

# techBASIC™ 3.3.1

## Reference Manual





A product of  
Byte Works®, Inc.  
<http://www.byteworks.us>

## Credits

techBASIC Programming  
Mike Westerfield

techBASIC Art  
Karen Bennett

Documentation  
Mike Westerfield

## Licenses

### **Random Number Generator:**

Copyright (c) 2006,2007 Mutsuo Saito, Makoto Matsumoto and Hiroshima University. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

\* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

\* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

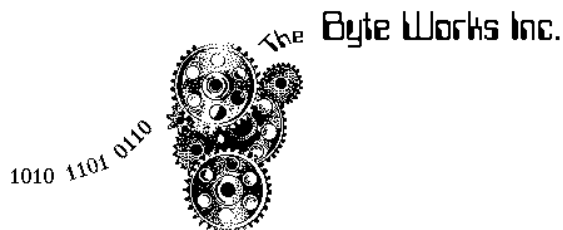
\* Neither the name of the Hiroshima University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Apple, iPhone, iPad and iPod are registered trademarks of Apple Computer, Inc.

The Byte Works is a registered trademark of The Byte Works, Inc.

techBASIC and techBASIC Sampler are trademarks of The Byte Works, Inc.



Copyright 2016  
By The Byte Works, Inc.  
All Rights Reserved





## Table of Contents

<b>Chapter 1 – Introducing techBASIC .....</b>	<b>1</b>
techBASIC vs. techBASIC Sampler .....	1
About the Manual.....	1
Typographical Conventions .....	1
Other Books and Reference Materials.....	2
<b>Chapter 2 – The iPad User Interface.....</b>	<b>5</b>
<b>Managing Programs .....</b>	<b>5</b>
Navigation in Folders.....	5
Editing Programs .....	6
Running Programs .....	6
Creating New Programs.....	6
Deleting and Renaming Programs.....	7
Moving Programs.....	8
<b>Changing Views.....</b>	<b>8</b>
<b>Editing Programs .....</b>	<b>9</b>
Find .....	10
Subs.....	11
Undo and Redo.....	11
Using Help.....	12
<b>Program Output .....</b>	<b>12</b>
Text Input and Output on the Console.....	12
Graphics Output .....	13
<b>Debugging Programs .....</b>	<b>14</b>
Setting and Clearing Breakpoints .....	14
Stepping Through Programs.....	14
Running and Pausing Programs .....	15
Stopping a Program .....	15
The Variable View .....	15
The Stack View .....	16
Understanding Errors .....	16
<b>Sharing and Syncing.....</b>	<b>16</b>
Sending Graphics to Photo.....	16
Sending Programs and Text to Mail.....	16
Moving Programs and Data To Your Desktop Computer .....	16
<b>About.....</b>	<b>18</b>
<b>Preferences .....</b>	<b>18</b>
<b>Chapter 3 – The iPhone and iPod User Interface .....</b>	<b>21</b>
<b>Changing Views.....</b>	<b>21</b>
<b>Managing Programs .....</b>	<b>21</b>
Navigation in Folders.....	22
Editing Programs .....	22
Running Programs .....	22
Creating New Programs.....	22
Deleting and Renaming Programs.....	23
Moving Programs.....	23

## Table of Contents

<b>Editing Programs .....</b>	<b>24</b>
Find .....	24
Undo and Redo.....	25
Using Help.....	25
<b>Program Output .....</b>	<b>26</b>
Text Input and Output on the Console.....	26
Graphics Output .....	26
<b>Debugging Programs .....</b>	<b>28</b>
Setting and Clearing Breakpoints .....	28
Stepping Through Programs.....	28
Running and Pausing Programs .....	28
Stopping a Program .....	29
The Stack View .....	29
Understanding Errors .....	29
<b>Sharing and Syncing.....</b>	<b>30</b>
Sending Graphics to Photo.....	30
Sending Programs and Text to Mail.....	30
Moving Programs and Data To Your Desktop Computer .....	30
<b>About.....</b>	<b>32</b>
<b>Preferences .....</b>	<b>33</b>
<b>Chapter 4 – Program Symbols .....</b>	<b>35</b>
<b>Character Set .....</b>	<b>35</b>
<b>Identifiers.....</b>	<b>35</b>
<b>Reserved Words .....</b>	<b>36</b>
<b>Reserved Symbols.....</b>	<b>36</b>
<b>Constants .....</b>	<b>36</b>
Decimal Integers .....	36
Hexadecimal Integers .....	37
Real Numbers .....	37
String Constants .....	38
Array Constants.....	38
<b>White Space .....</b>	<b>39</b>
<b>Comments.....</b>	<b>39</b>
<b>Chapter 5 – Types of Data .....</b>	<b>41</b>
<b>Integers .....</b>	<b>41</b>
<b>Reals.....</b>	<b>41</b>
Infinity .....	41
NaN .....	42
<b>Strings.....</b>	<b>42</b>
<b>Objects.....</b>	<b>42</b>
<b>Chapter 6 – BASIC Programs .....</b>	<b>45</b>
<b>The Anatomy of a BASIC Program .....</b>	<b>45</b>
Subroutines .....	45
Line Numbers and or label .....	45
Multiple Statements on One Line.....	46
Continuation Lines .....	46
<b>Chapter 7 – Declaring Variables and Types .....</b>	<b>49</b>
<b>What Is a Type?.....</b>	<b>49</b>

The Kinds of Types.....	49
<b>Type Compatibility.....</b>	<b>51</b>
Numeric Type Compatibility.....	51
Array Type Compatibility.....	52
Strings.....	53
Objects.....	53
<b>Default Types .....</b>	<b>53</b>
<b>Declaring Types and Variables .....</b>	<b>55</b>
<b>Chapter 8 – Expressions and Assignments.....</b>	<b>59</b>
<b>Expressions.....</b>	<b>59</b>
Kinds of Expressions.....	59
Evaluating Expressions.....	60
Terms.....	71
L-Values .....	73
<b>The Assignment Statements.....</b>	<b>74</b>
<b>Mathematical Functions .....</b>	<b>75</b>
<b>Array Functions.....</b>	<b>82</b>
<b>String Functions .....</b>	<b>85</b>
<b>Chapter 9 – Control Statements .....</b>	<b>91</b>
<b>Looping.....</b>	<b>91</b>
<b>Making Decisions .....</b>	<b>95</b>
<b>Jumping Around .....</b>	<b>98</b>
<b>Handling Errors.....</b>	<b>99</b>
<b>Stopping a Program .....</b>	<b>100</b>
<b>Clearing the Workspace .....</b>	<b>100</b>
<b>Chapter 10 – Input and Output .....</b>	<b>103</b>
<b>Printing Text .....</b>	<b>103</b>
<b>Reading Text.....</b>	<b>113</b>
<b>Imbedding Data In The Program .....</b>	<b>116</b>
<b>Chapter 11 – Disk Input and Output.....</b>	<b>119</b>
<b>File Names .....</b>	<b>119</b>
Path Names.....	119
The Default Prefix.....	120
The Sandbox.....	121
<b>File Numbers .....</b>	<b>121</b>
<b>File Input and Output Examples .....</b>	<b>121</b>
Line Oriented Text Files.....	121
Binary Files.....	122
Backtracking in Files .....	126
Reading An Entire File.....	128
Random Access Files .....	129
<b>Opening and Closing Files .....</b>	<b>130</b>
<b>Reading and Writing Files.....</b>	<b>131</b>
<b>Dealing With Directories and Files.....</b>	<b>132</b>
<b>Chapter 12 – Subroutines .....</b>	<b>137</b>
<b>GOSUB Subroutines.....</b>	<b>137</b>
<b>Subroutines and Functions.....</b>	<b>138</b>

## Table of Contents

SUB and FUNCTION Parameter Lists .....	138
Local Variables and Types .....	142
Recursion with SUB and FUNCTION .....	142
<b>Chapter 13 – techBASIC Events .....</b>	<b>145</b>
<b>The Two Kinds of techBASIC Programs.....</b>	<b>145</b>
Classic BASIC Programs .....	145
Event Driven Programs .....	145
<b>Handling Events .....</b>	<b>146</b>
<b>Chapter 14 – System Classes .....</b>	<b>161</b>
Date .....	161
Email .....	162
Err .....	164
Event .....	165
Math.....	167
System .....	181
<b>Chapter 15 – Sensor and Communication Classes.....</b>	<b>187</b>
Audio .....	187
BLE .....	189
BLE Central Communications.....	189
BLE Slave Communications.....	190
BLEATTRequest.....	192
BLECharacteristic .....	193
BLEDescriptor.....	194
BLEMutableCharacteristic.....	195
BLEMutableService .....	197
BLEPeripheral.....	198
BLEPeripheralManager .....	204
BLEService.....	207
Comm .....	208
HiJack.....	216
Sensors .....	218
<b>Chapter 16 – Graphics Classes.....</b>	<b>229</b>
Callout.....	229
Graphics .....	230
Image .....	246
Plot.....	258
Tiling Plots .....	258
Colors .....	260
PlotFunction .....	282
PlotMesh .....	284
PlotPoint .....	284
PlotSurface.....	285
PlotVector.....	286
<b>Chapter 17 – GUI Classes .....</b>	<b>287</b>
Activity .....	287
Annotation .....	288
Button .....	289

ColorPicker .....	293
Control.....	295
DatePicker.....	298
ImageView.....	300
Label.....	301
MapView .....	301
Picker.....	306
Progress .....	308
SegmentedControl.....	310
Slider .....	313
Stepper .....	316
Switch.....	317
Table .....	319
TextField .....	323
TextView .....	326
WebView .....	328
<b>Appendix A – Error Messages .....</b>	<b>333</b>
Compiler Errors.....	333
Runtime Errors.....	337
<b>Appendix B – Character Sets .....</b>	<b>347</b>
The ASCII Character Set .....	347
<b>Index .....</b>	<b>349</b>



# Chapter 1 – Introducing techBASIC

Welcome to techBASIC! techBASIC is an implementation of BASIC for iOS devices designed specifically for collecting, analyzing and displaying numeric information. With techBASIC, you can collect information from the iPhone's accelerometer, process it with powerful array manipulation operations, and display the information with interactive graphs that respond to your touch and swipe gestures.

If you are new to techBASIC, this is not the manual to start with. Instead, start with the *Quick Start Guide to techBASIC*. Read that guide cover to cover; it will get you up and going on the program quickly.

This manual is a complete reference manual for techBASIC and techBASIC Sampler. This is the manual you will want to keep on your iPad or iPhone, or perhaps print a copy for your lab. It is laid out for easy reference, cataloging all of the features of the program in logically grouped sections. There is a comprehensive index to help find information, and lots of short sample programs to show how language features work. To get the most from this manual, flip through to get a feel for how it is organized, then read sections that cover topics that are interesting to you or unfamiliar to you. Later, as you have questions about how to do specific things with techBASIC, or need a sample to use as a framework for a new program you are writing, you can refer back to this manual.

---

## techBASIC vs. techBASIC Sampler

This manual covers both techBASIC and techBASIC Sampler. They are, for the most part, the same program. techBASIC is a paid program that gives you all of the power of a technical computing environment right away, while techBASIC Sampler is a free demo version that disables editing existing programs or creating new ones. You can upgrade techBASIC Sampler to have all of the capabilities of techBASIC at any time through an in-app purchase. For the rest of this manual, we'll refer to techBASIC, but except for editing, everything applies equally well to techBASIC Sampler.

---

## About the Manual

Chapters 2 and 3 cover the user interface for techBASIC. Chapter 2 covers the iPad, while Chapter 3 covers the iPhone and iPod. While the capabilities of techBASIC are the same on all of these platforms, the user interface varies a bit because of the differing screen sizes, so it is split into two chapters. If you have both platforms, start with the chapter that covers the platform you will use most often. You probably won't need to read the other chapter, although you might refer to it for a specific detail.

Chapters 4 through 12 cover the techBASIC programming language. If you are already familiar with one of the Microsoft implementations of BASIC, like Visual Basic or QuickBASIC, you should skip reading these chapters with one exception: read through the sections of Chapters 4, 7, 8 and 12 that cover arrays. techBASIC implements mathematical operations on arrays that are missing from the Microsoft implementations, and adds array constants, which are not implemented in the original implementations of BASIC or in ANSI Standard BASIC.

Chapters 13 through 17 cover the build-in classes used by techBASIC for many of its more interesting capabilities, in particular graphics and access to the sensors built into your iOS device. It is worth browsing through this chapter carefully, especially the sample programs.

---

## Typographical Conventions

BASIC is a case insensitive language, so you can enter keywords and identifiers using uppercase or lowercase letters. In this book, reserved words are always shown in uppercase. This is done to make it easier for you to pick out the reserved words, not because you need to use uppercase letters in your own programs.

Sample programs and short sections of programs are shown using Courier font, like this:

```
PRINT "Hello, world."
```

When elements of the BASIC language are used in the text, they use the same font.

Sections that describe a statement, function or method start with a model statement that shows the structure of the statement. These are shown in bold Courier, like this:

```
SELECT CASE expression  
[ CASE case-range [ ',' case-range ]* ]*  
[ CASE ELSE ]  
END SELECT
```

Uppercase reserved words must be entered exactly as shown, other than using lowercase if you prefer. Items in lowercase, such as **expression**, indicate places where you type something different—in this case, perhaps a variable name. Brackets indicate optional items. And when followed by an asterisk, indicate the item can be used zero or more times. For **SELECT CASE**, you can use as many **CASE** clauses as you like, but only one **CASE ELSE**. Items in quote marks, like the comma, indicate punctuation that must be typed exactly as shown.

As seen in the discussion, when a specific element from a model statement is discussed, it is shown in the same font as the model statement.

Model statements are precise, but may be difficult to get used to at first. All of the statements have sample programs that show them in actual use.

Changes from previous versions of the manual are marked with change bars in the margin, like this paragraph.

---

## Other Books and Reference Materials

This section lists a lot of books, but don't be intimidated by the list. You don't actually need any of them to use techBASIC, and very few people will ever use all of them. This list is here to give you some ideas for further exploration, not as a list of required books! There are also lots of great alternatives.

If you are new to BASIC, you will need to supplement this manual with a good beginner's book on the BASIC programming language. BASIC is a very common language, and lots of good books exist for it. Try to find one that covers general BASIC, not a specific implementation, like *Illustrating BASIC*.

If you already know some flavor of BASIC, start with the *Quick Start to techBASIC* books to get an overview, then refer back to this manual to fill in details that interest you.

There are also lots of great books that use the BASIC programming language for scientific computing, which, after all, is what BASIC was designed to do. A few, like the author, have been around for a while, and may only be available in libraries or as out of print books, but they are worth the effort to tack down.

### Building iPhone & iPad Electronic Projects

Mike Westerfield

O'Reilly, Sebastopol, CA 2013

While techBASIC is a general purpose programming language, capable of writing any type of program, it's optimized for technical programming. Key parts of technical programming are accessing sensors, communicating with TCP/IP, communications with Bluetooth LE, and using devices like HiJack.

This new book from O'Reilly Media covers all of these topics with fun projects like building a moisture meter with HiJack or collecting data from a model rocket flight. It's available now as a prerelease electronic book, and is due out in print form in September 2013.



Quick Start Guide to techBASIC

Mike Westerfield

Byte Works, Inc., Albuquerque, New Mexico, 2011

This short book gets people who already know a little BASIC up and running with techBASIC quickly. It is available in two versions, one for the iPad and one for the iPhone and iPod. Both are free, and available from the Byte Works web site at <http://www.byteworks.us>.

Illustrating BASIC

Donald G. Alcock

Cambridge University Press, 1977

A fun introduction to programming in BASIC, with an unusual format. Currently available on Amazon.com.

Celestial BASIC: Astronomy On Your Computer

Eric Burgess

Sybex, Berkeley, CA, 1982

This is one of my favorite programming books of all time. If you're at all interested in astronomy, it's worth the effort to find a copy of this classic book. It has a great collection of simple BASIC programs that perform a wide variety of calculations, like planet positions, moon phases, dates for Easter, and so forth.

BASIC Computer Games

David H. Ahl, Ed.

Workman Publishing, New York, 1978

An amazingly diverse collection of short BASIC games. This old book is worth chasing down, too.



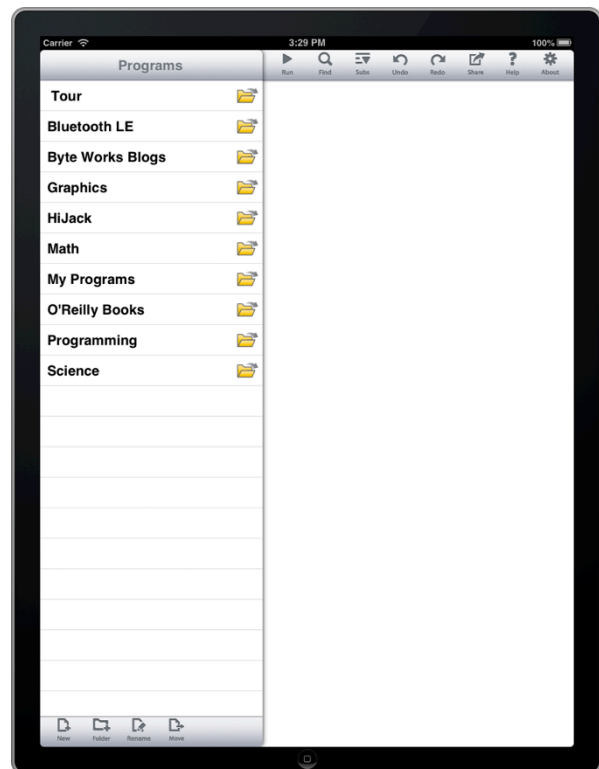
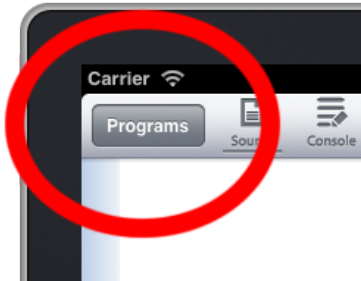
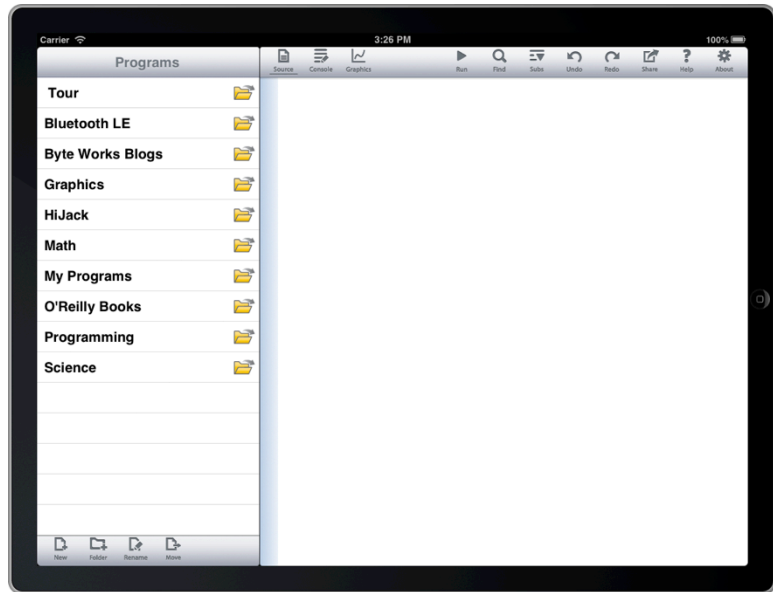
## Chapter 2 – The iPad User Interface

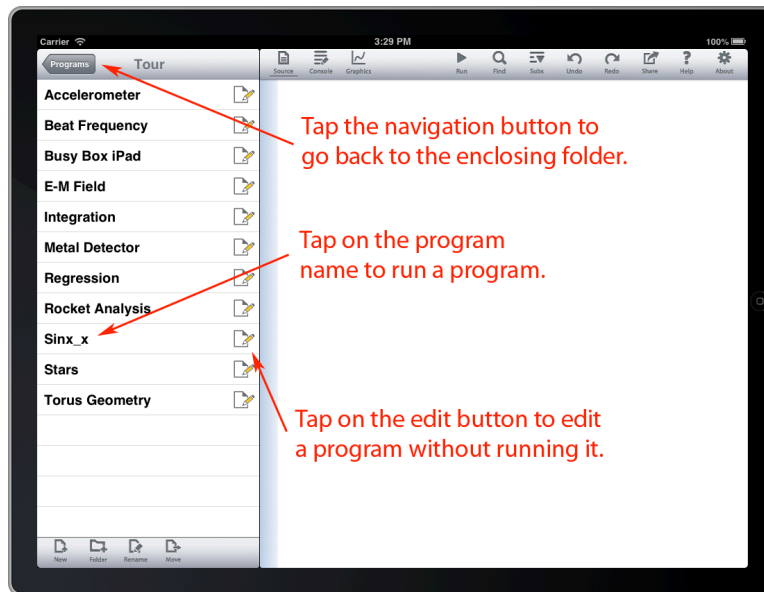
### Managing Programs

Sample programs, programs you write, and programs that you import from other sources are all listed in the Programs view. They are organized in folders, also shown in the Programs view. In landscape mode, this list of programs is shown on the left side of the screen. In portrait mode, the programs are in a drop-down view accessible by pressing the Programs button. Tapping outside of the view will dismiss the view.

### Navigation in Folders

Tap the name of a folder or the folder button to open a folder. Once a folder is open, the folder name will appear in a button at the top left of the program list. Tap this button to close the current folder, moving up a level to the folder that contains it.





---

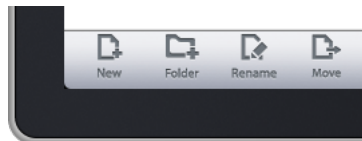
### Editing Programs

Tap the Edit button to the right of the program to see the source code for the program. The source code is displayed in the Source view.

---

### Running Programs

Tap the name of a program to run the program. For example, tap Sinx\_x to run the program, which will shift the view to the Graphics view to show a plot.

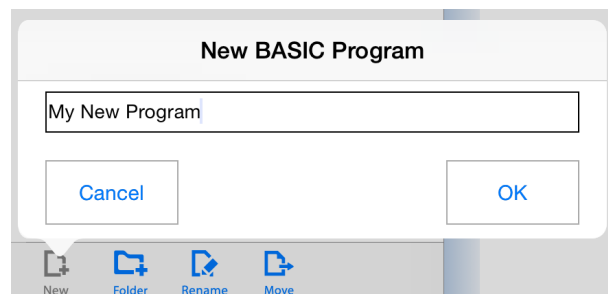


---

### Creating New Programs

Tap the New button at the bottom of the Programs view to create a new program. A dialog appears asking for the name of the new program. Enter the name and tap OK. The Source view is shown with a blank screen, ready for program entry.

See Editing Programs, later in this chapter, for features related to editing new and existing programs.



## Deleting and Renaming Programs

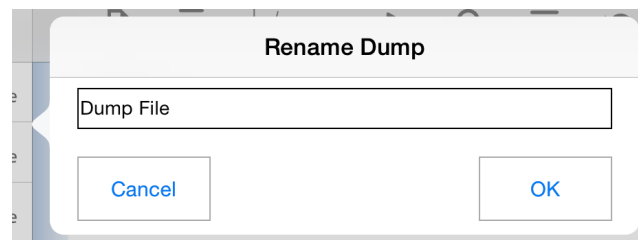
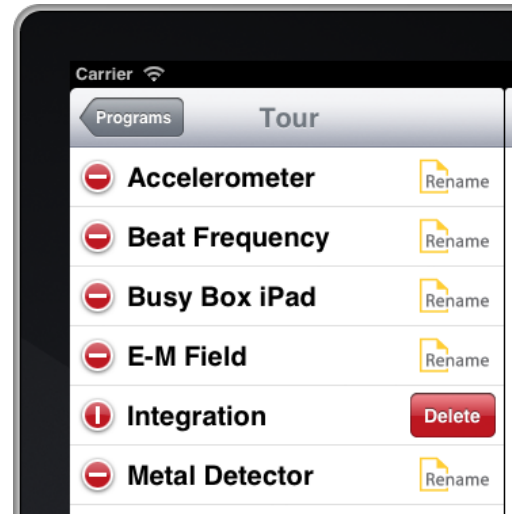
Tap the Rename button at the bottom of the Programs view to delete or rename an existing program. This changes the program view as shown to the right, adding delete dials and rename buttons.

Tapping a delete dial rotates it sideways and presents a Delete button to the right. Tapping that button deletes the program. There is no way to undo deleting a program, so be sure you really want to delete it before confirming the delete.

Deleting sample programs is a special case. Deleting a sample program definitely deletes the sample, but it can be recovered. See *Preferences*, later in this chapter, for an option to recover deleted or edited samples.

There is also a shortcut for deleting a program. Before you even tap the Edit button, swipe left across the program you want to delete. This presents the Delete button right away, skipping the steps of tapping Edit and the delete dial.

The other option while editing is to rename a program or folder. Tap the Rename button, and you will see a dialog showing the current name of the program. Change the name to whatever name you prefer and tap OK to rename it. Programs are always alphabetized, so look for the program in its new location.



Press the Done button that replaced the Edit button to return to the normal Programs view.

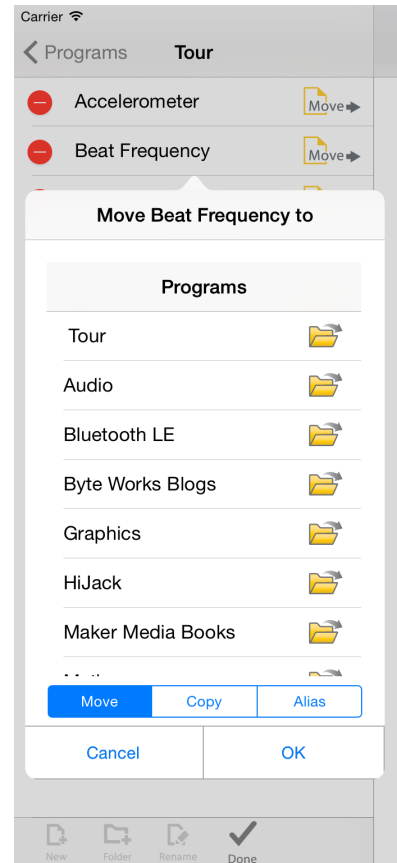
## Moving Programs

Tapping the Move button changes the program list to expose a Move button, as well as the same Delete button available when renaming programs and folders. Tapping one of the Move buttons to the right of a program name brings up a dialog allowing the program to be moved, copied, or aliased.

Start by selecting the new location for the program by navigating to the proper folder in the Programs list. This works just like navigating folders in the standard Programs list, but only folders are listed.

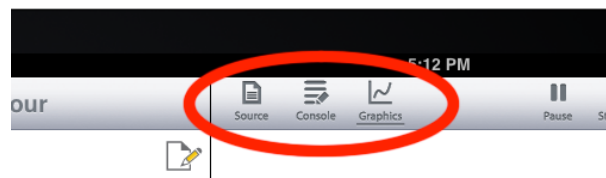
Select Move, Copy or Append. Move will move the file from its current location to the new one. Copy creates a new working copy of the program, while Alias creates an alias of the original. They look similar, but there is a key difference. A copy of a program is a new instance of the program. Changing the program does not change the original. An alias is just a name for the original program in a new place. Changing the alias will change the original program, too. Aliases are used extensively in the sample programs, allowing the same program to appear in different topical folders. For example, the Accelerometer appears in the Tour, O'Reilly Books and Science folders. Two of these are aliases, so the program itself only has to exist in one place.

Tap OK to move, copy or alias the program, or Cancel to exit without making a change.



## Changing Views

There are three views available. These are sometimes selected automatically based on other actions, and sometimes selected due to commands in the program. You can also use one of the three view selection buttons to select a view at any time. The current view is marked with an underline.



The Source view shows the currently selected program. techBASIC will switch to the source view whenever you tap an Edit button to edit a program or when the debugger hits a breakpoint and needs to show you where the breakpoint occurred. The source view is covered in *Editing Programs*, right after this section.

The Console view looks a lot like the Source view, but it is actually the place where techBASIC writes text programs print using the PRINT statement, as well as the place you can type responses that are read with the input statements, line LINE INPUT. techBASIC switches to the Console view when a program starts to run, and also under program control when the program executed a

```
system.showConsole
```

call. The Console view is discussed in *Text Input and Output on the Console*, later in this chapter.

The Graphics view is an interactive display showing the plots and graphics created by a program. techBASIC does not switch to the Graphics view automatically, but many of the sample programs do. They switch to the Graphics view using the

```
system.showGraphics
```

call. The Graphics view is covered in *Graphics Output*, later in this chapter.

---

## Editing Programs

With the exception of horizontal scrolling, editing programs works just like editing text in other iPhone applications, so it won't be covered in detail here. Tap on the source code to begin editing, and use the Done button to dismiss the keyboard when editing is complete. There are a few things worth mentioning, though.

While the electronic keyboard that is always available on the iPad is perfectly adequate for many tasks, if you will be writing or editing long programs on the iPad, a physical keyboard is a great accessory. A physical keyboard allows you to move about in the program using arrow keys, select text by pressing shift and using a movement key, and copy and paste with the Command-C and Command-P keyboard shortcuts. Here is a complete list of supported keyboard shortcuts:

⌘C	Copy
⌘V	Paste
⌘X	Cut
⌘F	Open the Find dialog
⌘G	Find the next occurrence of the find string (the Find dialog is not opened)
⌘]	Indent (adds two spaces to the start of each selected line)
⌘[	Outdent (removes up to two spaces from the start of each selected line)
⌘→	Move to the end of the line
⌘←	Move to the start of the line
⌘↑	Move to the start of the file
⌘↓	Move to the end of the file
⌘-shift→	Select to the end of the line
⌘-shift←	Select to the start of the line
⌘-shift↑	Select to the start of the file
⌘-shift↓	Select to the end of the file
option→	Move to the end of the word or number
option←	Move to the start of the word or number
option↑	Move to the start of the line, or the previous line if at the end of a line
option↓	Move to the end of the line, or the next line if at the end of a line
option-shift→	Select to the end of the word or number
option-shift←	Select to the start of the word or number
option-shift↑	Select to the start of the line, or the previous line if at the end of a line
option-shift↓	Select to the end of the line, or the next line if at the end of a line

If you are using the software keyboard, techBASIC tries to help out with a code completion bar that appears right over the standard keyboard. This context sensitive bar looks at the program as you type and tries to predict what you might type next. If you see a helpful symbol, tap it to place the symbol in your file. You can turn this feature off in preferences.

Finally, programs are line oriented. Wrapping lines is generally unnatural. For that reason, each program line appears unwrapped. Scroll horizontally to see parts of the line that are not visible.

## Find

The Find button is available anytime the Source view is selected. Tapping the Find button brings up a search and replace dialog like the one shown.

Enter the text you want to locate in the Find text edit box. If you want to replace this text with something else, rather than just locate it, enter the replace string in the Replace text edit box.

The three switches let you turn the search options on or off.

The search always starts at the location you were editing when Find was tapped. If the search reaches the end of the program without finding the target string, Wrap Around allows the search to jump to the top of the program and start over.

BASIC is a case insensitive language. Still, there are times when you may want to search for text with a specific letter case. Turn on the Case Sensitive option to require matching not only the letters in the Find string, but also the case of the letters.

It is frequently useful to find just the places where a short variable name is used, perhaps an index variable named `i`, but not to find everywhere where the letter `i` is used. The Whole Word option helps. It causes the search to ignore the Find string unless it starts and ends with a character that is not an alphabetic letter or number. For example, searching for `i` in the text

```
FOR i = 1 TO 10
  i4 = i*4
  PRINT i4
NEXT
```

will find the highlighted text, but not the letter `i` in the identifier `i4` or in the `PRINT` token.

With the desired Find and Replace strings entered, and the proper options selected, tapping the Find button searches for the next occurrence of the Find string. Tapping it again will continue to the next occurrence, and so on.

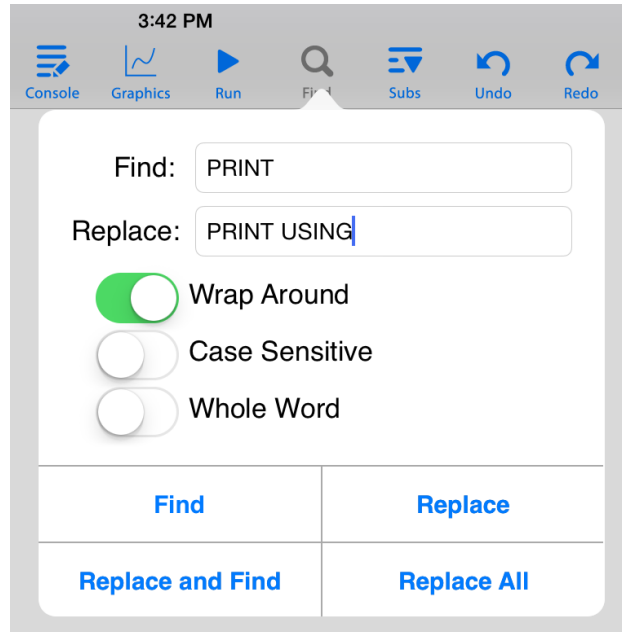
Tapping Replace replaces whatever text is selected—even if it is not the Find string—with the Replace string.

Tapping Replace and Find replaces the current selection with the Replace string, then find the next occurrence of the Find string after the replaced text. This option, in combination with Find, is great for stepping through a program, replacing some but not all of the occurrences of the Find string.

Replace All begins searching from the top of the program, replacing each occurrence of the Find string with the Replace string.

Tap off of the dialog to dismiss it.

If you are using a hardware keyboard, you can continue to find the last string you entered using command-G.





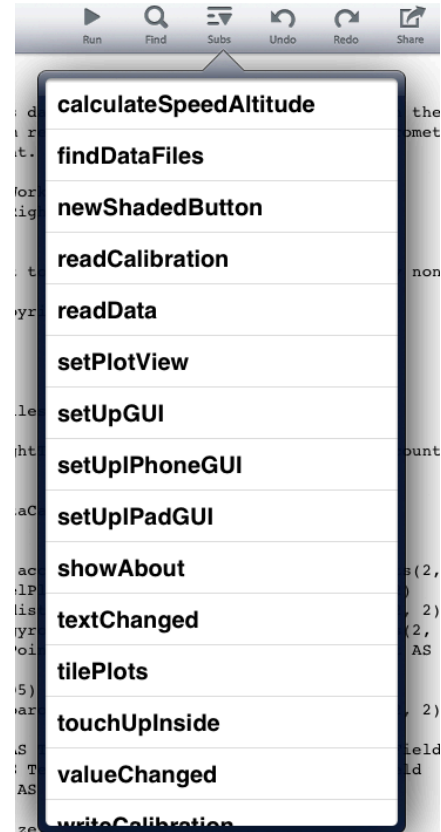
## Subs

The Subs button displays a list of all of the subroutines and functions in the current program. The image shows the subroutines in the Rocket Analysis sample from the Tour folder. Tapping on a program will select the line containing the program and scroll the display to the program. It's a great way to rapidly jump to a specific location in longer programs.

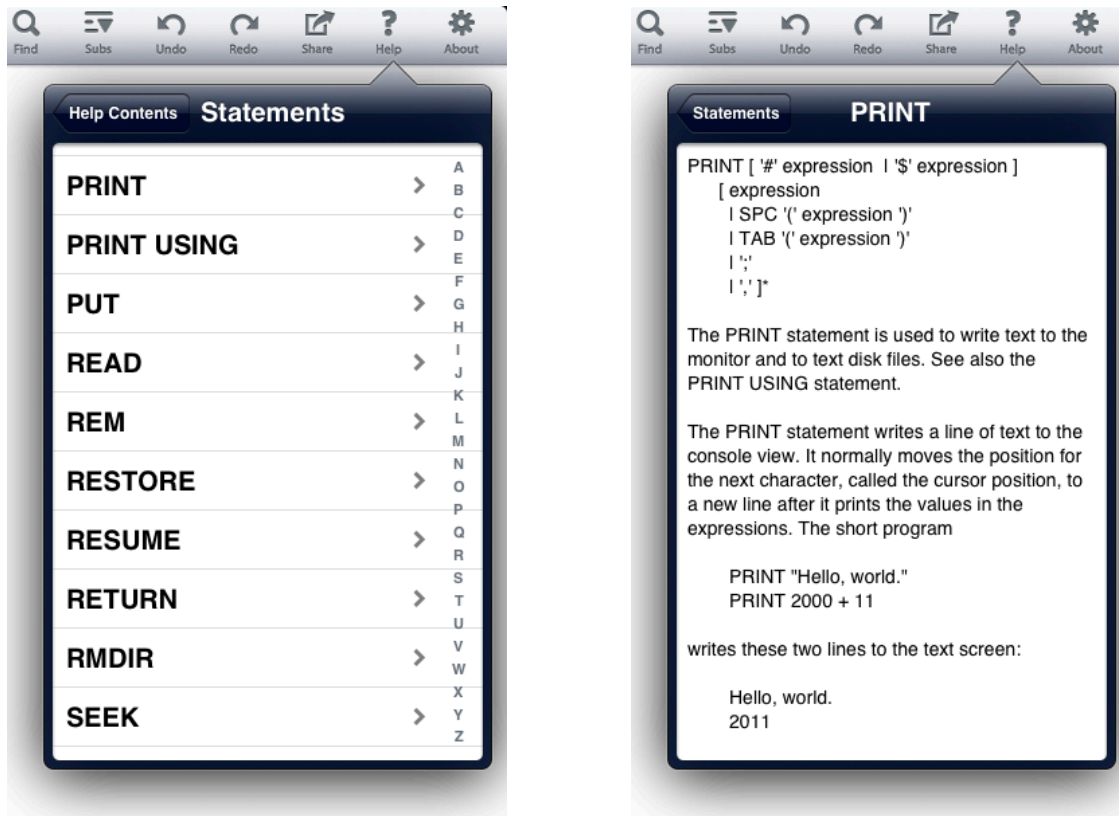
## Undo and Redo

Tap on these buttons to undo or redo changes to the program.

There is no limit, other than memory, on the number of levels of Undo and Redo. Dismissing the keyboard does, however, empty the undo buffer.



## Using Help



Even programmers who use a language forty (or more!) hours a week don't remember every command, or every detail about every command, in the languages they use. The Help facility gives you all the technical details about every statement, function and class in techBASIC. Tap on a topic to expand on the topic, or use the back button at the top of the dialog to move back a level.

While Help has listings for all of the language, it does not include figures, samples or information about the user interface. Help is a great quick reference while writing a program, but it's not a bad idea to keep a copy of this manual in iBooks for situations where more detail is needed.

## Program Output

### Text Input and Output on the Console

PRINT and other output statements write text to the Console view or to files stored in the iPad's memory. When text is written to the Console view, it is displayed in a scrollable, but not editable, text view. New text is always placed at the end of existing text.

Statements like `LINE INPUT` let you type text into a running program. Any prompt is placed at the end of the text in the console, then the program waits for you to type a response and press the return key.

As an example, try the program in the snippet. It prints a prompt asking for a starting number; enter an integer value. It then asks for an ending number. Enter another integer, but don't make it too much bigger than the first. All of the integers from the first to the last are printed.

If you don't follow instructions well, and the program is still printing numbers, press the Stop button that shows up while the program runs. The Stop button is covered later in *Stopping a Program*.

Snippet

```

INPUT "Enter a starting number: "; first
INPUT "Enter an ending number: "; last
FOR i = first TO last
  PRINT i
NEXT

```

There is no fixed limit to the number of characters that can appear in the Console view, but it will become sluggish with too many characters. The Clear command shows up while in the Console view. Tap this button to delete all text in the console view.

The command `System.clearConsole` can be used in programs to do the same thing.

## Graphics Output

The output from programs that draw or plot functions appears on the Graphics view. Plots, incidentally, don't show up until the program completes. That doesn't mean that the plot is static, though!

To experiment with these commands, run the Beat Frequency sample from the Tour folder. The plot from that sample is shown here.

The swipe gesture is a natural way to move graphics around on the screen. Swipe in any direction, and the entire plot will move along with your finger.

