE)

REM PRINT to the screen to see the console.

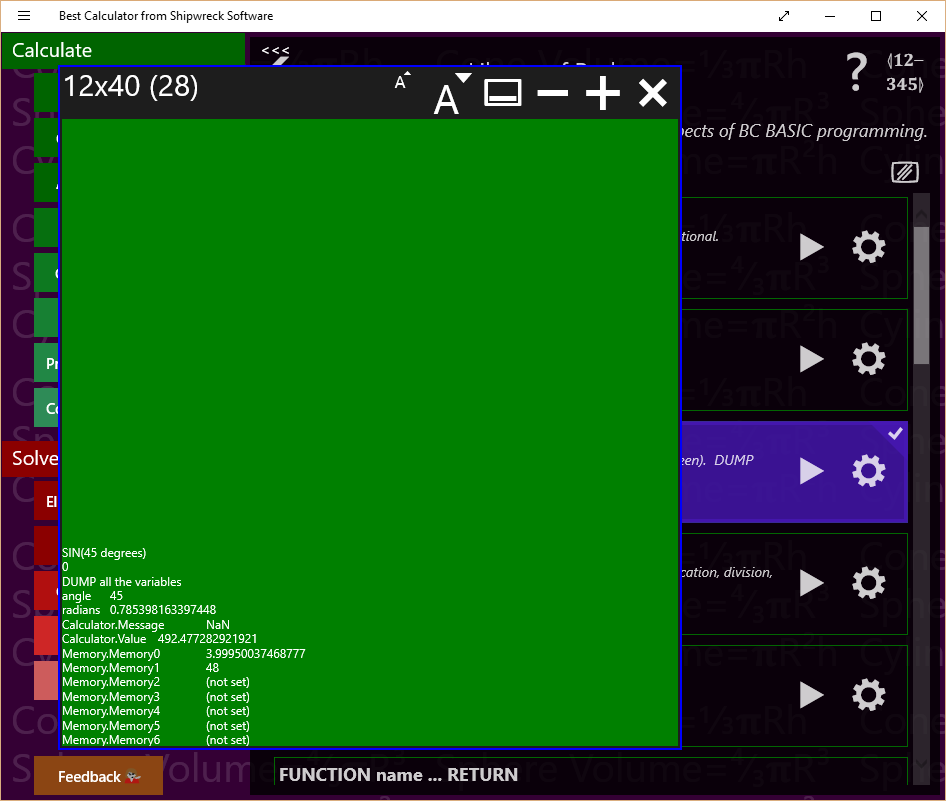
PRINT " "

CONSOLE "SIN(45 degrees)", SIN(RADIANS)

CONSOLE "DUMP all the variables"

DUMP

The variables are then printed to the scrolling console.



You might need to press the console button () to see the console.

## FOR <variable> = <start> TO <end> [STEP <step>] … NEXT <variable>

Use the FOR and NEXT statements to form loops. The variable is the name of the *loop variable*; it will start at the <start> value. Each time through the loop, it will be incremented by the <step> amount (the default is 1) until it’s more than the <end> amount. The start, end and step values are all expressions.  
If the <step> value is negative, then the loop is changed slightly. The variable starts at the start value, is decremented by the <step> value, and the loop ends when the variable is less than, not greater than the <end> value.

The end of the loop is the NEXT <variable> statement; the variable is the exact same as in the FOR statement. For example FOR i=1 TO 10 is ended at a later NEXT I statement.

Common errors:

1. Never GOTO or GOSUB into the middle of a FOR … NEXT loop, and never jump out.
2. FOR … NEXT loops can be nested inside each other, but always nest them correctly. The first FOR must match the last NEXT
3. You can reuse a variable name in several loops, but not nested.
4. You should not modify the variable inside the loop.
5. You will always go through the loop at least once

**Example of a simple FOR loop:**

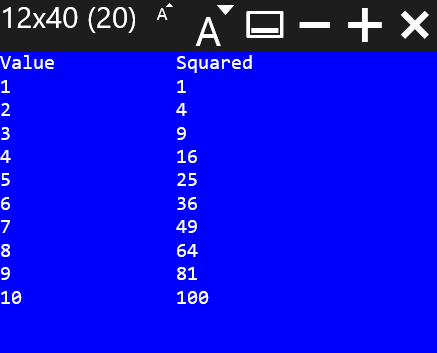
PRINT “Value”, “Squared”

FOR I=1 TO 10

PRINT I, I\*\*2

NEXT I

The example prints a table of numbers and their squares.



**Example of going backwards through a loop:**

PRINT “Value”, “Squared”

FOR I=10 TO 1 STEP -1

PRINT I, I\*\*2

NEXT I

To go backwards, you have to specify a negative STEP value and TO value that’s less than the starting value. In the example, the STEP value is -1, and the TO value (1) is less than the start value (10).

**Example of a nested FOR loop:**

CLS GREEN

PRINT “Value \*\*2 \*\*3 \*\*4 \*\*5”

FOR N=1 TO 10

PRINT AT N+2,1 N

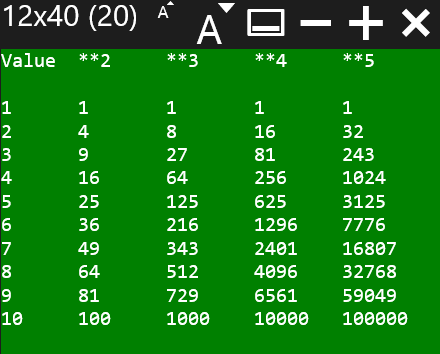
FOR POWER = 2 TO 5

PRINT AT N+2, (POWER-1)\*8 N\*\*POWER

NEXT POWER

NEXT N

This function prints a table of numbers; each number is printed along with the number raised to a power of 2, 3, 4 and 5.



## FUNCTION <name> ( <args> ) … RETURN [<value>]

Creates a function with the given name (expressions are not allowed), taking the arguments. A single value can be returned.

When you CALL a function, you pass in data; that data is then given the names in the function. The names in the argument list are only valid in the function.

It’s considered a best practice to have a single RETURN in a function; it’s easier to understand a function that has only a single return point. However, it’s also sometimes easier to RETURN early.

**Example of using FUNCTION to print to the screen:**

CALL Hello(“Mom”)  
Hello(“Dad”)  
  
FUNCTION Hello (name)  
PRINT “Hello ”;name  
RETURN

In the example, the function HELLO will print whatever value is passed in. It’s called twice, and therefore the program will print out two messages: Hello Mom and Hello Dad.

## GOSUB <linenumber> and RETURN

*Tip: you should almost always use a function instead of using GOSUB. Many earlier versions of BASIC (including Sinclair BASIC and MSW BASIC) used GOSUB instead of FUNCTIONs.*

GOTO and GOSUB both jump to the line number indicated. GOSUB remembers where you jumped from. When the program encounters a RETURN statement, the program will continue at the line after the GOSUB line. You can nest a GOSUB inside a GOSUB routine.

It’s considered good practice that each block of code that you will GOSUB to has a single entry point and usually will have just the one RETURN. Otherwise, your code will get very complex.