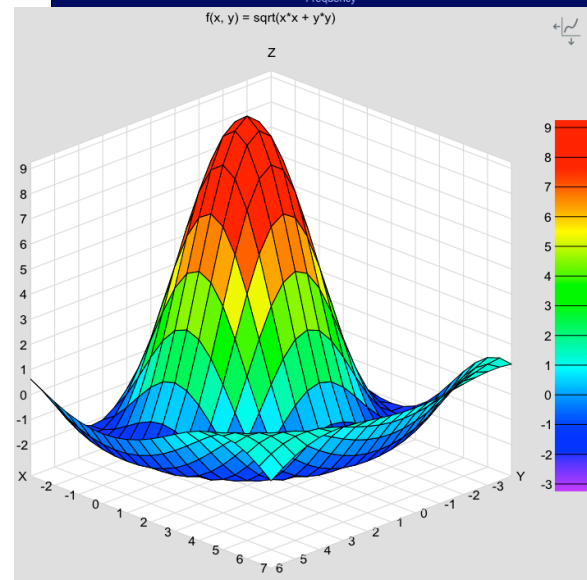
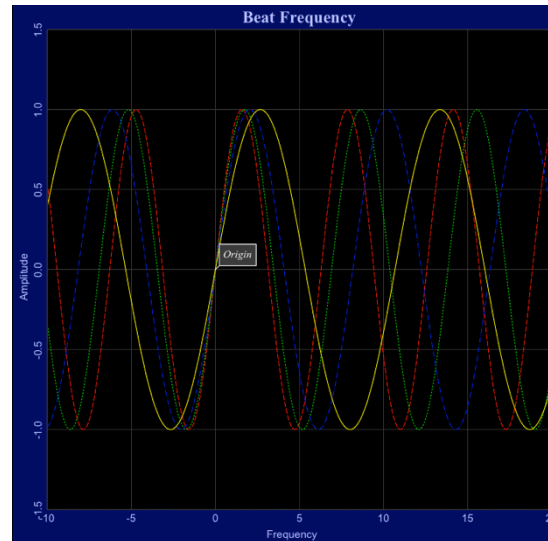
Pinching zooms or shrinks the image. There are actually three different pinch effects, depending on the angle of the pinch. If you pinch vertically, the Y-axis shrinks or expands, but the X-axis is unchanged. Pinching horizontally does just the opposite, shrinking or expanding just the X-axis. Pinching along the diagonal shrinks and expands both axis, maintaining the current aspect ratio.

Looking at the program, you can see that techBASIC is not plotting specific points like some other programs, it is actually plotting a function. This allows the program to recompute the values for the plot as you zoom in. As a result, the level of detail available is essentially unlimited—you can zoom in on a detail as close as you like.

The small text box marking the origin is a callout. Callouts let you label portions of the plot; they move right along with the plot as you swipe and pinch. You can also create new callouts at any time by tapping. If your tap is within about one finger width of a plotted line, the callout will snap to the closest point on the line. Callouts created interactively generally show the coordinates of the point tapped, but it is possible to provide callouts with specific text. Two examples are presented in Chapter 16, when the `newPlot` method is described. One of these examples is the Stars sample. If you run this sample and tap on a star, the star's name is displayed.

To experiment with three-dimensional plots, run the `Sinx_x` sample from the Tour folder. You should see a plot like the one shown here.

Three-dimensional plots present a unique problem for panning and zooming. Do you want to pan and zoom the data on the axis, like we did for two-dimensional plots, or the axis itself, so you can zoom in on the entire function?



When the program starts, techBASIC will actually pan and zoom the function, not the axis. Unlike a two-dimensional plot, though, we want to pan our three-dimensional function along any of the three axis, but there are only two axis on the screen. To pan along a specific axis, use a wipe gesture parallel to the axis. A vertical swipe, for example, will move the function up and down, while a swipe parallel to the X axis scoots it back and forth along that axis. Pinches are similar to two-dimensional pinches. A vertical pinch changes the range of the function, adjusting Z in this case. Horizontal pinches change the domain, expanding or shrinking the X-Y plane while maintaining the original aspect ratio. A diagonal pinch changes all of the axis at the same time.

The button at the top right of the plot is used to change the behavior of the pinch and swipe gestures. The button shows arrows moving the axis; tap the button, and that's what will happen. The button will also change appearance to remind you that you can switch back to manipulating the function by tapping it again.

With the mode changed, try swiping the plot again. This time, the axis moves in the direction of the swipe. Pinching in any direction changes the size of the axis.

Three-dimensional plots can also be rotated. There are two basic rotation gestures.

First, to twirl the plot about an axis pointed out of the screen, place two fingers on the screen like you are about to pinch, but rotate instead. The plot will twist around the origin of the axis.

You can also twist about an axis on the screen by using two fingers to swipe. Imagine the plot inside a clear track ball. This rotation works as if you swiped the track ball, spinning it about its axis.

Manipulating functions this way is a powerful way to understand the geometry of the function, but this is also an amazing visualization tool. Run the Stars sample, which displays most of the stars within 10 parsecs of our sun. The static display is interesting, but the lack of three dimensions limits the value considerably. Now use a two-finger swipe to rotate the sphere of stars. The motion of rotating the stars allows the relative positions to jump right out of the two-dimensional screen.

---

## Debugging Programs

There is a run button on the button bar that can run the currently selected program. While the program is running, this button is replaced by a series of buttons. These are used to pause, cancel or debug the running program. This section describes the function of each of these buttons, as well as the stack and variable display used to understand the program as it runs.




---

## Setting and Clearing Breakpoints

Setting a breakpoint lets you stop a program at a specific location, examining the current values for variables and watching them change as you step line-by-line through the program. The blue bar to the left of the Source view is the breakpoint bar. Tap on the breakpoint bar to set a breakpoint; tap again to clear it. A line with a breakpoint displays a blue marker like the one shown here.

```
DIM p AS Plot
p = graphics.newPlot
p.setTitle("f(x, y) = 10
p.setGridColor(0.85, 0.8
p.setsurfacestyle(3)
p.setAxisStyle(5)
```

With a breakpoint set, running the program will stop it at the first breakpoint encountered.

---

## Stepping Through Programs

Once a program is stopped at a breakpoint or with the Pause button, you can step through the program one line at a time to see the program flow. The currently executing line is marked by a gold triangle, like the one seen here after stepping two lines past the breakpoint.

Step Over steps one line at a time, even if the line being executed calls a subroutine or function that executes many lines before returning.

```
DIM p AS Plot
p = graphics.newPlot
p.setTitle("f(x, y) = 10
p.setGridColor(0.85, 0.8
p.setsurfacestyle(3)
p.setAxisStyle(5)
```

Step In works just like Step Over on lines that don't call a subroutine or function, but if the line does make a call, execution pauses at the first line of the subroutine or function.

Step Out is generally used from inside a subroutine or function. The program runs at full speed until the subroutine or function ends, then pauses at the first line encountered after returning. If Step Out is used from the main program, the program runs at full speed to completion.

In all cases, the program will stop if a breakpoint is encountered. For example, stepping over a subroutine with a breakpoint will still stop at the breakpoint.

## Running and Pausing Programs

The Run button can be used to run the currently selected program. While the program is running, a pause button replaces the run button. The program may complete so quickly that the pause button is not noticed, but on longer programs or programs in an infinite loop, tapping the Pause button stops the program as if the next executing line had a breakpoint.

Once a program is stopped, either by tapping Pause or by encountering a breakpoint, tapping the Run button causes the program to continue execution from the current location. It will run at full speed until the Pause button is pressed, a breakpoint is encountered, or the program completes.

## Stopping a Program

Use the Stop button to immediately stop execution of a running program. This works whether the program is stopped by the Pause button or a breakpoint, or running at full speed. An error message will be displayed at the line that was executing when the Stop button was pressed.

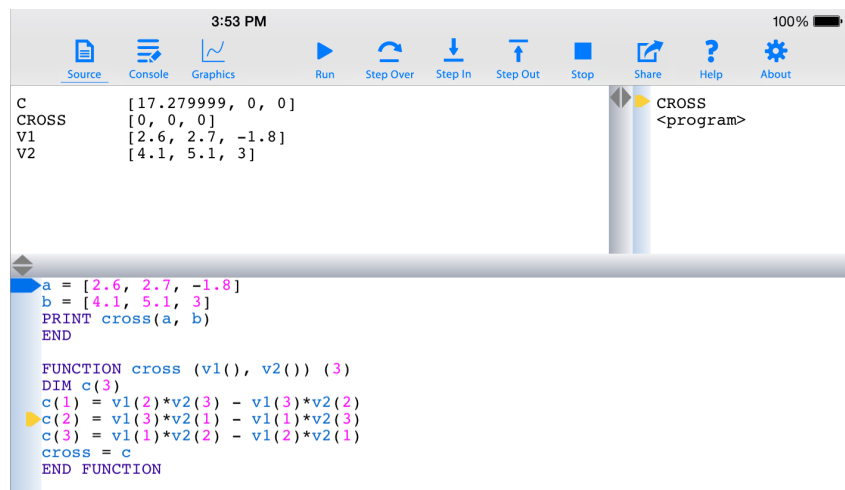
## The Variable View

When a program is stopped, two new subviews show up in the Source view. The view at the top left is the Variable view. It shows all of the variables in the current program scope; that is, all of the ones in the current subroutine, function, or the main program.

In the sample shown here, the program is currently in a function called `cross`, computing the cross product of two vectors. There are four variables in the function. The first is `C`, a matrix used to hold the result inside the function. The first element of `C` has been set to 17.279999, while the last two have yet to be changed from the initial values of 0. `CROSS` is the name of the function itself, showing the value that will be returned to the caller. The last two variables are the parameters that were passed to the function. These values change as the program steps.

There are times when the variable view is not wide enough to show the complete value of a variable. There are two ways to deal with this. The first is to slide the gray divider to the right, increasing the width of the variable view. The second is to tap the variable, which brings up a detail window showing the full value of the variable. This still may not provide enough space for a really large array or very long string, but you can scroll the value in the detail view to see the rest of the value.

The Variable view can be scrolled to show additional variables if there are too many to show at one time. Dragging the horizontal gray divider will also increase or decrease the height of the Variable view as needed.



## The Stack View

The Stack view appears immediately to the right of the Variable view. It shows the current call stack. The program itself always appears at the bottom of the list. Names of new subroutines and functions are pushed onto the top of the stack as they are called, and popped off of the list as they return.

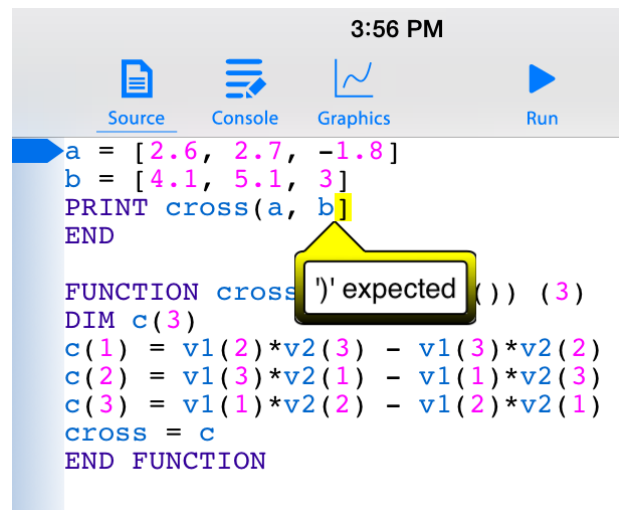
The gold triangle points to the current scope. The current scope is the one whose variables are shown in the Variable view, not necessarily the one that is executing. Tap the name of a scope to change the current scope. For example, tapping <program> in the sample views shown will change the scope to the program, displaying the values of A and B.

## Understanding Errors

When techBASIC detects an error, either during compilation or while running the program, the view shifts to the Source view and the error is shown. The approximate location of the error is highlighted in yellow, and an error message explains the problem techBASIC detected. In this case, the array constant was closed with a ) character rather than the expected ] character.

Tap the error message to dismiss it. The error is still highlighted in yellow until the program is edited. You can recall the error message by tapping on the highlighted area.

techBASIC error messages are listed in Appendix A. Each error message is accompanied by a brief description of the error. Common causes for the error and possible remedies are discussed when these may not be obvious from the error itself.



## Sharing and Syncing

### Sending Graphics to Photo

You can share any of the images on your graphics screen by tapping the Share button. This sends a copy of whatever you see in the Graphics view to the Photo, where you can save the image, show it to others, or e-mail it.

The resolution of the image is generally not the same as what you see on your screen. The image will be about 1 megapixel by default. You can change the image size using the techBASIC preferences.

### Sending Programs and Text to Mail

You can share program source and anything in the Console view by tapping the Share button. This creates a new email message containing the source code or console text, ready for the user to address and send.

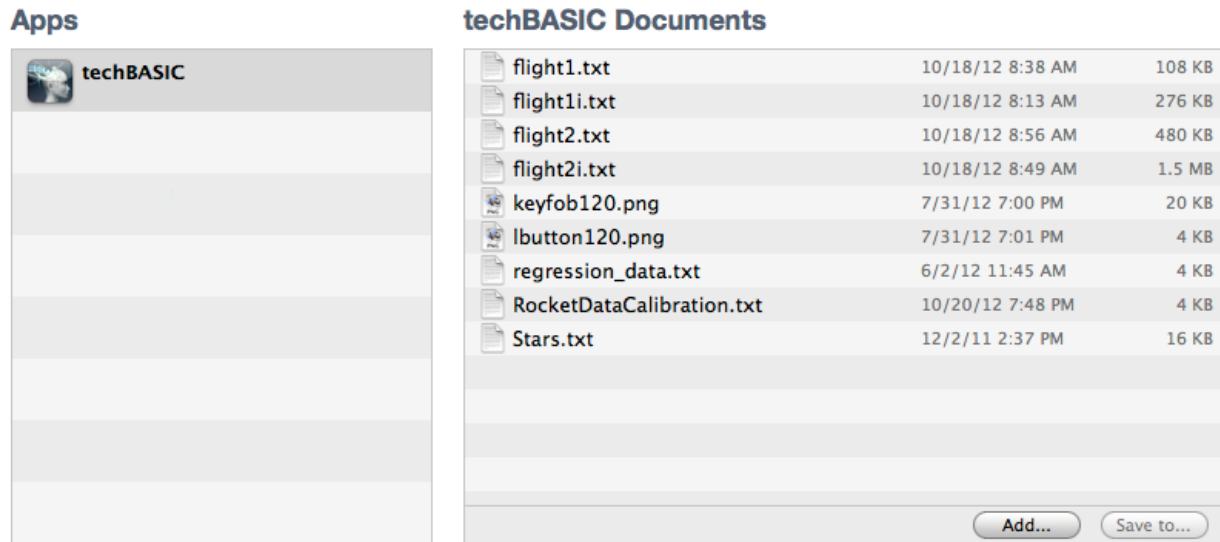
## Moving Programs and Data To Your Desktop Computer

There are many reasons to want to move a program or data from your iOS device to your desktop computer. You may want to share data with a colleague, use a program in another implementation of BASIC, or perhaps just edit a long program in a traditional keyboard and mouse environment.

Data files are stored in an area called the sandbox. iTunes can be used to move these files to and from your desktop computer. To move a data file, start iTunes and select your iPad from the list of devices. You will see a list of tabs across the top of the device window; select the tab named Apps. Near the bottom of the window, you will see a list of applications that can share data. Select techBASIC, and you will see a list of all data files.

## File Sharing

The apps listed below can transfer documents between your iPhone and this computer.



Data files are created by your programs or dragged into the sandbox using iTunes. Presumably you know what is in the file and what other programs can edit the file. If you are unsure of the contents of a file, use the Dump sample in techBASIC to view the contents, or check the techBASIC reference manual for details on file formats.

With the documents window open, the easiest way to move files to and from the iPad sandbox is to drag the file in or out of the document area. Traditional Open and Save dialogs are also available from the Add... and Save to... buttons. To remove a file, select the file and press the delete key on your keyboard.

techBASIC source files are not stored in the sandbox. This is due to a restriction placed on programs not written by Apple or by Apple subsidiaries. Except for Apple programs, no program is allowed to move executable code to and from an iOS device using iTunes. As a result, moving programs to and from techBASIC is considerably more cumbersome than moving data files.

To move a program from another device to your iPad:

- Place the source code in an otherwise blank email and send it to yourself.
- Read the email on the iPad.
- Tap twice on the email, then tap the Select All button to select all of the text.
- Tap the Copy button to copy the program.
- From techBASIC, tap the New button on the Programs list and enter a new program name.
- Paste the source into the new program.

To move a program from your iPad to another location:

- Display the source code by tapping the Source button.
- Tap the Share button. This places the contents of the source file in an email that you can send to yourself.
- Read the email on any other device and copy and paste the source to save it to the new device.



We apologize for the difficulty in performing this seemingly simple task. We will immediately add the ability to move programs to and from an iPad using iTunes (or through any other means) should Apple ever allow us to do so.

---

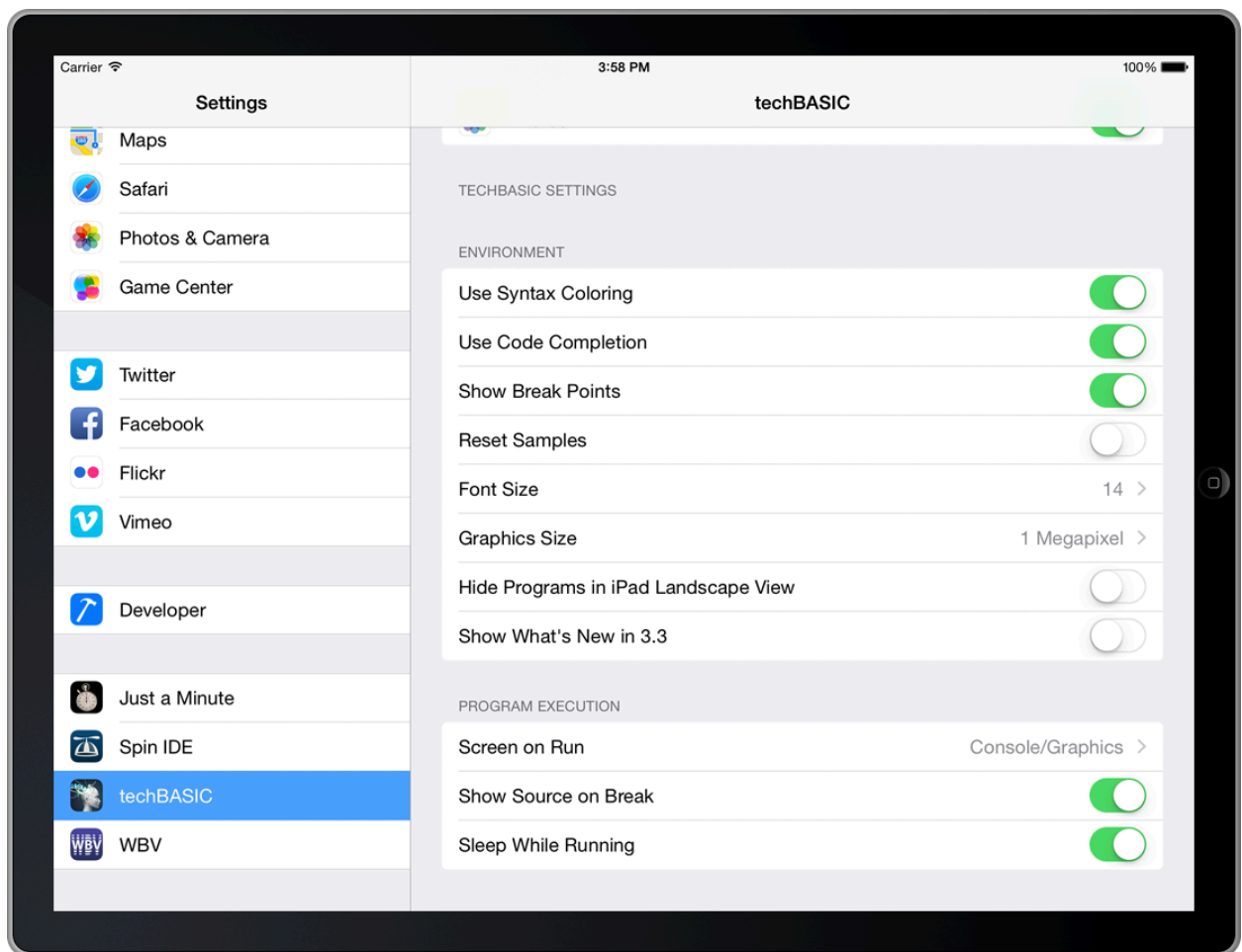
## About

The about button displays the current techBASIC version, copyright information, and has links to other programs from the Byte Works.

---

## Preferences

There are several preferences that can be accessed from the iPad Settings app. Select techBASIC from the list of apps and you will see a display like this one:



Use Syntax Coloring turns syntax coloring on and off. When off, all text is painted in black. The text will render and scroll a little quicker, although this should not normally be noticeable.

Use Code Completion turns the code completion keyboard input accessory on or off. When on, techBASIC will examine the program as you type and suggest words and symbols that might appear next in the program, reducing typing time.



Show Break Points turns the light blue breakpoint bar on and off. While hiding the breakpoint bar leaves you with no way to set or clear breakpoints, it also gives a little more room to see and edit programs.

You can change and delete the sample programs that come with techBASIC. Turn on Reset Samples to get the originals back. This will revoke any changes you have made, and restore any deleted samples. This only happens once, though, so you still have the ability to change and delete the restored samples. If you would like to use a sample as a basis for a new program, rename it. The renamed sample will not be overwritten when the samples are reset, and the original sample will be restored. The samples are only reset after techBASIC is shut down completely and restarted.

The console and source screens default to a font size of 14 points. Use this option to change the font size. The font size will only change after techBASIC is shut down and restarted.

Graphics size controls the size of graphics images exported to the Photo app by the Share button. The image size can be set to the screen size or to one, two, three or four megapixels.

The list of programs is normally shown all of the time in landscape view, while in portrait view it goes away when not in use. Use the Hide Programs in iPad Landscape View option to hide the program list when it is not in use in landscape view, too. This gives more room when editing or running programs.

A dialog appears when you start techBASIC that shows the changes since the last version. You can turn it off from a button in the dialog or from the Show What's New in 3.3 preference. You can also turn it back on with the preference.

Running a program normally shifts to the Console view if you are in the Source view, and stays on the Graphics view if that view is visible. Use the Screen on Run option to move to a specific view when a program starts to run. Remember that methods in the `System` class this can also control the view. These shift the view after it is initially set by techBASIC, and always override this setting.

Disable Show Source on Break to remain in the current view when a breakpoint is hit. You can step and trace from any view, and it may be more important to see what the program is printing, for example, than which line is executing.

Your iPad will sleep to preserve the battery if there is no interaction. Sleeping also pauses any program that is running, which may be a problem for a program that is collecting and processing data. Use the Sleep While Running option to disable sleep while a program is running. It is a good idea to have the iPad hooked up to a power source when this option is used for any length of time, especially if sensors like the accelerometer or location are being used.



## Chapter 3 – The iPhone and iPod User Interface

This chapter covers both the iPhone and iPod user interface. There really is no difference between the two from the standpoint of techBASIC. To save a lot of redundancy, both the iPod and iPhone are referred to as the iPhone for the rest of the chapter.

---

### Changing Views

There are five views available. These are sometimes selected automatically based on other actions, and sometimes selected due to commands in the program. You can also use one of the five view selection buttons on the tab bar to select a view at any time.

The Programs view shows your programs and provides buttons to edit the list of programs.

The Source view shows the currently selected program. techBASIC will switch to the source view whenever you tap a detailed disclosure key to edit a program or when the debugger hits a breakpoint and needs to show you where the breakpoint occurred. The source view is covered in *Editing Programs*.

The Console view looks a lot like the Source view, but it is actually the place where techBASIC writes text programs print using the `PRINT` statement, as well as the place you can type responses that are read with the input statements, line `LINE INPUT`. techBASIC switches to the Console view when a program starts to run, and also under program control when the program executed a

```
system.showConsole
```

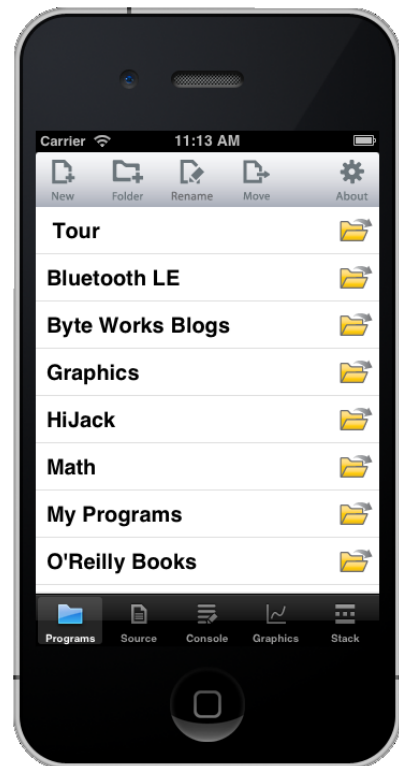
call. The Console view is discussed in *Text Input and Output on the Console*, later in this chapter.

The Graphics view is an interactive display showing the plots and graphics created by a program. techBASIC does not switch to the Graphics view automatically, but many of the sample programs do. They switch to the Graphics view using the

```
system.showGraphics
```

call. The Graphics view is covered in *Graphics Output*, later in this chapter.

The Stack view is used when debugging programs to see the call stack and view variable values. This is covered in *Debugging Programs*.




---

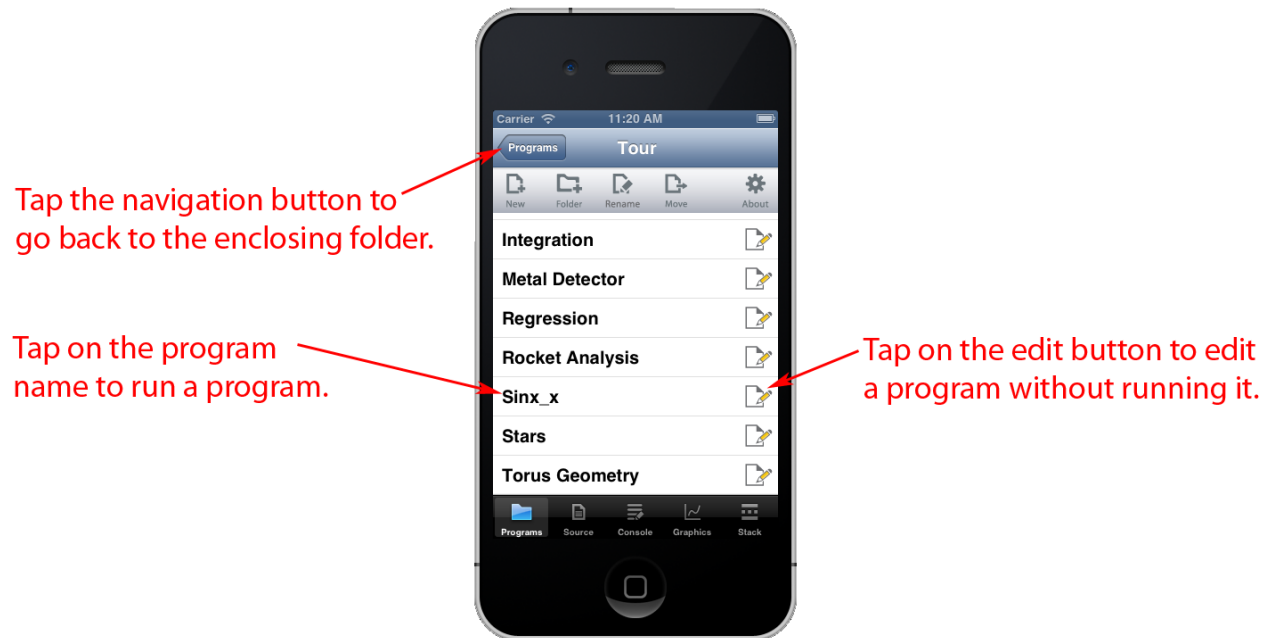
### Managing Programs

Sample programs, programs you write, and programs that you import from other sources are all listed in the Programs view. They are organized in folders, also shown in the Programs view.

---

## Navigation in Folders

Tap the name of a folder or the folder button to open a folder. Once a folder is open, the folder name will appear in a button at the top left of the program list. Tap this button to close the current folder, moving up a level to the folder that contains it.



---

## Editing Programs

Tap the Edit button to the right of the program to see the source code for the program. The source code is displayed in the Source view.

---

## Running Programs

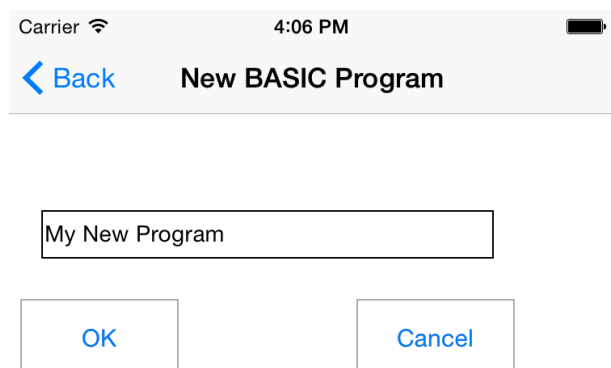
Tap the name of a program to run the program. For example, tap Sinx\_x to run the program, which will shift the view to the Graphics view to show a plot.

---

## Creating New Programs

Tap the New button at the top of the Programs view to create a new program. A view appears asking for the name of the new program. Enter the name and tap OK. The Source view is shown with a blank screen, ready for program entry.

See Editing Programs, later in this chapter, for features related to editing new and existing programs.



## Deleting and Renaming Programs

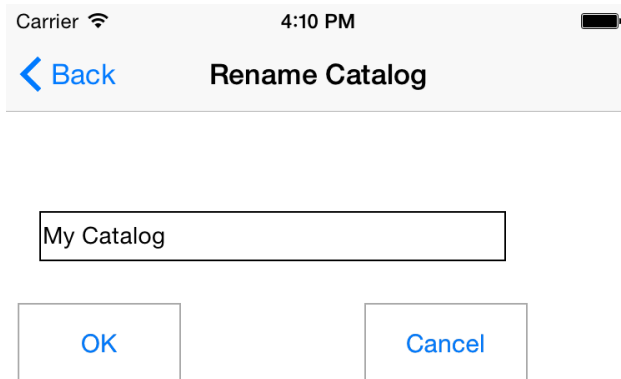
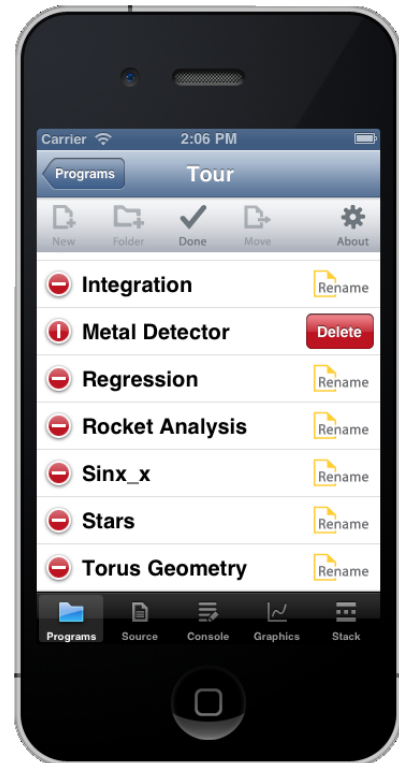
Tap the Rename button at the top of the Programs view to delete or rename an existing program. This changes the program view as shown to the right, adding delete dials and rename buttons.

Tapping a delete dial rotates it sideways and presents a Delete button to the right. Tapping that button deletes the program. There is no way to undo deleting a program, so be sure you really want to delete it before confirming the delete.

Deleting sample programs is a special case. Deleting a sample program definitely deletes the sample, but it can be recovered. See *Preferences*, later in this chapter, for an option to recover deleted or edited samples.

There is also a shortcut for deleting a program. Before you even tap the Edit button, swipe across the program you want to delete. This presents the Delete button right away, skipping the steps of tapping Edit and the delete dial.

Tapping the Rename button switches to a new view that allows a new name to be entered. Change the name to whatever name you prefer and tap OK to rename it. Programs are always alphabetized, so look for the program in its new location.

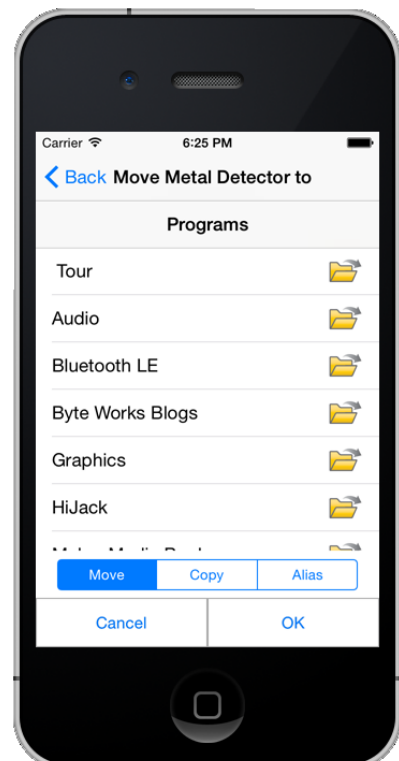


Press the Done button that replaced the Edit button to return to the normal Programs view.

## Moving Programs

Tapping the Move button changes the program list to expose a Move button, as well as the same Delete button available when renaming programs and folders. Tapping one of the Move buttons to the right of a program name brings up a view allowing the program to be moved, copied, or aliased.

Start by selecting the new location for the program by navigating to the proper folder in the Programs list. This works just like navigating folders in the standard Programs list, but only folders are listed.



Select Move, Copy or Append. Move will move the file from its current location to the new one. Copy creates a new working copy of the program, while Alias creates an alias of the original. They look similar, but there is a key difference. A copy of a program is a new instance of the program. Changing the program does not change the original. An alias is just a name for the original program in a new place. Changing the alias will change the original program, too. Aliases are used extensively in the sample programs, allowing the same program to appear in different topical folders. For example, the Accelerometer appears in the Tour, O'Reilly Books and Science folders. Two of these are aliases, so the program itself only has to exist in one place.

Tap OK to move, copy or alias the program, or Cancel to exit without making a change.

## Editing Programs

With the exception of horizontal scrolling, editing programs works just like editing text in other iPhone applications, so it won't be covered in detail here. Tap on the source code to begin editing, and use the Done button to dismiss the keyboard when editing is complete. There are a few things worth mentioning, though.

While the electronic keyboard that is always available on the iPhone is perfectly adequate for many tasks, if you will be writing or editing long programs on the iPhone, a physical keyboard is a great accessory. A physical keyboard allows you to move about in the program using arrow keys, select text by pressing shift and using a movement key, and copy and paste with the Command-C and Command-P keyboard shortcuts.

If you are using the software keyboard, techBASIC tries to help out with a code completion bar that appears right over the standard keyboard. This context sensitive bar looks at the program as you type and tries to predict what you might type next. If you see a helpful symbol, tap it to place the symbol in your file. You can turn this feature off in preferences.

Finally, programs are line oriented. Wrapping lines is generally unnatural. For that reason, each program line appears unwrapped. Scroll horizontally to see parts of the line that are not visible.

## Find

The Find button is available anytime the Source view is selected and editable. Tapping the Find button brings up a search and replace view like the one shown.

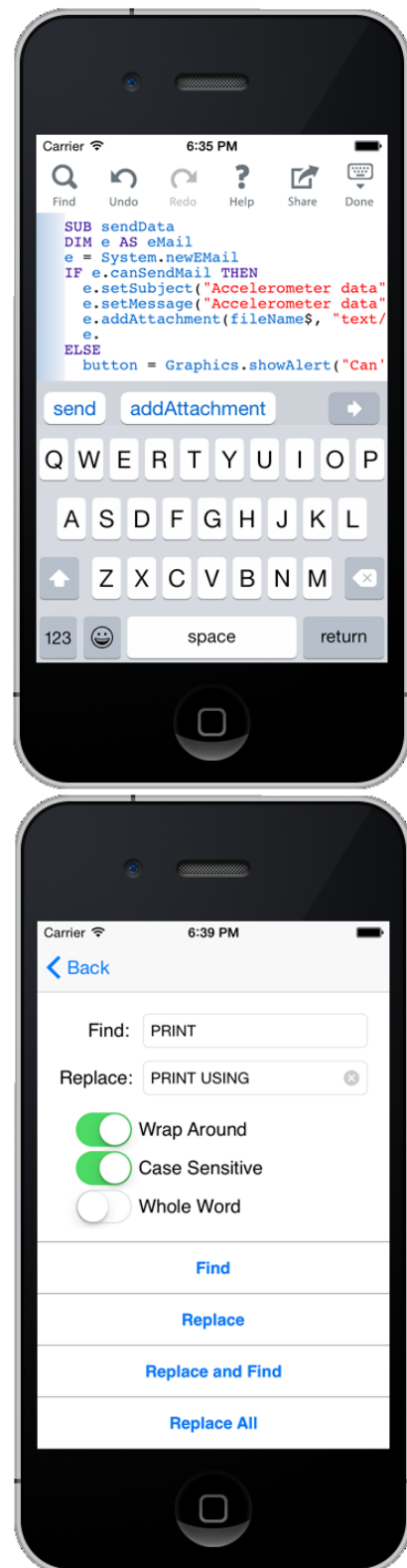
Enter the text you want to locate in the Find text edit box. If you want to replace this text with something else, rather than just locate it, enter the replace string in the Replace text edit box.

The three switches let you turn the search options on or off.

The search always starts at the location you were editing when Find was tapped. If the search reaches the end of the program without finding the target string, Wrap Around allows the search to jump to the top of the program and start over.

BASIC is a case insensitive language. Still, there are times when you may want to search for text with a specific letter case. Turn on the Case Sensitive option to require matching not only the letters in the Find string, but also the case of the letters.

It is frequently useful to find just the places where a short variable name is used, perhaps an index variable named `i`, but not to find everyplace



where the letter `i` is used. The Whole Word option helps. It causes the search to ignore the Find string unless it starts and ends with a character that is not an alphabetic letter or number. For example, searching for `i` in the text

```
FOR i = 1 TO 10
  i4 = i*4
  PRINT i4
NEXT
```

will find the highlighted text, but not the letter `i` in the identifier `i4` or in the `PRINT` token.

With the desired Find and Replace strings entered, and the proper options selected, tapping the Find button searches for the next occurrence of the Find string. Tapping it again will continue to the next occurrence, and so on.

Tapping Replace replaces whatever text is selected—even if it is not the Find string—with the Replace string.

Tapping Replace and Find replaces the current selection with the Replace string, then find the next occurrence of the Find string after the replaced text. This option, in combination with Find, is great for stepping through a program, replacing some but not all of the occurrences of the Find string.

Replace All begins searching from the top of the program, replacing each occurrence of the Find string with the Replace string.

If you are using a hardware keyboard, you can continue to find the last string you entered using command-G.

---

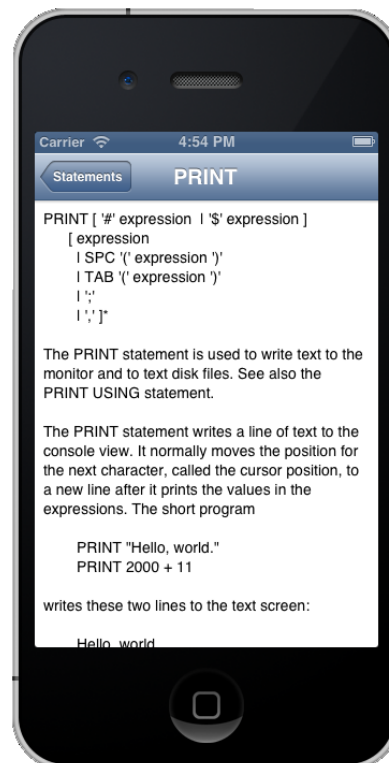
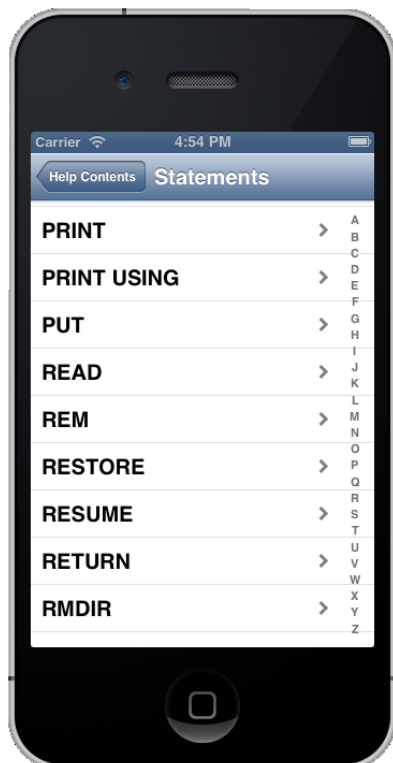
## Undo and Redo

Tap on these buttons to undo or redo changes to the program.