**Example of using GOSUB:**

REM Calculate the hypotenuse of a triangle  
A = 3  
B = 4  
**GOSUB** 100  
PRINT “”

DUMP  
STOP

100 REM Calculate the hypotenuse from A and B  
C=2 √ (A\*\*2 + B\*\*2)  
**RETURN**

## GOTO <linenumber>

Jumps to the line number indicated. Unlike GOSUB, you can’t RETURN from a GOTO.

GOTO is often considered harmful.

In BC BASIC, the primary values of a GOTO is compatibility with earlier versions of BASIC and to help “extend” the statements in an IF statement. BC BASIC IF statements can only conditionally run a single statement after the THEN, and do not have ELSE clauses.

## IF (<expression>) THEN <statement>

If the expression is TRUE (not zero), then the statement will be run; otherwise it will not be. Often the statement will be a GOTO or GOSUB. The expression often uses the comparison operators (< <> > plus AND OR and NOT).

**Example of an IF statement:**

a = 15

IF a > 12 THEN PRINT "A is more than a dozen"

In this example, the variable ‘a’ is set to the value 15. The PRINT part of the IF statement will only be executed if the variable a is greater than 12. Since 15 is more than 12, the PRINT statement is executed, and “A is more than a dozen” is printed.

## IMPORT FUNCTIONS FROM “program”

The IMPORT statement will read in all of the functions from a specified program in the same package. This lets you make a package with a common set of functions that all the programs in the package can use. This is useful when several programs in one package all need to perform the same calculation.

The STATISTICS samples do this. There’s a program called “Sample Size Library” that consists of a set of useful statistical functions (MarginOfError, SampleSize, GetZ, and more). The programs that you might bind onto a button then just IMPORT the functions from that program and can call them.

Technical details: as needed, the library program is compiled and the functions remembered. When the IMPORT FUNCTIONS FROM “program” is run, the functions are imported by name. All functions are imported automatically. You can’t pick just one or two functions from a program.

You always IMPORT a program from the same package. You cannot IMPORT FUNCTIONS from a different package. You can IMPORT as many programs as you like; newer imported functions will override older ones. Deliberately doing this is not a best practice.

**Example of the IMPORT statement:**

IMPORT FUNCTIONS FROM "Conversion Library"

The example is taken directly from the AU to Meters program in the Astronomy library. As soon as it’s run, the program can call any function from the “Conversion Library” program.

## INPUT <variable>

*Note: the expression <variable> = INPUT DEFAULT <value> PROMPT <prompt> is a more flexible and powerful way to read in data. The INPUT statement is included to improve compatibility with other versions of BASIC.*

The INPUT statement asks the user to enter a value.

**Example of the INPUT statement:**

REM The a=INPUT expression has more power

REM than INPUT statement. The expression version

REM lets you specify a prompt and a default.

REM The INPUT statement has no default value

REM and no prompt.

INPUT taxrate

REM The INPUT expression has box a default

REM and a prompt. The user has an easier time

REM remembering what to enter.

income = INPUT DEFAULT 40000 ↲

PROMPT “Enter your income for the year”

PRINT "Owe="; taxrate\*income

IF (taxrate\*income > 100) THEN ↲

PRINT “You owe more than 100”

## (LET) <variable> = <expression>

LET is the *assignment* statement. It sets (assigns) the value of the expression to a variable. The variable might or might not already exist. If the variable didn’t already exist, it will be created. If it did exist, the old value is discarded and overwritten with the new value. Some languages call this an *assignment* statement.

Variables start with a letter, and then can be any combination of letters, digits, and underscores. Variables are case sensitive; myage is different from MYAGE and is different from myAge. Expression can include requests for user input.

The word LET is optional in BC BASIC. You will find that it is required in many other variants of BASIC.

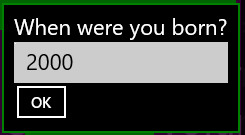
**Examples of the LET statement:**

LET year = 2015  
birth\_year = INPUT DEFAULT year - 15 ↲

PROMPT “When were you born?”  
age = year - birth\_year  
PRINT “”

DUMP

There are 3 LET statements in the example. The first uses the LET word (LET year = 2015). The other two leave off the LET word. When the INPUT expression is run, BC BASIC will pop up a dialog for the user to enter a value.



## PAUSE <frames>

Pauses the screen. This is useful when animating the screen. A value of 1 is one “frame”; there are about 60 frames per second. The value is not exact.

You can see an example of the PAUSE being used in the Colorful Countdown sample in the Quick Samples package.

## PRINT [AT row,col] <expression> [ (, or ;) [AT row,col] <expression]\*

The PRINT command will print one or more expressions to the output screen. BC BASIC remembers where the last thing was printed, and prints the next thing on the next line.

**PRINT expression** prints an expression (either a number of a string) onto the next line of the screen. This is the most common use of the PRINT command. If the screen is already full, nothing happens. The screen won’t clear or scroll to make room for the new text.

**PRINT expression; expression** to print two values next to each other

**PRINT expression, expression** to print values in columns

**PRINT AT [row, col] for exact placement**. You can print each expression at a particular point on the screen with the AT row, col syntax. The rows and column values start with 1,1 at the upper-left corner of the screen. You can tell how large the screen is with the Screen.H and Screen.W values.

**Simplest example of PRINT:**

PRINT “Hello, World”

**Example of PRINT with an AT:**

PRINT AT 1,1 “HELLO”; AT 2,4 “WORLD”

**Example of doing a PRINT with multiple expressions and commas:**

PRINT “HELLO”, “WORLD”

Note that there’s a big gap between the “HELLO” and the “WORLD”. The comma means to print at the next tab stop; those are each 16 characters apart.

**Example of doing a PRINT with multiple expressions and semicolons:**

PRINT “HELLO”; “WORLD”

With semicolons the two words are printed right next to each other without even a single space between them.

## RAND <seed> & the RND value

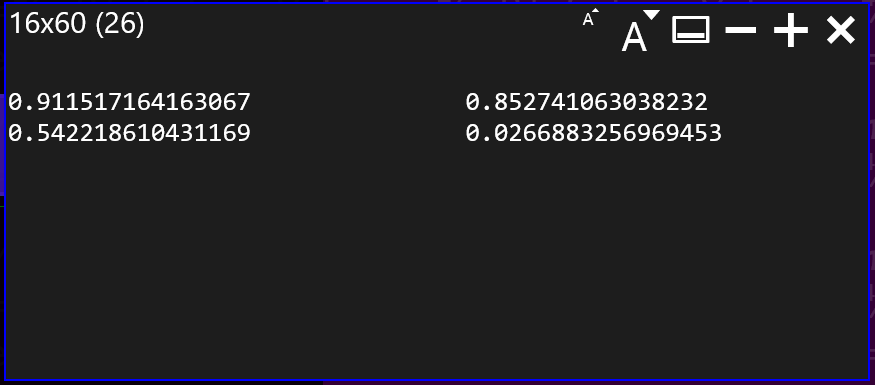
The RND value (like PI) acts like a variable in expressions; unlike PI (which is always the same value), RND will provide a stream of different random numbers between the values 0.0 and 1.0.

**Example of using RND:**

REM Print some random numbers

PRINT RND, RND, RND, RND

This prints numbers like so:



**Technical details**BC BASIC provides access to a single pseudo-random number generator; the stream of values is determined entirely the value of the initial seed value. All expressions in all functions are connected to the same stream of random numbers. The RAND statement will re-seed the random number generator. The seed value 0.0 is special; it will re-seed the generator with a time-dependant value. All other seed values simply reset the seed.