There is no limit, other than memory, on the number of levels of Undo and Redo. Dismissing the keyboard does, however, empty the undo buffer.

---

## Using Help



Even programmers who use a language forty (or more!) hours a week don't remember every command, or every detail about every command, in the languages they use. The Help facility gives you all the technical details about every statement, function and class in techBASIC. Tap on the Help button, available whenever you are editing a program, to bring up the help view. Tap on a topic to expand on the topic, or use the back button at the top of the dialog to move back a level.

While Help has listings for all of the language, it does not include figures, samples or information about the user interface. Help is a great quick reference while writing a program, but it's not a bad idea to keep a copy of this manual in iBooks for situations where more detail is needed.

## Program Output

### Text Input and Output on the Console

PRINT and other output statements write text to the Console view or to files stored in the iPhone's memory. When text is written to the Console view, it is displayed in a scrollable, but not editable, text view. New text is always placed at the end of existing text.

Statements like LINE INPUT let you type text into a running program. Any prompt is placed at the end of the text in the console, then the program waits for you to type a response and press the return key.

As an example, try the program in the snippet. It prints a prompt asking for a starting number; enter an integer value. It then asks for an ending number. Enter another integer, but don't make it too much bigger than the first. All of the integers from the first to the last are printed.

If you don't follow instructions well, and the program is still printing numbers, press the Stop button that shows up while the program runs. The Stop button is covered later in *Stopping a Program*.

#### Snippet

```
INPUT "Enter a starting number: "; first
INPUT "Enter an ending number: "; last
FOR i = first TO last
    PRINT i
NEXT
```

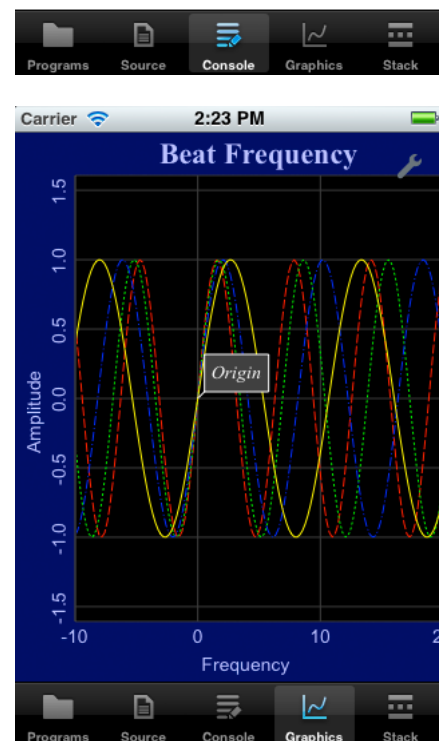
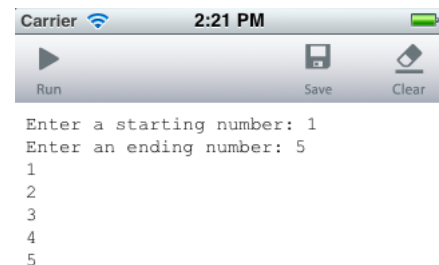
There is no fixed limit to the number of characters that can appear in the Console view, but it will become sluggish with too many characters. The Clear command shows up while in the Console view. Tap this button to delete all text in the console view.

The command `system.clearConsole` can be used in programs to do the same thing.

### Graphics Output

The output from programs that draw or plot functions appears on the Graphics view. Plots, incidentally, don't show up until the program completes. That doesn't mean that the plot is static, though!

To experiment with these commands, run the Beat Frequency sample. The plot from that sample is shown here.





The swipe gesture is a natural way to move graphics around on the screen. Swipe in any direction, and the entire plot will move along with your finger.

Pinching zooms or shrinks the image. There are actually three different pinch effects, depending on the angle of the pinch. If you pinch vertically, the Y axis shrinks or expands, but the X axis is unchanged. Pinching horizontally does just the opposite, shrinking or expanding just the X axis. Pinching along the diagonal shrinks and expands both axis, maintaining the current aspect ratio.

Looking at the program, you can see that techBASIC is not plotting specific points like some other programs, it is actually plotting a function. This allows the program to recompute the values for the plot as you zoom in. As a result, the level of detail available is essentially unlimited—you can zoom in on a detail as close as you like.

The small text box marking the origin is a callout. Callouts let you label portions of the plot; they move right along with the plot as you swipe and pinch. You can also create new callouts at any time by tapping. If your tap is within about one finger width of a plotted line, the callout will snap to the closest point on the line. Callouts created interactively generally show the coordinates of the point tapped, but it is possible to provide callouts with specific text. Two examples are presented in Chapter 16, when the `newPlot` method is described. One of these examples is the Stars sample. If you run this sample and tap on a star, the star's name is displayed.

To experiment with three-dimensional plots, run the `Sinx_x` sample. You should see a plot like the one shown on the next page.

Three-dimensional plots present a unique problem for panning and zooming. Do you want to pan and zoom the data on the axis, like we did for two-dimensional plots, or the axis itself, so you can zoom in on the entire function?

When the program starts, techBASIC will actually pan and zoom the function, not the axis. Unlike a two-dimensional plot, though, we want to pan our three-dimensional function along any of the three axis, but there are only two axis on the screen. To pan along a specific axis, use a wipe gesture parallel to the axis. A vertical swipe, for example, will move the function up and down, while a swipe parallel to the X axis scoots it back and forth along that axis. Pinches are similar to two-dimensional pinches. A vertical pinch changes the range of the function, adjusting Z in this case. Horizontal pinches change the domain, expanding or shrinking the X-Y plane while maintaining the original aspect ratio. A diagonal pinch changes all of the axis at the same time.

The tool button at the top right of the plot is used to change the behavior of the pinch and swipe gestures. Tapping the button brings up a small dialog; selecting Pan/Zoom Axis switches the behavior of swipes and pinches.

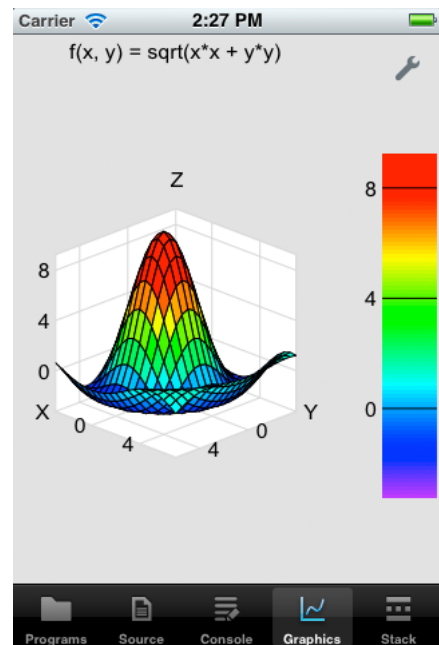
With the mode changed, try swiping the plot again. This time, the axis moves in the direction of the swipe. Pinching in any direction changes the size of the axis. Change the mode back using the same tool and button.

Three-dimensional plots can also be rotated. There are two basic rotation gestures.

First, to twirl the plot about an axis pointed out of the screen, place two fingers on the screen like you are about to pinch, but rotate instead. The plot will twist around the origin of the axis.

You can also twist about an axis on the screen by using two fingers to swipe. Imagine the plot inside a clear track ball. This rotation works as if you swiped the track ball, spinning it about its axis.

Manipulating functions this way is a powerful way to understand the geometry of the function, but this is also an amazing visualization tool. Run the Stars sample, which displays most of the stars within 10 parsecs of our sun. The static display is interesting, but the lack of three dimensions limits the value considerably. Now use a two-finger swipe to rotate the sphere of stars. The motion of rotating the stars allows the relative positions to jump right out of the two-dimensional screen.



## Debugging Programs

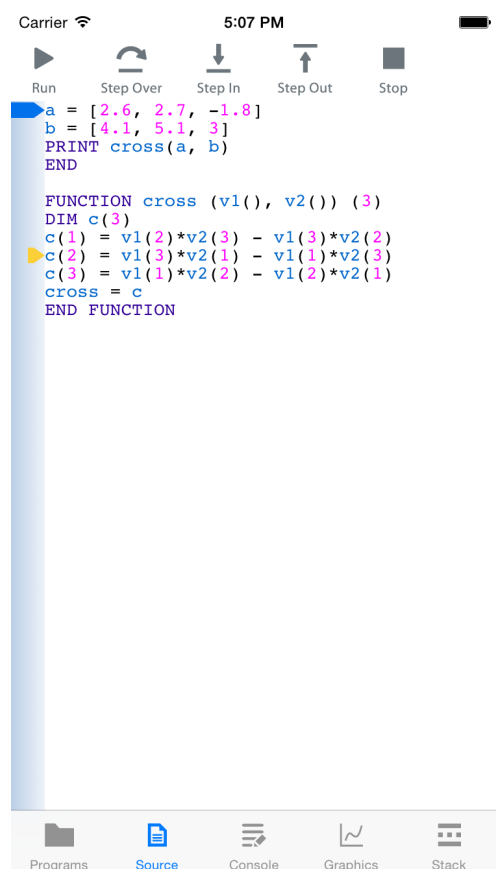


Running a program brings up a debug bar at the top of the Source, Console, Graphics and Stack view. This bar has a number of commands used to understand how your program works, or why it doesn't do what you expected. This bar generally shows up and vanishes to fast with short programs that you may not even notice it. The easiest way to make the bar stay put is to stop the program on the first line with a breakpoint.

### Setting and Clearing Breakpoints

Setting a breakpoint lets you stop a program at a specific location, examining the current values for variables and watching them change as you step line-by-line through the program. The blue bar to the left of the Source view is the breakpoint bar. Tap on the breakpoint bar to set a breakpoint; tap again to clear it. A line with a breakpoint displays a blue marker like the one shown here.

With a breakpoint set, running the program will stop it at the first breakpoint encountered.



### Stepping Through Programs

Once a program is stopped at a breakpoint or with the Pause button, you can step through the program one line at a time to see the program flow. The currently executing line is marked by a gold triangle, like the one seen here after stepping into the function.

Step Over steps one line at a time, even if the line being executed calls a subroutine or function that executes many lines before returning.

Step In works just like Step Over on lines that don't call a subroutine or function, but if the line does make a call, execution pauses at the first line of the subroutine or function.

Step Out is generally used from inside a subroutine or function. The program runs at full speed until the subroutine or function ends, then pauses at the first line encountered after returning. If Step Out is used from the main program, the program runs at full speed to completion.

In all cases, the program will stop if a breakpoint is encountered. For example, stepping over a subroutine with a breakpoint will still stop at the breakpoint.

### Running and Pausing Programs

While the program is running at full speed, a pause button replaces the run button. The program may complete so quickly that the pause button is not noticed, but on longer programs or programs in an infinite loop, tapping the Pause button stops the program as if the next executing line had a breakpoint.

Once a program is stopped, either by tapping Pause or by encountering a breakpoint, tapping the Run button causes the program to continue execution from the current location. It will run at full speed until the Pause button is pressed, a breakpoint is encountered, or the program completes.

## Stopping a Program

Use the Stop button to immediately stop execution of a running program. This works whether the program is stopped by the Pause button or a breakpoint, or running at full speed. An error message will be displayed at the line that was executing when the Stop button was pressed.

## The Stack View

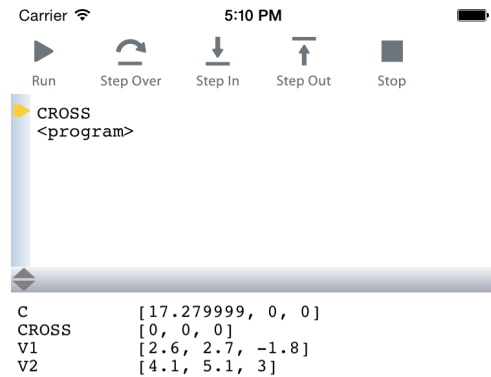
### The Variables Subview

The stack view is only useful when debugging a program. For the same cross product program shown for Setting and Clearing Breakpoints, the sack view shows a Variables subview on the bottom, and a Stack subview on the top. The Variables view shows all of the variables in the current program scope; that is, all of the ones in the current subroutine, function, or the main program.

In the sample shown here, the program is currently in a function called cross, computing the cross product of two vectors. There are four variables in the function. The first is C, a matrix used to hold the result inside the function. The first element of C has been set to 17.279999, while the last two have yet to be changed from the initial values of 0. CROSS is the name of the function itself, showing the value that will be returned to the caller. The last two variables are the parameters that were passed to the function. These values change as the program steps.

There are times when the variable view is not wide enough to show the complete value of a variable. Tapping the variable brings up a detail window showing the full value of the variable. This still may not provide enough space for a really large array or very long string, but you can scroll the value in the detail view to see the rest of the value.

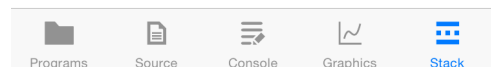
The Variable view can be scrolled to show additional variables if there are too many to show at one time. Dragging the horizontal gray divider will also increase or decrease the height of the Variable view as needed.



### The Stack Subview

The Stack subview appears immediately above the Variables view. It shows the current call stack. The program itself always appears at the bottom of the list. Names of new subroutines and functions are pushed onto the top of the stack as they are called, and popped off of the list as they return.

The gold triangle points to the current scope. The current scope is the one whose variables are shown in the Variables view, not necessarily the one that is executing. Tap the name of a scope to change the current scope. For example, tapping <program> in the sample views shown will change the scope to the program, displaying the values of A and B.



## Understanding Errors

When techBASIC detects an error, either during compilation or while running the program, the view shifts to the Source view and the error is shown. The approximate location of the error is highlighted in

```

a = [2.6, 2.7, -1.8]
b = [4.1, 5.1, 3]
PRINT cross(a, b)
END

FUNCTION cross ( ) (3)
DIM c(3)
c(1) = v1(2)*v2(3) - v1(3)*v2(2)
c(2) = v1(3)*v2(1) - v1(1)*v2(3)
c(3) = v1(1)*v2(2) - v1(2)*v2(1)
cross = c
END FUNCTION

```

yellow, and an error message explains the problem techBASIC detected. In this case, the array constant was closed with a ) character rather than the expected ] character.

Tap the error message to dismiss it. The error is still highlighted in yellow until the program is edited. You can recall the error message by tapping on the highlighted area.

techBASIC error messages are listed in Appendix A. Each error message is accompanied by a brief description of the error. Common causes for the error and possible remedies are discussed when these may not be obvious from the error itself.

---

## Sharing and Syncing

---

### Sending Graphics to Photo

---

You can share any of the images on your graphics screen by tapping the Tools button, then selecting the Send to Photo button from the dialog that appears. This sends a copy of whatever you see in the Graphics view to the Photo, where you can save the image, show it to others, or e-mail it.

The resolution of the image is generally not the same as what you see on your screen. The image will be about 1 megapixel by default. You can change the image size using the techBASIC preferences.

---

### Sending Programs and Text to Mail

---

You can share program source and anything in the Console view by tapping the Share button. The Share button is always visible in the console view, but you need to be editing a program to see it in the Source view. The Share button creates a new email message containing the source code or console text, ready for the user to address and send.

---

### Moving Programs and Data To Your Desktop Computer

---

There are many reasons to want to move a program or data from your iOS device to your desktop computer. You may want to share data with a colleague, use a program in another implementation of BASIC, or perhaps just edit a long program in a traditional keyboard and mouse environment.

Data files are stored in an area called the sandbox. iTunes can be used to move these files to and from your desktop computer. To move a data file, start iTunes and select your iPhone from the list of devices. You will see a list of tabs across the top of the device window; select the tab named Apps. Near the bottom of the window, you will see a list of applications that can share data. Select techBASIC, and you will see a list of all data files.



## About

The about button displays the current techBASIC version, copyright information, and has links to other programs from the Byte Works.

## Preferences

There are several preferences that can be accessed from the iPhone Settings app. Select techBASIC from the list of apps and you will see a view like the one shown here.

Use Syntax Coloring turns syntax coloring on and off. When off, all text is painted in black. The text will render and scroll a little quicker, although this should not normally be noticeable.

Use Code Completion turns the code completion keyboard input accessory on or off. When on, techBASIC will examine the program as you type and suggest words and symbols that might appear next in the program, reducing typing time.

Show Break Points turns the light blue breakpoint bar on and off. While hiding the breakpoint bar leaves you with no way to set or clear breakpoints, it also gives a little more room to see and edit programs.

You can change and delete the sample programs that come with techBASIC. Turn on Reset Samples to get the originals back. This will revoke any changes you have made, and restore any deleted samples. This only happens once, though, so you still have the ability to change and delete the restored samples. If you would like to use a sample as a basis for a new program, rename it. The renamed sample will not be overwritten when the samples are reset, and the original sample will be restored. The samples are only reset after techBASIC is shut down completely and restarted.

The console and source screens default to a font size of 14 points. Use this option to change the font size. The font size will only change after techBASIC is shut down and restarted.

Graphics size controls the size of graphics images exported to the Photo app by the Share button. The image size can be set to the screen size or to one, two, three or four megapixels.

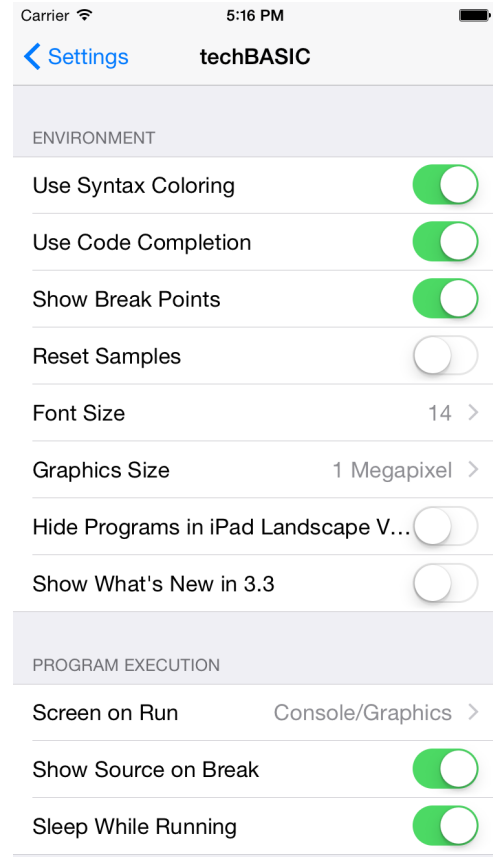
The Hide Programs in iPad Landscape View has no effect on the iPhone.

A dialog appears when you start techBASIC that shows the changes since the last version. You can turn it off from a button in the dialog or from the Show What's New in 3.3 preference. You can also turn it back on with the preference.

Running a program normally shifts to the Console view if you are in the Source view, and stays on the Graphics view if that view is visible. Use the Screen on Run option to move to a specific view when a program starts to run. Remember that methods in the `System` class this can also control the view. These shift the view after it is initially set by techBASIC, and always override this setting.

Disable Show Source on Break to remain in the current view when a breakpoint is hit. You can step and trace from any view, and it may be more important to see what the program is printing, for example, than which line is executing.

Your iPad will sleep to preserve the battery if there is no interaction. Sleeping also pauses any program that is running, which may be a problem for a program that is collecting and processing data. Use the Sleep While Running option to disable sleep while a program is running. It is a good idea to have the iPad hooked up to a power source when this option is used for any length of time, especially if sensors like the accelerometer or location are being used.







## Chapter 4 – Program Symbols

BASIC programs are made up of a series of program symbols called tokens. Tokens are the words used to write a program. They consist of identifiers, symbols, and constants. These tokens form lines, which are also a fundamental part of the BASIC language—some commands, like `SUB`, must appear at the start of a line; most commands are restricted to appearing on a single line; and some commands cannot have anything following them on a line.

---

### Character Set

All techBASIC source code is entered using UTF-8 character encoding; techBASIC supports any Unicode character. For all practical purposes, the standard US keyboard limits the values to the ASCII character set, but it is possible to use international keyboards or to edit a file from your desktop computer. So long as the files are saved in UTF-8, the source file is acceptable in techBASIC.

See <http://en.wikipedia.org/wiki/UTF-8> for a description of the UTF-8 character set.

UTF-8 is equivalent to the ASCII character set when the characters used are restricted to those contained in the ASCII character set, so if international characters are not needed, any ASCII editor will work with techBASIC.

---

### Identifiers

Identifiers in BASIC start with an alphabetic character or underscore and are followed by zero or more alphabetic characters, numeric characters, or underscores. The last character in the identifier is an optional type character. A single underscore with no other characters is not allowed; the single underscore is used as a line continuation character.

BASIC is a case insensitive language, which means that the identifiers `matrix` and `Matrix` are the same identifier. Alphabetic characters are not restricted to the ASCII character set—any Unicode character that is identified as an alphabetic character may be used. These vary by local, but generally, anything that is normally considered a letter will work in an identifier.

There is no practical limit to the number of characters in an identifier. The only limit is available memory.

Type characters indicate the default data type for the identifier. If no type character is used, the default type is single-precision real. You can override the default type using the `DIM` statement.

The type character, if it appears at all, becomes a part of the identifier. For example, `R` and `R!` both default to a data type of single-precision real, but they are different identifiers, and cannot be used interchangeably.

The type characters and their equivalent BASIC data types are shown in the table below. The internal formats for the data types are described in detail in the next chapter.

Character	Type
~	BYTE
%	INTEGER
&	LONG
!	SINGLE
#	DOUBLE
\$	STRING

Some examples of legal BASIC identifiers are shown below. They each represent a different identifier.

MAIN	ARRAY	my_var	S\$	I%	B~	
_subroutine		X1	_\$	D#	L&	R!

## Reserved Words

Reserved words are identifiers that have special meaning in BASIC. A reserved word can only be used for the meaning that BASIC assigns to it, except that reserved words can appear in comments or string constants. The reserved words in techBASIC are shown below.

ABS	AND	AS	ASC	AT	ATAN	BASE
BITAND	BITOR	BITNOT	BITXOR	BREAK	BYREF	BYTE
BYVAL	CALL	CASE	CDBL	CHDIR	CHR	CINT
CLEAR	CLNG	CLOSE	CON	CONT	COS	CSRLIN
CSNG	CURDIR	DATA	DEF	DET	DIM	DIR
DISPOSE	DO	DOT	DOUBLE	ELSE	END	EOF
ERROR	EXISTS	EXP	FN	FOR	FRE	FUNCTION
GET	GOSUB	GOTO	HEX	HOME	HTAB	IDN
IF	INPUT	INT	INTEGER	INV	ISDIR	KILL
LBOUND	LEFT	LEN	LET	LINE	LOC	LOF
LOG	LONG	LOOP	MAT	MID	MKDIR	MOD
NAME	NEXT	NOT	NULL	ON	OPEN	OR
POP	PRINT	PUT	READ	REM	RESTORE	RESUME
RETURN	RIGHT	RMDIR	RND	SEEK	SELECT	SGN
SHARED	SIN	SINGLE	SIZE	SIZEOF	SPC	SQR
STEP	STOP	STR	STRING	SUB	TAB	TAN
TEXT	THEN	TO	TRN	TYPE	UBOUND	UNTIL
USING	VAL	VTAB	WEND	WHILE	ZER	

## Reserved Symbols

Reserved symbols are the punctuation of the BASIC language. Reserved symbols are used as mathematical operators, for forming array subscripts and parameter lists, for separating statements, and so forth. With some restrictions, reserved symbols can also be used in comments and string constants. See the sections below for details.

The reserved symbols in techBASIC are:

!	:	;	+	-	&	.
*	/	^	<	>	=	@
(	)	[	]	#	,	

## Constants

Constants are used to place numbers, arrays and strings into the source code of the program. Each kind of constant has its own unique format, so they are discussed separately.

## Decimal Integers

Decimal integers come in two sizes, referred to as integer and long integer.

Integers consist of one to five digits. The number represented must range from 0 to 32767. You may use a leading - character to form a negative number, although the - character and the number are technically two separate tokens. In practice, this technical distinction is not important.

If the number exceeds 32767, the number becomes a long integer. Long integer constants can range from 32768 to 2147483647.

The table below shows some examples of legal decimal constants.

0	58	32767	32768	400000
---	----	-------	-------	--------

---

## Hexadecimal Integers

Hexadecimal numbers are integers represented in base sixteen, rather than the more familiar base ten. Hexadecimal numbers are made up of the digits 0 to 9 and the letters a to f or A to F. In techBASIC, hexadecimal numbers are distinguished from decimal numbers by a leading \$ character.

As with decimal integers, hexadecimal integers can be long or short. Any hexadecimal constant with 5 or more digits—even if those digits are zero—is a long integer constant. Hexadecimal constants with four or less digits are integer constants.

Integers and long integers are stored in two's complement notation, so hexadecimal constants can be negative. Any value with the most significant bit set is a negative number.

If you are not familiar with hexadecimal notation and two's complement notation you can find out more in most assembly language books and many general computer science books.

The table below shows some legal hexadecimal constants, along with their decimal equivalent and the kind of integer created.

hexadecimal	decimal	size
\$0	0	integer
\$00000	0	long
\$000A	10	integer
\$0010	16	integer
\$8000	-32768	integer
\$08000	32768	long
\$7FFF	32767	integer
\$7FFFFFFF	2147483647	long
\$FFFF	-1	integer

---

## Real Numbers

Floating-point constants are used to represent numbers that do not have an integral value, or that cannot be represented using an integer because they are too large or too small. The general format is a sequence of digits, followed by a decimal point, followed by another sequence of digits, and an exponent, as in

```
3.14159e-14
```

The exponent can start with either an uppercase E, or a lowercase e, as shown.

The format for floating-point constants can vary quite a bit from this general form. You can leave out the digit sequence before or after the decimal point, as in 1.e10 or .1e10. In fact, you can leave off the exponent, too, as in 1. or .1. You must have either an exponent or a decimal point, but if you specify an exponent, you can omit the decimal point, as in 12e40.

All of the real numbers discussed so far result in a single-precision constant. To specify a double-precision constant, use D or d for the exponent character instead of E or e. For example,

```
PRINT USING "#.#####"; 3.1415926535897932
```

prints

```
3.1415927410125732
```

but

```
PRINT USING "%.#####"; 3.1415926535897932D0
```

prints

```
3.1415926535897931
```

---

### String Constants

String constants consist of any sequence of characters except the end of line enclosed in quote characters. To form a string with a quote character, use two quote characters in a row.

Here are some legal string constants:

```
"Hello, world."  
"  
" "  
""That's good!""
```

---

### Array Constants

Arrays are fully supported data types in techBASIC, making it easy to do vector and matrix operations in a natural way. Part of this support is allowing array constants, although the name is a bit misleading.

Array constants start with the [ character and have zero or more array elements separated by commas, then end with the ] character. For example, the vector

$$\hat{i} + 2\hat{j}$$

can be represented as the matrix constant

```
[1, 2]
```

For matrices and higher-ordered arrays, an element can be another vector constant or array constant. One way to assign a 3x3 identity matrix is

```
I = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
```

That doesn't look particularly natural, though, so array constants are allowed to span multiple lines. An end of line can be used after any bracket, comma, or value in an array constant. This allows the more natural identity matrix

```
I = [[1, 0, 0],  
     [0, 1, 0],  
     [0, 0, 1]]
```

When the number of elements in each row varies, missing elements are filled in with zero. Thus

```
I = [[1],
```

```
[0, 1],
[0, 0, 1]]
```

also assigns a 3x3 identity matrix.

Array constants can contain expressions for the elements of the array. Here is a rotation matrix specified using an array constant.

```
alpha = 3.1415926535/4
R = [[cos(alpha), sin(alpha), 0],
     [-sin(alpha), cos(alpha), 0],
     [0, 0, 1]]
```

If any element of an array constant is a double-precision constant or an expression that returns a double-precision value, the resulting array constant consists of double-precision numbers. If none of the elements is double-precision, the array constant is an array of single-precision numbers.

---

## White Space

White space characters consist of the characters created by the space bar and tab key.

White space characters can appear between any two tokens, but may not be used between the characters of a token.

---

## Comments

---

**! any-ascii-characters**

**REM any-ascii-characters**

Comments provide a way to insert text in a program that the compiler ignores. techBASIC allows absolutely any character to appear between the comment token and the end of the line, although the editor effectively limits the allowed characters to the ASCII character set.

There are two distinct comment commands in techBASIC. REM is an almost universal comment command in implementations of BASIC. The ! character is less common, but not unusual; it is supported in the original version of BASIC and in versions of BASIC that follow the ANSI BASIC standard. Both of these work the same way, and are equivalent in all respects.



## Chapter 5 – Types of Data

This chapter describes those BASIC data types which are built into the language. The next chapter covers derived and user-defined data types.

Some of the information in this chapter deals with the way information is stored internally in the memory of the computer. This information is provided for very advanced programmers who need to understand the internal format of the data, primarily for reading and writing binary files. If it does not make sense to you, or if you will not be using the information, simply ignore it.

---

### Integers

techBASIC supports three different types of integers, `BYTE`, `INTEGER` and `LONG`.

Integers defined as `BYTE` are the smallest. `BYTE` values require one byte of storage. The allowed values range from 0 to 255. By default, identifiers ending with the character `~` are defined as `BYTE` integers.

`INTEGER` variables require two bytes of storage. `INTEGER` values can range from -32768 to 32767. By default, identifiers ending with the character `%` are defined as `INTEGER` variables.

Integers defined as `LONG` require four bytes of storage. `LONG` values range from -2147483648 to 2147483647. By default, identifiers ending with the character `&` are defined as `LONG` variables.

Internally, all integers are represented as binary values. Negative integers are represented as two's complement numbers. All of the integers that occupy more than one byte of storage are written to files with the least significant byte first, proceeding to the most significant byte.

---

### Reals

techBASIC supports two storage formats for real numbers.

`SINGLE` numbers are stored in the IEEE floating point number format, with the least significant byte first. They require four bytes of storage. `SINGLE` values are accurate to seven decimal digits, and allow exponents with a range of about  $1e-38$  to  $1e38$ . By default, identifiers ending with the character `!` or without a type character are defined as `SINGLE` variables.

`DOUBLE` numbers require eight bytes of storage each. They are stored least significant byte first, using the IEEE floating point format. They are accurate to fifteen decimal digits, and allow exponent ranges from  $1e-308$  to  $1e308$ . By default, identifiers ending with the character `#` are defined as `DOUBLE` variables.

---

### Infinity

The IEEE number format supports infinity. This is a specially coded value that indicates the number is too large to represent. It is signed, so you can have either positive infinity or negative infinity. The value prints as `Infinity` or `-Infinity`, depending on the sign.

All numeric operations in techBASIC support this number, and do reasonable things with it based on the mathematical concept of infinity. For example, adding any number to infinity is still infinity, while dividing a number by infinity gives zero.

In some cases, your calculations may actually work fine, even if an intermediate result is infinity. In other cases, detecting a value of infinity is simply an indication that you need to tune your algorithms or switch from `SINGLE` to `DOUBLE`.

If your program treats a numeric overflow as an error, you can check for the overflow by comparing the result with infinity. The code snippet shows one way to create a value to compare with, and handles both negative and positive infinity. This particular example generates a run time error when the number `X` is infinity. Unless

intercepted by an `ON ERROR GOTO` error handler, `ERROR 2` will generate an error message that prints “Integer overflow.”

### Snippet

```
inf = 1E50
IF ABS(x) = inf THEN ERROR 2
```

---

## NaN

The IEEE number format supports an error number that prints as NaN; this stands for “not a number.” It is generated whenever a numeric result is simply not valid. An example is taking the square root of a negative number.

All numeric operations in techBASIC support this number. In general, it propagates through the calculation, so that all of the values that are based on NaN are also NaN. For example, adding any number to NaN results in NaN. Comparisons to NaN are defined such that NaN is not equal to anything, including another NaN.

In most cases, NaN indicates an error condition, but the fact that it doesn’t trigger an immediate halt to the program or force you to write a tricky `ON ERROR GOTO` handler gives you more options for handling the error gracefully in your program.

For those cases where you want to stop the program when NaN is detected, the code snippet shows how to test for the condition and flag a run time error. Unless intercepted by an `ON ERROR GOTO` error handler, `ERROR 19` will generate an error message that prints “Type mismatch.”

### Snippet

```
NaN = SQR (-1)
IF Math.isNaN(x) = NaN THEN ERROR 1
```

---

## Strings

Strings are stored internally as a sequence of Unicode characters. When written to or read from a file, UTF-8 character encoding is used.

By default, identifiers ending with the character `$` are defined as `STRING` variables.

---

## Objects

Objects are the instantiations of specific classes within techBASIC. Objects are returned by some of the functions described beginning in Chapter 14.

### Snippet

```
system.showGraphics

DIM p AS Plot
p = Graphics.newPlot
p.setTitle("f(x, y) = sqrt(x*x + y*y)")
p.setGridColor(0.85, 0.85, 0.85)
p.setsurfacestyle(3)
p.setAxisStyle(5)

dim func as PlotFunction
func = p.newFunction(function f)

p.setTranslation3D(-4, -3, -2)
p.setScale3D(1, 1, .8)
END
```



```
FUNCTION f(x, y)
d = SQR(x*x + y*y)
IF d = 0 THEN
  f = 10
ELSE
  f = 10*sin(d)/d
END IF
END FUNCTION
```

|



## Chapter 6 – BASIC Programs

---

### The Anatomy of a BASIC Program

BASIC programs are made up of a series of lines, each of which can have zero or more BASIC statements. Blank lines are allowed, and multiple statements can appear on the same line if the statements are separated by a colon character, with a few exceptions noted in the descriptions of individual statements.

The program begins execution with the first line, continuing sequentially unless the normal flow of execution is changed by one of the statements that is encountered.

Here is a simple BASIC program that prints a string to the screen.

```
PRINT "Hello, world."
```

Classic BASIC programs end when the last line of the program is reached or when a `STOP` statement is encountered. Event driven programs end when a `STOP` statement is encountered. See Chapter 13 for a comparison of the two kinds of programs.

---

### Subroutines

BASIC programs can contain subroutines and functions, declared with the `SUB` and `FUNCTION` statement. Subroutines and functions are referred to as procedures in situations where either one is allowed. These procedures normally appear after the main program, one after the other, although it is possible to intermix the procedures and program statements in any order. The main program can call procedures, which can in turn call other procedures.

---

### Line Numbers and or label

Lines start with an optional line number. The line number is an integer in the range 1 to 65535 appearing at the beginning of the line. A label is an identifier followed by a colon.

There are two fundamentally different ways to use line numbers, and in some important ways they are incompatible.

Traditional BASIC programs require a line number on each program line. If you are importing an old BASIC program, it may be easier to leave the line numbers in the program; they will do no harm.

Like most BASICs implemented after the early 1980's, techBASIC does not require line numbers. Line numbers are still available, but they are usually not used unless you need a destination for a branching statement, such as `GOTO` or `ON ERROR`. Line numbers do not have to be sequential, and in fact, they don't even have to be unique. It's common to see the same line number used in various procedures within a program. Since there is no requirement that the line numbers be unique, you can copy a subroutine from one program and paste it into another without worrying about conflicts between the line numbers.

Line numbers appearing in the same procedure or in the program should be unique within that part of the program, although the compiler does not enforce this restriction. For example, the subroutine

```
SUB Duplicate
10 PRINT "Start"
   GOTO 10
10 PRINT "Done"
END
```

will not generate an error, although it will perform an infinite loop. When duplicate line numbers appear in the same part of a program, the first is always found and subsequent numbers are invisible to the compiler.

Labels provide a way to create destinations for branching statements that have meaningful names instead of just numbers. They can, and generally should, be used instead of line numbers.

A label that appears at the start of a line, serving as a destination for a branching statement, is followed by a colon. The colon tells the compiler the identifier is a label, not a variable at the start of an assignment or subroutine call. Labels that appear in branching statements are not followed by a colon. This is shown in the following example, which also shows one of the few remaining reasons for using GOTO statements in a BASIC program. They work very well for exiting loops early, as this program shows by faking an error when processing the value 7.

```
FOR i = 1 to 10
    errorCode = Process(i)
    IF errorCode <> 0 THEN
        PRINT "Error: "; errorCode
        GOTO Out
    END IF
    PRINT i
NEXT
Out:
END

FUNCTION Process(i)
    Process = 0
    IF i = 7 THEN
        Process = 1
    END IF
END FUNCTION
```

---

### Multiple Statements on One Line

Normally, each BASIC statement appears on a separate line in the program. In most cases, it is technically possible to place more than one statement on a line if you separate the statements with the : character. For example, the line

```
IF a < b THEN c = a : a = b : b = c
```

uses this feature to execute three separate statements instead of one when a is less than b. In techBASIC, it would be more common to use a block IF statement to do the same thing, like this:

```
IF a < b THEN
    c = a
    a = b
    b = c
END IF
```

---

### Continuation Lines

There are situations where a line in BASIC can be so long it becomes unwieldy. In these situations, a single underscore character may be used at the end of a line, and the remainder of the BASIC line may be placed on the next physical line. This is treated as a single physical line of BASIC.

For example, a subroutine statement can have enough parameters to be quite long, as in

```
SUB BLEDiscoveredPeripheral (time AS DOUBLE, peripheral AS BLEPeripheral,
services() AS STRING, advertisements(,) AS STRING, rssi)
```

This can be split across several lines to make it easier to read.

```
SUB BLEDiscoveredPeripheral (time AS DOUBLE, _
                             peripheral AS BLEPeripheral, _
                             services() AS STRING, _
                             advertisements(,) AS STRING, _
                             rssi)
```

Lines may be broken anywhere a space character can appear in the BASIC program.  
Any characters after the \_ character are ignored.



# Chapter 7 – Declaring Variables and Types

---

## What Is a Type?

---

### The Kinds of Types

---

#### Simple Types

At the simplest level, a type is a kind of value that can be stored in a variable. techBASIC supports six types. The name of each type is itself a type, and can be used as a type in DIM statement. The six types are:

BYTE	A single integer byte with the range 0 to 255.
INTEGER	A two byte signed integer with a range of -32768 to 32767.
LONG	A four byte signed integer with a range of -2147483648 to 2147483647.
SINGLE	A four byte single-precision floating-point number with exponents of approximately 1E-38 to 1E38.
DOUBLE	An eight byte double-precision floating-point number with exponents of approximately 1E-308 to 1E308.
STRING	Strings are sequences of up to 2147483647 characters. Each string requires one byte of storage per character, plus an overhead of five bytes. One of the five bytes of overhead is used to mark the end of the string; it is a zero value that appears after the last character. The other four bytes are a pointer to the first character of the string; this is the value actually stored in the string variable.

#### Arrays

Arrays are sequences of the same type of variable. A particular variable is selected using a subscript, which is a numeric value that specifies which of the various variables should be used.

BASIC supports multiply subscripted arrays, too. Multiple subscripts are separated by commas. Here is an example that shows the creation and initialization of a unit matrix with 5 columns and 5 rows. This is a bit contrived, since an array constant or the IDN function could have been used to get the same result.

```
DIM a(5, 5)
FOR row% = 1 TO 5
  FOR column% = 1 TO 5
    a(row%, column%) = 0.0
  NEXT
  a(row%, row%) = 1.0
NEXT
```

By default, array subscripts start with 1 and end with the value shown in the DIM statement. If an array is used before a DIM statement dimensions it, the upper subscript defaults to 10. The DIM statement can be used to change either the lower or upper bound on an array subscript, and the BASE statement can be used to change the default lower subscript from 1 to 0.

Many functions and mathematical operations return arrays. The number of subscripts will never change, but the range of the subscripts frequently will change. For example, this snippet shows an array that starts as a 2x2 array, but after the multiplication operation, it is a 5x5 array. The SIZE, UBOUND and LBOUND functions can be used to determine the actual range of subscripts for an array.

## Chapter 7: Declaring Variables and Types

```
DIM a(2, 2)
a = [[3],
     [5],
     [7],
     [9],
     [11]]*[[1, 1, 2, 3, 5]]
PRINT a
```

These samples show three different ways to look at the contents of an array. The first sample shows manual indexing with numeric subscripts. For two-dimensional arrays, remember that the first subscript is the row, while the second is the column. The array constants in the second example show multiplication of a 5x1 array of prime numbers with a 1x5 Fibonacci sequence. When printed, the row-column relationship is easier to see:

3	3	6	9	15
5	5	10	15	25
7	7	14	21	35
9	9	18	27	45
11	11	22	33	55

In the vast majority of cases, the base type for an array is one of the number types, but techBASIC also allows arrays of strings.

The total length of any one array is limited to 2147483647 bytes. Array subscripts can range from -2147483648 to 2147483647, although a single array cannot support the entire range of subscripts. The number of subscripts in an array is also limited to 2147483647.