*Tip: the random number generator is not suitable for any cryptographic or security uses.*

## REM comment words to the end of the line

The REM (remark) statement lets you put comments into your code. The words after the REM, and up to the carriage return, are entirely ignored by the program,

**Example of a REM comment statement:**

**REM** Calculate the hypotenuse given A and B  
C=2 √ (A\*\*2 + B\*\*2)

The REM statement will help you understand what your program does. The common practice is to explain the *why* of a program, not the *how*. It’s also common to comment on unusual or clever mathematical techniques. For example, some of the programs in the Real Estate section specifically refer to the exact regulations that are being implemented; this helps later on when you need to verify that the program is implemented in accordance with the laws.

## STOP and END

The STOP statement halts execution of the program. If rerun, the program will start from the very beginning again. It’s common practice for programs to have their main logic at the start, and a number of subroutines (called by using GOSUB) at the end. A STOP statement is placed after the main logic, and before the subroutines.

**Example of using STOP to return a value to the calculator:**

IMPORT FUNCTIONS FROM "Conversion Library"

from = Calculator.Value

m = ConvertToMeters(from, "au")

Calculator.Message = "Convert " + from ↲

+ " au into " + m + " meters"

STOP m

When the example is done, it returns the “m” value to the calculator. The calculator then prints the value into the display. The example is from the Space and Astronomy built-in package. It will not just work if you type it into a new program. To work, it requires that you have a program “Conversion Library” in your package.

END is permitted as a synonym for STOP. It is added for compatibility with other version of BASIC including the first Dartmouth BASIC program (see the historical note).

# Extensions Reference

BC BASIC includes *extensions* to the core BASIC language. The extensions let you

* Set and get values from the calculator screen
* Perform advanced math
* Set and get memory values that persist from one session to another and roam to your other computers
* Get information about the screen

## Calculator.Value and Calculator.Message Extension

You can get and set the current numeric calculator value with the Calculator.Value value.

You can set and get the Calculator.Value; it is always a double. If you try to set it to a string value, the string will be converted to a double (e.g., the string “3.14” becomes the double 3.14; the string “apple” becomes the double NaN).

The Calculator.Message is the small display above the calculator value. When you start Best Calculator, it will say “System test passed”.



You can set (but not get) the Calculator.Message value.

**Examples of using the Calculator.Value and Calculator.Message extensions:**

value = Calculator.Value

retval = value \* Math.PI

Calculator.Message = “Converted diameter ” ↲

+ value + “ to circumference”

Calculator.Value = retval

First the variable called “value” is set to the current value in the calculator window. It’s multiplied by PI. Then a message is printed to the calculator window, and the calculator value is set to the variable called “retval”.

## Memory Extension

You can read and write numeric values to any of the calculator memory. In the example, there are 8 unnamed memory cells (Memory0 to Memory7) plus two named cells (PipeHeight with a value of 45.2 and PipeFraction with a value of 54.4).

Some named memory values are displayed in the Memory screen of Best Calculator. Display the Best Calculator memory screen by tapping the  menu item.

### Memory values

There are three ways to access a cell: by number, by name, and by simple name.

**Access cells by number**: Memory[<expression>]. The named cells can also be accessed by number; in the picture, PipeHeight is the cell right after Memory7 and is accessed as Memory[8].

**Access cells by name**: Memory[<expression>]. For example, the PipeHeight cell can be accessed as Memory[“PipeHeight”]

**Access cells by simple name**: Memory.<constant\_name> where the constant looks like a variable name (without double quotes) and not a string or number. The memory cell name must be compatible with the rules for variable names. For example, the name can’t have spaces or start with a number. The PipeHeight cell can be accessed as Memory.PipeHeight.

### Memory functions

**Is the memory set?** You can tell if a memory cell is set or not in two ways:

Memory.GetOrDefault(<expression>, <default value>) returns either the memory value (if it’s set) or the supplied default value if not.

Memory.IsSet(<expression>) returns true or false (1 or 0) if the memory value is already set.

These functions are commonly used to let you “smart initialize” a value. For example, some calculations use a seldom-changed value (for example, money conversions). You can use Memory.GetOrDefault as the default value in an input expression and then save the value that the user enters.

**Example money conversion program:**

REM

REM The defaults here are roughly the conversion

REM rate from yen to australian dollars.

REM 1 yen is about 0.011 australian dollar;

REM 10000 yen is about 110 australian dollars.

REM

prompt1 = “Conversation rate <from> to <to>” ↲

+ “[e.g., yen to australian dollars]”

prompt2 = “Amount to convert [e.g., amount in yen]”

rate = INPUT DEFAULT Memory.GetOrDefault ↲

("ConversionRate", 0.011) PROMPT prompt1

Memory.ConversionRate = rate

amount = INPUT DEFAULT Memory.GetOrDefault ↲  
 ("ConversionAmount", 10000) PROMPT prompt2

Memory.ConversionAmount = amount

value = amount \* rate

Calculator.Message = "Convert " + amount + ↲

" at a rate of " + rate + " is " + value

Calculator.Value = value

### Memory technical details

**The calculator memory is both persistent and roaming**. Persistent means that it keeps its value between program runs; it’s never automatically reset to zero or other default states. Roaming means that the data roams between your sessions on different computers.

The DUMP command will print out all the memory values to the scrolling console screen.

Best Calculator Memory display will show the first 10 memory slots. However, you can actually access more than that with BC BASIC. Memory slots in BC BASIC can be numbered up to 100. You can show the Best Calculator Memory screen by pressing the Memory  button on the left menu.

Interesting cases for programmers:

1. The memory can only store doubles; it does not store strings.
2. Using indexes less than 0 or more than 100 will silently fail, as will numeric indexes which are not integer (e.g., 3.5). These reads will always return a no such value.
3. An integer index and the string version of the index will refer to the same cell. For example, Memory[1] and Memory[“1”] refer to the same memory cell.
4. If you used a named cell and there isn’t a cell already with that name, the cell won’t be visible in the display. If the user then renames a cell in the memory display with that name, future reads and writes will be to that user-named cell. BC BASIC doesn’t place any limit on how many cells there are. Given any particular name, BC BASIC will prefer to save and load from the visible memory cells, but will use the non-visible cells if it has to. BC BASIC won’t ever change the name of a cell; that’s up to you.
5. There isn’t any way to delete memory cells. Once set, the memory cell is created forever. You can, of course, override the memory value.

**Examples of using the Memory extension:**

CLS

REM

REM You can use integer index values

REM

Memory[0] = Memory[0] + 1

Memory[1] = Memory[1] +10

PRINT "Numeric Index: "; Memory[0]; " "; Memory[1]

REM

REM You can use simple index names

REM

Memory.PipeHeight = Memory.PipeHeight + 1

PRINT "Simple name: "; Memory.PipeHeight

REM

REM You can use index names with square brackets

REM

Memory["PipeHeight"] = Memory["PipeHeight"] + 1

PRINT "Const Index: "; Memory["PipeHeight"]

REM

REM You can use variables and expressions

REM in the index name

REM

name = "PipeHeight"

Memory[name] = Memory[name] + 1

PRINT "Variable Index: "; Memory[name]

prefix = "Pipe"

suffix = "Height"

Memory[prefix + suffix] = ↲

Memory[prefix + suffix] + 1

PRINT "Expression Index: "; ↲

Memory[prefix + suffix]

REM Some memory isn't set

a = Memory.NotSet

isset = Memory.IsSet ("PipeHeight")

isnotset = Memory.IsSet ("NotSet")

ns = Memory.NotSet =Memory.NotSet

REM Memory.GetOrDefault returns either the

REM memory value or the default value   
REM depending on whether the memory was

REM set or not.

default = Memory.GetOrDefault ("NotSet", 34)

notdefault = ↲

Memory.GetOrDefault ("PipeHeight", 34)

DUMP

## Math Extension

BC BASIC includes a full set of math functions and constants divided into categories for trigonometry, rounding, logarithm and power functions and other functions.

### Math Trigonometry functions (Math.Sin (radians) and more)

BC BASIC does all trigonometry calculations in radians. The function Math.DtoR (degrees) will convert degrees to radians and Math.RtoD(radians) will convert radians to degrees.

|  |  |
| --- | --- |
| **Function** | **Notes** |
| Math.Acos(value) | Calculates the inverse of Math.Cos; given a value will compute the corresponding angle in radians. |
| Math.Asin(value) | Calculates the inverse of Math.Sin; given a value will compute the corresponding angle in radians. |
| Math.Atan(value) | Calculates the inverse of Math.Tan; given a value will compute the corresponding angle in radians. Note that Math.Tan of 90° is infinite. |
| Math.Atan2(y, x) | Calculates the inverse tangent of given an Y and X value. Note that Y is given first. This matches most common program languages including C#, Java, Fortran, C and JavaScript. Unlike the Math.Atan function, Math.Atan2 can handle angles of 90° |
| Math.Cos (radians) | Calculates the cosine of an angle given in radians |
| Math.Cosh (radian) | Calculates the hyperbolic cosine of an angle given in radians |
| Math.DtoR (degrees) | Converts degrees to radians |
| Math.RtoD (radians) | Converts radians to degrees |
| Math.Sin (radians) | Calculates the sin of an angle given in radians |
| Math.Sinh (radians) | Calculates the hyperbolic sin of an angle given in radians |
| Math.Tan (radians) | Calculates the tangent of an angle given radians |
| Math.Tanh (radians) | Calculates the hyperbolic tangent of an angle given radians |

### Math rounding and sign functions (Floor(), Round() and more)

|  |  |
| --- | --- |
| **Function** | **Notes** |
| Math.Abs (value) | Calculate the absolute value of a number. |
| Math.Ceiling (value) | Calculates the ceiling of a number. The ceiling is the number rounded up to the nearest integer. For example, Math.Ceiling (2.2) is 3. Ceilings of negative numbers round up (e.g., to be closer to zero), so Math.Ceiling (-2.2) is -2. |
| Math.Floor (value) | Calculates the floor of a number. The floor is the number rounded down to the nearest integer. For example, Math.Floor (2.8) is 2; Math.Floor (-2.8) is -3. |
| Math.Frac (value) | Returns the fractional part of a number (the part after the decimal sign). Math.Frac(3.456) is 0.456.  For negative numbers, Math.Frac returns the difference between the number at the next higher number. For example, Math.Frac(-7.4) is 0.6.  Math.Floor(value) + Math.Frac(value) is equal to the original value. |
| Math.Max (value, …) | Returns the largest value of a set of numbers. You may give one or more values to Math.Max() |
| Math.Min (value, …) | Returns the smallest value of a set of numbers. You may give one or more values to Math.Max() |
| Math.Mod (v1, v2) | Returns the remainder when v1 is divided by v2. For example, Math.Mod(7,3) is 1 because 3 goes into 7 2 times with a remainder of 1. Math.Mod(7.6, 3.1) is 1.4 because it’s the remainder after 3.1 is multiplied by 2. |
| Math.Round (value) | Calculates the rounded value of a number. The rounded value is the one closest to an integer. Math.Round (2.2) is 2; Math.Round (2.8) is 3. If a number is a “.5” number, it is rounded down (technically, rounded towards zero; Math.Round (2.5) is 2, and Math.Round (-2.5) is -2.) |
| Math.Sign (value) | Return the sign of a number. The sign is 1 for positive values, -1 for negative values, and 0 for zero. |
| Math.Truncate (value) | Calculates the truncated value of a number. The truncated value is the integer value closest to zero. For positive numbers, this is like Math.Floor (for example, Math.Truncate (2.8) is 2). For negative numbers, this is like Ceiling (for example, Math.Truncate (-2.8) is -2, the integer closer to zero) |

### Math logarithm and power functions (Math.Log, Math.Exp, and more)

|  |  |
| --- | --- |
| **Function** | **Notes** |
| Math.Exp(value) | Calculates the value *e*value for any given value. This is the reverse of the Math.Log function |
| Math.Log (value) Math.Log (value, base) | Can do two different calculations. When given just one value, calculates the natural (base *e*) logarithm of the given value. When given two numbers, calculates the logbase of the value. |
| Math.Log2 (value) | Calculates the base-2 logarithm of a number. This is useful when dealing with computer math.  Simple example: you’re writing a program, and certain variable will hold a number from 0 to 934. How many bits are needed to hold this value? Answer: Math.Log2(934) is about 9.87; rounding up, you discover than you will need a 10-bit field to hold the number.  Sophisticated example: You need to store 200 numbers, each of which is a value 0 to 11 (inclusive, so there are 12 total values). Assuming best bit packing but no compression, how much space do you need? The answer is that 200 \* Math.Log2(12) = 717 bits; divide by 8 to to discover that your data will fit into 90 bytes of space. |
| Math.Log10(value) | Calculates the base-10 logarithm of a number. This is useful when rounding a number up to the nearest power-of-ten.  For example, you want the first power of ten (e.g., 10, 100, 1000) of 783. Math.Log10 (783) is 2.89; rounded up this is 3. Math.Pow (10, 3) is 1000, and that’s the closest larger power of ten of the number. |
| Math.Pow (x, y) | Calculates xy. For example, Math.Pow (10, 3) is 1000. |
| Math.Sqrt(value) | Calculates the square root of the value. |

### Other Math functions (Math.Factorial and Math.IsNaN)

**The Math.Factorial Function**

Math.Factorial(value) is the *n!* function. For any given integer, it returns the product of all the integers less than or equal to *n*. For example, Math.Factorial(5) is 5 \* 4 \* 3 \* 2 \* 1, or 120.

Math.Factorial returns NaN (not a number) for any input that isn’t an integer or is less than zero. Math.Factorial(0) is 1.

**Example of using Math.Factorial**

REM simple binding for X! so it's on the

REM main calculator page

x=Math.Factorial(Calculator.Value)  
STOP x

**The Math.NaN function**

Math.IsNaN(value) returns true (1) when the given value evaluates to a NaN value and 0 otherwise.

NaN values will propagate their values in BC BASIC, and don’t compare equal to each other. The only simple way to tell a number is a NaN value is to use the Math.IsNaN function.

### Math.PI and Math.E values

Two constants, Math.PI and Math.E are available from the Math extension.