You can calculate the number of bytes used by an array by multiplying the size of one element of the array in bytes by the number of elements in each subscript. For example, the array in the code snippet uses 100 bytes; 4 bytes for each `SINGLE` value multiplied by 5 rows multiplied by 5 columns. Arrays of strings use 6 bytes per string entry, plus two bytes per character.

### Objects

techBASIC has limited support for objects through a number of built-in classes, defined beginning in Chapter 14.

You can define variables to hold references to any of these objects using the `DIM` statement. This example shows the definition of a variable to hold a Graphics object, which, in turn, is used to create a function that will be displaced in spherical coordinates.

```
System.showGraphics

DIM p AS Plot
p = Graphics.newPlot
p.setBorderColor(1, 1, 1)
p.setSurfaceStyle(3)
p.setAxisStyle(2)

DIM func AS PlotFunction
func = p.newSpherical(function f)

p.setTranslation3D(0, 0, -5)
p.setTranslation(0, 2.5)
END
```



```

FUNCTION f(theta, phi)
  f = 5
END FUNCTION

```

---

## Type Compatibility

With the plethora of new types available in techBASIC, it's important to understand when two types are compatible. If two types are compatible, their values can be used interchangeably: a value can be assigned to a variable or passed as a parameter if their types are compatible, and two values can be compared if their types are compatible.

There are actually two shades of type compatibility. Two values are type compatible if they are completely interchangeable. Two values are assignment compatible if one value can be assigned to a variable of the other type, or when the value can be passed as a parameter. Since BASIC automatically converts numbers of one type to another, assignment compatibility is not as restrictive as type compatibility. This distinction between assignment compatibility and type compatibility is only important for numbers.

---

## Numeric Type Compatibility

All of the numeric types are assignment compatible with each other: you can freely mix numbers of different types. For two numbers to be type compatible, though, the numbers must be exactly the same kind.

There are three situations where the difference between type compatibility and assignment compatibility is important.

First, some of the automatic type conversions can lead to a loss of precision. If you assign 1.9 to an integer variable, the value that is stored is 1. The same is true when you use the `SINGLE` value 1.9 as an array subscript: the number is first converted to an integer value 1, and the integer result is used to determine the array element to select. This can lead to strange results if roundoff error causes a value to be slightly less than the expected integer. For example, try this:

```

FOR i = 0 TO 10
  a(i) = i
NEXT

sum = 0
x = 3/1300
FOR i = 1 TO 1300
  sum = sum + x
NEXT i
PRINT a(sum)

```

Obviously, the program should print 3. Actually, it prints 2. The reason is that 3/1300 isn't stored exactly, it's stored as approximately 0.002307692. Summing this value 1300 times does not give 3, it gives 2.99997, and when truncated, the index used to access the array is 2, not 3.

In general, your program will be both faster and less prone to bugs of this kind if you always use integer or long integer values when calculating an array subscript. If you must use floating-point subscript values, add 0.5 to the subscript to reduce the chance of this kind of roundoff error.

The second place the difference between type compatibility and assignment compatibility is important is when the numeric values involved cannot be converted. For example, assigning the `SINGLE` value 3E5 to an `INTEGER` variable causes a run-time error, since an `INTEGER` cannot hold values larger than 32767.

The third place this difference is important is when you are passing parameters by reference to procedures. For example,

## Chapter 7: Declaring Variables and Types

```
i = 4
Change(i)
PRINT i
END

SUB Change(BYREF x)
x = x*2
END SUB
```

prints the value 8; the subroutine changed the value of the original variable that was passed as a parameter. This only happens when a variable is passed by reference, though.

The reason this change is important for type compatibility is that variables that are passed by reference must match the type of the parameter exactly—no conversion of any kind is done. If a subroutine expects an `INTEGER` parameter, you cannot pass a `SINGLE` variable by reference.

---

### Array Type Compatibility

Like numbers, arrays can be assigned, can be passed as parameters, can be returned by functions—both build-in functions like `IDN` and functions defined in the program—and can be the result of mathematical operations, such as matrix addition. Assigning a numeric array to another numeric array follows the same rules discussed above for numbers. For example, assigning an array of single-precision real numbers to an array of integers truncates each value, just as assigning a single-precision real number to an integer truncates the value. This program illustrates the conversion:

```
DIM i(3) AS INTEGER, a(3) AS SINGLE
a = [1.1, 2.2, 3.3]
i = a
PRINT i
```

The program prints

```
1                2                3
```

Arrays can be assigned even if number of elements in each subscript differ. The following assignment is perfectly legal:

```
DIM a(3), b(5)
a = [1.1, 2.2, 3.3]
b = a
PRINT b
```

The program prints

```
1.1                2.2                3.3
```

The array `b` has been converted silently from an array with 5 elements to an array of 3 elements to properly hold the result of the assignment.

Arrays with differing numbers of subscripts are not assignment compatible. For example, the following program will fail when executed.

```
! This will not work!
DIM a(3), b(5, 5)
a = [1.1, 2.2, 3.3]
b = a
PRINT b
```

An array can be passed as a parameter to a procedure if the array value is assignment compatible with the procedure parameter. If the parameter is by reference, the types of the elements of the arrays must also match.

Arrays of strings are assignment compatible with each other if the number of subscripts matches. Arrays of strings are not assignment compatible with numbers.

---

## Strings

Strings are always compatible with each other. Strings are not compatible with any other type.

Strings can be converted to numbers, and numbers can be converted to strings, using the `STR` and `VAL` functions.

---

## Objects

Objects are instantiations of classes. All classes in techBASIC are built-in, providing services like graphics objects that can still be manipulated after the program finishes executing. These objects can be stored in variables, just as other values can be stored in variables. There is no default type for objects. Each object variable must be declared with a `DIM` statement before any value is stored in the object. The object variable is declared using the class name. For example,

```
DIM p AS Plot
DIM pf AS PlotFunction
```

declares an object variable which holds a plot, and another which holds a graph. Once initialized by

```
p = Graphic.newPlot
pf = p.newFunction(FUNCTION f)
```

the plot and function can be manipulated, perhaps changing the background color or the style of line used to draw the function. These object variables are not type compatible; you cannot assign a `Plot` to a `PlotFunction`.

---

## Default Types

There are two ways to create a named variable in BASIC. The most direct way is with the `DIM` statement. The `DIM` statement is traditionally used to dimension arrays, but it can also be used to create a variable with any type you like. For example, the `DIM` statement

```
DIM i AS INTEGER
```

creates a new variable called `i`, and makes this variable an `INTEGER`.

The most common way to create a new variable, though, is to simply use it. BASIC guarantees that the value will be created and initialized to 0 or an empty string, as appropriate. This doesn't really matter in most cases, since a variable is almost always assigned an initial value the first time it is used.

There needs to be some way to assign a type to a new variable, though. BASIC uses a special set of characters that can appear at the end of any identifier. If the variable is declared by using it in an expression, this trailing character determines the variable's type.

## Chapter 7: Declaring Variables and Types

For example, % is used for integers. The program

```
FOR i% = 1 TO 10
  PRINT i%
NEXT
```

creates the variable I% when the FOR loop starts. Since the last character in the variable's name is %, this variable is an INTEGER.

Here's a complete list of the type characters in techBASIC.

Character	Type
~	BYTE
%	INTEGER
&	LONG
!	SINGLE
#	DOUBLE
\$	STRING

Variables that don't have a type character are declared as SINGLE.

If the type character is used, it becomes a part of the variable name. Using the variable without the type character will create a completely different variable. This short program creates two distinct variables, as the PRINT statement proves when you run the program.

```
DIM i AS INTEGER
i% = 4
i = 5
PRINT i%, i
```

While a type character at the end of an identifier becomes a part of the variable name, type characters cannot be used anywhere else in the variable name. Only one is allowed, and that one type character, if it is used at all, must be the last character in the variable's name.

If a variable is created by the DIM statement, the type you give in the DIM statement overrides any type character, or the absence of a type character. In the example just shown, the variable I would have been a SINGLE variable if the DIM statement was not in the program, since variables without a type character default to SINGLE. In this case, though, the variable I is an INTEGER variable. It's just as possible to create a variable called I% that holds a SINGLE value using the same idea, but of course anyone who does this for anything but a prank deserves to have their fingers smacked with a ruler. Any BASIC programmer will assume that a variable with a trailing % character is an INTEGER; making it something else could cause confusion.

Arrays can also be defined by using the array, rather than with the DIM statement. The number of subscripts matches the number used in the expression where the array first appears. The maximum subscript is always 10. For example, encountering the statement

```
a(1, 1) = 11
```

without first encountering a DIM statement creates an array of SINGLE values. The array has two subscripts, and each can range from 1 to 10, so there are 100 SINGLE values in the array.

It is not possible to create an array and a variable with the same name. It is also not possible to use the same name for a subroutine or function as an array or variable.

---

## Declaring Types and Variables

---

**BASE [ 0 | 1 ]**

The **BASE** statement sets the default lower bounds for arrays. Without the **BASE** statement, the default lower bounds for an array is 1. The program

```
DIM a(4)
```

creates an array with four elements having subscripts 1 to 4. The program

```
BASE 0
DIM a(4)
```

creates an array with five elements having subscripts 0 to 4.

The **BASE** statement can only be used once in a program, and only 0 and 1 are allowed for the default lower bounds. The **BASE** statement must appear before any array is used in a statement, including **DIM**. See the **DIM** statement for a way to set the lower bounds for an array to something other than 0 or 1.

---

**DIM [ SHARED ] identifier [ subscript [ TO subscript ] ] [ AS type ] [ ', ' identifier [ subscript [ TO subscript ] ] [ AS type ] ]\***

The **DIM** statement is used to create a variable. Variables can be created by simply using them in a BASIC expression, but there are two situations where you need more control over how the variable is created than you get when you simply use a variable. In addition, many programmers find that dimensioning each and every variable is a good way to document what variables are used in a program and how they are used—a comment just before or after the **DIM** statement is very handy for remembering how a program's data structures are used.

### Dimensioning Arrays

The first situation where you need control over how a variable is created is dimensioning an array. This is the most common use for **DIM**, and it's also the traditional use that gives the statement its name. To dimension an array, give the name of the array with the maximum value for each subscript. For example,

```
DIM v(5), a(5, 5)
```

dimension two arrays. The first is an array with five **SINGLE** values, subscripted from 1 to 5. The second array has two subscripts, each ranging from 1 to 5. The full array contains 25 **SINGLE** values.

As with any variable, you can use type characters to specify the type of the elements in the array. For example,

```
DIM a#(7, 7)
```

creates an array of 49 **DOUBLE** values. See *Default Types*, earlier in this chapter, for a complete discussion of type characters. You can also specify a specific type using **AS**; this is covered in *Assigning a Type With AS*, right after this section.

While the default lower bounds on an array subscript is 1, there are two ways to change this value. The most direct is to give both the lower and upper bound when declaring the array, as in

```
DIM livingPeople(1880 TO 2011)
```

to declare an array for the birth year of all living people in a census. The lower bound can be any integer value, so long as it is less than or equal to the upper bound.

## Chapter 7: Declaring Variables and Types

The second way to change the bounds is with the `BASE` statement. This statement exists because many implementations of BASIC start the lower bounds at 0 by default, rather than 1. Placing

```
BASE 0
```

At the start of a program causes techBASIC to mimic this behavior. Using `BASE`, our original example becomes

```
BASE 0
DIM v(5), a(5, 5)
```

This creates arrays that index from 0 to 5 rather than 1 to 5; `V` holds 6 values while `A` holds 36 values. The `BASE` statement can only be used once, and the only allowed values are 0 or 1.

In most situations it makes more sense to use an `INTEGER` value for array subscripts, but it is possible to use any numeric value. Values are always converted to `INTEGER` before being used as a subscript. For floating-point values, the value is truncated, so a subscript of 3.999 is treated as the `INTEGER` value 3. The potential problems of this sort of round-off error are the main reason floating-point values should not normally be used for array subscripts. Calculating the array subscript also takes much longer using floating-point arithmetic; in some programs the speed difference can be dramatic.

In most cases `DIM` statements appear right at the start of a program or subroutine, and the subscripts are constant values. These normal use rules come about because it makes sense to organize programs with the `DIM` statements at the beginning, and in most cases the size of an array is fixed. There are situations where it makes sense to use an expression for the size of an array, though, and occasionally it even makes sense to imbed the `DIM` statement in the program.

For example, here's an array that uses the value stored in `I%` to determine the size of the array. This value might be read from a disk file just before reading the numbers for the array, or it might be entered by the person using the program.

```
DIM speed(i%)
```

An array can also be dimensioned in multiple places, or multiple times. This technique can be used to grow an array gradually as the need arises. There are two things to be aware of when using this technique. The first is that, while the number of elements in each subscript of the array can be changed when an array is redimensioned, the number of subscripts cannot. For example

```
DIM a(4)
DIM a(7)
```

is OK, but

```
DIM a(4)
DIM a(4, 4) ! Not valid.
```

will result in a redimensioned array error. The other caution is that redimensioning an array resets all of the values in the array back to zero or the empty string.

See *List Files Example* on page 125 for a program that puts this idea to use.

### Assigning a Type With `AS`

The `AS` clause is used to assign a type to a variable. `AS` is followed by the name of a type. This can be something as simple as the name of a default type or as complex as the name of an object.

For example,

```
DIM i AS INTEGER, j AS INTEGER, k AS INTEGER
```

creates three `INTEGER` variables that don't need `%` as the last character of the variable name. You can use this idea with any of the built-in types. The built-in types are `BYTE`, `INTEGER`, `LONG`, `SINGLE`, `DOUBLE` and `STRING`. See *Default Types*, earlier in this chapter, for a complete discussion of these types.

While it rarely if ever makes sense to do so, you can use the `AS` clause to override the type of a variable. For example,

```
DIM cost$ AS SINGLE
```

creates a variable named `cost$`; this would normally be a string, but because of the `AS` clause, the variable is `SINGLE`.

The `DIM` statement can also be used to define a variable to hold an object, as this short program shows.

```
System.showGraphics

DIM p AS Plot
p = Graphics.newPlot
p.setTitle("sin(2*theta)*cos(2*theta)")

DIM func AS PlotFunction
func = p.newPolar(FUNCTION f)
func.setColor(1, 0, 0)
func.setStyle(2)
END

FUNCTION f(theta)
f = SIN(2*theta)*COS(2*theta)
END FUNCTION
```

### Using Default Types With DIM

The last use of the `DIM` statement is to create a variable using its normal default type. This is entirely optional, since `BASIC` will create the variable for you and initialize it to zero the first time it is used in the subroutine or program, but this is a convenient way to create a dictionary of your variables and describe how they are used in the program.

The statements

```
DIM i%: REM Loop/index variable
DIM interest: REM Annual interest rate
```

create two variables, the `INTEGER` variable `i%` and the `SINGLE` variable `interest`, and give a clue as to how these variables are used in the program.

### DIM SHARED

In `techBASIC`, a variable declared at the program level is automatically accessible from any subroutine or function that does not have an explicit `DIM` statement for the variable. In some other dialects of `BASIC`, variables declared at the program level are not accessible inside a subroutine unless some other action is taken. One of these actions is to use the `SHARED` qualifier immediately after `DIM`. `techBASIC` allows the use of the `SHARED` qualified on a `DIM` statement to allow careful programmers to write programs that will run in multiple dialects of `BASIC`. It has no meaning in `techBASIC`; it is allowed, but ignored.





## Chapter 8 – Expressions and Assignments

---

### Expressions

BASIC works on values, manipulating, storing, and retrieving information stored in the bytes of your computer's RAM and in files that can be moved to other computers. Almost all of the statements in BASIC that accept a value as a parameter allow you to use an expression. An expression can be something as simple as the number 1, or as complicated as a complex mathematical formula. This chapter describes how expressions work.

---

### Kinds of Expressions

There are four different kinds of expressions in BASIC. They can be intermixed, and putting them together follows the same underlying rules, so all four are described here as a group. The difference between them lies simply in the result they produce.

#### Mathematical Expressions

Most BASIC commands and statements work on numbers. A mathematical expression is any expression that results in a number, whether that number is an `INTEGER`, `LONG`, `SINGLE` or `DOUBLE` value.

When you see the term **expression** in the model for a BASIC statement, it generally means that the expression is a mathematical expression. In a few cases, like the `LET` statement, it can also mean that the expression can be a mathematical expression or a string expression. Those cases are usually obvious. After all, you need to be able to assign values to string variables, just as you assign values to numerical variables, so `LET` must support string expressions as well as mathematical expressions. Just as obviously, you can't take the square root of a string, so the expression accepted by `SQR` must be a mathematical expression. In any case, the description of the command will point out what kinds of expressions are valid by telling you what the operation is and by explicitly stating when strings or pointers are allowed.

#### Logical Expressions

Some commands test to see if a condition is true or false. The `IF` statement is the classic example. The condition is called a logical expression.

At one level, logical expressions are exactly the same as mathematical expressions. Both are calculated the same way. Both result in a number. The difference is not how they are created, but how they are used.

While the result of a logical expression is a number, what is needed is a logical value—either true or false. To make this jump, BASIC treats any number whose value is zero as false, and any other value as true.

Operations that return a logical value, like the `OR` operation, always return 0 for false and 1 for true. While it is not technically required, most implementations of BASIC seem to follow this rule. In techBASIC, the number is always returned as an `INTEGER`. In most situations this doesn't make much difference.

There is one place where conversion of numbers can lead to some unexpected results. Be careful of floating-point values used as logical values. The value 0.01, for example, is not a zero, so by itself it has a logical value of true. If you save the value in an integer, though, it converts to zero, changing the result to false. This causes the following program to print `TRUE` the first time, and `FALSE` the second, even though strict logic requires both values to be the same.

```
b = 0.1
b% = 1
PrintLogical(b OR 0)
PrintLogical(b% OR 0)
END
```

## Chapter 8: Expressions and Assignments

```
SUB PrintLogical(b)
  IF b THEN
    PRINT "TRUE"
  ELSE
    PRINT "FALSE"
  END IF
END SUB
```

### Array Expressions

Array expressions are expressions that result in an array rather than a simple numeric value. Like most scientific implementations of BASIC, techBASIC programs can add, subtract, multiply, and otherwise operate on entire arrays in a single operation. At times, these array expressions use numeric expressions as an argument, and at other times, an array expression is passed to a function that returns a numeric value.

### String Expressions

String expressions return strings, as the name implies. Generally the string is the result of a function, like `LEFT`, but the concatenation operation `&` also works on strings, combining them to form a longer string.

---

## Evaluating Expressions

To help us analyze how expressions are constructed, we'll divide the discussion into two categories. This section will discuss the various operations that accept two numbers, arrays or strings and return a single number, array or string. These are technically known as binary operators. In the next section, we'll discuss terms, which are numbers, variables, and operations that work on a single value and return a single result. As we'll see, you can always think of a term as a single value, and these values can be combined with the operations in this section two at a time.

### Operator Precedence

When you write a mathematical formula, you expect that some operations are performed before others. For example, if you see

$$1 + 2 \times 3$$

you expect the result to be 7, because multiplication is always done before addition. The technical term for this choice of order is operator precedence. Operations with a higher precedence are always done before those with a lower precedence.

The following table shows operator precedence for all of the operations in BASIC, both the binary operations (those that take two arguments, like addition and division) described in this section and the unary operations (those that take a single argument, like `NOT`) described with terms. The operations at the top of the table have a higher precedence, and are always performed first.

When operations have the same precedence, the operation is always done in left-to-right order. Normally this doesn't matter, but in some numerically sensitive equations it can make a difference.

Operations By Precedence					
.	<sup>1</sup>	( )	<sup>2</sup>		
+	<sup>3</sup>	-	<sup>3</sup>	NOT	BITNOT
^					
*		/		MOD	
+		-		&	
<<		>>			
=		<		>	<= >= <>
AND					
OR					
BITAND					
BITXOR					
BITOR					

Notes for the operator precedence table:

- <sup>1</sup> Used to access methods in an object.
- <sup>2</sup> In this table, the parentheses indicate accessing an array.
- <sup>3</sup> These are the unary versions of the operations. For example, -X uses the unary subtraction operation.

You can use parentheses to change the order of operations. For example,

```
PRINT 1 + 2 * 3
```

prints 7, but

```
PRINT (1 + 2) * 3
```

prints 9.

## Binary Conversions

techBASIC supports five different kinds of numbers: `BYTE`, `INTEGER`, `LONG`, `SINGLE` and `DOUBLE`. Binary conversions are the rules used to perform operations on different kinds of numbers. These rules tell you both what the result will be, and in some cases how the calculation is performed.

Most of the time these differences are not important. BASIC converts numbers back and forth as needed without causing much trouble. There are a few situations where the difference is important, though. Understanding them could save you hours of staring at a program that *ought* to work, but just doesn't seem to give you the answer you expect.

Unless otherwise noted, when two numbers of the same kind are used with a binary operation, the result is a number of the same kind, too. For example, if you add two integers, as in `I% + J%`, the result is also an integer.

The one universal exception is `BYTE`. techBASIC treats `BYTE` variables as a special case of `INTEGER` that uses less storage. Operations involving `BYTE` values always work as if the values were `INTEGER`.

When you mix two different kinds of number in the same operation, the numbers are *first* converted to the same number type, *then* the operation is performed. For example, `R * I%`, where `R` is a `SINGLE` number, is carried out by converting `I%` to a `SINGLE` value, then doing the multiplication. The result is `SINGLE`.

Binary array operations follow the same rules as binary variables, applying the rule to each element of the array.

The following table shows how binary conversions are carried out. Since `BYTE` numbers are always converted to `INTEGER`, they are not shown in the table. The rightmost column shows both the number format the values are converted to before the calculation is performed and the kind of number the operation returns.

If one value is...	and the other is...	the calculation is...
DOUBLE	SINGLE	DOUBLE
DOUBLE	LONG	DOUBLE
DOUBLE	INTEGER	DOUBLE
SINGLE	LONG	SINGLE
SINGLE	INTEGER	SINGLE
LONG	INTEGER	LONG

### Unary Conversions

There are many places in techBASIC where a number is converted from one type to another. For example, this happens during binary conversions, described above. Unary conversions are also made when you assign a number that is one type to a variable of another type using a LET statement, or when you pass a value as a parameter. You can also force a unary conversion using the functions CINT, CLNG, CSNG and CDBL. In all of these cases, the conversion from one number type to another is done in exactly the same way. This section describes how these conversions are performed.

#### Converting DOUBLE to SINGLE

Converting a DOUBLE value to a SINGLE value results in a loss of precision in the number, which drops from about 16 decimal digits to about 7 decimal digits. In some cases, this loss is completely transparent. Both formats can represent the number 4.5 with complete accuracy, so converting the DOUBLE value 4.5 to the SINGLE value 4.5 doesn't lose any accuracy at all. On the other hand, converting the DOUBLE value 4.500000001 to SINGLE will result in the number 4.5.

DOUBLE values also support a larger exponent range than SINGLE values. The exponent range for SINGLE is about 1E-38 to 1E38. If the DOUBLE number is too close to zero to represent with the smallest available SINGLE exponent, the result is 0.0. If the DOUBLE value is too large to represent with the largest available SINGLE exponent, the result is infinity or negative infinity.

#### Converting DOUBLE to LONG

The value is first converted to an integer by rounding down to the largest integer that is less than or equal to the original value. Some typical values are:

DOUBLE	LONG
-100.6	-101
-99.2	-100
-0.1	-1
0.1	0
3.3	3
3.99	3

The maximum range for LONG values is -2147483648 to 2147483647. After truncating, if the double value is outside this range, the program stops with a run time error.

#### Converting DOUBLE to INTEGER

The rules for converting DOUBLE to INTEGER are essentially the same as for converting DOUBLE to LONG. The only difference is that INTEGER values have a smaller range than LONG values, so overflows that result in a run time error can occur with numbers that are valid for LONG. The valid range for INTEGER values is -32768 to 32767; if the truncated DOUBLE value is outside this range, a run time error stops the program.

#### Converting SINGLE to DOUBLE

Converting SINGLE to DOUBLE always works, and there is no loss of precision.

#### Converting SINGLE to LONG

Converting a SINGLE value to a LONG value follows the same rules as converting a DOUBLE value to a LONG value.

#### Converting SINGLE to INTEGER

Converting a SINGLE value to an INTEGER value follows the same rules as converting a DOUBLE value to an INTEGER value.

#### Converting LONG to DOUBLE

All LONG values can be represented with no loss of precision by a DOUBLE value. The conversion always works, with no possible error or loss of precision.

#### Converting LONG to SINGLE

Converting a LONG value to a SINGLE value always works, with no possibility of an error, but there can be a loss of precision. The mantissa of a SINGLE value uses 24 bits, which gives about 7 significant decimal digits. LONG values larger than 16777216 or smaller than -16777216 cannot be stored without loss of precision in a SINGLE value. The least significant bits are lost.

If you decide to verify this range, be sure to convert the SINGLE value back to LONG before printing it. Only the first seven significant digits of SINGLE values are normally printed, and you need to see eight digits to verify there was no loss of precision.

#### Converting LONG to INTEGER

Converting a LONG value to an INTEGER value works for any value in the range -32768 to 32767. A LONG value outside this range triggers a run time error.

#### Converting INTEGER to DOUBLE

All INTEGER values can be represented with no loss of precision by a DOUBLE value. The conversion always works, with no possible error or loss of precision.

#### Converting INTEGER to SINGLE

All INTEGER values can be represented with no loss of precision by a SINGLE value. The conversion always works, with no possible error or loss of precision.

#### Converting INTEGER to LONG

All INTEGER values can be represented with no loss of precision by a LONG value. The conversion always works, with no possible error or loss of precision.

#### Converting BYTE to Any Other Type

BYTE values are always converted to INTEGER values before any operation is performed. The result is always an integer in the range 0 to 255.

## Chapter 8: Expressions and Assignments

### Converting Any Other Type to BYTE

Conversion of any value to a BYTE always starts by converting the value to an INTEGER. If the original value is outside the range -32768 to 32767, the conversion triggers a run time error.

Once the value has been reduced to an INTEGER, the least significant 8 bits of the two's complement integer value are used. If the value is in the range 0 to 255, the result is exact. If the value is outside that range, the result seems strange unless you are familiar with the way integers are stored. If you would like to explore how integers are stored, refer to any assembly language programming book, or any general computer science text that discusses two's complement notation.

For positive numbers the value that results is the same as you would get from the expression

```
i% - 256*INT(i%/256)
```

For negative numbers the value is the same as the result of this expression:

```
65536 + i% - 256*INT((65536 + i%)/256)
```

### Addition

The symbol for addition is +.

Addition works on both numbers and arrays.

When you add two numbers, the addition operation returns the sum of the two numbers. For example, 1 + 1 returns 2.

If you are adding integers, and the result is larger than 32767 or smaller than -32768, the result cannot be an integer value. If the value is not in the range -32768 to 32767, the result is not reliable, but no runtime error occurs. For example, 30 + 30 returns the integer 60, but 30000 + 30000 returns -5536. The same thing happens with LONG values, although the range is somewhat larger. LONG values can range from -2147483648 to 2147483647. If the result of an addition of long integers falls outside this range, the value is not valid.

SINGLE and DOUBLE values can overflow, too. If the result of an addition is too close to zero to represent, the value returned will be 0.0. If the value is too large or too small to represent, the result will be infinity or negative infinity. All of the various math operations in techBASIC know how to handle infinity in a reasonable way. Adding infinity to any other value except negative infinity gives infinity, and adding negative infinity to any value except positive infinity gives negative infinity. Adding infinity to negative infinity results in a value called "not a number," which prints as NaN. This is handled reasonably, too. NaN added to any other value still returns NaN.

Arrays can also be added. Any two arrays that have the same number of subscripts, where each subscript has the same number of elements, can be added. It is not a requirement that the upper and lower bounds of the subscripts be the same. For example, this program works just fine.

```
DIM a(3), b(-3 TO -1)
a = [1, 2, 3]
b = [-3, -2, -1]
PRINT a + b
```

Arrays are added by adding the corresponding elements in each array. The above addition is completely equivalent to

```
DIM c(3)
FOR i% = 1 TO 3
  c(i%) = a(i%) + b(i% - 4)
NEXT i%
```

### String Concatenation

Strings are concatenated, not added. The symbol for concatenation is &. When you concatenate two strings, the second is tacked onto the end of the first. For example,

```
PRINT "Hello, " & "world."
```

prints the string “Hello, world.” The concatenation operator also works on arrays of strings, provided the arrays have the same number of subscripts and each subscript has the same number of elements. (See Addition, above, for details on how arrays are manipulated.)

### Subtraction

The symbol for subtraction is -. Subtraction works with numbers and arrays.

When you subtract two numbers, the result is the number on the left minus the number on the right. For example,  $4 - 10$  returns -6.

Just as with addition, overflows with INTEGER or LONG arguments will result in an invalid value.

Just as with addition, subtraction of SINGLE or DOUBLE values that result in a number too close to zero to represent returns 0.0, while results too large or too small to represent return infinity or negative infinity.

The table below shows how infinity and NaN behave for subtraction. NaN stands for “not a number,” and indicates that the result of an operation is not a valid real number. “Any value” refers to any number, including infinity or NaN, that is not listed explicitly in the table.

This...	minus this...	gives this...
NaN	any value	NaN
any value	NaN	NaN
Infinity	Infinity	NaN
-Infinity	Infinity	-Infinity
Infinity	-Infinity	Infinity
-Infinity	-Infinity	NaN
Infinity	any value	Infinity
any value	Infinity	-Infinity
-Infinity	any value	-Infinity
any value	-Infinity	Infinity

Any two arrays that have the same number of subscripts, where each subscript has the same number of elements, can be subtracted. It is not a requirement that the upper and lower bounds of the subscripts be the same. For example, this program works just fine.

```
DIM a(3), b(-3 TO -1)
a = [1, 2, 3]
b = [-3, -2, -1]
PRINT a - b
```

Arrays are subtracted by subtracting the corresponding elements in each array. The above subtraction is completely equivalent to

```
DIM c(3)
FOR i% = 1 TO 3
  c(i%) = a(i%) - b(i% - 4)
NEXT i%
```

## Multiplication

The symbol for multiplication is \*. Multiplication works with numbers and arrays.

Multiplying two numbers returns their product. For example, `4 * 5` returns 20.

If the product of two `INTEGER` numbers is outside the range -32768 to 32767, the result is not valid. If the product of two `LONG` numbers is outside the range -2147483648 to 2147483647, the program is also not valid.

If the product of two `SINGLE` or two `DOUBLE` values is too close to zero to represent, the result is 0.0. If the values are too large or too small to represent, the result is infinity or negative infinity.

The table below shows how infinity and NaN behave for multiplication. NaN stands for “not a number,” and indicates that the result of an operation is not a valid real number. “Any value” refers to any number, including infinity or NaN, that is not listed explicitly in the table.

This...	times this...	gives this...
NaN	any value	NaN
any value	NaN	NaN
Infinity	any positive value	Infinity
Infinity	any negative value	-Infinity
Infinity	0	NaN
any positive value	Infinity	Infinity
any negative value	Infinity	-Infinity
0	Infinity	NaN
-Infinity	any positive value	-Infinity
-Infinity	any negative value	Infinity
-Infinity	0	NaN
any positive value	-Infinity	-Infinity
any negative value	-Infinity	Infinity
0	-Infinity	NaN

Two arrays can be multiplied if both have two subscripts, and the number of elements in the second subscript of the first array matches the number of elements in the first subscript of the second array. The result is a two-dimensional array whose first subscript has the same number of elements as the first subscript of the first array, while the second subscript has the same number of elements as the second subscript of the second array. Said another way, two arrays can be multiplied if the first is *m* by *n* and the second is *n* by *p*; the result is an *m* by *p* array. This follows the normal rules of matrix multiplication, as this example shows:

```
DIM A(3, 3), B(3, 1), C(3, 1)
A = [[1, 1, 2],
     [3, 5, 8],
     [13, 21, 34]]
B = [[3],
     [5],
     [7]]
C = A*B
PRINT C
```

Keep in mind that a 3x1 matrix is not the same thing as a vector (a singly subscripted array).

Any array can also be multiplied by a number. This operation multiplies each element of the array by the same constant. For example, multiplying the identity matrix by 4 yields a matrix with 4 along the main diagonal.

```
DIM A(5, 5)
A = IDN(5) * 4
```



## Division

The symbol for division is `/`.

Division divides the number to the left of the operator by the number to the right. For example, `4.8/1.5` returns 3.2.

Division always returns a `SINGLE` or `DOUBLE` value. After binary conversions, if the operands are `INTEGER` or `LONG`, the values are converted to `SINGLE` and the result is `SINGLE`.

If the result is too close to zero to represent, the result is 0.0. If the values are too large or too small to represent, the result is infinity or negative infinity. These print as `Infinity` and `-Infinity`.

The table below shows how infinity and NaN behave for division. NaN stands for “not a number,” and indicates that the result of an operation is not a valid real number. “Any value” refers to any number, including infinity or NaN, that is not listed explicitly in the table.

This...	divided by this...	gives this...
NaN	any value	NaN
any value	NaN	NaN
any positive value	0	Infinity
any negative value	0	-Infinity
Infinity	any positive value	Infinity
Infinity	any negative value	-Infinity
-Infinity	any positive value	-Infinity
-Infinity	any negative value	Infinity
any value	Infinity	0.0
any value	-Infinity	0.0
Infinity (+ or -)	Infinity (+ or -)	NaN

## Modulo

The symbol for the modulo operation is `MOD`.

`MOD` returns the remainder from an integer division. The operation `x MOD y` is equivalent to `x - y*INT (x/y)`. If both arguments are integers, the result is an integer; if either argument is a long, float or double, the result is a long.

The second argument must not be zero.

## Exponentiation

The symbol for exponentiation is the carrot character, `^`.

Exponentiation raises the number to the left of the operator to the power of the number to the right. For example, `3 ^ 4` is `3 * 3 * 3 * 3`, or 81.

Exponentiation always returns a `SINGLE` or `DOUBLE` value. After binary conversions, if the operands are `INTEGER` or `LONG`, the values are converted to `SINGLE` and the result is `SINGLE`.

If the result is too close to zero to represent, the result is 0.0. If the values are too large or too small to represent, the result is infinity or negative infinity. These print as `Infinity` and `-Infinity`.

Raising negative number to a power is not mathematically valid using the real number system unless the power is an even integer. Raising a negative number to an even integer will result in a valid answer. For example,

```
PRINT (-3.0) ^2
```

prints the value 9. Raising a number to a floating-point number can cause issues, though. For example,

```
a = -2
```

## Chapter 8: Expressions and Assignments

```
b = 2.000001
PRINT a^b
```

prints NaN, which is correct, since the exponent is not an even integer. The almost identical

```
a = -2
b = 2.0000001
PRINT a^b
```

prints 4, though. That's because the precision for SINGLE values is not large enough to represent the last digit, and b rounds to 2.

The table below shows how zero, infinity and NaN behave for exponentiation. “Any value” refers to any number, including infinity or NaN, that is not listed explicitly in the table.

This...	raised to the power...	gives this...
any value	0	1
NaN	any value	NaN
any value	NaN	NaN
any negative value	any value except even integers	NaN
Infinity (+ or -)	any value	NaN
any value	Infinity	Infinity
any value	-Infinity	0.0

### Bit Shift Left

The symbol for bit shift left is <<.

Bit shift left shifts the binary bits in a two's complement integer left by the specified number of bits. The least significant bits are filled with zero bits. If the number of bits to shift is zero or negative, the value is not changed.

For example, the operation

```
x = 3 << 2
```

shifts the bit pattern 00000011 left by two positions, giving a bit pattern of 00001100, which has a numeric value of 12.

If the first operand is LONG or DOUBLE, the result will be LONG. For all other argument types, the result is INTEGER.

### Bit Shift Right

The symbol for bit shift right is >>.

Bit shift right shifts the binary bits in a two's complement integer right by the specified number of bits. The most significant bits are filled with the original sign bit. If the number of bits to shift is zero or negative, the value is not changed.

For example, the operation

```
x = -6 >> 1
```

shifts the bit pattern 11111010 right by one position, giving a bit pattern of 11111101, which has a numeric value of -3.

If the first operand is LONG or DOUBLE, the result will be LONG. For all other argument types, the result is INTEGER.

**BITNOT**

The symbol for the bitwise not operation is BITNOT.

Bitwise not reverses the binary bits in a two's complement integer.

For example, the operation

```
x = BITNOT 6
```

converts the bit pattern 00000110 to 11111001, which has a numeric value of -7.

If the operand is LONG or DOUBLE, the result will be LONG. For all other argument types, the result is INTEGER.

**BITAND**

The symbol for the bitwise and operation is BITAND.

Bitwise and performs an and operation on the binary bits in two two's complement integers. If the corresponding bits in both integers are 1, the result bit is also 1. If either or both bits are zero, the result bit is also zero.

For example, the operation

```
x = 10 BITAND 6
```

performs an and operation on the bit patterns 00001010 and 00000110, yielding 00000010, which has a numeric value of 2.

If either operand is LONG or DOUBLE, the result will be LONG. For all other argument types, the result is INTEGER.

**BITOR**

The symbol for the bitwise or operation is BITOR.

Bitwise or performs an or operation on the binary bits in two two's complement integers. If the corresponding bits in either integer is 1, the result bit is also 1. If both bits are zero, the result bit is also zero.

For example, the operation

```
x = 10 BITOR 6
```

performs an and operation on the bit patterns 00001010 and 00000110, yielding 00001110, which has a numeric value of 14.

If either operand is LONG or DOUBLE, the result will be LONG. For all other argument types, the result is INTEGER.

**BITXOR**

The symbol for the bitwise exclusive or operation is BITXOR.

Bitwise exclusive or performs an exclusive or operation on the binary bits in two two's complement integers. If the corresponding bits in the integers differ, the result bit is 1. If both bits are the same, the result bit is zero.

For example, the operation

```
x = 10 BITXOR 6
```

performs an and operation on the bit patterns 00001010 and 00000110, yielding 00001100, which has a numeric value of 12.

## Chapter 8: Expressions and Assignments

If either operand is `LONG` or `DOUBLE`, the result will be `LONG`. For all other argument types, the result is `INTEGER`.

### AND

The `AND` operator logically combines two arguments, returning false (an `INTEGER 0`) if either of the arguments is 0, and true (an `INTEGER 1`) if both of the arguments are not zero. Both infinity and NaN (not a number, indicating a result that is not a valid number) are treated as true.

`AND` is generally used with other logical arguments. For example, to check to see if the value `A` lies between `LOW` and `HIGH`, you could use the test

```
IF low < a AND a < high THEN Process(A)
```

This condition first tests to see if `LOW` is less than `A`, then checks to see if `A` is less than `HIGH`. If both conditions are true, `AND` returns true and the subroutine `PROCESS` is called. If either condition is not true, `AND` returns false, and subroutine `PROCESS` is not called.

### OR

The `OR` operator logically combines two arguments, returning false (an `INTEGER 0`) if both of the arguments are 0, and true (an `INTEGER 1`) if either of the arguments are not zero. Both infinity and NaN (not a number, indicating a result that is not a valid number) are treated as true.

`OR` is generally used with other logical arguments. For example, to check to see if a character is either an uppercase or lowercase alphabetic character, you could use this test:

```
a$ = LEFT(line$, 1, 1)
IF a$ >= "A" AND a$ <= "Z" OR a$ >= "a" AND a$ <= "z" THEN
    GetWord(line$)
END IF
```

This condition first tests to see if `a$` is an uppercase character (`a$ >= "A" AND a$ <= "Z"`), then checks to see if `a$` is a lowercase letter (`a$ >= "a" AND a$ <= "z"`). If either of these tests is true, the result of the `OR` operation is true, and `GetWord` is executed. If both conditions are false, the first letter of `line$` is not an alphabetic character, and `GetWord` is not called.

### Comparison Operators

Comparison operators are used to compare two values. They return true (an `INTEGER 1`) if the comparison is true, and false (an `INTEGER 0`) if the comparison is not true. You can compare numbers or strings.

For example, `4.9 < 5` is true, so the result is an integer 1. `"Fred" >= "Sam"` is not true, so the result is an integer 0.

There are six comparison operators. The symbol, what the operation does, some examples and the result of the compare are shown in the table below.

Symbol	Operation	Example	Result
<	less than	-3 < 6	1
		6.1 < 6.1	0
		9 < 6	0
<=	less than or equal	-4 <= 4	1
		7 <= 7	1
		43 <= 16	0
>	greater than	2 > 7	0
		-10 > -10	0
		-10 > -20	1
>=	greater than or equal	2 >= 8	0
		3.14 >= 3.14	1
		6.1 >= 6.0	1
=	equal	9 = 9	1
		9 = -9	0
<>	not equal	3 <> 3	0
		4 <> 3	1

Some BASICs allow the multi-character comparison operators with the characters in any order. For example, >< is allowed instead of <>. Spaces are also allowed between the characters. techBASIC supports these conventions, but you should generally use the standard form for the operations shown in the table.