**Example of using Math.PI and Math.E:**

REM Converts a circle AREA to DIAMETER

REM area = Math.PI \* R\*\*2, which means

value = Calculator.Value

retval = 2 \* SQR (value / Math.PI)

Calculator.Message = “Converted area “ + value + “ to diameter”

Calculator.Value = retval

## Screen.H and Screen.W Extension

The Screen.H and Screen.W provide the height and width, in fixed-width characters, of the screen. The most common use is to help lay out text to fix a particular screen size.

**Example: using Screen.H and Screen.W to print in the middle of the screen:**

CLS MAGENTA

PrintCenter (“Hello, world!”)

FUNCTION PrintCenter (str)

lmargin = 1+INT (( Screen.W - LEN str) / 2)

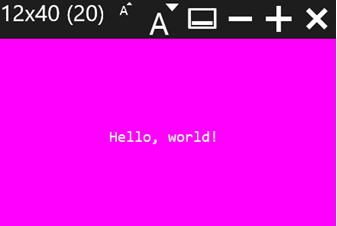
IF (lmargin < 1) THEN lmargin = 1

row = INT ((Screen.H) / 2)

PRINT AT row,lmargin str

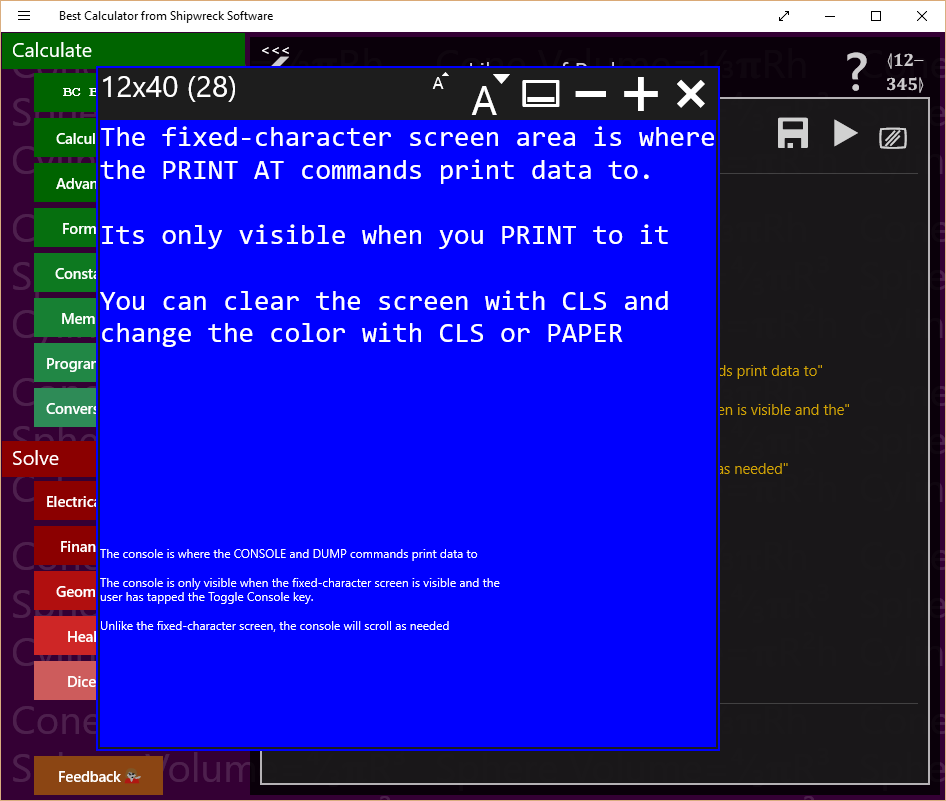
RETURN

The resulting output shows a neatly-centered message.



# Graphics and BC BASIC

BC BASIC has two ways to print to the screen. The main way is the PRINT and CLS and PAPER commands; they print to the “Fixed-character” screen. The CONSOLE and DUMP commands write to the console.



In the example, the area of the screen in blue with large type is the fixed-character screen. It’s called that because each character prints at the same width. This lets you make tables and diagrams more easily.

Underneath is the console. The primary use of the console is debugging. Normally the console is not visible.

## Fixed character screen commands

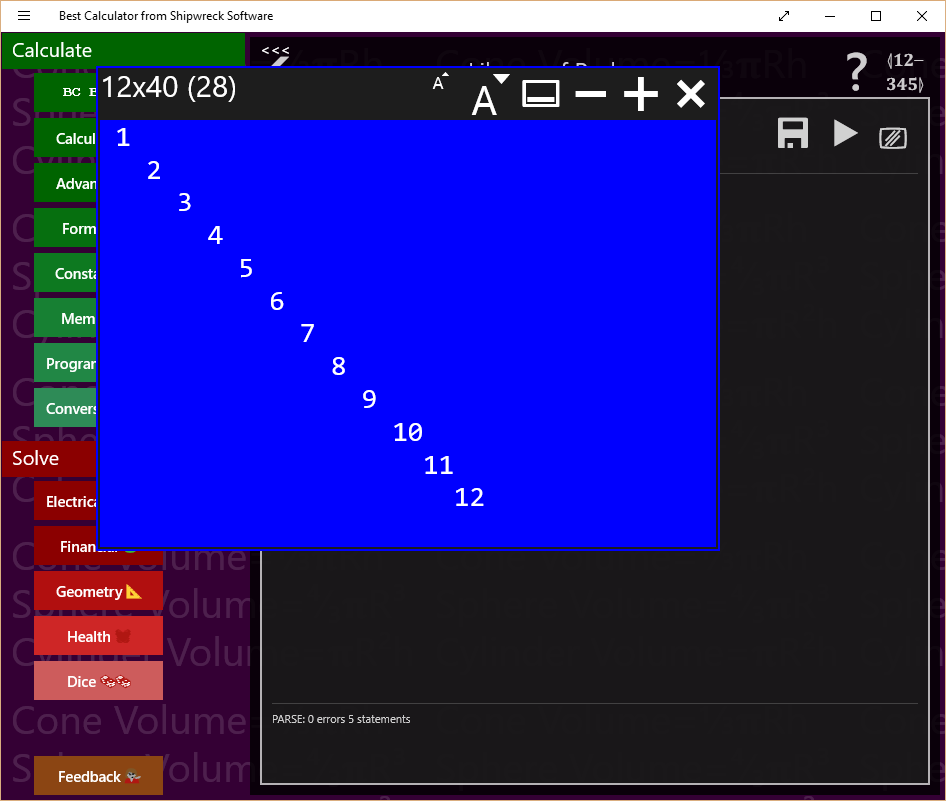
PRINT [AT line, column] <expression>   
A comma between expressions will print each expression at 16-character positions

**Simple example:**

CLS BLUE

R = 1  
10 PRINT AT R, 2\*R R  
R = R + 1  
IF (R < 20) THEN GOTO 10

The following output is produced:



**Example that prints circles on the screen:**

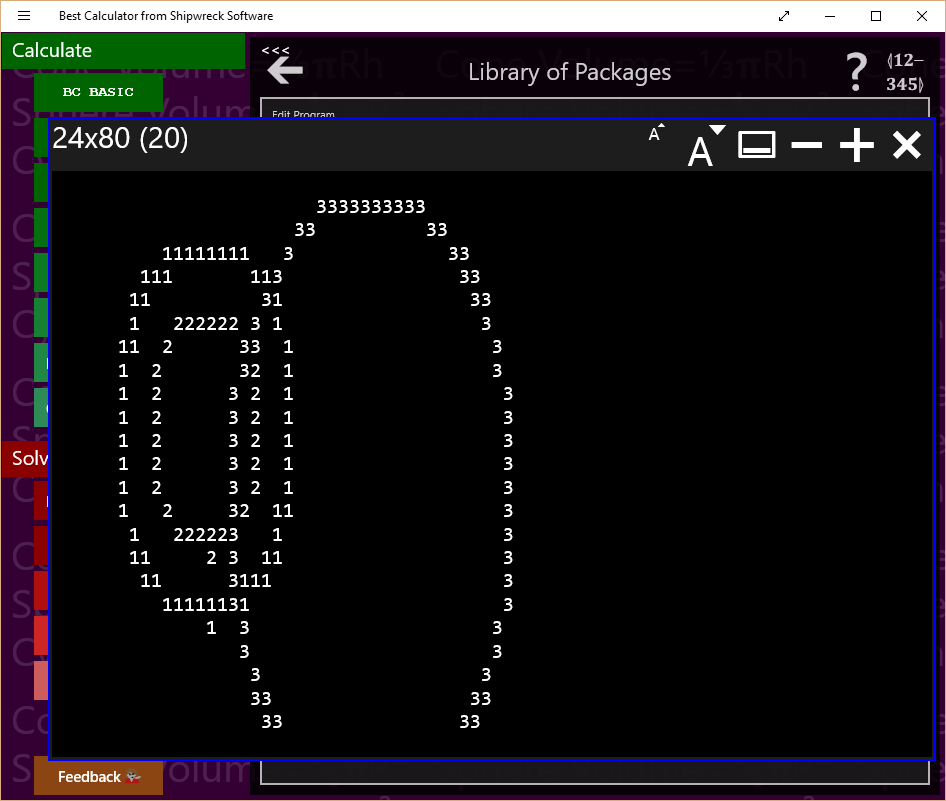
REM  
REM PLOT 3 CIRCLES  
REM  
  
CHAR = 1  
R = 8  
CX = 15  
CY = 12  
GOSUB 1000

CHAR = 2  
R=5  
GOSUB 1000  
  
CHAR=3  
R=13  
CX=30  
CY=15  
GOSUB 1000

STOP

1000 REM DO A CIRCLE USING CHAR R CX AND CY  
S = 0  
1010 REM TOP OF LOOP  
COL = R \* SIN(S) + CX  
ROW = R \* COS(S) + CY  
PRINT AT ROW,COL CHAR  
S = S + 0.05  
IF (S < 7) THEN GOTO 1010  
RETURN

The circle program makes this output



## Console Commands

The console commands work on the console, a scrolling list of small-font output. The primary console commands are CONSOLE (writes to the console), DUMP (writes the name and value of all of the BC BASIC variables to the console) and the CLS and PAPER commands (which clear and possible change the color).

**Example of using CLS to clear the screen and change the color:**

CLS YELLOW

# Using the library, step by step

Once you have written a simple program, you might want to write more programs, and keep them all. BC BASIC includes simple Library functionality to keep all your programs. The Library also includes a series of sample programs for you to use.

In this first example, we’ll write a simple program to convert square feet to acres. The steps are listed in the diagram and will be described in detail in each section.

## Add a new package for your program

Press the BC BASIC button and select Library. The *Library of Packages* screen will pop up. A package is a bundle of individual *programs*; you’re going to make a single new *package* that contains a simple *program*. The program will convert from square feet to acres.

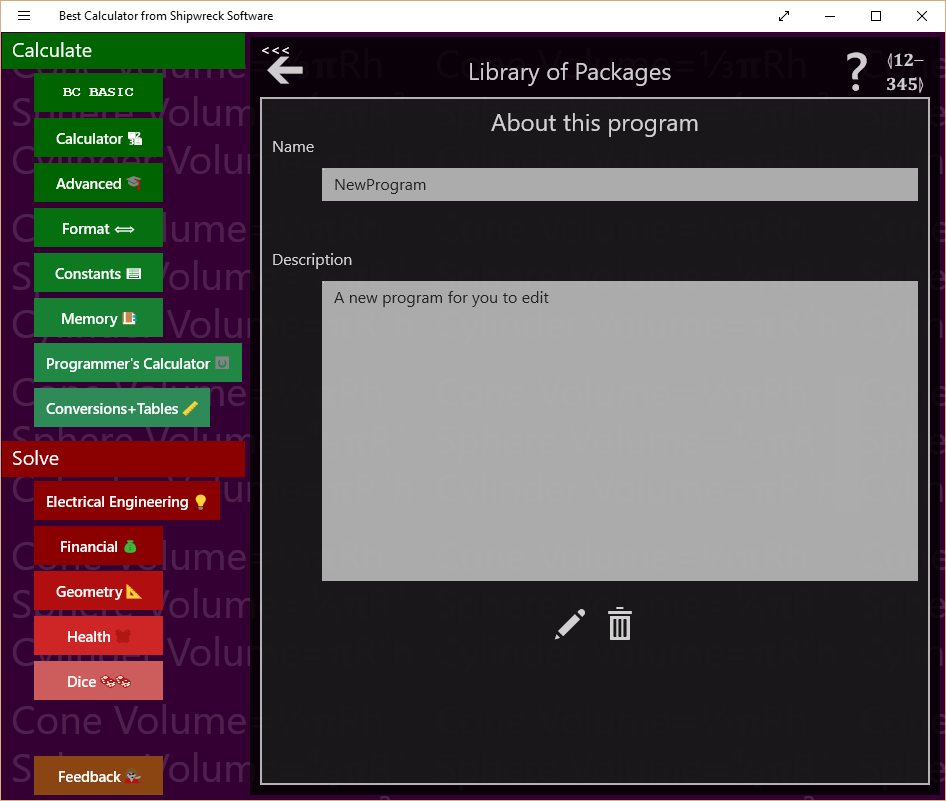
First, you need to make the *package* that your programs will be part of.

Tap the + button to add a new package. A new package will be created with a default name of “New Package”.

After you’ve written more programs and want to create more packages, you’ll probably want to rename this package. Do that by tapping the package’s GEAR button (). Then change the name and description. The changes take place right away. Tap the BACK ARROW to get back to the list of packages.

## Add a new program to your package

Your new package is now ready for you to add your new program. Tap the package to see the list of programs in the package, and then tap the + to add a new program. It will be given the name NewProgram . It’s also got a description and some code.