Numbers are compared using normal rules for arithmetic. Strings, on the other hand, are compared more or less by alphabetical order. For example, “Fred” is less than “Sam”. There are some surprises, though, because the characters are compared using their ASCII character orders. Uppercase letters are always less than lowercase letters, so “fred” is greater than “Sam”. The ASCII character chart also includes characters other than alphabetical characters. The ASCII character chart is shown in Appendix B.

If the first letter of a string matches, the compare continues with the next letter. For example, “Sam” is less than “Susan”. If all of the characters match, but the strings have different lengths, the longer string is greater than the shorter string. This means that “Fred” is less than “Frederick”.

Of course, if all of the characters match and the strings are the same length, the strings are equal.

---

## Terms

The first part of this chapter dealt with expressions from the standpoint of traditional mathematical operations like addition, comparisons, and parentheses, just as these operations are generally used in algebra. In algebra, these operations use numbers or variables as arguments. For example,

$$4 + Y$$

is a perfectly reasonable statement in algebra, and it’s perfectly acceptable in BASIC, too.

In BASIC, the numbers that are used by the mathematical operations covered earlier in this chapter are called terms, and they can be many things besides just numbers or variables. In each case, though, the term is fully evaluated, giving a resultant number or string, before any of the operations discussed earlier is performed.

## Constants

Constants include numbers, arrays and strings. Numbers can be integers, long integers, single-precision floating-point or double-precision floating-point. Integers and long integers can be written in either standard decimal form or using hexadecimal notation.

Chapter 4 describes the format and limitations for constants.

## Chapter 8: Expressions and Assignments

### Unary Math Operations

You can use a `-` operation before any term. This operation doesn't require a number to the left. In affect, the BASIC term

```
-v
```

works as if you typed

```
0 - v
```

When it is used this way, the `-` operation is technically referred to as unary subtraction to distinguish it from the similar subtraction operator.

You'll generally use this operation to indicate a negative constant or to change the sign of a variable, as in

```
x = -4
y = -x
a = [1, 2, 3]
b = -a
```

but it's perfectly legal to use the operation in the middle of an expression. For example, it is legal to write

```
z = x - -4
```

The effect is exactly the same as

```
z = x + 4
```

Eliminating the extra operation by using `+` rather than two `-` operators saves a small amount of space and time. Still, there are rare cases where the program makes more sense if the natural operations are left in place, and if clarity is more important than a byte of space and a little speed, it might make sense to use the unary subtraction operator in the middle of an expression.

There is also a unary operation for `+`. It rarely makes sense to use it in a program, since it doesn't actually do anything but occupy space and time.

### NOT

`NOT` is the unary negation operation for logical values. BASIC uses numbers for logical values, assigning false the value of 0 and treating any number other than 0 as true. The `NOT` operation returns true if its operand is false, and false if the operand is true. From a strictly mathematical standpoint, `NOT` returns the `INTEGER` 0 if the operand is nonzero, and it returns the `INTEGER` 1 if the operand is zero.

In practice, `NOT` is usually used with logical operations or variables used to store logical values. A common example uses a variable `DONE%` to indicate if a loop has completed its work. The loop continues until `DONE%` is true.

```
done% = 0
WHILE NOT done%
    HandleMovement
WEND
```

### Array Subscripts

Elements of an array are selected by enclosing the array subscripts in parentheses after the array name. The subscripts are expressions, and can contain functions and other array elements. In all cases, the subscripts are evaluated first, in left to right order, then the corresponding element of the array is extracted.

If an array has more than one subscript, the subscripts are separated by commas.

Here is an example of arrays in an expression that computes the length of a multidimensional vector.

```
length = 0.0;
FOR i% = LBOUND(vector, 1) TO UBOUND(vector, 1)
    length = length + vector(i%)*vector(i%)
NEXT
length = SQR(length)
```

This is for illustration; in practice it would be faster to simply take the square root of the dot product:

```
length = SQR(DOT(vector))
```

### Using BASIC Functions

BASIC has many built in functions. These functions return either a number or string. Most of the functions are described later in this chapter, broken down into sections based on whether they deal primarily with strings, arrays or numbers.

Most functions require one or more arguments, called parameters. For example, the SQR function returns the square root of another number. Taking the square root of 4 is written like this:

```
SQR(4)
```

The parameter is 4.

This entire sequence is a single term. The parameter is an expression, and it can include anything you see in any other expression, including calls to other BASIC functions. The parameters are evaluated first, then the function is called, and finally the value returned by the function is used in the expression.

In some rare cases, the actual order in which the parameters are evaluated can affect the way the program operates. It's generally best to rethink the program so this doesn't happen. For the rare cases where it matters, techBASIC evaluates parameters in left to right order.

### Using FUNCTION Functions

Like most modern BASICs, techBASIC supports user-defined multi-line functions using the `FUNCTION` statement. Chapter 12 tells how to create these functions, and gives examples of how they are used. Functions defined with the `FUNCTION` statement behave just like built in BASIC functions in an expression.

---

## L-Values

In a few places in this manual, you will see a reference to something called an l-value. This is a rather descriptive term borrowed from the C language. It means any expression that can appear on the left side of an equal sign in a `LET` statement. In practice, it's any expression that gives the location of a value in memory.

The remainder of this section gives a very technical description of just what is and is not an l-value. Whether you wade through this description to get a full understanding of l-values or not, keep in mind that the concept is simpler than the description. An l-value is any expression that gives a place where a value is stored in memory.

L-Values are required for the location to store a value with a `LET` or `MAT` statement and for some kinds of parameters passed to subroutines and functions.

## Chapter 8: Expressions and Assignments

The simplest l-value is the name of a variable. Constants, such as 4.5, are not l-values. Think of it this way: you can store 7.1 in the variable X, but you can't store 7.1 in the number 4.5.

Arrays are a series of l-values, and an element of an array is an l-value. For example, A(X) is an l-value. An array is also an l-value.

All other expression operations are not l-values. Even something as innocent as enclosing a value in parentheses or putting a + operation in front of an l-value yields an expression that is not an l-value. For example,

```
LET +x% = 4 : ! Illegal!
```

is not a legal BASIC statement, since the expression to the left of the = operator in a LET statement must be an l-value.

---

## The Assignment Statements

---

### [ LET ] l-value '=' expression

The expression to the right of the = operation is evaluated and stored in the location given by the l-value.

LET is optional, and is almost always omitted from BASIC programs. It has been part of the language since the original implementation, though, and has been kept by virtually every implementation so old programs will still work.

If the expression yields a number of any kind, and the l-value is a different kind of number, the number is converted to the proper type before it is stored. If the expression is a floating-point value and the l-value is an INTEGER or LONG value, the number is truncated to the largest integer that is less than or equal to the value of the expression. For example,

```
i% = 3.9
PRINT i%
```

prints 3.

If you assign a value to an INTEGER or LONG that is too long for the variable, a math error is generated. For example,

```
r = 40000.0
i% = r
```

generates an error.

If you assign a DOUBLE to a SINGLE, and the DOUBLE value is too large to be represented as a SINGLE, the result is infinity. The lines

```
d# = 1D40
d = d#
PRINT d#, d
```

prints

```
1.0000000E40      Infinity
```

Strings can also be assigned to other strings. Numbers cannot be assigned to strings, and strings cannot be assigned to numbers.

The snippet shows a few examples that you can experiment with to see how this works.



Snippet

```
s$ = "Hello, world."
h$ = LEFT(s$, 5)
PRINT h$
x = 3.2
y = 1.6
length = SQR(x*x + y*y)
PRINT length
```

**[ MAT ] l-value '=' expression**

The array expression to the right of the = operation is evaluated and stored in the array given by the l-value.

MAT is optional, and is almost always omitted from techBASIC programs. It has been part of the language since the original implementation, though, and has been kept by virtually every implementation so old programs will still work. Many current implementations of BASIC that support MAT still require its use.

Array assignment is valid if the array to assign has the same number of subscripts as the array the result is stored in. It is not required that each subscript have the same range of elements, or even the same number of elements. Assigning an array can change the number of elements in the array. While this may seem a little odd, this is a very powerful feature in BASIC that lets you concentrate on the matrix algebra, and not worry so much about the exact number of range of subscripts. Combined with the fact that BASIC will predefine any array with the appropriate number of subscripts the first time it is used, fairly natural-looking programs are possible. Here is an example that shows this concept in action.

```
A = [[1, 1, 2],
      [3, 5, 8],
      [13, 21, 34]]
B = [[3],
      [5],
      [7]]
X = A*B
PRINT X
```

This is not a fragment of a program—despite the lack of DIM statements or leading MAT tokens, this is a fully functioning BASIC program that multiplies a 3x3 matrix by a 3x1 matrix, printing the resulting 3x1 matrix, which is

```
22
90
382
```

Assigning one array to another can change the number of elements in each subscript, but it will never change the number of subscripts or the lower bound for the subscripts. In situations where it might not be clear, or could change based on program flow, use the UBOUND function to determine the resulting size of the array.

Assigning an array to a variable that holds a number, or a number to an array, is not allowed.

---

## Mathematical Functions

---

**ABS '(' expression ')'**

Returns the absolute value of the argument.

The argument must be a numeric type. The type of the result is the same as the type of the argument. For example, if the argument is SINGLE, the result is SINGLE; if the argument is INTEGER, so is the result.

The absolute value is the same as the argument if the argument is zero or positive, and the negative of the argument if the argument is negative. For example, ABS (4) is 4, and so is ABS (-4) .

The absolute value of negative infinity is infinity. The absolute value of NaN is NaN.

**ACOS '(' expression ')'**

Returns the arc cosine of the argument. The angle is returned in radians.

If the argument is `DOUBLE`, the result is also `DOUBLE`. For `SINGLE`, `LONG` and `INTEGER` arguments, the result is `SINGLE`.

The arc cosine is the number whose cosine is the same as the argument. The result of the `ACOS` function is the angle between the X axis and a line from the origin and a point in the X-Y plane.

The `ACOS` function always returns a result between 0.0 and  $\pi$ .

Snippet

```
theta = ACOS(x/hypotenuse)
```

**ANGLE '(' expression ',' expression ')'**

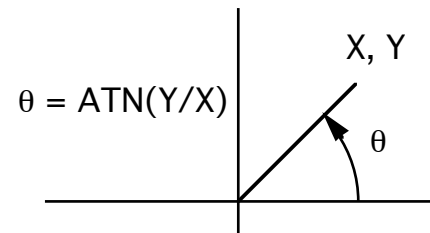
Returns the angle between the X axis and a line to the given point. The first argument is the X coordinate of the point, while the second argument is the Y coordinate of the point. Both may not be zero. The angle is returned in radians, and is in the range  $-\pi$  to  $\pi$ .

If either of the arguments is `DOUBLE`, the result is also `DOUBLE`. For `SINGLE`, `LONG` and `INTEGER` arguments, the result is `SINGLE`.

The angle is equivalent to a two-argument arc tangent, returning a number whose tangent is the same as the ratio of the arguments.

Snippet

```
theta = ANGLE(x, y)
```

**ASIN '(' expression ')'**

Returns the arc cosine of the argument. The angle is returned in radians.

If the argument is `DOUBLE`, the result is also `DOUBLE`. For `SINGLE`, `LONG` and `INTEGER` arguments, the result is `SINGLE`.

The arc sine is the number whose sine is the same as the argument. The result of the `ASIN` function is the angle between the X axis and a line from the origin and a point in the X-Y plane.

The `ASIN` function always returns a result between  $-\pi/2$  and  $\pi/2$ .

Snippet

```
theta = ASIN(y/hypotenuse)
```

**ATAN '(' expression ')'**

Returns the arc tangent of the argument. The angle is returned in radians.

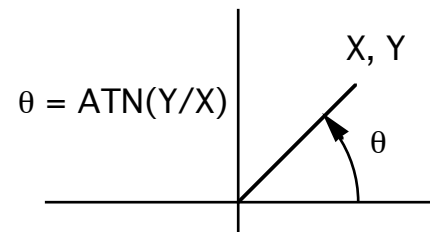
If the argument is `DOUBLE`, the result is also `DOUBLE`. For `SINGLE`, `LONG` and `INTEGER` arguments, the result is `SINGLE`.

The arc tangent is the number whose tangent is the same as the argument. For any particular argument, there are actually an infinite number of answers. For example, the tangent of  $\pi/4$  (roughly 0.785398) is 1, so the arc tangent of 1.0 is approximately 0.785398. As with any angle, though, adding  $2\pi$  to the angle gives an equivalent angle.

The result of the `ATAN` function is the angle between the X axis and a line from the origin and a point in the X-Y plane. The value of the argument is the Y coordinate of the point divided by the X coordinate.

The `ATAN` function always returns a result between 0.0 and  $\pi/2$  for positive arguments, and 0.0 and  $-\pi/2$  for negative values. It can't tell if a positive number should be a point where both X and Y are positive, or a reflected angle where both X and Y are negative.

See `ANGLE` for a way to get the arc tangent with a range of  $-\pi$  to  $\pi$ .



Snippet

```
theta = ATAN(y/x)
```

---

**CDBL '(' expression ')'**

Converts any numeric argument to a DOUBLE value.

CDBL is generally used in an expression to force the calculation to be performed using double-precision floating-point operations. For example, if you are about to multiply two real values, and would like to maintain as many significant digits as possible, you could use CDBL to force one of the arguments to DOUBLE before doing the multiply, like this:

```
product# = CDBL(x)*y
```

This gives a potentially different result from the statement

```
product# = x*y
```

Without CDBL, the calculation is performed using a single-precision multiply, truncating the result to approximately 7 significant decimal digits. This result is extended to a double-precision value. Using CDBL, *x* is extended to double-precision immediately. This also forces *Y* to double-precision. See *Binary Conversions*, earlier in this chapter, for the complete explanation of why. The multiplication produces a result that has about 14 significant decimal digits.

At first glance, it might seem like these extra digits have no meaning. In fact, there are many numerically sensitive algorithms that depend on exactly this kind of extra precision at just the right point in the calculation. And the extra digits are real in at least one sense—computer based multiplication always doubles the number of significant digits, but these are generally discarded before you see the result.

You don't need this function when assigning a value to a DOUBLE variable. Conversion between numeric types is generally automatic. CDBL is only needed for extraordinary situations like the one described, where the precision of a number must be changed within a calculation.

---

**CINT '(' expression ')'**

Converts any numeric argument to an INTEGER value.

CINT is used to force an integer value at a particular spot in a calculation. One example is finding the remainder from division. As long as the numbers involved are not larger than the range of integers, this will return the remainder from dividing *A%* by *B%*:

```
a% = 15
b% = 4
r% = a% - b%*CINT(a%/b%)
```

Without CINT, the division returns 3.75, the multiplication yields 15.0, and *r%* is set to 0. With the CINT operation, 3.75 is converted to the integer 3, and *r%* is set to 3.

When the argument is a floating-point number, CINT truncates the value to convert to an integer. The result is the largest integer that is less than or equal to the floating-point value. For example, CINT(4.6) gives 4, while CINT(-4.6) gives -5.

Converting a number that is too large to an integer gives an error. The valid range for integers is -32768 to 32767.

---

**CLNG '(' expression ')'**

Converts any numeric argument to a LONG value.

CLNG is sometimes used to convert INTEGER values to LONG, extending the precision of a calculation. You might do this to avoid an error due to an integer overflow. For example, in the expression

## Chapter 8: Expressions and Assignments

```
I = 30000
L& = I + I
```

you know the integer value will overflow. Using `CLNG` to force one of the terms to `LONG` right away avoids the overflow.

```
I = 30000
L& = CLNG(I) + I
```

`CLNG` can also be used to convert floating-point arguments to `LONG` to force integer math. See `CINT` for an example based on this idea.

When the argument is a floating-point number, `CLNG` truncates the value to convert to an integer. The result is the largest integer that is less than or equal to the floating-point value. For example, `CLNG(4.6)` gives 4, while `CLNG(-4.6)` gives -5.

Converting a number that is too large to fit into a long integer gives an error. The valid range for long integers is -2147483648 to 2147483647.

---

### **COS '(' expression ')'**

Returns the cosine of the argument. The argument is expressed in radians.

If the argument is `DOUBLE`, the result is also `DOUBLE`. For `SINGLE`, `LONG` and `INTEGER` arguments, the result is `SINGLE`.

The `COS` function is not accurate for very large angles. By very large angles, we mean angles larger than about 1300 radians for `SINGLE` arguments, and 2E8 radians for `DOUBLE` arguments, although the accuracy drops off gradually as the angle increases. For this reason, it is best to keep angles between  $-2\pi$  and  $2\pi$  whenever possible. For very large arguments, `COS` always returns 0.0.

Refer to any book that covers trigonometry for a discussion of the cosine.

#### Snippet

```
x = length * COS(theta)
```

---

### **COSH '(' expression ')'**

Returns the hyperbolic cosine of the argument.

If the argument is `DOUBLE`, the result is also `DOUBLE`. For `SINGLE`, `LONG` and `INTEGER` arguments, the result is `SINGLE`.

Refer to any book that covers trigonometry for a discussion of the hyperbolic cosine.

#### Snippet

```
x = COSH(theta)
```

---

### **COT '(' expression ')'**

Returns the cotangent of the argument. The argument is expressed in radians.

If the argument is `DOUBLE`, the result is also `DOUBLE`. For `SINGLE`, `LONG` and `INTEGER` arguments, the result is `SINGLE`.

`COT(x)` is equivalent to `SIN(x) / COS(x)`.

Refer to any book that covers trigonometry for a discussion of the cotangent.

---

### **CSC '(' expression ')'**

Returns the cosecant of the argument. The argument is expressed in radians.

If the argument is `DOUBLE`, the result is also `DOUBLE`. For `SINGLE`, `LONG` and `INTEGER` arguments, the result is `SINGLE`.

`CSC(x)` is equivalent to `1/SIN(x)`.

Refer to any book that covers trigonometry for a discussion of the cosecant.

---

**CSNG '(' expression ')'**

Converts any numeric argument or pointer argument to a SINGLE value.

---

**DEG '(' expression ')'**

Converts the argument from radians to degrees.

If the argument is DOUBLE, the result is also DOUBLE. For SINGLE, LONG and INTEGER arguments, the result is SINGLE.

DEG (x) is equivalent to  $x * 180 / \pi$ .

---

**EXP '(' expression ')'**

Returns the natural exponent of the argument.

If the argument is DOUBLE, the result is also DOUBLE. For SINGLE, LONG and INTEGER arguments, the result is SINGLE.

The natural exponent is the result of raising e to a power. The number known as e is approximately 2.71828. The exponent is also the inverse of the natural logarithm, LOG. For values that are valid for both functions, EXP (LOG (x)) always returns x.

The EXP function is frequently used to manipulate powers, such as interest rates. For example, if you earn 4% per year on a passbook savings account for 10 years, the value of your initial investment m is given by

$$v = m * \text{EXP}(10.0 * \text{LOG}(1.04))$$

Snippet

```
FUNCTION Power10(x)
! Returns 10 raised to a power.
Power10 = EXP(x*LOG(10.0))
END FUNCTION
```

---

**INT '(' expression ')'**

Returns the largest integer value that is less than or equal to the argument.

For INTEGER and LONG arguments, INT returns the argument. The value returned is still an INTEGER for INTEGER arguments, and LONG for LONG arguments.

For SINGLE and DOUBLE arguments, the value returned is still SINGLE or DOUBLE, but any fraction part is lost. The value returned is the largest integer that is less than or equal to the argument.

The table shows some results for various floating-point arguments.

expression	result
INT(5.4)	5.0
INT(3.99)	3.0
INT(-0.1)	-1.0
INT(-10.9)	-11.0

---

**LOG '(' expression ')'**

Returns the natural logarithm of the argument.

If the argument is DOUBLE, the result is also DOUBLE. For SINGLE, LONG and INTEGER arguments, the result is SINGLE.

The natural logarithm is not defined for zero or negative arguments. If the argument is less than or equal to zero, LOG returns NaN.

Snippet

```
PRINT LOG(10)
```

---

**LOG10 '(' expression ')'**

Returns the base 10 logarithm of the argument.

## Chapter 8: Expressions and Assignments

If the argument is `DOUBLE`, the result is also `DOUBLE`. For `SINGLE`, `LONG` and `INTEGER` arguments, the result is `SINGLE`.

The logarithm is not defined for zero or negative arguments. If the argument is less than or equal to zero, `LOG10` returns `NaN`.

### Snippet

```
PRINT LOG10(10)
```

---

### **LOG2 '(' expression ')'**

Returns the base 2 logarithm of the argument.

If the argument is `DOUBLE`, the result is also `DOUBLE`. For `SINGLE`, `LONG` and `INTEGER` arguments, the result is `SINGLE`.

The logarithm is not defined for zero or negative arguments. If the argument is less than or equal to zero, `LOG2` returns `NaN`.

### Snippet

```
PRINT LOG2(10)
```

---

### **PI**

Returns  $\pi$  to double precision.

### Snippet

```
PRINT SIN(PI/4)
```

---

### **RAD '(' expression ')'**

Converts the argument from degrees to radians.

If the argument is `DOUBLE`, the result is also `DOUBLE`. For `SINGLE`, `LONG` and `INTEGER` arguments, the result is `SINGLE`.

`RAD(x)` is equivalent to  $x * \pi / 180$ .

---

### **RND '(' expression ')'**

`RND` returns a pseudo-random number greater than or equal to zero and less than 1.0. The value returned is always `SINGLE`.

While the rest of this discussion refers to the values `RND` returns as random numbers, they really aren't random. Pseudo-random numbers is the technical term that refers to functions like `RND`, which return sequences of numbers with no apparent pattern. Of course, there is a pattern—but it's a pattern that can't be detected by a series of tests for randomness. The result is a series of numbers that can be used for tasks like shuffling a deck of cards, and that will produce results as good as shuffling by hand.

Each time `RND` returns a value, the value is computed using a formula, and is based on the last value returned. The original value determines the sequence of numbers you get. This original value is called the seed. There is a way to specify the seed for `RND`, which we'll look at in a moment. In most cases, though, you should let `RND` pick its own seed. It bases the seed on the current date and time.

There are three ways to call `RND`. If the argument is a positive value, `RND` returns a random number. Subsequent calls return other seemingly unrelated random numbers.

If you call `RND` with an argument of zero, it returns the same value it returned on the previous call. This is a useful shortcut when you need to use the same random value in several places in an program.

If you call `RND` with a negative argument, the argument is used as a new seed for the random number generator. After producing a series of numbers, calling `RND` with the same negative argument will cause `RND` to regenerate the same sequence of numbers. This is a very useful feature when you are debugging a program that uses `RND`: By temporarily placing a line like

```
r = RND(-1.0)
```

at the start of the program, it will always generate the same series of numbers, making bugs easier to reproduce.

While RND is pretty good for simple uses, see rand and normal in the Math class for more sophisticated random number generation.

Snippet

```
! Print 10 random numbers
FOR i = 1 TO 10
  PRINT RND(1.0)
NEXT
! Print 10 different random numbers based on our seed.
PRINT RND(-1.0)
FOR i = 1 TO 9
  PRINT RND(1.0)
NEXT
! Print the same 10 random numbers again.
PRINT RND(-1.0)
FOR i = 1 TO 9
  PRINT RND(1.0)
NEXT
```

---

**SEC '(' expression ')'**

Returns the secant of the argument. The argument is expressed in radians.

If the argument is DOUBLE, the result is also DOUBLE. For SINGLE, LONG and INTEGER arguments, the result is SINGLE.

SEC(x) is equivalent to 1/COS(x).

Refer to any book that covers trigonometry for a discussion of the secant.

---

**SGN '(' expression ')'**

Returns the sign of the value, or one of -1, 0 or 1, depending on the argument. If the argument is zero, SGN returns 0. If the argument is less than zero, SGN returns -1. If the argument is greater than zero, SGN returns 1.

The type of the result is the same as the type of the argument. For example, if the argument is SINGLE, the result is SINGLE; if the argument is INTEGER, so is the result.

Snippet

```
! Jump to various spots based on the sign of the number
ON 2 + SGN(x) GOTO 10, 20, 30
```

---

**SIN '(' expression ')'**

Returns the sine of the argument. The argument is expressed in radians.

If the argument is DOUBLE, the result is also DOUBLE. For SINGLE, LONG and INTEGER arguments, the result is SINGLE.

The SIN function is not accurate for very large angles. By very large angles, we mean angles larger than about 1300 radians for SINGLE arguments, and 2E8 radians for DOUBLE arguments, although the accuracy drops off gradually as the angle increases. For this reason, it is best to keep angles between  $-2\pi$  and  $2\pi$  whenever possible. For very large arguments, SIN always returns 0.0.

Refer to any book that covers trigonometry for a discussion of the sine.

Snippet

```
y = length*SIN(theta)
```

---

**SINH '(' expression ')'**

Returns the hyperbolic sine of the argument.

If the argument is DOUBLE, the result is also DOUBLE. For SINGLE, LONG and INTEGER arguments, the result is SINGLE.

Refer to any book that covers trigonometry for a discussion of the hyperbolic sine.

### Snippet

```
x = SINH(theta)
```

---

### SQR '(' expression ')'

Returns the square root of the argument.

If the argument is `DOUBLE`, the result is also `DOUBLE`. For `SINGLE`, `LONG` and `INTEGER` arguments, the result is `SINGLE`.

The square root of a number is the number that, multiplied by itself, gives the argument. For example, the square root of 4 is 2, since  $2 * 2$  is 4.

The square root function is not defined for negative numbers, and returns NaN if the argument is negative.

### Snippet

```
hypotenuse = SQR(x*x + y*y)
```

---

### TAN '(' expression ')'

Returns the tangent of an angle. The argument is expressed in radians.

If the argument is `DOUBLE`, the result is also `DOUBLE`. For `SINGLE`, `LONG` and `INTEGER` arguments, the result is `SINGLE`.

The `TAN` function is not accurate for very large angles. By very large angles, we mean angles larger than about 1300 radians for `SINGLE` arguments, and  $2E8$  radians for `DOUBLE` arguments, although the accuracy drops off gradually as the angle increases. For this reason, it is best to keep angles between  $-2\pi$  and  $2\pi$  whenever possible. For very large arguments, `TAN` always returns NaN.

The tangent tends toward infinity as the argument approaches  $\pi/2$ . If the argument gets too close to  $\pi/2$ , `TAN` returns infinity. If the argument gets too close to  $-\pi/2$ , the result will be -infinity.

Refer to any book that covers trigonometry for a discussion of the tangent.

### Snippet

```
altitude = baseLine*TAN(theta)
```

---

### TANH '(' expression ')'

Returns the hyperbolic tangent of the argument.

If the argument is `DOUBLE`, the result is also `DOUBLE`. For `SINGLE`, `LONG` and `INTEGER` arguments, the result is `SINGLE`.

Refer to any book that covers trigonometry for a discussion of the hyperbolic tangent.

### Snippet

```
x = TANH(theta)
```

---

## Array Functions

---

### CON '(' expression [ ',' expression ] \* ')'

`CON` returns a constant array, where each element of the array is 1. The array consists of `DOUBLE` values.

Each expression value is evaluated and converted to an `INTEGER` by truncation. See *Unary Conversions*, earlier in this chapter, for a detailed description of this process. There must be at least one expression, but as many expressions as desired can be specified, separated by commas.

The resulting array will have the same number of subscripts as the number of expressions. Each subscript will have the number of elements specified by the expression. The lower bound on each subscript will be 1 unless `BASE` has been used to set the default lower bound to 0, in which case the lower bound will be 0. The upper bound is the lower bound plus the number of elements in the subscript.



Snippet

```
! Initialize a 10 element array with the value 1.
A = CON(10)
! Set up a matrix with 3 rows and 5 columns. Each element is 7.2.
B = 7.2*CON(3, 5)
```

**DET '(' array-expression ')'**

Returns the determinant of a square matrix.

The array passed as the argument to the DET function must have two subscripts, and each subscript must have the same number of elements. The upper and lower bounds do not need to match, so long as the number of elements is the same. If the array is an array of DOUBLE, the result type is also DOUBLE. For SINGLE, LONG and INTEGER arrays, the result is SINGLE.

Snippet

```
! Find the volume of the parallelepiped whose edges are defined by the
! vectors (1.2, 1.7, 1.8), (1.7, 2.5, 2.5), (3.1, 1.4, 1.5).
A = [[1.2, 1.7, 1.8],
     [1.7, 2.5, 2.5],
     [3.1, 1.4, 1.5]]
PRINT DET(A)
```

**DOT '(' array-expression ',' array-expression ')'**

Returns the dot product of two vectors.

The arrays passed as arguments must have a single subscript, and must have the same number of elements. The upper and lower bounds do not need to match, so long as the number of elements is the same. If either array is an array of DOUBLE, the result type is also DOUBLE. For SINGLE, LONG and INTEGER arrays, the result is SINGLE.

The dot product of two vectors is obtained by multiplying each element by the corresponding element in the other array, then adding the results.

Snippet

```
! Find the length of the vector 3.7i + 4.8j + 1.3k.
v = [3.7, 4.8, 1.3]
PRINT SQR(DOT(v, v))
```

**IDN '(' expression [ ',' expression ] ')'**

IDN returns an identity matrix. An identity matrix is a square matrix with the value 1 along the main diagonal, and 0 everywhere else. The array consists of DOUBLE values.

Each expression value is evaluated and converted to an INTEGER by truncation. See *Unary Conversions*, earlier in this chapter, for a detailed description of this process. There must be at least one expression, which specifies the number of elements in each subscript of the array. If there are two expressions, they must yield the same integer size.

Snippet

```
! Set up a 3x3 identity matrix.
I = IDN(3)
```

**INV '(' array-expression ')'**

INV returns the inverse of a matrix. The result is always a matrix containing double values.

The argument must be a square matrix, e.g. a two-dimensional array where the number of elements is the same for both subscripts.

The inverse of a matrix is the matrix that, multiplied by the original matrix, yields the identity matrix. A runtime error, "The matrix is singular.", is thrown if the inverse cannot be calculated.

## Chapter 8: Expressions and Assignments

### Snippet

```
! Enter the X and Y values. Any number may be used, so long as there are
! the same number of elements in each array.
x = [60, 61, 62, 63, 65]
y = [3.1, 3.6, 3.8, 4, 4.1]

! Solve for the regression coefficients.
n% = UBOUND(x, 1)
sumX = DOT(CON(n%), x)
sumY = DOT(CON(n%), y)
sumXY = DOT(x, y)
sumXX = DOT(x, x)
A = [[n%, sumX],
      [sumX, sumXX]]
b = [[sumY], [sumXY]]
s = INV(A)*b

! Try a sample solution.
PRINT "For X=64, Y is approximately "; s(1, 1) + s(2, 1)*64
```

---

### **LBOUND** '(' identifier ',' expression ')'

LBOUND returns the lower bound for an array subscript.

The identifier must be the name of an existing array. The expression value is evaluated and converted to an INTEGER by truncation. See *Unary Conversions*, earlier in this chapter, for a detailed description of this process. The expression specifies the subscript of the array for which to find the bound, starting with 1 for the first subscript. It is an error if the value is less than 1 or greater than the number of subscripts in the array.

The lower bounds defaults to 1. If BASE is used to set the default lower bound to 0, the default is 0. Arrays can also be dimensioned with a specific lower bound using the DIM statement.

### Snippet

```
FOR i% = LBOUND(a, 1) TO UBOUND(a, 1)
  a(i%) = i%
NEXT
```

---

### **SIZE** '(' identifier [ ',' expression ] ')'

SIZE returns the number of elements for an array subscript, or the total number of elements in the entire array.

The identifier must be the name of an existing array. If present, the expression value is evaluated and converted to an INTEGER by truncation. See *Unary Conversions*, earlier in this chapter, for a detailed description of this process. The expression specifies the subscript of the array for which to find the size, starting with 1 for the first subscript. It is an error if the value is less than 1 or greater than the number of subscripts in the array.

If the expression is omitted, the value returned is the total number of elements in all subscripts of the array.

SIZE for a specific subscript is always equivalent to UBOUND - LBOUND + 1 for the same subscript. SIZE with no expression is the equivalent of the product of SIZE for each subscript in the array.

---

### **TRN** '(' array-expression ')'

TRN returns the transpose of a matrix. The result is always the same type as the argument.

The argument must be a two-dimensional array.

The transpose of a matrix is a matrix with each element flipped about the main diagonal. The transpose of

```
[[ 1, 2, 3, 4],
 [ 5, 6, 7, 8],
 [ 9, 10, 11, 12]]
```

is

```
[[ 1, 5, 9],
 [ 2, 6, 10],
 [ 3, 7, 11],
 [ 4, 8, 12]]
```

---

**UBOUND** '(' identifier ',' expression ')'

UBOUND returns the upper bound for an array subscript.

The identifier must be the name of an existing array. The expression value is evaluated and converted to an `INTEGER` by truncation. See *Unary Conversions*, earlier in this chapter, for a detailed description of this process. The expression specifies the subscript of the array for which to find the bound, starting with 1 for the first subscript. It is an error if the value is less than 1 or greater than the number of subscripts in the array.

Snippet

```
FOR i% = LBOUND(a, 1) TO UBOUND(a, 1)
  a(i%) = i%
NEXT
```

---

**ZER** '(' expression [ ',' expression ]\* ')'

ZER returns an array where each element of the array is 0. The array consists of `DOUBLE` values.

Each expression value is evaluated and converted to an `INTEGER` by truncation. See *Unary Conversions*, earlier in this chapter, for a detailed description of this process. There must be at least one expression, but as many expressions as desired can be specified, separated by commas.

The resulting array will have the same number of subscripts as the number of expressions. Each subscript will have the number of elements specified by the expression. The lower bound on each subscript will be 1 unless `BASE` has been used to set the default lower bound to 0, in which case the lower bound will be 0. The upper bound is the lower bound plus the number of elements in the subscript.

Snippet

```
! Initialize a 3 element vector with the value 0.
a = ZER(3)
! Set up a matrix with 3 rows and 5 columns.
b = ZER(3, 5)
! Initialize a 3x4x5 array with zero.
c = ZER(3, 4, 5)
```

---

## String Functions

---

**ASC** '(' string-expression ')'

Returns the value for the first character in the string. If there are no characters in the string, ASC returns 0.

Characters are represented internally as a number. For example, the character A is stored as the number 65. The ASC function returns this number.

ASC supports the Unicode character set; the ASCII character set is a subset of the Unicode character set.

See Appendix B for a complete list of the ASCII character set.

See also `CHR`, which returns the ASCII character corresponding to a given number.

---

### Snippet

```
FUNCTION toUpper(s$) AS STRING
! Return the string as uppercase letters.
! NOTE: This illustrates the ASC function, but UCASE is a better
! way to conver a string to uppercase letters.
s2$ = ""
WHILE LEN(s$) > 0
    c$ = LEFT(s$, 1)
    s$ = RIGHT(s$, LEN(s$) - 1)
    IF (c$ >= "a") AND (c$ <= "z") THEN
        c$ = CHR(ASC("A") + ASC(C$) - ASC("a"))
    END IF
    s2$ = s2$ & c$
WEND
toUpper = s2$
END FUNCTION
```

---

### **CHR '(' expression ')'**

### **CHR\$ '(' expression ')'**

Returns the character for a given number. The character is returned as a string of length 1.

The expression value is evaluated and converted to an **INTEGER** by truncation. See *Unary Conversions*, earlier in this chapter, for a detailed description of this process.

Characters are represented internally as a number. For example, the character A is stored as the number 65. Given the number, **CHR** returns the character.

The character value for 0 is used internally to mark the end of a string. **CHR**(0) returns a string of length 0.

See Appendix B for a complete list of the ASCII character set.

See also **ASC**, which returns the number the first character of a string. There is also a code sample showing **CHR** and **ASC** used together.

---

### **HEX '(' expression ')'**

Returns the hexadecimal value of the expression as a string.

**BYTE** and **INTEGER** values are returned as a four-digit hexadecimal value using uppercase letters. For example, **HEX**(1000) returns "03E8". **LONG** values are returned as eight-digit hexadecimal strings. **SINGLE** values are first converted to **INTEGER**, then returned as before; **DOUBLE** values are first converted to **LONG**.

---

### **LCASE '(' string-expression ')'**

### **LCASE\$ '(' string-expression ')'**

Returns the input string with all uppercase characters converted to lowercase.

### Snippet

```
! Compare two strings of different case for equality.
INPUT reply$
IF LCASE(reply$) = "yes" THEN
    PRINT "Yes"
ELSE
    PRINT "No"
END IF
```

---

### **LEFT '(' string-expression ',' expression ')'**

### **LEFT\$ '(' string-expression ',' expression ')'**

Returns the leftmost characters in a string.

The second parameter is evaluated and converted to a LONG. See *Unary Conversions*, earlier in this chapter, for a detailed description of this process. If this value is negative or larger than the number of characters in the string that is passed as the first parameter, LEFT returns the string. If this value is positive and less than the number of characters in the string, the specified number of characters is returned, beginning with the first character of the string.

One use of LEFT is to peel characters from a string, processing them one by one. See the code snippet for ASC for an example.

Snippet

```
! Remove the first word from S$.
i% = 1
WHILE (i% < LEN(s$)) AND (MID(s$, i%, 1) <> " ")
    i% = i% + 1
WEND
w$ = LEFT(s$, i% - 1)
```

---

**LEN '(' string-expression ')'**

Returns the number of characters in a string. The terminating null character that marks the end of the string is not a part of the string, and is not counted by LEN. The number of characters is returned as a LONG.

techBASIC uses the character CHR(0) to mark the end of a string. If you imbed this character in a string, LEN will return the number of characters appearing before this character.

Snippet

```
l% = LEN(s$)
```

---

**LTRIM '(' string-expression ')'**

**LTRIM\$ '(' string-expression ')'**

Returns the string with any leading space characters removed from the string.

For example,

```
LTRIM("    Hello")
```

returns the string "Hello".

---

**MID '(' string-expression ',' expression ',' expression ')'**

**MID\$ '(' string-expression ',' expression ',' expression ')'**

Returns characters from any position in a string.

The last two expressions give the index of the first character to return, counting from one, and the number of characters to return, respectively. Both of these values are converted to LONG before being used. See *Unary Conversions*, earlier in this chapter, for a detailed description of this process.

For example,

```
MID("Hello, world.", 8, 5)
```

returns the string "world".

If the index is larger than the number of characters in the string, MID returns the empty string. For example,

```
MID("Hello, world.", 14, 5)
```

returns the string "", a string with no characters.

## Chapter 8: Expressions and Assignments

If the number of characters from the index to the end of the string is smaller than the number of characters specified by the last parameter, all available characters from the index character to the end of the string are returned. For example,

```
MID("Hello, world.", 8, 20)
```

returns the string "world."

---

**POS '(' string-expression ',' string-expression [ ',' expression ] ')'**

Returns the position of one string in another string.

The first expression is the string to search, while the second is the string to search for (target string). If the target string exists in the search string, the position of the first occurrence is returned, where the first character is 1. For example

```
POS("This is a test. This is only a test.", "test")
```

returns 11, while the call

```
POS("This is a test. This is only a test.", "This")
```

returns 1.

The last parameter is optional. If used, it is the position within the search string to start the search, again with the first character being 1. The value is converted to `LONG` before being used. See *Unary Conversions*, earlier in this chapter, for a detailed description of this process. For example, the call

```
POS("This is a test. This is only a test.", "test", 12)
```

returns 32.

POS returns 0 if the target string could not be found for any reason. This could simply be that it does not exist in the search string, as in

```
POS("This is a test. This is only a test.", "apple")
```

or

```
POS("This is a test. This is only a test.", "test", 33)
```

Zero is also returned if the start location is given, but it is less than 1 or greater than the length of the search string.

POS returns 1 if the target string is the empty string, as in

```
POS("This is a test. This is only a test.", "")
```

While POS is the name for the string search function in ANSI BASIC, the same functionality is implemented in other variations of BASIC using different names, including INSTR and MATCH.

---

**RIGHT '(' string-expression ',' expression ')'**

**RIGHT\$ '(' string-expression ',' expression ')'**

Returns the rightmost characters in a string.

The second parameter is evaluated and converted to an `LONG`. See *Unary Conversions*, earlier in this chapter, for a detailed description of this process. If this value is negative or larger than the number of characters in the string

that is passed as the first parameter, `RIGHT` returns the string. If this value is less than the number of characters in the string, the specified number of characters is returned from the end of the string.

For example,

```
RIGHT("Testing, 1, 2, 3", 7)
```

returns the string "1,2,3", while

```
RIGHT("Testing, 1, 2, 3", 50)
```

returns the entire input string.

One use of `RIGHT` is to return what's left of a string after one character or a sequence of characters has been peeled off of the start of the string. See the code snippet for `ASC` for an example.

---

**RTRIM** '(' string-expression ')'

**RTRIM\$** '(' string-expression ')'

Returns the string with any trailing space characters removed from the string.

For example,

```
RTRIM("Hello ")
```

returns the string "Hello".

---

**STR** '(' expression ')'

**STR\$** '(' expression ')'

`STR` accepts any number and returns the number as a string. The format matches the format the `PRINT` statement uses to print strings.

The actual value returned depends to some extent on the type of the number. For example, the `SINGLE` value returned by `STR(1000000000.0)` will be returned as the string `1.000000E9`, while the same value returned as a `LONG`, as in `STR(CLng(1000000000.0))` returns the string `1000000000`.

See `PRINT` for a description of the formatting rules.

---

**UCASE** '(' string-expression ')'

**UCASE\$** '(' string-expression ')'

Returns the input string with all lowercase characters converted to uppercase.

Snippet

```
! Compare two strings of different case for equality.
INPUT reply$
IF UCASE(reply$) = "YES" THEN
    PRINT "Yes"
ELSE
    PRINT "No"
END IF
```

---

**VAL** '(' string-expression ')'

`VAL` returns the value of a string; it accepts any string and returns the equivalent number.

`VAL` examines the string beginning with the first character. It forms the largest possible string that is a valid number, then converts this value to a `DOUBLE` value. Leading signs, decimal points and exponents are allowed, but no leading or imbedded spaces can be used. The exponent character can be `E`, `e`, `D` or `d`. Allowing `D` or `d` for the

## Chapter 8: Expressions and Assignments

exponent character is consistent with the syntax for double-precision constants in the program, although it makes no difference in the `VAL` function, since all results are double-precision.

If there are no characters in the string, or if the first character cannot be a part of a number, `VAL` returns 0.0.

This function...	...returns this number.
<code>VAL( "4" )</code>	4.0
<code>VAL( "-1e45" )</code>	-1E45
<code>VAL( "1d2" )</code>	100.0
<code>VAL( " 4" )</code>	0.0
<code>VAL( "7th" )</code>	7.0



## Chapter 9 – Control Statements

This chapter describes the statements that control the order statements are executed, and in some cases, whether a statement is executed or not.

---

### Looping

---

```
DO [ WHILE expression | UNTIL expression ]
  [ statement ]*
LOOP [ WHILE expression | UNTIL expression ]
```

DO-LOOP loops are a powerful looping construct that you can use any time you need to perform an action multiple times, but the number of times you need to loop can't be calculated in advance. It is typically used to loop while some condition is true.

The DO-LOOP structure is a very flexible loop statement that replaces several simpler statements in other programming languages, adding capability at the same time. In fact, the WHILE-WEND loop is a special case of the DO-LOOP loop. This flexibility comes at a price, though: It's tough to grasp how the statement works until you think it through carefully.

To understand this loop, we'll examine its four simplest parts first, then see how they can be combined.

The first form of the DO-LOOP is equivalent to the WHILE-WEND. Using the same example from the description of WHILE-WEND, the DO version looks like this:

```
DO WHILE NOT EOF(1)
  INPUT #1, a$
  PRINT a$
LOOP
```

This loop starts by checking a condition to see if it is true; in this case, we're checking to see if the end of a file has been reached. If the condition is true—if the end of file has not been reached—the statements between DO and LOOP are executed, then the condition is tested again. This process repeats until the end of file is reached, at which time the loop is finished and the statement after LOOP is executed.

In this example, we looped while we were not at the end of the file. It's a bit more natural to think of this loop as looping until we get to the end of the file, and that's how the second version of this loop works. In BASIC, this condition looks like this:

```
DO UNTIL EOF(1)
  INPUT #1, a$
  PRINT a$
LOOP
```

The main advantage of this form of the DO statement is the natural way the condition is expressed. Leaving out the NOT makes the statement easier to read and easier to think about. Programs that are easy to read and understand are easier to write, easier to debug, and easier to modify than programs that are written in an unnatural way.

In this situation, we needed to test the condition before the loop started; after all, it is possible that the file might be empty, so we start off at the end of the file, and never want to execute the contents of the loop. There are also situations where you know the loop must be executed at least once. Many times these situations arise when the value to be tested doesn't even exist until the loop executes at least one time. In these situations, the test needs to be performed after the statements in the body of the loop have executed once. That's exactly what the next two versions of this loop do.

## Chapter 9: Control Statements

One example is reading user input until all of it has been processed. A program to read and process strings until an empty string is returned could look like this:

```
DO
  LINE INPUT a$
  IF LEN(a$) > 0 THEN
    Process(a$)
  END IF
LOOP UNTIL LEN(a$) = 0
```

As with the DO statement with the condition at the top, you can use WHILE to reverse the sense of the loop. The loop would do exactly the same thing if you coded it as

```
DO
  LINE INPUT a$
  IF LEN(a$) > 0 THEN
    Process(a$)
  END IF
LOOP WHILE LEN(a$) > 0
```

It is possible to use a condition both at the top and bottom of a loop. For example, you could write a loop to cheer Karen up like this:

```
DO UNTIL Karen_feeling$ = "happy"
  ThrowParty
LOOP UNTIL broke
```

This loop starts off checking to see if Karen is happy. If so, nothing is done. If Karen is not happy, the body of the loop executes. Once it executes, we check to see if there is any money left. If not, the loop is finished. If there is money left, control returns to the top of the loop, where we check to see how Karen is doing. The process repeats until Karen is happy or we go broke.

While this is legal, it's usually easier to understand a loop that keeps all of the conditions together. This loop does pretty much the same thing as the first one.

```
DO UNTIL (Karen_feeling$ = "happy") OR broke
  ThrowParty
LOOP
```

The difference is that we check to see if we're broke first—probably a good idea in this case. If you really have two conditions, and one must be tested before the loop starts, but the other can't be, splitting them up can make a program shorter and more efficient. If the conditions can be put together, though, it's a good idea to do so.

---

**FOR identifier '=' expression TO expression [ STEP expression ]**  
**[ statement ]\***  
**NEXT [ identifier ] [ ',' identifier ]\***

The FOR-NEXT statement is used to loop over a series of statements a fixed number of times. If you need to loop over a series of statements, but can't calculate in advance how many times you will need to loop, use the DO-LOOP or WHILE-WEND statement.

A typical FOR-NEXT statement would print all of the numbers from 1 to 10, like this:

```
FOR I = 1 TO 10
  PRINT I
NEXT
```

The statement begins by assigning 1 to the loop control variable I. All of the statements up to the NEXT statement are then executed. While this example only shows one statement, there are typically many statements between the FOR and NEXT statements.

As you can see, the loop control variable can be used inside the loop. While it is legal to change the value of the loop control statement inside the FOR loop, it's generally not a good idea to do so. Changing the loop control variable can create subtle and difficult to locate bugs.

When the NEXT statement is reached, the loop control value is incremented by 1. If the new value is less than or equal to the second expression, 10 in our example, the statements between FOR and NEXT are executed again. This process repeats until the loop control variable's value is larger than 10; at that point, execution continues with the statement immediately after the NEXT statement.

While the FOR loop requires the stop and start values to be known in advance, they do not have to be fixed constants like those in the first example. For example, assume you need to change all of the characters in a string to uppercase letters. With an appropriately defined function called TOUPPER somewhere else in your program, you could use a FOR loop to change the string, like this:

```
r$ = ""
FOR i% = 1 TO LEN(s$)
    r$ = r$ + toUpper(MID(s$, i%, 1))
NEXT
s$ = r$
```

In the examples so far, the loop control variable is incremented by 1 each time through the loop. It is possible to tell the FOR loop to step by some other increment using the optional STEP clause. A classic example is drawing a circle using trigonometric functions. This loop steps around the circle using radians, dividing the circle into 360 one degree lines.

```
Circle(100, 100, 100)

SUB Circle(radius, xCenter, yCenter)
pi = 3.1415926535
x = xCenter + radius
y = yCenter
FOR angle = 0 TO 2*pi STEP pi/180
    x2 = xCenter + radius*COS(ANGLE)
    y2 = yCenter + radius*SIN(ANGLE)
    graphics.drawLine(x, y, x2, y2)
    x = x2
    y = y2
NEXT
END SUB
```

STEP values can be calculated, like the start and stop values. STEP values can also be negative, which reverses the direction of the loop. For example, this loop will count down from 10, rather than up.

```
FOR i = 10 TO 1 STEP -1
    PRINT i
NEXT
```

You can optionally include the loop control value on the NEXT statement. This provides a quick check; if the loop control variables don't match, the program stops with an error. You can also put more than one loop control variable on the next statement, separating them with commas. In fact, you can simply use the comma to indicate that the NEXT statement applies to more than one loop control variable.

The examples below are all legal, and show the various ways you can code the NEXT statement.

## Chapter 9: Control Statements

```
! Matrix addition the slow way: C = A + B
FOR i% = 1 TO 10
  FOR j% = 1 TO 10
    C(i%, j%) = A(i%, j%) + B(i%, j%)
  NEXT
NEXT

FOR i% = 1 TO 10
  FOR j% = 1 TO 10
    C(i%, j%) = A(i%, j%) + B(i%, j%)
  NEXT j%
NEXT i%

FOR i% = 1 TO 10
  FOR j% = 1 TO 10
    C(i%, j%) = A(i%, j%) + B(i%, j%)
NEXT j%, i%

FOR i% = 1 TO 10
  FOR j% = 1 TO 10
    C(i%, j%) = A(i%, j%) + B(i%, j%)
NEXT ,
```

The FOR loop control variable can be any numeric type, including BYTE, INTEGER, LONG, SINGLE or DOUBLE. It must be a single value, though, not an element of an array.

When you exit a FOR loop, the value of the FOR loop control variable is guaranteed to be the first value that would fail the loop termination test. For example, in the array addition above, both `i%` and `j%` will be 11 when the loops finish.

It is very poor form, but technically legal, to jump into or out of a FOR loop with a GOTO statement. It is not legal, and will cause an error, to jump into a FOR loop in such a way that the FOR statement is not encountered before the NEXT statement.

---

**WHILE expression**  
[ **statement** ]\*  
**WEND**

The WHILE-WEND loop is a simpler, more efficient version of a DO-LOOP that loops until a condition is met. It is typically used when you need to loop over a series of statements, but don't know in advance how many times you will need to loop. When the number of times you need to loop is known, use the FOR-NEXT statement.

The expression must result in a numeric value. If the result is not zero, execution continues with the first statement past the WHILE statement. Once WEND is reached, the process repeats. If the expression result is zero, the loop is skipped. In this case, execution continues with the statement just past WEND.

It is very poor form, but technically legal, to jump into or out of an executing WHILE loop.

### Snippet

```
!-----
!  
! PrintFile - Print a text file  
!  
! Parameters:  
!   name - name of the file to print  
!  
!-----
```

```

SUB PrintFile (name$)
OPEN name$ FOR INPUT AS #1
WHILE NOT EOF(1)
    INPUT #1, a$
    PRINT a$
WEND
CLOSE #1
END SUB

```

---

## Making Decisions

---

**IF expression THEN statement**

**IF expression GOTO line-number**

```

IF expression THEN
[ ELSE IF expression ]*
[ ELSE ]
END IF

```

The IF statement is used to execute other statements when a specific condition exists. One form of the IF statement also allows you to check a series of conditions, selecting the appropriate alternative.

BASIC has evolved over the years, leaving us with two rather different variations of the IF statement. The classic BASIC IF statement is contained completely on a single line. The expression right after IF is evaluated. It must result in a number. If this number is zero, execution continues with the line after the IF statement. If the result of the expression is anything except zero, the statement after THEN is executed.

Here's an example of the classic IF statement in action:

```

life_force = life_force - sword_hit
IF life_force <= 0 THEN player_died

```

One of the interesting features of the classic IF statement is that it executes everything from THEN to the end of the line, even if there are multiple statements. You could put this to use to implement the classic shell sort.

```

DO
    swap = 0
    FOR i% = 1 TO array_size - 1
        IF a(i%) < a(i% + 1) THEN t = a(i%) : a(i%) = a(i% + 1) : a(i% + 1) =
t : swap = 1
    NEXT
LOOP WHILE swap

```

Before structured programming statements like the DO-LOOP and IF-THEN-ELSE statements were added to BASIC, it was very common to see statements like

```

10 IF a < b THEN GOTO 40
20   c = b
30 GOTO 50
40   c = a
50 REM

```

It was so common, in fact, that a shorthand was invented. When you have an IF statement with THEN GOTO, you can omit the THEN, as in

## Chapter 9: Control Statements

```
10 IF a < b GOTO 40
```

This form of the IF statement isn't used much anymore, though. Its been replaced by the block IF statement, which can span multiple lines. Recoding this example with the block IF statement gets rid of the line numbers and makes the program considerably easier to follow.

```
IF a < b THEN
    c = b
ELSE
    c = a
END IF
```

This form of the IF statement allows multiple statements between the IF and ELSE, and again between the ELSE and END IF. In fact, you can even imbed other IF statements.

The ELSE clause is optional. As in this example, it is used when you need to do one thing if the condition is true, and another if the condition is not true. If you don't need to do anything when the condition is false, you can leave the ELSE out entirely.

The last important option is the ELSE IF statement, used when the IF statement must choose between several alternatives. Here's an example that changes the direction of a ball when it hits the side of the screen. x represents the ball's position, and vx the speed. The same statements would appear right after these to handle the vertical direction.

```
x = x + vx
IF x <= 0 THEN
    vx = -vx
    IF x < 0 THEN x = -x
ELSE IF x >= 320 THEN
    vx = -vx
    IF x > 320 THEN x = 640 - x
END IF
```

You can use more than one ELSE IF clause, stringing them out to handle as many different variations on a possibility as you like. You can also mix ELSE IF with ELSE, but if you do, the ELSE clause must appear after all ELSE IF clauses. When multiple ELSE IF clauses are used, the program tests them in order. When one matches, the statements between it and the next ELSE IF, ELSE or END IF are executed, and all remaining ELSE IF and ELSE clauses are skipped.

It's possible to use the ELSE IF clause to do different things based on the value of a variable, such as this color complement example.

```
IF color = red THEN
    color = green
ELSE IF color = orange THEN
    color = blue
ELSE IF color = yellow THEN
    color = violet
ELSE IF color = green THEN
    color = red
ELSE IF color = blue THEN
    color = orange
ELSE
    color = violet
END IF
```

When you see a series of ELSE IF statements like this that compare the same value over and over to various possibilities, though, it's time to consider the SELECT CASE statement.

---

```
SELECT CASE expression
[ CASE case-range [ ' , ' case-range ]* ]*
[ CASE ELSE ]
END SELECT
```

The SELECT CASE statement is used when you want to do one of several different things, making the choice based on a single value. For example, to change a color to its complementary color, you could use

```
SELECT CASE color
  CASE red
    color = green
  CASE orange
    COLOR = blue
  CASE yellow
    COLOR = violet
  CASE green
    COLOR = red
  CASE blue
    COLOR = orange
  CASE violet
    COLOR = yellow
END SELECT
```

The expression after SELECT CASE is evaluated one time, then compared to a succession of values. As soon as a match is found, the statements following the matching CASE are executed. There can be more than one statement after each CASE statement, even though our example only shows a single statement after each CASE. Once the statements are executed, control passes to the statement after END SELECT.

If no matching statements are found, control skips to the statement after END SELECT without executing any of the imbedded statements. If you need a catch-all case, use the CASE ELSE clause, like this:

```
FOR i = 1 TO 10
  PRINT i
  SELECT CASE I
    CASE 1: PRINT "st"
    CASE 2: PRINT "nd"
    CASE 3: PRINT "rd"
    CASE ELSE: PRINT "th"
  END SELECT
NEXT
```

This example also shows a compact and easy to read way to code a SELECT CASE when all of the actions fit on a single line. It is much easier to scan this SELECT CASE statement for a particular situation to see how it is handled than it is to scan the more verbose form of the first example. When multiple statements appear after each CASE, though, it is usually easier to read the program if the statements following the CASE appear on separate lines.

If you need to execute the same series of statements for several alternate values, separate the values with commas, like this:

## Chapter 9: Control Statements

```
SELECT CASE MID(a$, 1, 1)
  CASE "A", "E", "I", "O", "U"
    PRINT "vowel"
  CASE "W", "Y"
    PRINT "sometimes vowel"
  CASE ELSE
    PRINT "consonant"
END SELECT
```

You can also compare a value to a range of values. This is particularly useful with strings and floating-point numbers.

```
SELECT CASE wavelength
  CASE 0.0 TO 1E-11: PRINT "Gamma rays"
  CASE 1E-11 TO 1E-9: PRINT "X-rays"
  CASE 1E-9 TO 4E-7: PRINT "Ultraviolet"
  CASE 4E-7 TO 7E-7: PRINT "Light"
  CASE 7E-7 TO 1E-5: PRINT "Infrared"
  CASE 1E-5 TO 100: PRINT "Short wave radio"
  CASE 100 TO 1E4: PRINT "Radio"
  CASE 1E4 TO 1E38: PRINT "Long wave radio"
END SELECT
```

If you are used to languages like C and Java, that last example deserves a second look. Unlike most languages, BASIC can handle floating-point and strings, and due to the fact that it supports ranges, it handles them quite well. As with single values, you can code several ranges, or even ranges mixed with discrete values, on a single CASE statement. Multiple values are separated by commas.

It is possible, and with ranges even likely, that more than one CASE clause will match the value from the SELECT CASE statement. The CASE clauses are examined in the order they appear in the program. The statements after the first matching CASE clause are executed; statements after subsequent CASE clauses are not executed, and in fact, the conditions are not even tested. Once a matching CASE clause is found and the statements executed, control jumps immediately to the statement after END SELECT without examining any other possibilities.

Because of this, it is important that the CASE ELSE clause appear after all CASE clauses, just before the END SELECT statement.

---

## Jumping Around

---

### **GOTO line-number**

The GOTO statement is used to jump immediately to another line in the program.

While there is nothing fundamentally wrong with the GOTO statement, its overuse can lead to programs that are almost impossible to read or debug. The GOTO statement should be avoided unless it results in a dramatic increase in performance.

```
      GOTO 40
DATA 1.65235, 30.656, 5.6665, 3.1556
...
DATA 1.45564, 28.667, 4.4453, 3.1327
40 !
```

---

### **ON expression GOTO line-number [ ',' line-number ]\***

The ON-GOTO statement uses an index to jump to one of several locations in a program. In modern BASICs, it has largely been replaced by the SELECT CASE statement, which is considerably easier to read and debug.



The expression is evaluated, then truncated to an integer. Counting from one, one of the line numbers is selected from the list of line numbers immediately after GOTO, and the program jumps to that line. If there are no matching line numbers, execution continues with the line after the ON-GOTO statement.

#### Snippet

```

ON ERROR_NUMBER GOTO 10, 20, 30
PRINT "Unknown error"
GOTO 40
10 PRINT "I don't know how to "; verb$
GOTO 40
20 PRINT "You can't go "; directions$; " from here."
GOTO 40
30 PRINT "You are too weak to move"
40 REM

```

---

## Handling Errors

### ERROR expression

The ERROR statement is used to trigger a run time error. The parameter is the number of the error to trigger. See Appendix A for a complete list of error numbers and the corresponding messages.

If there is an ON ERROR GOTO error handler active when ERROR is used, control passes to the ON ERROR GOTO error handler. From there, the Err class can be used to read the error number.

The principal use for the ERROR statement is in ON ERROR GOTO error handlers. It allows you to pass any error your error handler is not designed to handle back to BASIC. See the description of ON ERROR GOTO for an example that shows how to use ERROR effectively in an ON ERROR GOTO error handler. Pay special attention to the fact that

```
ON ERROR GOTO 0
```

should be used before ERROR statements that appear inside an ON ERROR GOTO error handler. If you don't turn ON ERROR GOTO handling off before using ERROR, it will jump right back to the ON ERROR GOTO handler, generally causing an infinite loop.

The Err class is used to determine the specific nature of an error. See page 162 for a description of this class.

---

### ON ERROR GOTO line-number

When BASIC encounters any condition this manual calls an error, the normal reaction is to stop the program and display an error message. ON ERROR GOTO gives you a way to intercept these errors, handle them, and continue with the program.

ON ERROR GOTO doesn't do much when it is executed. In fact, all that happens is that the line number is recorded. If an error never occurs, nothing is ever done with the line number. If an error occurs, though, execution immediately jumps to the line you specified. From there, you can detect what error occurred using the Err.number method from the Err class, and where it occurred using Err.erl. If it's something you want to handle, you can deal with the error, then pop back to the line where the error occurred using RESUME.

The ON ERROR GOTO statement can appear anywhere in the program, but the line number where the error is handled must appear in the main program, not in a SUB or FUNCTION.

You can use more than one ON ERROR GOTO statement in the program. If an error occurs, execution continues with the line number specified by the most recently executed ON ERROR GOTO statement.

Using a line number of 0 turns off ON ERROR GOTO error handling.

The snippet shows a short program that deliberately generates an error by assigning a value that is too large to be an integer. This error is trapped and corrected by the error handler at line 99. The next error is one the error handler is not designed to handle, though, so it uses the ERROR statement to flag the error in the normal way.

## Chapter 9: Control Statements

Pay special attention to the line:

```
ON ERROR GOTO 0
```

in the error handler. Using a line number of 0 turns ON ERROR GOTO error handling off. This must be done before using ERROR to flag the error. If the original error handler is still in effect when ERROR is encountered, the program will jump right back to the start of the error handler, causing an infinite loop.

### Snippet

```
ON ERROR GOTO 99
x = 40000
i% = x
PRINT i%
END

99 IF Err.number = 2 THEN
    x = 32767
    RESUME
END IF
ON ERROR GOTO 0
ERROR Err.number
```

---

### **RESUME**

The RESUME statement is used in an ON ERROR GOTO error handler to pass control back to the statement where the error occurred. Execution begins at the statement where the error occurred, not at the line. For example, in the line

```
IF i > 0 THEN j% = i
```

if i has the value 40000, the statement generates an error because i is larger than 32767, which is the largest value j% can hold. RESUME starts with the assignment j% = i; it does not repeat the IF check.

See ON ERROR GOTO for an example of an error handler that uses RESUME to recover from an error.

---

## Stopping a Program

### **END**

END is used to mark the end of a program. The most common use is just before the first SUB or FUNCTION statement.

END is also the first token in several special token pairs. See IF, SELECT CASE, FUNCTION and SUB for descriptions of END used with another token.

---

### **STOP**

STOP stops a running program.

---

## Clearing the Workspace

### **CLEAR**

Erases all types, variables and strings. Variables are removed whether they were created with the DIM statement or by being used without encountering a DIM statement.

This statement is generally used to completely reset a program. This is occasionally handy during debugging or when writing an `ON ERROR GOTO` error handler.



## Chapter 10 – Input and Output

There are many kinds of input and output on a modern computer. BASIC has built-in commands to deal with two of these: disk input and output and input from the keyboard with output to the computer's monitor. BASIC also has a set of commands to read data imbedded in the program itself. While you could argue that reading data from the program isn't really input, the commands look and work a lot like the other input commands, and are lumped in with them here.

While the iPhone and iPad do not have a disk drive in the physical sense of the word, they do have solid state memory that software treats as a disk drive. The files you write using the commands in this section can also be synchronized with your desktop computer using either iCloud or iTunes. In either case, the result is a normal file on your desktop computer.

There is a lot of overlap between commands that read the keyboard and write to the console, and commands that read and write disk files. At the same time, these are very different operations. This chapter covers commands that are generally used with the keyboard and monitor, as well as the commands that read data imbedded in the program. The next chapter covers commands that are generally used to read and write disk files.

---

### Printing Text

---

```
PRINT [ '#' expression | '$' expression ]
      [ expression
        | SPC '(' expression ')'
        | TAB '(' expression ')'
        | ';'
        | ',' ]*
```

The **PRINT** statement is used to write text to the monitor and to text disk files. It has two major forms. The simplest to use is covered in this section; it's easy to understand, but doesn't give a lot of control over how numbers are formatted. **PRINT USING** gives a great deal of control over how numbers are formatted, but it is a bit harder to understand and use. **PRINT USING** is covered in the next section.

The **PRINT** statement writes a line of text to the console view. It normally moves the position for the next character, called the cursor position, to a new line after it prints the values in the expressions. The short program

```
PRINT "Hello, world."
PRINT 2000 + 11
```

writes these two lines to the text screen:

```
Hello, world.
2011
```

### Printing Strings

String output is relatively simple. All of the characters in the string are printed, one after the other, to the text screen or disk file.

Keep in mind that string expressions work, too. The line

```
PRINT LEFT("Hello, world.", 5)
```

prints

```
Hello
```

### Printing **BYTE**, **INTEGER** and **LONG** Values

Any expression that evaluates to one of these integer number formats prints the result as decimal digits with no leading zeros or spaces. If the number is less than zero, it is printed with a leading - character.

The table shows several examples. The `PRINT` statement in the first column prints the line shown in the second column.

PRINT Statement	Output Line
PRINT 320	320
PRINT 0	0
PRINT 4000 - 5000	-1000
PRINT CINT (45.67)	45
PRINT CLNG (40)	40
PRINT \$E12000	14753792
PRINT -2147483647 - 1	-2147483648

### Printing **SINGLE** and **DOUBLE** Values

Floating-point numbers print three different ways, depending on the value of the number.

The most common representation for a floating-point number displays a whole number, a decimal point, and the decimal fraction. Negative numbers are preceded by the - character. For example,  $\pi$  to seven significant digits is written 3.141593. This is the format used for **SINGLE** and **DOUBLE** values whose absolute value is greater than or equal to 0.01 and smaller than  $1E8$  for **SINGLE** values, or  $1D13$  for **DOUBLE** values. Numbers whose absolute value is smaller than 0.01, and **SINGLE** numbers whose absolute value is greater than or equal to  $1E8$ , or **DOUBLE** numbers whose absolute value is greater than or equal to  $1D13$ , print in scientific notation.

**SINGLE** values are precise to slightly more than seven significant decimal digits, so that is the number of digits that print. If there are more than seven significant digits the number is rounded to seven significant digits before printing. Any trailing zeros appearing after the decimal point are removed; if all of the digits appearing after the decimal point are zero, the decimal point is also removed. Numbers smaller than 1.0 always print with a leading zero. After following these rules, whatever is left of the number is printed.

**DOUBLE** values are precise to slightly more than 13 decimal digits, so they print thirteen significant digits. The same rules are used for removing trailing zeros after the decimal point, and for removing the decimal point itself if there are no non-zero digits after the decimal point.

Scientific notation is used for numbers that are very close to zero and very far away from zero. In scientific notation, the number is multiplied or divided by 10 until the absolute value is greater than or equal to 1.0 and less than 10.0. This portion of the number is called the mantissa. For **SINGLE** numbers, seven significant digits are printed; for **DOUBLE** numbers, eight significant digits are printed. Unlike numbers in standard form, the decimal point and all trailing zeros are printed, even if all of the digits to the right of the decimal point are zero. The mantissa is followed by the letter E and the exponent for the number. Raising 10 to the power of the exponent and multiplying the result by the mantissa gives the actual number. For example, 1000000000 prints as 1.000000E9, while 0.001 prints as 1.000000E-3.

The table shows several examples. The `PRINT` statement in the first column prints the line shown in the second column.

PRINT Statement	Output Line
PRINT .00123456789	1.234568E-3
PRINT .0123456789	0.01234568
PRINT .123456789	0.1234568
PRINT 1.23456789	1.234568
PRINT 12.3456789	12.34568
PRINT 12345.6789	12345.68
PRINT 123456.789	123456.8
PRINT 1234567.89	123456.8
PRINT 1234567.89	1234568
PRINT 12345678.9	12345680
PRINT 123456789.0	123456E8
PRINT 1.23456789123456789D-3	1.2345679E-3
PRINT 1.23456789123456789D-2	0.01234567891234
PRINT 1.23456789123456789D-1	0.1234567891234
PRINT 1.23456789123456789D0	1.234567891234
PRINT 1.23456789123456789D12	12345678912.34
PRINT 1.23456789123456789D13	1.2345679E13
PRINT 1.001	1.001
PRINT 1.000001	1.000001
PRINT 1.0000001	1
PRINT 1.0000007	1.000001
PRINT -10000 * 10000	-1.000000E8
PRINT EXP(1)	2.718282

### Printing Arrays

While you may opt to print arrays by looping over the subscripts, giving maximum control over the output, there is also a simple way to print the contents of any array.

```
a = [1, 2, 3]
PRINT a
```

Works just fine, printing the elements of the array, then enough spaces to form a 16 character column, then the next element. The output would look like this:

```
1           2           3
```

The array is followed by two blank lines.

For arrays with multiple subscripts, each row of the array is printed, then a blank line, then the next row. The total number of blank lines at the end of the array is still two. The program

```
A = [[2, 3, 5],
      [7, 11, 13],
      [17, 19, 21]]
PRINT INV(A)
```

prints

```
0.2           -0.4           0.2
-0.925        0.5375        -0.112
0.675         -0.1625       -0.0125
```

with two blank lines following the last line.

## Chapter 10: Input and Output

If an array has more than two subscripts, the blocks of values formed by all but the first subscript are printed, and so forth until only two subscripts remain, then the array is printed as a two-dimensional array. Thus this program:

```
A = [[ [1, 2], [3, 4]],  
      [[5, 6], [7, 8]],  
      [[9, 10], [11, 12]]]  
PRINT A
```

prints

1	2
3	4
5	6
7	8
9	10
11	12

followed by two blank lines.

### Printing Multiple Expressions With Commas and Semicolons

You can print more than one number or string using a single `PRINT` statement by separating the expressions with commas or semicolons.

Semicolons simply separate the expressions. The printed expressions are crammed together with no intervening spaces. A good example of semicolons in action is

```
PRINT "The circumference of a circle whose diameter is "; R; " is "; PI*R;  
", "
```

If `R` is 3 and `PI` is 3.1415926, this statement prints

```
The circumference of a circle whose diameter is 3 is 9.424778.
```

Commas tab the next item to the next tab position. Tabs appear in each column divisible by 16. The leftmost column is numbered 1. Commas give you a simple way to quickly create tables, like this table that prints the sine for each angle from 1 to 20 degrees.

```
dtr = 3.1415926/180.0  
FOR a = 1 TO 20  
  PRINT a, SIN(a*dtr)  
NEXT
```

These tables are created by printing the correct number of spaces to the console, not by using tab characters.

You can use more than one comma in a row to skip multiple columns. For that matter, you can use multiple semicolons, too, or even mix commas and semicolons.



### Controlling Spaces Using SPC and TAB

SPC and TAB are special functions you can use inside a PRINT statement. They are only valid in a PRINT statement. Each returns a string with a varying number of spaces.

SPC takes a single parameter, which it evaluates and converts to LONG. If the result is negative, it is replaced by 0. SPC returns a string with the resulting number of spaces. This short example shows how SPC can be used to create a table of numbers that are right justified in any desired field size. WIDTH% is the width of the columns.

```
width% = 12
dtr = 3.1415926/180.0 : ! Converts degrees to radians.
h1$ = "Angle"
h2$ = "Cosine"
PRINT SPC(WIDTH% - LEN(h1$));h1$; SPC(width% - LEN(h2$));h2$
h1$ = "-----"
h2$ = "-----"
PRINT SPC(width% - LEN(h1$));h1$; SPC(width% - LEN(h2$));h2$
FOR a = 0 TO 10
    v1$ = STR(a)
    v2$ = STR(COS(a*dtr))
    PRINT SPC(width% - LEN(v1$));v1$; SPC(width% - LEN(v2$));v2$
NEXT
```

TAB also returns a string with some number of spaces. TAB evaluates the parameter, then subtracts the column number where the next character will be printed, counting columns from 1. If the result is negative, it is replaced with zero. TAB returns a string with the specified number of spaces. The effect is that TAB inserts spaces so the next character you print will appear in the column you specify.

Here's the same program you just saw, but this time the second column is left justified using TAB.

```
width% = 12
dtr = 3.1415926/180.0 : ! Converts degrees to radians.
h1$ = "Angle"
h2$ = "Cosine"
PRINT SPC(width% - LEN(h1$));h1$; TAB(width% + 4);h2$
h1$ = "-----"
h2$ = "-----"
PRINT SPC(WIDTH% - LEN(h1$));h1$; TAB(width% + 4);h2$
FOR a = 0 TO 10
    v1$ = STR(a)
    v2$ = STR(COS(a*dtr))
    PRINT SPC(width% - LEN(v1$));v1$; TAB(width% + 4);v2$
NEXT
```

### Controlling New Lines With Semicolons

In all of the examples shown so far, PRINT always starts at the beginning of a fresh line, and always finishes by moving to the start of a new line. You can prevent this behavior by ending the PRINT statement with a semicolon. This leaves the cursor at the end of the line you were printing, rather than skipping to a fresh line. The next PRINT statement, or any other statement that sends characters to the console, picks up where the PRINT statement left off.

## Chapter 10: Input and Output

### Snippet

```
DIM A$(6)
A$(1) = "red"
A$(2) = "orange"
A$(3) = "yellow"
A$(4) = "green"
A$(5) = "blue"
A$(6) = "violet"
FOR I% = 1 TO 6
    PRINT "The ";I%;
    SELECT CASE I%
        CASE 1
            PRINT "st";
        CASE 2
            PRINT "nd";
        CASE 3
            PRINT "rd";
        CASE ELSE
            PRINT "th";
    END SELECT
    PRINT " color in the rainbow is ";A$(I%);"."
NEXT
```

### **Printing Blank Lines**

A PRINT statement with no parameters at all skips to the start of a new, fresh output line. If the cursor starts at the beginning of a line, this prints a blank line. If the cursor is on a line that already contains characters, perhaps because a semicolon appeared at the end of the last PRINT statement, the PRINT statement with no parameters finishes the current line, setting the cursor so the next printed character will appear at the start of the next line.

### **Printing To Disk Files**

If the first thing after the PRINT statement is a # character, the printed text will go to a disk file rather than the text screen. The # character is followed by an INTEGER expression; this value must match one of the files currently open for output. See OPEN in the next chapter to see how to open a file for output.

Each character that would have been written to the text screen is written to the disk file instead. Whenever the PRINT statement would have skipped to the start of a fresh line, the character CHR(10) is written to the disk file.

### **Printing To Strings**

If the first thing after the PRINT statement is a \$ character, the printed text will go to a string rather than the text screen. The value that follows \$ must be an expression that could normally appear to the left of the = sign in an assignment statement for a string.

For example, to add tab-like spacing to a string, use

```
PRINT $ a$ 1, 2, 3
```

This creates a string with spaced inserted between the numeric values, just as they would be in the text printed to the text screen.

Each character that would have been written to the text screen is placed in the string instead. The character CHR(10) is placed at the end of the string unless the PRINT ends with the ; character.

---

```
PRINT [ '#' expression | '$' expression ] USING format-string ';' expression
      [ ( ',' | ';' ) expression ]* ( ',' | ';' )
```

---

There are two forms of the `PRINT` statement. The `PRINT USING` statement gives accurate control over exactly how values will be displayed. The standard `PRINT` statement, described above, doesn't give much control over the output, but it is much easier to use than `PRINT USING`.

Every `PRINT USING` statement has a format string right after `USING`. This format string contains normal text plus format models, which are sequences of special characters that describe how a value should be printed. The normal text is printed just as it appears; the format models are replaced by one of the values that follow the semicolon that appears after the format string. If there is more than one expression, you can separate the expressions with either commas or semicolons; unlike `PRINT`. With the `PRINT` statement, the comma and semicolon serve different purposes, but with `PRINT USING`, both punctuation marks simply separate expressions.

You can use a semicolon or comma after all of the expressions to indicate that a carriage return should not be printed. As with the semicolon used with the simple `PRINT` statement, this causes the next characters printed by a `PRINT` statement or an `INPUT` prompt to appear on the same line, right after the last character printed by the `PRINT USING` statement.

There are a wide variety of format models available. The sections that follow start with the most basic format model for a number, `#`, and gradually add the other characters that can be mixed with the `#` character to control the way numbers are formatted. After covering the format models used with numbers, the string format models are explained. Each section ends with a list of examples that show a format model as a format string, a value, and the characters that are printed. To keep things simple, these examples only show one format model per line, and don't use expressions, but multiple format models can be used in a single format string, and other characters can be used, too. The examples all use a vertical bar just before and just after the format model. These bars are not required, or even common, in real format strings; they are included here so you can clearly see spaces and what happens when a value is too large for the format model.

Examples of real format strings that mix text and multiple format models appear at the end of the description of the `PRINT USING` statement.

## Formatting Numbers

One or more `#` characters set up a format model for a number. The number of `#` characters used determines the minimum width of the output field. If the number doesn't need all of the output field, spaces are printed before the number; if there isn't enough room for the number, the entire value is still written.

If the value is a floating-point number, the number is rounded to the closest integer value. This doesn't convert the value to an `INTEGER`, it simply rounds the floating-point number to the closest whole number.

Format Model	Value	Prints
" ## "	1	1
" ## "	12	12
" ## "	123	123
" ### "	-3	-3
" ### "	45.67	46
" ### "	0.25	0

## The Decimal Point

Replacing one of the `#` characters in a numeric format model with a period turns the output from an integer to a fixed point output. The classic use for this format model is to print a dollar amount with exactly two digits to the right of the decimal point.

If a number is too large to represent in the available space, it prints as `#` characters.

`SINGLE` numbers are precise to slightly more than seven significant digits, and `DOUBLE` numbers are significant to more than 13 significant digits. If you provide space for more significant digits than are available, you will get something, but the extra digits are artifacts of the number conversion, not valid values. This is shown in the last example in the table, which gets the eighth digit right—more by accident than design—but the ninth significant digit is clearly more than the available precision for a `SINGLE` value.

Format Model	Value	Prints
" ###.## "	1	1.00
" ###.## "	12	12.00
" ###.## "	123	##.##
" ###.## "	-3.4	-3.4
" ###.## "	-23.4	##.##
" ###.## "	45.678	45.68
" ###.## "	9.999	10.00
" ###.## "	0.0049	0.00
" ###.##### "	123.456789	123.456787

### Adding Commas

Replacing one or more of the # characters (except the first one!) with a comma causes the number to be printed with a comma between every three digits of the whole number part, counting left from the decimal place. While it isn't required, it makes sense to put the commas in the same positions they will print. This makes the format model easier to read.

Substituting a comma for a # character does not extend the number of characters available to print values, so be sure you leave enough room for the number's significant digits and for the comma characters. If you need to print eight significant digits, as shown in the first example of the table, you will need a total of ten characters in the format model—eight for the numeric digits, and two for commas. As the second example shows, the number of commas is not the issue. The issue is how many commas the PRINT USING statement will need to insert to represent the value.

Format Model	Value	Prints
" ##,###,### "	1e6	1,000,000
" #,##### "	1e6	1,000,000
" ##,###,###.## "	1.2345678912D7	12,345,678.91
" ##,###.##### "	1234.5678D0	1,234.5678

### Controlling Positive and Negative Signs

By default, if a number is positive no sign is printed, and if a number is negative a - character is printed before the first digit. You can change this behavior two ways.

First, you can indicate that both + and - characters should be used for the sign by substituting a + character for the first # character. The number will always be preceded by a sign.

The second option is to replace the last # character with a + character or - character. If you use a - character, the last character will be a - for negative numbers and a space for positive numbers. If you use a + character, the last character will still be - for negative numbers, but it will be + for positive numbers.

Leading - signs are not used in format models. A - character appearing before a format model is treated like any other character that is not used in a format model: It is simply printed.

Format Model	Value	Prints
" #####.## "	-1.23	-1.23
" +###.## "	-1.23	-1.23
" +###.## "	1.23	+1.23
" ###.##- "	-1.23	1.23-
" ###.##- "	1.23	1.23
" ###.##+ "	-1.23	1.23-
" ###.##+ "	1.23	1.23+

### Dollar Signs

Replacing the first # character with a \$ character causes a \$ to be printed before the number and all leading spaces. Replacing the first two # characters with two \$ characters causes a single \$ character to be printed right before the first digit.

If you are using both a leading + sign and a leading \$ or \$\$, you can put them in any order, so long as they all come before the first # character.