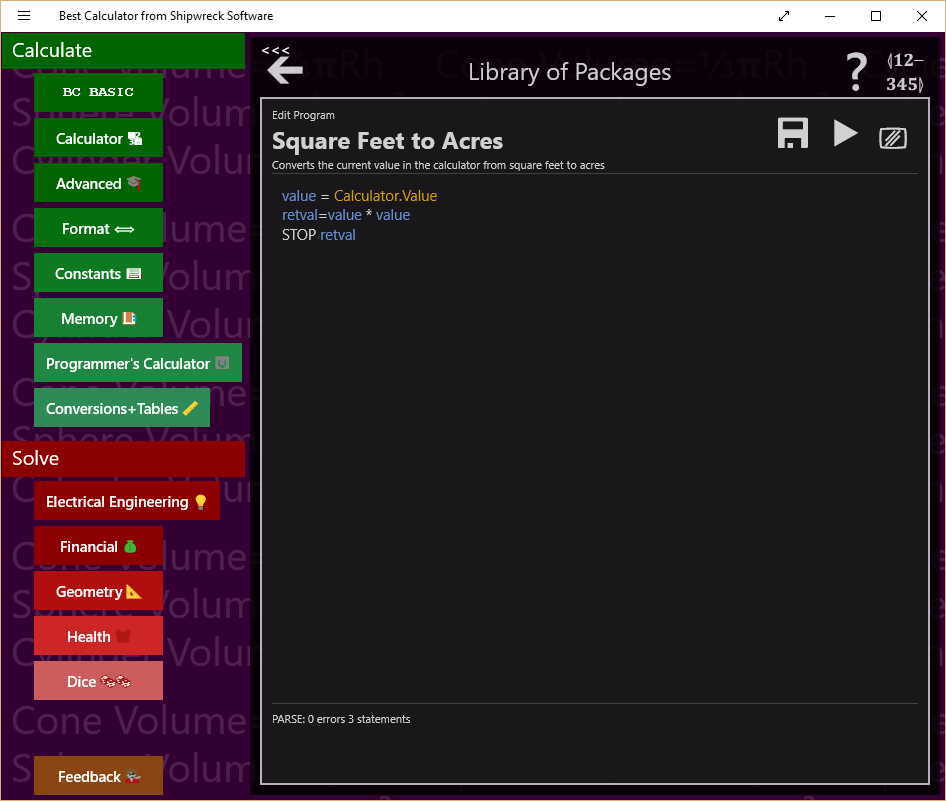
Tap the program’s GEAR () button to bring up the About this program screen.

Change the name to “Square Feet to Acres” and the description to “Converts the current value in the calculator from square feet to acres”. You don’t have to do anything special to save the name and description; they are saved automatically when your program is saved.

You can also set the key label value. If you bind a program to a programmable key, this string will be displayed. Set it now to “FT>ACRE”. The string has to be short to fit onto a button.

## Edit the program to do the conversion

To get to the edit screen, you can either tap the EDIT button at the bottom of the About this program screen, or you can tap the BACK ARROW button to get back to the Programs list and then tap the program’s EDIT button.

The program starts out with a sort of mini-sample. You’ll be deleting the mini-sample code and replacing it with your own.

Most conversion programs follow the same pattern:

1. Get a number from the calculator display. This is the number of square feet.
2. Multiple or divide it to get the new value. To convert square feet to acres, just divide by 43560.
3. Output a string to the calculator display to say what we’ve done
4. Return the numeric value so it’s set into the calculator

The program to do these is:

value = Calculator.Value

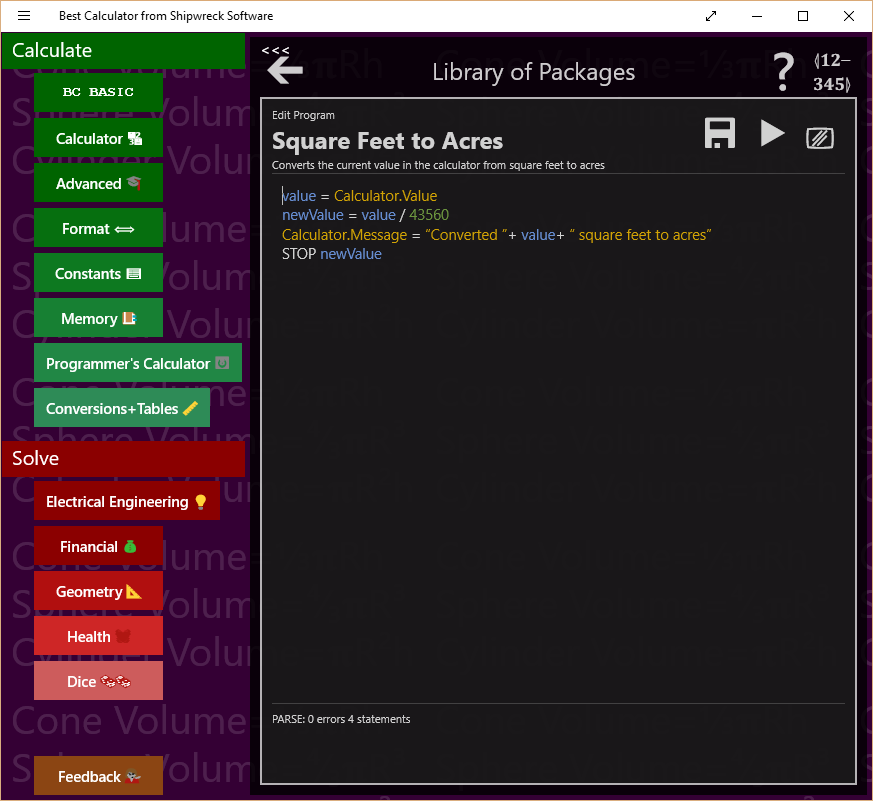
newValue = value / 43560

Calculator.Message = “Converted ”+ value ↲

+ “ square feet to acres”

STOP newValue

Each line corresponds to one of our steps. Enter this code into the program area.



## Run the program to test it

Before we run the program, we need to have a “known good” conversion. Type “Convert 10000 square feet to acres” into a web search; it should tell you that 10000 square feet is 0.229568 acres.

To test the program, tap the Calculator button; this dismisses the BC BASIC programming area and pops up the calculator. Type in the starting value of 10000.

Now tap the BC BASIC button. The BC BASIC programming area pops up again, right where you were. Tap the RUN button to run the program. Your program is automatically saved when you run the program. You can also press the F5 key to run your program.

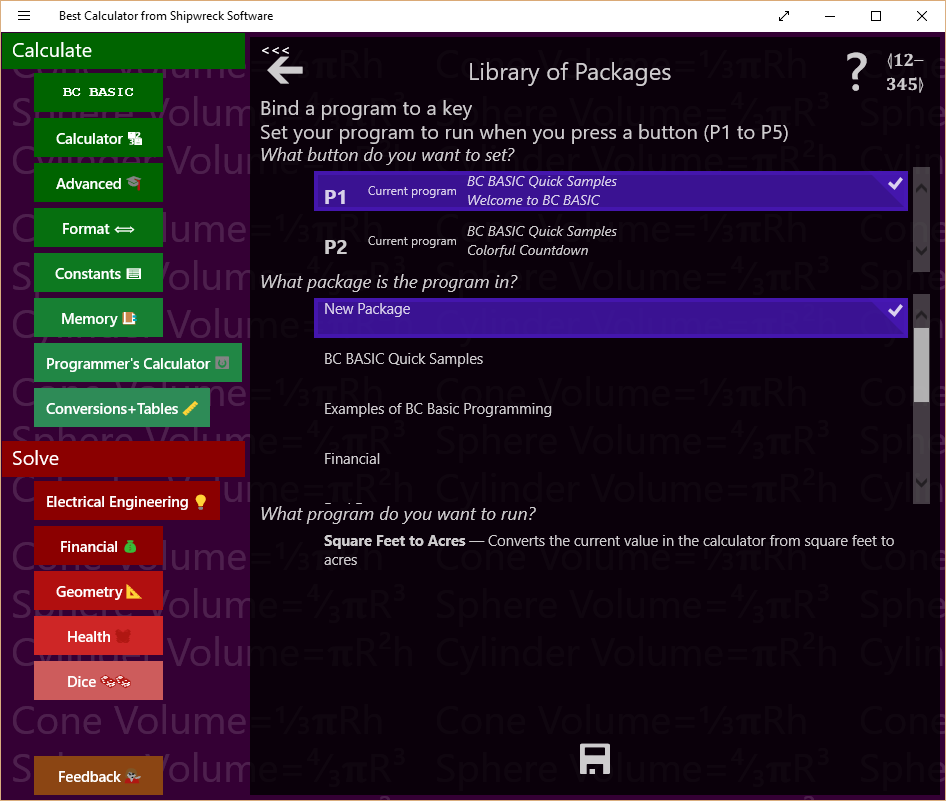
*The results go to the calculator screen, which is not visible.* That’s why it looks like nothing happened. Tap the Calculator button again to see the calculator. You should see this:

The program works! It’s done the conversion, and reminded you of what exactly it did.

## Bind the program to a key

Now we’re going to *bind* the program to one of the programmable keys on the calculator. They are the ones marked P1 to P4. Best Calculator comes with these keys already programmed to some common tasks.

In any of the programming dialogs, tap the BIND key (); it’s the one in the upper-right corner. It brings up the binding dialog.

To bind a key, you first pick the key to bind, the package and program to bind to, and then press save. You can also set what the button should say.

The first question is *What button do you want to bind to?* Tap one of the buttons in the button list (labeled P1, P2, P3 and so on) to pick a button to bind to. People often just pick button P1. The button list tells you what package and program the button is currently bound to. This helps you pick the right button to use.

The second question is *What package is the program in?* All the possible packages are listed. As you tap on a package, the next list changes to show the programs in that package. Tap on **New Package** to pick your new package. If you’ve change the name of the package, pick the new name.

The third question is *What program do you want to run?* Tap **Square feet to acres** to select your new program.

You can optionally set the label of the button. The default value is set from the “Key Label” value when you update the program data (the “About this program” screen). You should have already set it to FT>ACRE.

Lastly, **be sure to tap the SAVE button** (). Your selection isn’t saved until you press save.

Now verify that the button works how it should. Tap the Calculator button to see the calculator again. Now enter a value into the calculator. You might want to enter 10000 since you already know how many acres it is. Now press the button you bound (probably the P1 button). The value in the calculator screen should be replaced with 0.229568 (and there is a message that it just converted 10000 square feet to acres).

## Next steps

Now you’ve seen the dialogs and screens that you need to use to create a BC BASIC program. Your next step is to try it! Pick a problem that you have where you work, at home, for your hobby, or your schoolwork. Conversion programs are often a great way to start; lots of times you have to convert one value to another.

If you need to make a conversion program, take a look at the code in the Astronomy package. It demonstrates a more advanced way how to make a single central library program that handles lots of conversions. Or you can just write each one just like you did this one.

Sometimes you have to enter several numbers. The Arc Length program shows how you can prompt the user for several values. The Money Conversion program in the Quick Samples library shows how you can ask the user for input and remember the last value entered. By setting a default value for the value to be entered, you can really make your work flow go faster.

# Index to BC BASIC Manual

- (subtract), 29, 30

( ) (parenthesis)

in expressions, 28

when calling a function, required, 28

\*(multiply), 30

\*\* (raise to the power), 29

/ (divide), 30

[ ] (square brackets)

in expressions, 28

+ (add), 30

< (less than), 31

<= (less than or equal), 31

<> (not equal), 31

= (equal), 31

> (greater than), 31

>= (greater than or equal), 31

≅ (approximately equal), 31

² (square), 29

³ (cube), 29

⁴ (fourth power), 29

ABS, 35

ACS, 34

AND, 31

ASN, 34

*assignment*, 53

ATN, 34

BASIC

history, iii

bind, 12

Calculator.Message, 59

Calculator.Value, 59

CALL, 44

CHR, 37

CLS, 44

clearing the screen manually, 16

CODE, 37

Comment (using REM), 57

Console, 22

display or hide (toggle), 23

CONSOLE, 46

COS, 34

double

numbers stored as doubles, 25

DUMP, 46

END, 57

Escape key, 21

EXP, 34

exponential notation, 25

expressions, 27

F5 key, 22

Fixed character size screen, 22

floats, 25

Font size

indicator, 22

larger, 23

smaller, 23

FOR, 47

FUNCTION, 50

GOSUB, 50

GOTO, 51

IF - THEN, 51

IMPORT FUNCTIONS, 52

import package, 12

infinite loop

stopping, 22

INPUT (operator)

details, 33

INPUT (operator, preferred), 31

INPUT (statement, not preferred), 53

INT, 35

Integers, 25

LEFT, 35

LEN, 36

LET, 53

library

how to display properties, 10

made up of packages, 9

Line ending

\n, 24

\r, 24

\v, 24

LN, 34

Math.Abs, 65

Math.Acos, 64

Math.Asin, 65

Math.Atan, 65

Math.Atan2, 65

Math.Ceiling, 65

Math.Cos, 65

Math.Cosh, 65

Math.DtoR, 65

Math.E, 69

Math.Exp, 67

**Math.Factorial**, 68

Math.Floor, 66

Math.Frac, 66

Math.Log, 67

Math.Log10, 67

Math.Log2, 67

Math.Max, 66

Math.Min, 66

Math.Mod, 66

**Math.NaN**, 68

Math.PI, 69

Math.Pow, 68

Math.Round, 66

Math.RtoD, 65

Math.Sign, 66

Math.Sin, 65

Math.Sinh, 65

Math.Sqrt, 68

Math.Tan, 65

Math.Tanh, 65

Math.Truncate, 66

Memory.<constant\_name>, 61

Memory.GetOrDefault, 61

Memory.IsSet, 61

Memory[<expression>], 61

MID, 35

NEXT, 47

NOT, 31

OR, 31

Output screen, 22

close, 23

larger, 23

smaller, 23

package

about, 14

delete, 15

how to add, 76

how to display properties, 11

how to export, 15

how to import, 12

made up of programs, 16

PAUSE, 54

PI, 28

PRINT, 55

program

about, 18

add, 16

delete, 19

edit, 17

edit dialog, 20

edit from the About dialog, 19

how to add, 77

how to bind to a program key, 80

how to run, 79

properties, 17

run, 17

running from edit dialog, 21

saving while editing, 21

program list

how to display, 10

RAND (set random seed), 56

REM, 57

RIGHT, 35

RND, 28, 56

save (export) package, 15

screen

clear, 16

Screen.H, 69

Screen.W, 69

SGN, 35

SIN, 34

Smart quotes

and strings, 26

SQR, 34

statement, 38

statement terminator

not required in BC BASIC, 24

STOP, 57

String constants, 26

TAN, 34

Unicode

for minus signs, 26

VAL, 38

Variables, 26

ending in $ (dollar sign), 26