Format Model	Value	Prints
" \$###.## "	-1.23	\$ -1.23
" \$###.## "	1.23	\$ 1.23
" \$+###.## "	-1.23	\$ -1.23
" \$+###.## "	1.23	\$ +1.23
" \$\$###.## "	1.23	\$ \$1.23
" \$\$###.## "	-1.23	\$ -\$1.23
" \$+\$#.## "	1.23	\$ +\$1.23

### Filling Spaces in Numbers

If the format model leaves more space to the left of the decimal point than is needed, the extra space is filled with blanks. In some applications, such as writing checks, it's a good idea to fill in any spaces so someone else doesn't fill them in for you later!

Replacing the first # character with an \* prints an \* in the first space. Replacing the first two # characters with \*\* prints an \* in all leading spaces.

You can mix \* characters, \$ characters and + or - characters in any order, so long as they all appear before the first # character, and so long as pairs of \* or \$ characters appear as a pair. In all cases, dollar signs, positive signs and negative signs are placed in the available space just as they always were, and \* characters fill any remaining spaces.

Format Model	Value	Prints
" *###.## "	-1.23	* -1.23
" *###.## "	1.23	* 1.23
" **###.## "	-1.23	**-1.23
" **###.## "	1.23	**1.23
" **\$#.## "	-1.23	\$*-1.23
" \$\$\$#.## "	1.23	\$**1.23
" \$+*.## "	-1.23	\$*-1.23
" \$+*.## "	1.23	\$*+1.23
" \$\$.## "	1.23	\$**1.23
" \$\$.## "	-1.23	\$*-1.23
" \$\$.## "	1.23	\$**+1.23
" \$\$.## "	-1.23	\$*-1.23

### Formatting Numbers In Scientific Notation

Following any number format with ^ prints the number in scientific notation. The exponent prints as the letter e, a + or - sign, and the numeric exponent. Use four ^ characters to hold any SINGLE exponent, and five to hold any DOUBLE exponent. The format model ##.#####^ will print all significant digits of any SINGLE number, along with the sign and exponent. The format model ##.#####^ does the same for DOUBLE values.

A minimum of three characters are needed to print a one digit exponent; one character for the "e", one for the sign, and one for the digit. Because of this minimum size, you must use at least three ^ characters to get scientific notation.

Format Model	Value	Prints
" ##.###^ "	1	1.000e+0
" ##.###^ "	-0.1234	-1.234e-1
" +#.#####^ "	123.45678	+1.234568e+02

### Formatting Strings

There are three format models for strings. The & character prints an entire string, printing all characters, regardless of the size of the string. The ! character prints the first letter of a string. Two backslash characters with any number of intervening spaces prints as many of the characters as will fit. The backslashes count, so a backslash,

## Chapter 10: Input and Output

two spaces and a backslash prints the first four characters from a string. If the format model is wider than the string, the string is printed, then blanks are printed to fill the available space.

Format Model	Value	Prints
" & "	"testing"	testing
" ! "	"testing"	t
" \\ "	"testing"	te
" \\ \\ "	"testing"	tes
" \\ \\ "	"testing"	test
" \\ \\ "	"testing"	testing
" \\ \\ "	"testing"	testing
" \\ \\ "	"testing"	testing

### Mixing Text and Format Models

All of the examples so far show mixing of text and format models, but only to show where the value being printed started and stopped, making it clear where spaces were printed. In actual programs it's common to see a format string with more than one format model, and to see significant text mixed in with the format models.

For example, the program

```
pv = 100
y = 7
i = 10
fv = pv*EXP(y*LOG(1 + i/100))
PRINT USING "$$###.## compounded for # years at #% interest returns
$$###.##."; pv, y, i, fv
```

prints

```
$100.00 compounded for 7 years at 10% interest returns $194.87.
```

### Printing Format Characters as Text

You may need to print one of the special format characters from the format string. To prevent PRINT USING from using a character as a format character, precede it with the underscore character. To print an underscore, place two underscore characters in the format string.

For example,

```
PRINT USING "_\#_\"; 45
```

prints

```
\45\
```

even though a backslash character is usually used as a fixed length string format model.

### Too Many and Too Few Format Models

If there are fewer values than format models, printing stops when the first extra format model is found. For example,

```
PRINT USING "|#|#|#|"; 1, 2
```

prints

```
|1|2|
```

Too many values for the available format models reuses the format model. This is actually a useful feature, allowing you to create multi-column tables with a single format model. The line

```
PRINT USING "|#|"; 1, 2, 3
```

prints

```
|1||2||3|
```

The only problem with a format model that will cause the program to stop with an error is providing a string to a number specifier or providing a number to a string format model, or an array for either one.

### Printing To Disk Files

Using # followed by a number between PRINT and USING redirects the text that would normally be printed to a disk file. The value that follows # must match one of the files currently open for output. See OPEN in the next chapter to see how to open a file for output.

Each character that would have been written to the text screen is written to the disk file instead. The character CHR(10) is written to the disk file at the end of each line.

### Printing To Strings

Using \$ followed by a string variable between PRINT and USING prints the text that would normally appear on the console to a string. The value that follows \$ must be a n expression that could normally appear to the left of the = sign in an assignment statement for a string.

For example, to format a number as dollars and cents, properly rounding the value to two decimal digits, use PRINT USING to create the value, then strip any extra spaces from the left side of the string with LTRIM, like this:

```
PRINT $ a$ USING "$$#,###.##" ; 4.321 ;
a$ = LTRIM(a$)
```

At this point, the string variable a\$ contains the text \$4.32, with no leading spaces.

Each character that would have been written to the text screen is placed in the string instead. The character CHR(10) is placed at the end of the string unless the PRINT USING ends with the ; character.

---

## Reading Text

---

```
INPUT [ '#' expression ',' ] [ string-expression ';' ] l-value [ ',' l-value ]*
```

INPUT is used to read values from the console or a disk file. These values can be numbers or strings. Multiple inputs on the same typed line or line from a disk file are separated with commas.

### Reading from Disk Files

INPUT normally reads from the keyboard. To read from a disk file, follow INPUT with the # character and a value that matches the file number for an open file. For example, the statement

```
INPUT #4, A$
```

reads a string from a disk file.

## Chapter 10: Input and Output

See OPEN for details on file numbers.

### Prompts

INPUT will print a prompt on the console to indicate it expects input. If you don't do anything special this prompt character is a question mark. For example, if you use the statement

```
INPUT theName$
```

INPUT will print a ? character on the screen, then wait for a typed response. You can supply your own prompt string, followed by a semicolon, like this:

```
INPUT "What is your name? "; theName$
```

You can use string expressions for the prompt, not just string constants. If the first string is followed by a semicolon, it is treated as a prompt and printed. If it is followed by a comma, it is treated like an input variable. In that case, you get the ? prompt and the INPUT statement expects a string value on the input line.

Here's a couple of examples to illustrate this subtle point. This first INPUT statement reads two comma separated strings from the console:

```
INPUT a$, theName$
```

This INPUT statement uses a\$ as a prompt, reading one string from the console:

```
a$ = "What is your name?"  
INPUT a$; theName$
```

Of course, you don't want ? characters showing up all over the console while reading a disk file. If you've specified a file number using the # parameter INPUT won't print a default prompt. You can still add your own prompt, perhaps as a debugging aid, and that will still get printed to the text screen.

If you don't want a prompt and you're reading input from the keyboard, code an empty string as the prompt, like this:

```
INPUT ""; theName$
```

### Multiple Inputs

You can read several values with a single INPUT statement. For each value you are reading, the INPUT statement scans the text typed in the console, forming a chunk of characters. This chunk of characters starts with the first unread character and extends until a comma or end of line mark is found. All of the characters that are read are converted into a string or a number, depending on the type of the parameter. The resulting value is saved.

The rules used to convert the text to numbers are the same as the rules used to convert program symbols to numbers. You can put spaces before the first character of the number and after the last, but not between the characters of the number. You can use leading plus or minus signs, decimal points, and exponents.

Putting this all together, here's an example that shows how to read three values from a line.

```
INPUT "Enter a point in 3 dimensions: "; X, Y, Z
```

This INPUT statement prints the prompt, then looks for three numbers. The numbers are separated by commas. One acceptable response is

```
45, .098, 2.99E4
```



Several things can go wrong with the input process. The worst is typing more inputs than the program expects. For example, if you use the `INPUT` statement

```
INPUT "Please enter your name: "; A$
```

and the response is

```
Fred Pennymaker, Jr.
```

the `INPUT` statement sees two values, one before the comma and one after. This causes the program to stop with a run-time error. You can intercept it with an `ON ERROR GOTO`, but that's a lot of work for such a simple error.

Almost as bad is trying to read a number, but getting input that isn't a number. For example, if the `INPUT` statement

```
INPUT A
```

gets the response

```
4ever
```

it will choke, printing the error message "Number expected: Reenter". The program won't move on until it gets a valid number.

If the `INPUT` statement expects more inputs than it gets, it waits for more input. If the statement

```
INPUT "Please enter your city and state: "; CITY$, STATE$
```

gets the response

```
Indianapolis
```

the program waits for more input. If the person using the program realizes what's happening, and types

```
Indiana
```

the program continues along with no problems. The end of line marker is a perfectly acceptable substitute for a comma. But a blank screen and an apparently frozen program can be fairly confusing for an unsuspecting user of the program.

All of these problems mean that `INPUT` is a great command for quickly hacking out a solution to a problem when you are the only person who would use the program, or, at worst, you will be available to help and train the people who will use the program. For programs that will be used more than a few times, though, it's worth the extra work to use `LINE INPUT` and parse the input yourself.

---

**LINE INPUT [ '#' expression ',' ] [ string-expression ';' ] l-value [ ',' l-value ]\***

`LINE INPUT` is almost the same command as `INPUT`. The only difference is that it doesn't use the comma to mark multiple inputs. It can still use multiple inputs, but each input must be typed on a separate line.

`INPUT` is a simple, effective way to get typed responses into a program. It works well if the person using the program understands its limitations. But there are many limitations to the `INPUT` statement, both in terms of error handling and because you can never use a comma in an input string.

`LINE INPUT` solves these problems. For reading lines of text, it can't be beat. The problems arise when you need to read values from the line—in other words, your application would work better with `INPUT`, but you need to create a program that will handle bogus typed responses better than `INPUT` allows.

In this situation, you can use `LINE INPUT` to read lines of text, then parse it yourself. This takes a fair amount of work, and the techniques involved vary greatly from application to application. In general, though, keep in mind that the `VAL` command can convert strings to numbers.

If your application requires extremely sophisticated parsing, consider compiler books. They deal with the issue of taking a text stream, breaking it into parts, handling errors, and acting on the result in great detail. For most applications, you won't need all of the techniques you'll find in a good compiler book, but there is no better place to learn how to handle text.

---

## Imbedding Data In The Program

The statements in this section are used to read data that is imbedded in a program. For example, a program that calculates the positions of planets might encode the orbital parameters for the planets in `DATA` statements, reading them into arrays using `READ`.

---

### **DATA any-ascii-characters**

`DATA` statements provide input for `READ` statements. The information is stored exactly as you type it, more or less like a comment. This text is scanned by the `READ` statement using exactly the same rules as for the `INPUT` statement.

In general, you will place one or more values in a `DATA` statement, separating the values with commas.

`DATA` statements can hold either strings or numbers. The strings can be enclosed in quote marks, but this isn't required. Using quote marks preserves leading space characters and the case of characters; without them, leading spaces are removed and lowercase letters are converted to uppercase.

These sample `DATA` statements review the rules for coding numbers and strings, which are the same for `DATA` statements as for lines in `BASIC` or for `INPUT`. Any of the `DATA` statements can be read as a string. The first three lines, which contain two strings each, contain `DATA` statements that can only be read into a string; attempting to read a non-number into a numeric value will cause an error. Since the data is scanned like an input statement, comments and continuation lines cannot appear on a `DATA` statement.

#### Snippet

```
DATA Albany, New York
DATA Seattle, Washington
DATA "Santa Fe", "New Mexico"
DATA 1, $400, -32767
DATA 1000000, -123456
DATA 3.1415, .5, 1e9, -3.4D-6
```

---

### **READ 1-value [ ', ' 1-value ]\***

`READ` reads one or more values from `DATA` statements.

If `READ` is in a `SUB` or `FUNCTION`, the `DATA` statement must be in the same `SUB` or `FUNCTION`, and if the `DATA` statement is in the main program, the `READ` must be there, too. `DATA` statements in other parts of the program are only visible to a `READ` in the same part of the program.

The first `READ` statement reads the first value from the first `DATA` statement, placing the value from the `DATA` statement into the first variable in the `READ` statement. This process continues, with the `READ` statement reading data sequentially until all of the parameters have been filled in. The next `READ` statement picks up where the first left off, reading the next available piece of data from a `DATA` statement. The number of parameters in each `READ` and `DATA` statements don't have to match. If a `READ` statement only reads, say, two of the available four pieces of data in a `DATA` statement, the next `READ` starts with the third value from the `DATA` statement. Conversely, if a `READ` statement reads two values, but only one is left in the current `DATA` statement, it skips ahead to read another value from the next available `DATA` statement.

There are two possible errors, either of which will stop the program with a run time error. You cannot read more data than is available, although you can use `RESTORE` to start over. You also can't read a number if the data supplied is not a valid number after removing leading and trailing spaces.

Snippet

```
! Wavelength data
DATA "red", "orange", "yellow", "green", "blue", "violet"
DATA 760, 647, 585, 575, 491, 424, 380
DIM colors$(6)
DIM wavelength(7)
! Read the data
FOR i = 1 TO 6
    READ colors$(i)
NEXT
FOR i = 1 TO 7
    READ wavelength(i)
NEXT
! Print the table
PRINT "Color      Wavelength (nm)"
PRINT "-----"
FOR i = 1 TO 6
    PRINT USING "\      \ ### to ###";colors$(i), wavelength(i),
wavelength(i + 1)
NEXT
```

---

**RESTORE**

`RESTORE` resets the pointer used to track which element of a `DATA` statement will be read next. After `RESTORE`, the next `READ` statement reads the first piece of data from the first available `DATA` statement.

`RESTORE` only resets the `DATA` statement pointer for the procedure in which it appears. For example, if you have `DATA` statements and `READ` statements in the main program and use `RESTORE` in a subroutine, the `READ` statement in the main program is not affected.

Snippet

```
DATA 1, 2, 3, 4
FOR i = 1 TO 4
    RESTORE
    FOR j = 1 TO i
        READ n
        PRINT n,;
    NEXT
    PRINT
NEXT
```



# Chapter 11 – Disk Input and Output

This chapter covers disk access, including commands used to read from and write to disk, commands used to manipulate files on a disk, and commands used to manipulate the directory structure of a disk.

While the iPhone, iPod and iPad do not have a disk drive in the physical sense of the word, they do have solid state memory that software treats as a disk drive. The files you write using the commands in this chapter can also be synchronized with your desktop computer using either iCloud or iTunes. In either case, the result is a normal file on your desktop computer.

Many of the commands normally used to read and write information to the console can also be used to read and write text to disk files. See the previous chapter for information about these commands. The commands described in the last chapter that can also read or write disk files are `INPUT`, `LINE INPUT`, `PRINT` and `PRINT USING`.

---

## File Names

Each file written by a program has a name. For the most part, you can use any string you want for a file name, so long as it does not have the slash character, `/`. Unlike identifiers in BASIC, file names are case sensitive on iOS. This means the file names `Fred` and `fred` refer to different files. There is a maximum length for a file name, but at 1024 characters, it is unlikely to be a problem.

If your files always lived on the iPhone or iPad, that would be the only thing to worry about for a file name. If you move the files to other computers, though, there are a few other considerations.

Not all file systems are case sensitive. While the process of synchronizing your files with another computer will take care of this issue, you can save yourself some time and confusion by not relying on case sensitivity to distinguish between different files. Other file systems also place more restrictions on the length of files names. In general, these should be kept to 255 characters or less for maximum portability, but that's not a particularly onerous restriction.

Other file systems have different restrictions on characters used in file names. For maximum portability, don't start a file name with a space or period, and don't use any of these characters in a file name:

Character	Name
<code>/</code>	forward slash
<code>\</code>	back slash
<code>?</code>	question mark
<code>%</code>	percent sign
<code>*</code>	asterisk
<code>"</code>	quote
<code>&lt;</code>	less than
<code>&gt;</code>	greater than
<code>:</code>	colon

File names typically end with a period and a three-letter file extension that designates the type of data in the file. For the most part, if you are creating files that will be filled with information using `PRINT` or `PRINT USING`, the file extension should be `.txt` (text); if the file is written using `PUT`, the extension should be `.bin` (binary). There are specific situations where other file extensions will make sense. For example, if you are writing an image file in PNG format, you will be using the `PUT` command, but in this case, the file should end in `.png`.

---

## Path Names

Files are organized in folders and directories on disk. Every application and data file on your iOS computer exists in a file stored in a directory, which also has a name, just like files. These directories are often contained in other directories. The lowest level directory on the device, the one at the root of this directory tree, is frequently

## Chapter 11: Disk Input and Output

called the root directory. A few special characters aside, each file is ultimately referred to by a full path name that starts with the root directory and lists all of the subdirectories needed to get to the file.

As an example of path names, we'll assume we have a disk with information about various cities. The information for each city is stored in a file whose name matches the name of the city. Of course, there are situations where two cities have the same name, although they will be in different countries or different states within the same country.

Consider the case of Manhattan, Kansas. It's located in the United States, on the continent of North America. We'll choose the name `Earth` for the disk itself. On the disk `Earth` is a directory named `NorthAmerica`; this in turn contains a directory named `UnitedStates`, which has a directory named `Kansas`. The directory `Kansas` has a file named `Manhattan` that holds information about that city. The full path name is

```
/Earth/NorthAmerica/UnitedStates/Kansas/Manhattan
```

Another file containing information about another city by the same name might have the path name

```
/Earth/NorthAmerica/United States/Illinois/Manhattan
```

The directory `UnitedStates` contains at least two directories, one named `Kansas` and the other named `Illinois`. Each of these directories contains a file named `Manhattan`. The fact that the file name is the same shows why the full path name is needed to uniquely identify a file.

---

### The Default Prefix

In practice, we don't use the full path name. There is a prefix called the default prefix that identifies a specific directory. In the example above, we might identify a file by first setting the default prefix to

```
/Earth/NorthAmerica/United States/Kansas:
```

and then using the file name

```
Manhattan
```

This makes a lot of sense if we're going to access several other files from the same directory. For example, if we're doing a comparison of cities in Kansas, we might also be accessing files named `Topeka`, `KansasCity` and `Wichita`. All of these would be in the same directory as `Manhattan`. By setting the default prefix to `Kansas` first, we can use the short file names for the various cities.

Another way to use the default prefix is to use partial path names. If you are comparing `Manhattan, Kansas` to `Manhattan, Illinois`, you can set the default prefix to

```
/Earth/NorthAmerica/United States/
```

and use the partial path names

```
Kansas/Manhattan
```

and

```
Illinois/Manhattan
```

to access information about the two cities. The difference between the partial path name and a file named `Manhattan` on a disk named `Kansas` is that the partial path name does not start with a slash.

---

## The Sandbox

All of this is made even simpler on iOS by a security restriction that keeps each application from accessing the files for other applications, or for the iOS operating system itself. All of the data files and programs you create are restricted to a single folder known as the application sandbox. Your BASIC programs can only access the files in this directory. On iOS, all of your programs will use partial path names. If you do not expect to have a large number of files, you can even ignore path names. They are there to help organize your files if needed, but can be ignored if you prefer.

---

## File Numbers

Most of the commands that deal with files refer to the file by a number. This number usually appears right after a # character.

The file number is a number you pick when you open the file with the `OPEN` statement. It must be an integer value from 1 to 32767. While almost all of the examples in this section will use the constant 1, this is not a requirement. You can use an expression to calculate the file number, perhaps using a loop like

```
FOR i% = 1 TO 5
  CLOSE #i%
NEXT
```

to close several open files.

---

## File Input and Output Examples

Rather than inserting short, generally meaningless examples of file input and output throughout the chapter, most of the examples of file input and output are collected here. While the examples tend to be short, they still illustrate the basic techniques used to create, write and read files in techBASIC.

---

### Line Oriented Text Files

There are two very simple programs below. The first creates a new file and writes a few lines of text. It uses the simple `PRINT` statement to create the information, but you could also use `PRINT USING` or `PUT` to create the file.

The second program opens the file created by the first and writes the lines to the screen. `LINE INPUT` isn't stopped by commas, so it handles all of the lines gracefully. This shows the basic techniques for dealing with any file organized as a series of variable length lines of text.

`EOF` is used to test for the end of the file. The program can read any number of lines of text without knowing in advance how many lines are in the file.

After synchronizing to move the file to your desktop computer, you can open the file with almost any ASCII editor.

#### Program That Writes the File

```
! Create a new file and write some test lines.
OPEN "Test.txt" FOR OUTPUT AS #1
PRINT #1, "This is a test."
PRINT #1, "This is only a test."
PRINT #1, "If this had been a real program, we would have written
something useful."
CLOSE #1
```

### Program That Reads the File

```
! Read a text file and print its contents.
OPEN "Test.txt" FOR INPUT AS #1
WHILE NOT EOF(1)
    LINE INPUT #1, a$
    PRINT a$
WEND
CLOSE #1
```

---

### Binary Files

There is a fundamental difference between binary files and text files. In a text file, everything is converted to ASCII characters and saved in text representation. The `INTEGER` value 100, which normally uses two bytes of memory, is expanded to three one-byte characters. The conversion process takes time, generally requires more memory, and can lead to loss of precision if you are reading and writing floating-point values.

Binary files save information in the same raw internal format used to save the information in the computer's memory. An `INTEGER` takes two bytes, regardless of the value. `SINGLE` values always use four bytes, and there is no loss of precision as the number is written to disk and read back in. Since the number does not need to be converted to and from text, reading and writing the value is much, much faster than reading and writing text files, and since each value has the same length, you have the option of using random access files.

This program creates a new binary file and writes three `SINGLE` values to the file.

```
OPEN "Test.bin" FOR BINARY AS #1
a = [1.2, 3.4, 5.6]
FOR i% = 1 TO 3
    PUT #1,, a(i%)
NEXT
CLOSE #1
```

This program reads the file, writing the values to the text screen.

```
OPEN "Test.bin" FOR BINARY AS #1
WHILE NOT EOF(1)
    GET #1,, a
    PRINT a
WEND
CLOSE #1
```

Most of the programs in this section are rather short, but here are two very practical programs. The first reads any file and prints the contents. The contents of the file are printed both as hexadecimal values and, when possible, as ASCII values. You can use this program to examine the various files created by these samples to see exactly how they are stored on disk.

This sample is also one of the same files that comes with techBASIC. It is called Dump.



**File Dump Example**

```

! -----
!
! Dump a file
!
! This program prints the contents of any file in
! both hexadecimal and ASCII form.
!
! -----
!
! Set up the variables
!
DIM fileName AS STRING
DIM count AS LONG
DIM bytes(16) AS BYTE
DIM lineCount AS INTEGER
!
! Get the name of the file to dump
!
INPUT "File to dump: "; fileName
!
! Open and dump the file
!
OPEN fileName FOR INPUT AS #1
lineCount = 1
WHILE NOT EOF(1)
    GET #1,, bytes(lineCount)
    lineCount = lineCount + 1
    IF lineCount = 17 THEN
        PrintLine(count, BYTES(), 16)
        count = count + 16
        lineCount = 1
    END IF
WEND
IF lineCount <> 1 THEN
    PrintLine(count, bytes(), lineCount - 1)
END IF
CLOSE #1
END

! -----
!
! PrintLine - Print one line from the file
!
! Parameters:
!     count - number of bytes before this line
!     bytes - line of bytes
!     lineCount - number of bytes in this line
!
! -----
SUB PrintLine(count AS LONG , bytes() AS BYTE , lineCount AS INTEGER )

```

## Chapter 11: Disk Input and Output

```
!
! Print the file displacement
!
PrintByte(count/256)
PrintByte(count)
PRINT ":";
!
! Print the hexadecimal bytes
!
FOR group = 0 TO 3
  PRINT " ";
  FOR offset = 0 TO 3
    IF group*4 + offset < lineCount THEN
      PrintByte(bytes(group*4 + offset + 1))
    ELSE
      PRINT " ";
    END IF
  NEXT
NEXT
!
! Print the line as ASCII text
!
PRINT " '";
FOR offset = 1 TO 16
  IF offset <= lineCount THEN
    IF (bytes(offset) >= 32) AND (bytes(offset) < 127) THEN
      PRINT CHR(bytes(offset));
    ELSE
      PRINT " ";
    END IF
  ELSE
    PRINT " ";
  END IF
NEXT
PRINT ""
END SUB

! -----
!
! PrintByte - Print one byte
!
! Parameters:
!   b - byte to print
! -----
SUB PrintByte(b AS INTEGER )
DIM b1 AS INTEGER
!
b = b - 256*CINT(b/256)
b1 = b/16
b = b - b1*16
IF b1 > 9 THEN
  PRINT CHR(ASC("A") + b1 - 10);
ELSE
  PRINT CHR(ASC("0") + b1);
END IF
```

```

IF b > 9 THEN
  PRINT CHR(ASC("A") + b - 10);
ELSE
  PRINT CHR(ASC("0") + b);
END IF
END SUB

```

The second sample is also in the samples that come with techBASIC. This one is called Catalog; it lists all of the files in your sandbox.

### List Files Example

```

! -----
!
! Catalog
!
! This program lists all of the files in the iOS
! sandbox.
!
! -----
numberOfFiles = 0
Catalog("")
PRINT
PRINT "There are "; numberOfFiles; " files in the sandbox."
END

! -----
!
! Catalog - Recursively print all files in a
!           directory.
!
! Parameters:
!   directory - The directory to print. Pass an
!               empty string for the current
!               directory.
!
! -----
SUB Catalog (directory AS STRING)
  DIM fileName AS STRING
  DIM count AS INTEGER
  DIM directories(1) AS STRING, temp(1) AS STRING
  !
  ! Create the directory to catalog, making sure any
  ! name ends with /.
  !
  IF directory = "" THEN
    fileName = DIR("*")
  ELSE
    IF RIGHT(directory, 1) <> "/" THEN
      directory = directory & "/"
    END IF
    fileName = DIR(directory & "*")
  END IF

```

```
!
! Print all files in the directory, and remember all
! subdirectories.
!
WHILE fileName <> ""
    IF ISDIR(directory & fileName) THEN
        temp = directories
        DIM directories(count + 1) AS STRING
        FOR i = 1 to count
            directories(i) = temp(i)
        NEXT
        count = count + 1
        directories(count) = directory & fileName
    ELSE
        PRINT directory & fileName
        numberOfFiles = numberOfFiles + 1
    END IF
    fileName = DIR
WEND
!
! Print any subdirectories.
!
FOR i = 1 TO count
    Catalog(directories(i))
NEXT
END SUB
```

---

### Backtracking in Files

Many file formats hold a varying amount of information and use a length at the start of the information to tell the program reading the file what to expect. An example is the JPEG graphics format, which uses an integer value to indicate how much graphic information follows. Unfortunately, due to the fact the graphics files are frequently compressed as they are written to disk, you may not know how much graphic data there is until it is written!

A common way to handle this problem is to record your position in a file, write a dummy length, then write the data. Once the data has been written, you can determine how many bytes were written, then move back in the file and write the length. This common technique illustrates the use of the `LOC` (location) function to determine where you are in the file, the position parameter of the `PUT` function to write the length to a specific location in the file, and `EOF` and `SEEK` to move to the end of the file to continue writing a new block of information.

Assuming you have opened file 1 as an output file, your subroutine to write the graphics data can start with the statements

```
p1& = LOC(1)
L% = 0
PUT #1,, L%
```

This records the location in the file where the length of the data should be written, then writes a value of 0 to occupy the correct number of bytes for the length. The program will fill this value in later, when it is known.

Your program would continue with the code that actually writes the information, possibly calling subroutines to write some of the data. There is no need to keep track of how many bytes are written. Once all of the information is written, the program records the new file position, backs up and fills in the length of the data, then resets the file position to the position after all of the data, getting ready for any additional output. The code looks like this:

```

p2& = LOC(1)
L% = p2& - p1&
PUT #1, p1& + 1, L%
SEEK #1, LOF(1) + 1

```

The two programs that follow put this idea to work in a simple example. The first program reads numbers you type until you enter 0. These numbers are written to a file that starts with the number of `INTEGER` values in the file. It then repeats the process, so you end up with a file containing two variable length lists of numbers. The second program reads the file.

### Program That Writes the File

```

! -----
!
! Read two variable length lists of integers from the
! keyboard and write them to a file.
!
! The user indicates the end of a list by typing 0.
!
! In the file, each list of integers is preceded by
! the number of integers in the list.
!
! -----
!
KILL "Test.bin"
OPEN "Test.bin" FOR BINARY AS #1
PRINT "Enter any number of integers; enter 0 to end the list."
EnterNumbers(1)
PRINT "Enter another list of integers, again using 0 to end the list."
EnterNumbers(1)
CLOSE #1
END

! -----
!
! EnterNumbers - enter a variable length list of numbers
!
! Parameters:
!   file - file number to write the list to
!
! -----
!
SUB EnterNumbers(file AS INTEGER)
DIM offset AS LONG
DIM endOffset AS LONG
DIM count AS INTEGER
DIM value AS INTEGER
!
! Record the file position & reserve space for the length
!
offset = LOC(file)
count = 0
PUT #file,, count

```

## Chapter 11: Disk Input and Output

```
!  
! Get the list of integers and write them to the file  
!  
DO  
    INPUT ""; value  
    IF value <> 0 THEN PUT #file,, value  
LOOP UNTIL value = 0  
!  
! Go back and write the length  
!  
endOffset = LOC(file)  
count = (endOffset - offset)/2 - 1  
PUT #file, offset + 1, count  
SEEK #file, LOF(file) + 1  
END SUB
```

### Program That Reads the File

```
DIM count AS INTEGER, i AS INTEGER, value AS INTEGER  
  
OPEN "Test.bin" FOR BINARY AS #1  
WHILE NOT EOF(1)  
    GET #1,, count  
    PRINT "List of "; count; " numbers:"  
    FOR i = 1 TO count  
        GET #1,, value  
        PRINT USING "#####"; value  
    NEXT  
WEND  
CLOSE #1
```

---

### Reading An Entire File

Reading a file in small pieces is convenient, but generally slow, especially if you need to manipulate various pieces of a file in a more or less random order. It's often practical to read an entire file into memory at once, manipulate the file, and write it back to disk. The LOF (length of file) command makes this easy by reporting how many bytes or records are in a file.

The programs below puts this idea to work. The first program writes a random number of INTEGER values to a file. The number of integers is between 50 and 99. The second program reads this file into an array whose size is set after the size of the file is known. It then sorts the list of random numbers and writes them back to the file and to the text screen.

Compare this example to the next one, which uses RANDOM files to do the same thing.

### Program That Writes the File

```
KILL "Test.bin"  
OPEN "Test.bin" FOR BINARY AS #1  
FOR i% = 1 TO 50 * (1.0 + RND(1))  
    v% = CINT(10000 * RND(1))  
    PUT #1,, v%  
NEXT  
CLOSE #1
```

**Program That Reads the File**

```

DIM count AS INTEGER
DIM i AS INTEGER , j AS INTEGER
DIM v AS INTEGER
! Read the file
OPEN "Test.bin" FOR BINARY AS #1
count = LOF(1)/2
DIM a(count) AS INTEGER
FOR i = 1 TO count
    GET #1,, a(i)
NEXT
! Sort the numbers
FOR i = 1 TO count - 1
    FOR j = i + 1 TO count
        IF a(j) < a(i) THEN
            v = a(i)
            a(i) = a(j)
            a(j) = v
        END IF
    NEXT
NEXT
! Write the values
SEEK #1, 1
FOR i = 1 TO count
    PUT #1,, a(i)
    PRINT a(i), ;
NEXT
CLOSE #1

```

---

**Random Access Files**

Random access files use a fixed record size so a piece of information can be quickly located in the file. A common application is a database, such as a mailing list or recipe file. Our example will use a simple file of integers, performing a disk based sort on the file. Compare this with the previous sample, which does essentially the same thing, but reads the file into memory, sorts it in memory, then writes it back to disk.

There are advantages and disadvantages to both methods. Reading the entire file into memory is definitely faster. Some files are bigger than available memory, though, and in some cases you may not want to use all of the available memory even if the file is small enough. Your application may have need for other large chunks of memory, not leaving space for the file. Some files are so large that the time required to read the entire file and write it back to disk is also a serious problem. In all situations where the impact of loading the file into memory is inappropriate, random access files work very well.

The program below works on the same file of random integers produced by the previous example. It reads values from the file using random access, sorting the numbers and writing them back to the file.

**Random Access Program**

```

DIM count AS INTEGER
DIM i AS INTEGER, j AS INTEGER
DIM v1 AS INTEGER, v2 AS INTEGER

```

```

OPEN "Test.bin" FOR RANDOM AS #1 LEN 2
count = LOF(1)
FOR i = 1 TO count - 1
    GET #1, i, v1
    FOR j = i + 1 TO count
        GET #1, j, v2
        IF v2 < v1 THEN
            PUT #1, i, v2
            PUT #1, j, v1
            v1 = v2
        END IF
    NEXT
    PRINT v1, ;
NEXT
CLOSE #1

```

---

## Opening and Closing Files

---

### **CLOSE [ '#' expression ]**

Closes a file previously opened with OPEN.

If a file number is used, CLOSE closes the specific file specified by the expression. If no file number is used, CLOSE closes all files that have been opened by OPEN. CLOSE with no file number is a quick, easy way to close all open files.

See *File Input and Output Examples*, earlier in this chapter, for an example.

---

### **OPEN filename FOR io-kind AS '#' expression [ LEN expression ]**

Files must be opened before you can use most disk operations. The OPEN statement opens the file, assigning a file number to the file in the process. From the time the file is opened until you are finished with the file, all file commands will use the number you assign to identify the file. Once you are finished with a file, use CLOSE to close the file.

Files are also opened in one of five specific ways. You can read from a file opened for input, but you can't write to it, for example. If you open a file for input and need to write to it, you must close the file and open it again.

filename is the name of the file to open. See *File Names*, earlier in this chapter, for information about legal file names.

The file may be opened in any of the following ways by substituting the token shown for the io-kind field.

token	use
OUTPUT	The file is opened for output. If the file already exists, any old contents are lost.
INPUT	The file is opened for input. The file must already exist, but the file type does not matter. Input starts from the beginning of the file.
APPEND	The file is opened for output. If the file already exists, the old contents are not lost. New information is written after all of the old information.
RANDOM	The file is opened for random access. The LEN field is required; each record written to or read from the file will use that number of bytes.
BINARY	The file is opened for input and output. Old information is not lost. New information written immediately after the file is open will overwrite the information at the start of the file.

The value following # is used in subsequent file commands to identify the opened file. This value can range from 1 to 32767. No two open files may use the same file number, but once the file is closed, the number is available for use by another OPEN statement.



If used, the `LEN` expression gives the internal buffer size used to cache input and output. This field is required for random access files, and matches the length of one random access record. For all other file types, larger values use more RAM but generally result in faster disk input and output, while lower values save RAM but result in slower input and output.

Opening a file for `INPUT` or `APPEND` implies that the file already exists. For all of the other file kinds, the file may exist or may not exist.

See *File Input and Output Examples*, earlier in this chapter, for an example.

---

## Reading and Writing Files

---

### **EOF '(' expression ')'**

Returns 0 if there is unread information in a file, and -1 if there is not.

`EOF` is used to see if all of the information in a file has been read. You generally test for the end of the file, and if the end of the file has not yet been reached, you read and process more information.

See *Line Oriented Text Files*, earlier in this chapter, for an example.

---

### **GET [ '#' expression ',' [ expression ] ',' ] 1-value**

Reads a single value from the keyboard or a disk file.

The first expression is the file number, assigned when the file is opened with `OPEN`.

The next expression is the location in the file to write read the value. For random access files, this is the record number; for all other files, this is a byte number. In both cases, the first value in the file is numbered 1.

If no file is specified, the variable must be a string. A single character is read from the console, converted to a string, and saved in the variable. If no characters have been typed, `GET` waits for a character before returning.

If a file is given, `GET` reads binary information from the file. While strings are still treated as single characters, any other data type can be read, including integers or real numbers.

The ability of `GET` to read records as well as the simple data types makes it a very powerful choice for binary file input. Coupled with `PUT`, you can quickly write and read records in their internal, binary format. While you are restricted to fixed length record in random access files, there is no such restriction with other file types, so you can read any data written by other programs, too. If all else fails, `GET` can read the file byte by byte.

`GET` is used in several of the examples in *File Input and Output Examples*, earlier in this chapter.

See `PUT` for information on the format expected for various values. Note that reading a string will always read a single character from the stream, returning it as a one-character string.

---

### **LOC '(' expression ')'**

Returns the location in a file.

Random access files use fixed length records. In a random access file, `LOC` returns the number of the record most recently read or written. Contrast this with all other kinds of files, where `LOC` returns the number of bytes that have been read or written.

While `LOC` can be used for many different purposes, the classic purpose is to record the current position in a file. Combined with `SEEK`, this lets you write subroutines that can remember a file location and return to it at a later point.

See *Backtracking in Files*, earlier in this chapter, for an example.

---

### **LOF '(' expression ')'**

Returns the length of a file.

Returns the number of records in a random access file, or bytes in any other kind of file.

See *Reading An Entire File*, earlier in this chapter, for an example.

**PUT '#' expression ',' [ expression ] ',' l-value**

PUT writes values to files. It is usually used for binary or random access files, although technically it can be used with any file type.

The first expression is the file number, assigned when the file is opened with OPEN.

The next expression is the location in the file to write the value. For random access files, this is the record number; for all other files, this is a byte number. In both cases, the first value in the file is numbered 1.

**l-value** is the value to write to the file.

The actual bytes written for a particular value depend on the byte of the variable.

type	format
byte	A single byte is written.
integer	Two-byte signed integer value, least significant byte first (little-endian).
long	Four-byte signed integer value, least significant byte first (little-endian).
single	Four-byte IEEE format floating-point number.
double	Eight-byte IEEE format floating-point number.
string	The string is converted to bytes using UTF-8 encoding. The bytes are written to the output file in the order they appear in the converted string.
array	A header is written, followed by the values in the array. The specific format is subject to change. If the specific format is important for your application, write the individual elements of the array.

PUT is used in several of the examples in *File Input and Output Examples*, earlier in this chapter.

**SEEK '#' expression ',' expression**

Sets the file so the next read or write occurs at the position indicated by the second expression.

For random access files, the file is divided into chunks based on the length specified when the file is opened. For all other file types, the file is divided into bytes. In each case, the first chunk is numbered 1, with the remaining chunks numbered sequentially.

SEEK is used in several of the examples in *File Input and Output Examples*, earlier in this chapter.

---

## Dealing With Directories and Files

---

**CHDIR pathname**

Change the default prefix to **pathname**.

Use this command to set the default prefix inside a program. After using this command, you can use partial path names or file names to open or manipulate files.

See *File Names*, earlier in this chapter, for an explanation of file names, path names and the default prefix.

The snippet shows a subroutine that moves up one level to the directory containing the current directory.

Snippet

```
SUB Parent
d$ = CURDIR
separator$ = LEFT(d$, 1)
last% = 1
FOR i% = 1 TO LEN(d$) - 1
    IF MID(d$, i%, 1) = separator$ THEN last% = i%
NEXT
CHDIR LEFT(d$, last%)
END SUB
```

---

**CURDIR**

Returns the name of the current directory.

See *File Names*, earlier in this chapter, for an explanation of file names, path names and the current directory.

See CHDIR for a short sample that shows CURDIR in action.

---

**DIR [ '(' file-name ')' ]**

Returns file names from a directory.

The first call should specify a parameter. This can be the name of a specific file or the wildcard character “\*”. Full or partial path names may be used. DIR will return the name of the file if there is a file by the given name, or the name of the first file in the directory if the wildcard character is used.

If the wildcard character is used, subsequent calls may be made without a parameter. These calls return the names of the remaining files in the directory. When all files have been returned, DIR returns an empty string.

This sample program uses DIR to identify all of the files in a directory. It changes the names of all of the files to lowercase using the NAME statement. To understand how this works, consider a file named “FINANCES”. You can use the statements

```
name$ = "Finances"
NAME name$ AS name$
```

to change the file from all uppercase letters to an uppercase letter followed by lowercase letters. This works because file names are case insensitive when you look for an existing file, so the name “Finances” works perfectly well to open or identify the file “FINANCES”, but letter case is preserved when a file is created or renamed, so the command does change the case you see when you catalog the disk.

Snippet

```
dirName$ = DIR("")
WHILE dirName$ <> ""
  name1$ = dirName$
  name2$ = ""
  WHILE name1$ <> ""
    ch$ = LEFT(name1$, 1)
    name1$ = RIGHT(name1$, LEN(name1$) - 1)
    IF (ch$ >= "A") AND (ch$ <= "Z") THEN
      ch$ = CHR(ASC(ch$) - ASC("A") + ASC("a"))
    END IF
    name2$ = name2$ & ch$
  WEND
  NAME dirName$ AS name2$
  dirName$ = DIR
WEND
```

---

**EXISTS filename**

Checks to see if there is a file with the name `filename` on disk. If so, a non-zero value is returned; if not, zero is returned.

This function returns true whether the file is a directory or a data file. See ISDIR for a way to detect a directory.

---

**ISDIR filename**

Checks to see if there is a directory with the name `filename` on disk. If so, a non-zero value is returned; if not, zero is returned.

The snippet shows a program that lists all of the files in the iOS sandbox. It is also included as a sample in techBASIC; the sample is named Catalog. ISDIR is used to determine if a file is a data file or directory. For directories, the directory is opened recursively, and the files in the directory are printed, too.

## Chapter 11: Disk Input and Output

### Snippet

```
! -----
!
! Catalog
!
! This program lists all of the files in the iOS
! sandbox.
!
! -----
numberOfFiles = 0
Catalog("")
PRINT
PRINT "There are "; numberOfFiles; " files in the sandbox."
END

! -----
!
! Catalog - Recursively print all files in a
!           directory.
!
! Parameters:
!   directory - The directory to print. Pass an
!               empty string for the current
!               directory.
!
! -----
SUB Catalog (directory AS STRING)
DIM fileName AS STRING
DIM count AS INTEGER
DIM directories(1) AS STRING, temp(1) AS STRING
!
! Create the directory to catalog, making sure any
! name ends with /.
!
IF directory = "" THEN
    fileName = DIR("*")
ELSE
    IF RIGHT(directory, 1) <> "/" THEN
        directory = directory & "/"
    END IF
    fileName = DIR(directory & "*")
END IF
!
! Print all files in the directory, and remember all
! subdirectories.
!
WHILE fileName <> ""
    IF ISDIR(directory & fileName) THEN
        temp = directories
        DIM directories(count + 1) AS STRING
        FOR i = 1 to count
            directories(i) = temp(i)
        NEXT
        count = count + 1
        directories(count) = directory & fileName
    END IF
    fileName = DIR(directory & fileName & "*")
END WHILE
```

```

ELSE
    PRINT directory & fileName
    numberOfFiles = numberOfFiles + 1
END IF
fileName = DIR
WEND
!
! Print any subdirectories.
!
FOR i = 1 TO count
    Catalog(directories(i))
NEXT
END SUB

```

---

**KILL filename****RMDIR filename**

Deletes the file **filename**.

In some implementations of BASIC, RMDIR is used to delete directories and KILL is used to delete files. There is no distinction between these operations in techBASIC: The same command can delete a directory or a file. techBASIC supports both command names for compatibility, but RMDIR is simply an alias for KILL.

The samples in *File Input and Output Examples*, earlier in this chapter, create a file called `Test.txt`. You could delete this file from inside your BASIC program with the command

```
KILL "Test.txt"
```

---

**MKDIR pathname**

Creates a new directory with the name **pathname**.

**Snippet**

```
MKDIR "MyDirectory"
```

---

**NAME filename AS filename**

Renames the file, directory or disk. The first file name is the original file name, and the second is the new file name.

As an example, consider the problem of safely writing an important file. Even if you make the outrageous assumption that your program has no errors, and the equally outrageous assumption that techBASIC, the system software, and the various other programs running can never fail, and that the computer itself will never have an error, there's always the off chance the battery will be drained just as you're beginning to save an all-important data that you've spent months creating and hours modifying since the last backup. Murphy's Law pretty much assures you that the battery will fail at the worst possible moment, leaving the entire file unusable. One way to solve this problem is to never overwrite the original file until the modified one is safely saved to disk.

Assuming the original file's name is in the variable `fileName$`, and that `Write` writes the data to a new file, returning `TRUE` if there were no errors writing the file, a safe save looks like this:

```

IF Write("temp") THEN
    KILL fileName$
    NAME "temp" AS fileName$
END IF

```

The disadvantage of this kind of save is that the disk must have enough room for both the old and new versions of the file, but the distinct advantage is that a file error of almost any kind while writing the file leaves the original version untouched.

See `DIR` for an example of this command used to change the letter case of file names.



## Chapter 12 – Subroutines

---

### GOSUB Subroutines

Subroutines based on GOSUB are simple to implement and easy to understand. They are also a part of virtually every implementation of BASIC. GOSUB statements do require the use of line numbers and the line number doesn't tell you as much as a SUB name about what a call does. For these reasons, SUB is generally a better way to handle subroutines in techBASIC.

---

#### GOSUB line-number

Control jumps to the first line whose number matches `line-number`. `line-number` must be an integer constant or label. When a RETURN statement is encountered, control jumps to the statement after GOSUB. The following program illustrates this by printing 1, 2 and 3 using subroutine calls.

```

      i = 1
      GOSUB 10
      i = i + 1
      GOSUB 10
      i = i + 1
      GOSUB 10
      END

10 PRINT i
   END

```

Subroutines can be nested up to 4096 levels deep. Recursion is allowed so long as this limit is not exceeded. Here's a simple example of a recursive subroutine that calculates a positive integer exponent.

```

      x = 3
      r = 1
      e = 23
      GOSUB 10
      PRINT r
      END

10 IF e = 0 THEN RETURN
   e = e - 1
   r = r*x
   GOSUB 10
   RETURN

```

The variables used in the subroutine are identical to the variables used in the rest of the program, so in

```

      i = 4
      GOSUB 10
      PRINT i
      END

10 i = 5
   RETURN

```

the value printed is 5, not 4.

## Chapter 12: Subroutines

If GOSUB is used in a subroutine or function, the destination line must be in the same procedure. If GOSUB is used in the main program, the destination line must also be in the main program.

---

### **ON expression GOSUB line-number [ ',' line-number ]\***

The ON-GOSUB statement is similar to the ON-GOTO statement. It uses an index to jump to one of several locations in a program.

The expression is evaluated, then truncated to an integer. Counting from one, one of the line numbers is selected from the list of line numbers immediately after GOSUB, and the program does a GOSUB call to that line. If there are no matching line numbers, execution continues with the line after the ON-GOSUB statement.

Just as with the GOSUB statement, RETURN is used to return from the subroutine call. Control continues with the statement immediately after the ON-GOSUB statement.

#### Snippet

```
FOR i = 1 TO 3
    ON i GOSUB 10, 20, 30
NEXT
END

10 PRINT "one"
RETURN
20 PRINT "two"
RETURN
30 PRINT "three"
RETURN
```

---

### **POP**

Removes one GOSUB return address from the stack. In effect, this turns the most recent GOSUB into a GOTO.

---

### **RETURN**

Returns from the most recent GOSUB or ON-GOSUB, transferring control to the statement following the GOSUB statement.

See GOSUB for examples of RETURN.

---

## Subroutines and Functions

---

### **SUB and FUNCTION Parameter Lists**

Both SUB subroutines and FUNCTION functions support parameter lists. The rules for parameter lists are the same for both. In this section, subroutines and functions will be referred to as procedures, a name that encompasses both subroutines and functions.

Parameter lists follow the procedure name, enclosed in parentheses. A parameter list consists of one or more parameter declarations separated by commas. Each parameter declaration is a variable, optionally followed by AS and a type. If no type is given explicitly, the type is derived from the name of the variable.

For example, this function returns the hyperbolic sine of a value. Like all of the procedures in this section, it's shown with a simple test program, which shows how a parameter is coded when the procedure is called. The examples form very short programs, but they are complete and will run as shown, so you can type them in and try variations to explore how procedures work.

```
PRINT sinh(2)
END
```



```

FUNCTION sinh(x)
sinh = 0.5*(EXP(x) - EXP(-x))
END FUNCTION

```

The parameter doesn't have an AS type clause, so the type is assumed from the variable name. Just as with a variable anywhere else in the program, a name with no trailing type character is `SINGLE`. For that matter, the function itself returns a `SINGLE` value for the same reason.

There are two ways to create a similar function that takes a `DOUBLE` argument and returns a `DOUBLE` result. The first uses type characters, like this:

```

PRINT sinh#(2)
END

FUNCTION sinh#(x#)
sinh# = 0.5D0*(EXP(x#) - EXP(-x#))
END FUNCTION

```

The other method requires a bit more typing, but you don't have to type # after the name when you use the function. It looks like this:

```

PRINT sinh(2)
END

FUNCTION sinh(x AS DOUBLE) AS DOUBLE
sinh = 0.5D0*(EXP(x) - EXP(-x))
END FUNCTION

```

Arrays, strings and all numeric types are allowed as parameters. All of the numeric types and strings work exactly like the example of `SINGLE` and `DOUBLE` in the `sinh` function, above.

BASIC uses parentheses immediately after the name of the array, both when the procedure is declared and when it is called. Commas appear inside the parenthesis to indicate the number of subscripts in the array, but no expressions. Using `A(4)` as a parameter when you call a procedure passes the specific `SINGLE` value at that index, just as printing `A(4)` prints a specific `SINGLE` value from the array. Using `A()` passes the entire array to the procedure.

Arrays passed as parameters do not have to be a specific length, as the example shows. We can use the same procedure to handle arrays of several sizes. This example uses a single procedure to calculate the length of a vector, but the same idea can be used to handle matrices or higher-dimensional arrays.

```

DIM v2(1), v3(2)
v2 = [3, 4]
v3 = [2, 2, 2]
PRINT length(v2()), length(v3())
END

FUNCTION length(v())
squares = 0.0
FOR i% = 1 TO UBOUND(v, 1)
    squares = squares + v(i%)*v(i%)
NEXT
length = SQR(squares)
END FUNCTION

```

## Chapter 12: Subroutines

Multiply subscripted arrays are handled exactly the same way, as the next example shows. This example computes the determinant of a matrix with two or more rows and columns using cofactor reduction. The important point here isn't whether you know what cofactor reduction is, or even what the determinant of a matrix is. In fact, it's not even the subroutine itself—the built in `DET` function does the same job faster. The important point is that the sample shows clearly how a multi-dimensional array is passed as a procedure parameter. It also shows a clever use of BASIC's ability to handle variable dimensioned arrays, since the procedure calls itself with successively smaller arrays until the 2 by 2 case is reached.

```
a = [[2, 3, 5],
      [7, 11, 13],
      [17, 19, 21]]
PRINT Determinant(a())
PRINT DET(a)
END

FUNCTION Determinant(a(),)
IF UBOUND(a, 1) = 2 THEN
  DETERMINANT = a(1, 1)*a(2, 2) - a(1, 2)*a(2, 1)
ELSE
  DIM b(UBOUND(a, 1) - 1, UBOUND(a, 2) - 1)
  DIM i AS INTEGER, j AS INTEGER
  DIM r AS INTEGER, c AS INTEGER

  sign = 1.0
  sum = 0.0
  FOR i = 1 TO UBOUND(a, 1)
    r = 1
    FOR j = 1 TO UBOUND(a, 2)
      IF j <> i THEN
        FOR c = 2 TO UBOUND(a, 2)
          b(r, c - 1) = a(j, c)
        NEXT
        r = r + 1
      END IF
    NEXT
    sum = sum + sign*a(i, 1)*Determinant(b())
    sign = - sign
  NEXT
  Determinant = sum
END IF
END FUNCTION
```

### Passing Parameters by Reference and Value

There are two fundamentally different ways to pass a parameter to a subroutine. The first is called pass by value. When you pass a parameter by value, changes made inside the subroutine do not affect the original value. For example, the program

```
i = 4
j = 5
DoubleIt(i)
DoubleIt(j)
PRINT i, j
END
```

```
SUB DoubleIt(x)
  x = x + x
END SUB
```

prints 4 and 5; changes made to the variable X inside the subroutine do not change the original variable.

The second way to pass a parameter is by reference. When you pass a parameter by value, changes made inside the subroutine have no effect on the original value passed. To pass a value by reference, precede the parameter variable with BYREF. Recoding the sample,

```
i = 4
j = 5
DoubleIt(i)
DoubleIt(j)
PRINT i, j
END

SUB DoubleIt(BYREF x)
  x = x + x
END SUB
```

prints 8 and 10.

Arrays can be passed by reference or by value using the same idea.

### Using Parameters

Inside a procedure, a parameter works just like any other variable. You can use the parameters in expressions, change the value of a parameter, or pass the parameter as a parameter to yet another procedure.

Space used by parameters vanishes as soon as the procedure returns.

### Optional Parameters

Many of the predefined subroutines and functions have optional parameters. An optional parameter appears as an = sign followed by a default value. For example, the declaration for the newLabel method looks like this:

```
FUNCTION newLabel (x, y, width = 42, height = 21) AS Label
```

The last two parameters are optional with default values of 42 and 21. When creating a new label, the line

```
myLabel = newLabel(10, 10)
```

creates a new label with the upper left hand corner at 10, 10. The label will be 42 pixels wide and 21 pixels high. The line

```
myLabel = newLabel(10, 10, 80)
```

creates a label at the same location, but this one is 80 pixels wide. Finally,

```
myLabel = newLabel(10, 10, 80, 30)
```

creates the same label, but 30 pixels tall, perhaps to accommodate a larger font.

---

## Local Variables and Types

Variables declared inside the procedure survive until the procedure returns, but no longer. If the procedure is called again, an entirely new set of variables is allocated. This prevents you from storing values inside a subroutine for later use. For example, the program

```
FOR i = 1 TO 10
  Test
NEXT
END

SUB Test
  j = j + 1
  PRINT j
END SUB
```

looks, at first glance, like it might print the numbers 1 to 10. In fact, it prints 1 ten times. Every time the subroutine TEST returns to the main program, the value J vanishes; each time TEST is called again, a new variable named J is created and initialized to zero.

Variables from outside the procedure can be accessed from inside. If you are writing procedures you plan to use in other programs, it is a good idea to declare all variables locally so the procedure does not accidentally change a variable in the program. The program

```
x = 5
Test
PRINT x
END

SUB Test
  PRINT x
END SUB
```

prints 5 and 5. To guard the variable in the program, declare it locally. This version of the program

```
x = 5
Test
PRINT x
END

SUB Test
  DIM x
  PRINT x
END SUB
```

prints 0 and 5, leaving the program-level variable untouched.

---

## Recursion with SUB and FUNCTION

Recursion is a process where a procedure calls itself. It is usually used to break a problem down into smaller pieces. BASIC supports recursion, as you can see from the many examples throughout the book, including the determinant example from *SUB and FUNCTION Parameter Lists*, earlier in this chapter.

The only limit on recursion depth is the memory available in the variables buffer. Each time you call a procedure, some memory is used to store various values, like the location of the line to return to after the procedure

completes. Space is also used for parameters and local variables. If you try to call a procedure and there isn't enough memory left to store these values, your program will stop with an out of memory error.

---

**[ CALL ] identifier [ parameter-list ]**

Calls a subroutine defined by a SUB statement. See SUB for details, or *SUB and FUNCTION Parameter Lists* for several examples.

Like LET and MAT, CALL may be omitted. It is there for compatibility with other implementations of BASIC, but you can save some typing by just typing the name of the subroutine to call it.

---

**FUNCTION identifier [ parameter-definition-list ] [ "(" expression [ ',' expression ]\* ")" ] [ AS type]  
[ statement ]\*  
END FUNCTION**

Defines a function. The FUNCTION definition appears after the BASIC program, mixed with any SUB definition and other FUNCTION definition in any order.

The identifier is the name of the function. This is followed by the parameter list, if any, and the type returned by the function. The statements that appear between the FUNCTION statement and the END FUNCTION statement are executed as if they were a program, then the last value set for the function is returned to the caller.

The parameter list appears after the function name in parentheses. Parameter lists for FUNCTION and SUB procedures follow the same rules. These rules are discussed in *SUB and FUNCTION Parameter Lists*, earlier in this chapter.

If the function returns an array, the default dimensions follow the parameter list, just as they would in a DIM statement. If the function has no parameters but returns an array, an empty set of parenthesis can be used for the function's parameter list.

```
PRINT f
END

FUNCTION f() (2, 2)
f = [[1, 2], [3, 4]]
END FUNCTION
```

Last is the type of the function, coded as AS followed by a type name. If no type is given, the type is assumed from the function name, just as it is for a variable name. Following these rules,

```
FUNCTION f(x)
```

is a function that returns a SINGLE value.

```
FUNCTION f%(x)
```

and

```
FUNCTION f(x) AS INTEGER
```

both return an INTEGER value.

The value returned by the function is set by assigning a value to the function name. This can be done more than one time; the last value set is the one returned. If no value is set, 0 is returned for numeric functions, a null string for strings, and an array of zeros for arrays.

A function returns to the statement that called it when the END FUNCTION statement executes.

See *Local Variables and Types* for a discussion of local variables, and *Recursion with SUB and FUNCTION* for a discussion of recursion. These sections and *SUB and FUNCTION Parameter Lists*, appear earlier in this chapter; all three have extensive examples of functions.

---

```
SUB identifier [ parameter-definition-list ]  
[ statement ]*  
END SUB
```

Defines a subroutine.

The SUB definition appears after the BASIC program, mixed with any FUNCTION definitions and other SUB definitions in any order.

The identifier is the name of the subroutine, used when it is called. This is followed by the parameter list, if any. The statements that appear between the SUB statement and the END SUB statement are executed as if they were a program.

The parameter list appears after the subroutine name in parentheses. Parameter lists for FUNCTION and SUB procedures follow the same rules. These rules are discussed in *SUB and FUNCTION Parameter Lists*, earlier in this chapter.

Subroutines are called with the CALL statement. This is followed by the name of the subroutine and any parameters. The CALL token is optional.

A subroutine returns to the call location when the END SUB statement executes.

See *Local Variables and Types* for a discussion of local variables, and *Recursion with SUB and FUNCTION* for a discussion of recursion. These sections and *SUB and FUNCTION Parameter Lists*, appear earlier in this chapter; all three have extensive examples of subroutines.

## Chapter 13 – techBASIC Events

---

### The Two Kinds of techBASIC Programs

There are two fundamentally different ways to write a program in techBASIC. Both methods have their place, depending on what the program is supposed to do. Any program that contains one of the event handling subroutines described in Handling Events, later in this chapter, is an event driven program, while any program that does not include one of the event handling subroutines is a classic BASIC program.

---

#### Classic BASIC Programs

Classic programs run to completion, processing information from a file, sensor, or perhaps prompting the user for some form of text input. Classic programs include most programs that generate plots—the fact that the plot is interactive may make it seem like the program is still running, but in fact, the program can end and the plot can still be manipulated with swipes, pinches and taps.

---

#### Event Driven Programs

Event driven programs are programs that use a graphical user interface (GUI), including programs that use controls and programs that directly process tap, swipe and pinch events. The main part of an event driven program sets up the GUI and does any other initialization for the program before events begin to be processed. Instead of the program ending at that point, though, the program continues to execute. As the user taps, swipes, pinches, or types characters on the keyboard, these events are passed on to one of the event handling subroutines in the program. The program does not end until a `STOP` statement is executed or the user taps the Stop button on the techBASIC button bar.

Plots can be used in event driven

Adding a control object, such as a button, without adding an event handling subroutine does not create an event driven program! The control will be created, placed on the graphics screen, and then the program will end—at which point the control will be removed from the graphics screen.

There are numerous examples of event driven programs in Chapter 17, which describes the various controls that can be used to create a GUI. Here is a very simple paint program that shows how to process raw touch events.

#### Snippet

```
System.showGraphics
DIM x0, y0
Graphics.setColor(0, 0, 1)

DIM l AS Label
l = graphics.newlabel(10, 10, 400)
l.setText("Draw with finger strokes and taps.")

DIM quit AS Button
quit = graphics.newButton(10, 650)
quit.setTitle("Quit")
END

SUB touchesBegan (e AS Event)
p = e.where
x0 = p(1, 1)
y0 = p(1, 2)
END SUB
```

```

SUB touchesMoved (e AS Event)
p = e.where
x1 = p(1, 1)
y1 = p(1, 2)
Graphics.drawLine(x0, y0, x1, y1)
x0 = x1
y0 = y1
e.setConsumed(1)
END SUB

SUB tap (e AS Event)
p = e.where
x = p(1, 1)
y = p(1, 2)
Graphics.drawLine(x, y, x + 1, y)
e.setConsumed(1)
END SUB

SUB touchesEnded (e AS Event)
PRINT "Touch Ended"
e.setConsumed(1)
END SUB

SUB touchesCanceled (e AS event)
PRINT "Touch Canceled"
e.setConsumed(1)
END SUB

SUB touchUpInside (ctrl AS Button, when AS double)
STOP
END SUB

```

---

## Handling Events

Most GUI classes interact with the program by calling subroutines in the main program. For example, when the user taps a button, the program's `touchUpInside` subroutine is called with two parameters, the control where the event occurred and the time when the event was reported. The program can then respond in an appropriate way. Many communication classes, such as Bluetooth LE classes, return information using subroutines. For example, after asking for a list of available peripherals with the `BLE.startScan` method, the system begins looking for Bluetooth LE devices. As they are found, calls are made to `BLEDiscoveredPeripheral`.

In some situations, it may make sense to write a program that does not handle a particular event. For example, a program that does not care about every character typed in a `TextField` control, but does care when the user taps the return key on the keyboard after finishing an entry, can implement the `valueChanged` subroutine but not the `textChanged` subroutine. There is essentially no performance penalty when a subroutine is not included for the event; techBASIC does a very quick check to see if an appropriate subroutine exists and, not finding one, never generates an event in the first place.

This section collects information about all of the subroutines that handle events in one place for convenient reference. All of these events are also documented in the various classes used to create and manipulate the controls that generate the events, or in this chapter, where touch events are described. More extensive examples can be found in the individual class descriptions.



---

**SUB BLECharacteristicInfo (time AS DOUBLE, peripheral AS BLEPeripheral, characteristic AS BLECharacteristic, kind AS INTEGER, message AS STRING, err AS LONG)**

Called to return information found after a peripheral is asked about characteristics using the `discoverDescriptors`, `readCharacteristic` or `writeCharacteristic` methods.

**time** is the time when the event was generated. It is the number of seconds since the start of January 1, 2001 using Greenwich Mean Time. Note that it is possible for the time to jump a bit due to the clock being reset, either by the user or by synchronization with an external source.

**peripheral** is the `BLEPeripheral` whose characteristic was interrogated.

**characteristic** is the characteristic whose descriptors or value was found, or whose write has completed.

**kind** is one of the following values.

kind	Meaning
1	The <code>BLECharacteristicInfo</code> call is a response to a <code>discoverDescriptors</code> call.
2	The <code>BLECharacteristicInfo</code> call is a response to a <code>readCharacteristic</code> call.
3	The <code>BLECharacteristicInfo</code> call is a response to a <code>writeCharacteristic</code> call.
4	The peripheral received a request to start or stop providing notifications for the characteristic. This happens when the app calls <code>BLEPeripheral.setNotify</code> for this characteristic.

**message** is a human-readable description of the error, or an empty string if there was no error.

**err** is a system error number indicating the type of error, or 0 if there was no error.

The snippet shows an implementation of this call that will respond to all the three main three characteristic calls.

#### Snippet

```

SUB BLECharacteristicInfo (time AS DOUBLE, peripheral AS BLEPeripheral,
characteristic AS BLECharacteristic, kind AS INTEGER, msg AS STRING, err AS
LONG)
  IF kind = 1 THEN
    DIM descriptors(1) AS BLEDescriptor
    descriptors = characteristic.descriptors
    PRINT "Found descriptors for characteristic "; characteristic.uuid; ":"
    FOR i = 1 TO UBOUND(descriptors, 1)
      PRINT "  descriptor "; i; ": "; descriptors(i).uuid
    NEXT
  ELSE IF kind = 2 THEN
    DIM value(1) AS INTEGER
    PRINT "Value for characteristic "; characteristic.uuid; " = ";
    value = characteristic.value
    FOR i = 1 TO UBOUND(value, 1)
      PRINT RIGHT(HEX(value(i)), 2);
    NEXT
    PRINT
  ELSE IF kind = 3 THEN
    PRINT "Value written to characteristic "; characteristic.uuid;
    PRINT "; error code = "; err
  END IF
END SUB

```

---

**SUB BLEDescriptorInfo (time AS DOUBLE, peripheral AS BLEPeripheral, descriptor AS BLEDescriptor, kind AS INTEGER, message AS STRING, err AS LONG)**

Called to return information found after a value is read from a descriptor using the `readDescriptor` method or written to the descriptor using the `writeDescriptor` method.

**time** is the time when the event was generated. It is the number of seconds since the start of January 1, 2001 using Greenwich Mean Time. Note that it is possible for the time to jump a bit due to the clock being reset, either by the user or by synchronization with an external source.

**peripheral** is the `BLEPeripheral` whose descriptor was interrogated.

**descriptor** is the descriptor whose value was read or written.

**kind** is one of the following values.

---

kind	Meaning
1	The <code>BLEDescriptorInfo</code> call is a response to a <code>readDescriptor</code> call.
2	The <code>BLEDescriptorInfo</code> call is a response to a <code>writeDescriptor</code> call.

---

**message** is a human-readable description of the error, or an empty string if there was no error.

**err** is a system error number indicating the type of error, or 0 if there was no error.

The snippet shows an implementation of this call that will respond to both reads and writes.

#### Snippet

```
SUB BLEDescriptorInfo (time AS DOUBLE, peripheral AS BLEPeripheral,
descriptor AS BLEDescriptor, kind AS INTEGER, msg AS STRING, err AS LONG)
  IF kind = 1 THEN
    PRINT "Value for descriptor "; descriptor.uuid; ":";
    DIM value(1) AS INTEGER
    value = descriptors(i).value
    FOR i = 1 TO UBOUND(value, 1)
      PRINT RIGHT(HEX(value(i)), 2);
    NEXT
    PRINT
  ELSE IF kind = 2 THEN
    PRINT "Value written to descriptor "; descriptor.uuid;
    PRINT "; error code = "; err
  END IF
END SUB
```

---

**SUB BLEDiscoveredPeripheral (time AS DOUBLE, peripheral AS BLEPeripheral, services() AS STRING, advertisements() AS STRING, rssi)**

Called when a peripheral is discovered. Peripherals are discovered after a call to `BLE.startScan`, which begins a scan for available Bluetooth LE devices.

**time** is the time when the event was generated. It is the number of seconds since the start of January 1, 2001 using Greenwich Mean Time. Note that it is possible for the time to jump a bit due to the clock being reset, either by the user or by synchronization with an external source.

**peripheral** is the `BLEPeripheral` object for the discovered peripheral. Methods in the `BLEPeripheral` class may be used to learn more about the peripheral or to initiate communications with the device.

**services** is a list of the UUIDs for the services offered by the device. Services can be standard services with 16 bit UUIDs assigned by the Bluetooth standards committee (see <http://developer.bluetooth.org/gatt/services/Pages/ServicesHome.aspx>) or 128 bit UUIDs assigned by an individual device manufacturer. Services are typically used to determine if the Bluetooth LE device offers the information needed. For example, the UUID for a blood pressure sensor is 1810.

**advertisements** is a set of information returned by a device without having to query the device. This information can be anything at all. Names identify the various advertisements available. The information is returned in an array with two subscripts. Use `UBOUND` to find the number of elements in an array, which can be zero. Here is a short code snippet that will print the service names and values.

```
PRINT "  Advertisements:"
FOR i = 1 TO UBOUND(advertisements, 1)
  PRINT "    "; advertisements(i, 1); ": "; advertisements(i, 2)
NEXT
```

**rssi** (Received Signal Strength Indicator) is the strength of the radio signal for the peripheral.

---

**SUB BLEMutableCharacteristicInfo (time AS DOUBLE, peripheral AS BLEPeripheralManager, characteristic AS BLECharacteristic, kind AS INTEGER)**

Called to return information found after a central device subscribes to a characteristic or unsubscribes from a characteristic.

**time** is the time when the event was generated. It is the number of seconds since the start of January 1, 2001 using Greenwich Mean Time. Note that it is possible for the time to jump a bit due to the clock being reset, either by the user or by synchronization with an external source.

**peripheral** is the BLEPeripheralManager managing the characteristic whose status changed.

**characteristic** is the characteristic whose descriptors or value was found, or whose write has completed.

**kind** is one of the following values.

---

kind	Meaning
1	A central has subscribed to a mutable characteristic.
2	A central has unsubscribed from a mutable characteristic.

---

The snippet is from the BLEChat A sample, found in the O'Reilly Books folder in techBASIC. The subscription status is used to set the color of a label indicating if another device has subscribed to a local service.

Snippet

```
SUB BLEMutableCharacteristicInfo (time AS DOUBLE, _
                                peripheral AS BLEPeripheralManager, _
                                characteristic AS BLECharacteristic, _
                                kind AS INTEGER)

  IF kind = 1 THEN
    receiveStatus.setBackgroundColor(0, 1, 0): ! Subscription active: Status
    Green.
  ELSE
    receiveStatus.setBackgroundColor(1, 0, 0): ! Subscription inactive:
    Status Red.
  END IF
END SUB
```

---

**SUB BLEMutableServiceInfo (time AS DOUBLE, peripheral AS BLEPeripheralManager, service AS BLEService, kind AS INTEGER, message AS STRING, err AS LONG)**

Called to return information found after a central device subscribes to a service.

**time** is the time when the event was generated. It is the number of seconds since the start of January 1, 2001 using Greenwich Mean Time. Note that it is possible for the time to jump a bit due to the clock being reset, either by the user or by synchronization with an external source.

**peripheral** is the BLEPeripheralManager managing the service whose status changed.

**service** is the service the master device subscribed to.

**kind** is 0.

**message** is a human-readable description of the error, or an empty string if there was no error.

**err** is a system error number indicating the type of error, or 0 if there was no error.

---

**SUB BLEPeripheralInfo (time AS DOUBLE, peripheral AS BLEPeripheral, kind AS INTEGER, message AS STRING, err AS LONG)**

Called to report various information about the connection status of a peripheral or to report completion of a `discoverServices` call.

**time** is the time when the event was generated. It is the number of seconds since the start of January 1, 2001 using Greenwich Mean Time. Note that it is possible for the time to jump a bit due to the clock being reset, either by the user or by synchronization with an external source.

**peripheral** is the `BLEPeripheral` whose connection status has been updated or whose `discoverServices` call has completed. Methods in the `BLEPeripheral` class may be used to learn more about the peripheral or to initiate communications with the device.

**kind** is one of the following values, indicating the change in the status of the device.

kind	Meaning
1	A connection has been successfully completed with the peripheral. Use <code>BLE.connect</code> to initiate a connection request. <b>err</b> will be 0, and <b>message</b> will be an empty string.
2	A connection attempt initiated with <code>BLE.connect</code> failed. <b>err</b> and <b>message</b> indicate the reason for failure.
3	A previously connected peripheral has disconnected. <b>err</b> and <b>message</b> indicate the reason for the disconnection.
4	The peripheral's <code>discoverServices</code> method asks the device for a list of available services. This response indicates the device responded to the request. If the list of services was returned successfully, <b>err</b> will be 0. If there was an error getting the list of services, <b>err</b> and <b>message</b> describe the reason for the failure. Use the peripheral's <code>services</code> method to get the list of services.
5	At least one of the peripheral's services has changed. All services have been invalidated, and must be rediscovered.
6	The RSSI for the peripheral has been updated. This call is made after a <code>BLEPeripheral.rssi</code> call has returned a new RSSI value for the peripheral.
7	The device name for the peripheral has been updated.

**message** is a human-readable description of the error, or an empty string if there was no error.

**err** is a system error number indicating the type of error, or 0 if there was no error.

The snippet shows how to print the result from the various kinds of calls to this subroutine.

#### Snippet

```
SUB BLEPeripheralInfo (time AS DOUBLE, peripheral AS BLEPeripheral, kind
AS INTEGER, msg AS STRING, err AS LONG)
  SELECT CASE kind
    CASE 1
      PRINT "Connected to "; peripheral.bleName

    CASE 2
      PRINT "Failed to connect to "; peripheral.bleName; ": "; err

    CASE 3
      PRINT "Lost connection to "; peripheral.bleName; ": "; err

    CASE 4
      DIM services(1) AS BLEService, included(1) AS BLEService
      services = peripheral.services
      PRINT "Discovered services:"
      FOR i = 1 TO UBOUND(services, 1)
        PRINT "  services("; i; "): "; services(i).uuid
      NEXT
```

```

CASE 5
    ! Call a subroutine to rediscover all services.
    doDiscovery

CASE 6
    PRINT "RSSI = "; peripheral.rssi

CASE 7
    PRINT "New device name: "; peripheral.bleName
END SELECT
END SUB

```

---

**SUB BLEPeripheralManagerInfo (time AS DOUBLE, peripheral AS BLEPeripheralManager, kind AS INTEGER, readRequest AS BLEATTRequest, writeRequest () AS BLEATTRequest, message AS STRING, err AS LONG)**

Called to report various information relating to the peripheral manager.

**time** is the time when the event was generated. It is the number of seconds since the start of January 1, 2001 using Greenwich Mean Time. Note that it is possible for the time to jump a bit due to the clock being reset, either by the user or by synchronization with an external source.

**peripheral** is the BLEPeripheral whose connection status has been updated.

**kind** is one of the following values, indicating the change in the status of the device.

kind	Meaning
1	The peripheral manager has updated its state. Use the <code>BLEPeripheralManager.state</code> call to read the new state.
2	The peripheral manager has started advertising. This call is made after the app calls <code>BLEPeripheralManager.startAdvertising</code> to report the success or failure of the call. Check the error code, which will be zero if advertising was started successfully. The error code and message indicate the error, if any.
3	When a call to update a value is made, it can fail because the queue is full. A call with this kind code indicates there is space on the queue to resend the information.
4	This call is made when the master device wants to read a characteristic manually. Information about the read is in the <code>BLEATTRequest</code> object passed as the <b>readRequest</b> parameter; this parameter is NULL for all other values of <b>kind</b> . The program should respond by filling in the value in the <code>BLEATTRequest</code> object, then calling the peripheral manager's <code>repondToRequest</code> method to return the data to the master device. The second parameter of <code>respondToRequest</code> is an error code; this should be zero for a successful read, or one of the error codes listed with the <code>respondToRequest</code> method if the read could not be performed.
5	This call is made when the master device wants to write one or more characteristic values. The various write requests are passed in an array of <code>BLEATTRequest</code> objects in the <b>writeRequest</b> parameter; this array is empty for all other values of <b>kind</b> . The program should start by scanning each of the write requests to make sure the writes are allowed. If any write is not valid, the program should not perform any of the writes. If all writes can be performed, the program should perform the writes. Whether the writes can be performed or not, the program should call the peripheral manager's <code>respondToRequest</code> method with the first <code>BLEATTRequest</code> object in the <code>writeRequest</code> array and an appropriate error code.

**message** is a human-readable description of the error, or an empty string if there was no error.

**err** is a system error number indicating the type of error, or 0 if there was no error.

The snippet shows a `BLEPeripheralManagerInfo` subroutine that handles the various values of **kind**. It frequently calls other subroutines that your program must implement to handle the specific characteristics it implements.

Snippet

```

SUB BLEPeripheralManagerInfo (time AS DOUBLE, _
                             peripheral AS BLEPeripheralManager, _
                             kind AS INTEGER, _
                             readRequest AS BLEATTRequest, _
                             writeRequest () AS BLEATTRequest, _
                             message AS STRING, _
                             err AS LONG)

SELECT CASE kind
CASE 1
    PRINT "Peripheral state = "; peripheral.state

CASE 2
    IF err <> 0 THEN
        ! Advertising failed to start.
        handleFailureToAdvertise(message, err)
    END IF

CASE 3
    ! Writing stopped because the queue was full. Pick up where the error
    ! occurred.
    resumeWriting

CASE 4
    ! The master device wants to read some data. Fill in the data in the
    ! readRequest object, then return the object and any error code.
    error% = doRead(readRequest)
    peripheral.respondToRequest(readRequest, error%)

CASE 5
    ! The master device wants to write some data. Start by checking each
    ! write to see if it is valid, setting the error code to a non-zero
    ! value if any write cannot be performed.
    error% = 0
    FOR i = 1 TO UBOUND(writeRequest, 1)
        IF error% = 0 THEN
            error% = validateWrite(writeRequest(i))
        END IF
    NEXT

    ! If all data can be written, write it.
    IF error% = 0 THEN
        FOR i = 1 TO UBOUND(writeRequest, 1)
            doWrite(writeRequest(i))
        NEXT
    END IF

    ! Tell the master device what happened.
    peripheral.respondToRequest(writeRequest(1), error%)
END SELECT
END SUB

```

---

**SUB BLERetrievedPeripherals (time AS DOUBLE, kind AS INTEGER, peripherals()  
AS BLEPeripheral)**

The underlying calls that generated this callback have been removed from iOS. This callback will no longer be made in techBASIC, but leaving it in old programs will not cause an error.

---

**SUB BLEServiceInfo (time AS DOUBLE, peripheral AS BLEPeripheral, service AS BLEService, kind AS INTEGER, message AS STRING, err AS LONG)**

Called to return information found after a peripheral is asked about services using the `discoverCharacteristics` or `discoverIncludedServices` methods.

**time** is the time when the event was generated. It is the number of seconds since the start of January 1, 2001 using Greenwich Mean Time. Note that it is possible for the time to jump a bit due to the clock being reset, either by the user or by synchronization with an external source.

**peripheral** is the `BLEPeripheral` whose services were interrogated.

**kind** is one of the following values.

---

kind	Meaning
1	The <code>BLEServicesDiscovery</code> call is a response to a <code>discoverCharacteristics</code> call.
2	The <code>BLEServicesDiscovery</code> call is a response to a <code>discoverIncludedServices</code> call.

---

**services** is the service whose characteristics or included services were found.

**message** is a human-readable description of the error, or an empty string if there was no error.

**err** is a system error number indicating the type of error, or 0 if there was no error.

The snippet shows an implementation of this call that will respond to both `discoverCharacteristics` and `discoverIncludedServices`. It prints the characteristics or included services for the service.

#### Snippet

```
SUB BLEServiceInfo (time AS DOUBLE, peripheral AS BLEPeripheral, service
AS BLEService, kind AS INTEGER, msg AS STRING, err AS LONG)
  IF kind = 1 THEN
    DIM characteristics(1) AS BLECharacteristic
    characteristics = service.characteristics
    PRINT "Found characteristics for service "; service.uuid; ":"
    FOR i = 1 TO UBOUND(characteristics, 1)
      PRINT "  characteristic "; i; ": "; characteristics(i).uuid;
      PRINT ", properties = "; characteristics(i).properties;
      PRINT ", isBroadcasted = "; characteristics(i).isBroadcasted;
      PRINT ", isNotifying = "; characteristics(i).isNotifying
    NEXT
  ELSE IF kind = 2 THEN
    DIM services(1) AS BLEService
    services = service.includedServices
    PRINT "Found included services for service "; service.uuid; ":"
    FOR i = 1 TO UBOUND(services, 1)
      PRINT "  included service "; i; ": "; services(i).uuid
    NEXT
  END IF
END SUB
```

---

**SUB BLEState (time AS DOUBLE, value AS INTEGER)**

Reports a change in the state of the Bluetooth LE services. This can be used to tell if Bluetooth LE is available on a particular device, or if not, why.

`BLEState` events are generated whenever the state of the Bluetooth LE device changes. This generally occurs after Bluetooth LE is started using either the `BLE.BLEAvailable` call or the `BLE.startBLE` call.

**time** is the time when the event was generated. It is the number of seconds since the start of January 1, 2001 using Greenwich Mean Time. Note that it is possible for the time to jump a bit due to the clock being reset, either by the user or by synchronization with an external source.

**value** indicates the new state of the Bluetooth LE service. It is one of the following:

value	State
0	The state is unknown. It will be updated again soon.
1	The connection with the system was momentarily lost. The state will be updated again soon.
2	The device does not support Bluetooth LE. iPhones support Bluetooth LE beginning with the iPhone 4s. iPads support Bluetooth LE beginning with the third generation iPad.
3	The application is not authorized to use Bluetooth LE.
4	Bluetooth LE is powered off. Use the Settings app, General tab, Bluetooth tab to enable Bluetooth.
5	Bluetooth LE is ready for use.

The `BLE.BLEAvailable` call can also be used to detect if Bluetooth LE is available.

---

**SUB cellSelected (ctrl AS Control, time AS DOUBLE, row AS INTEGER, section AS INTEGER)**

The cell selected event is generated when the user selects a cell in a Table object. `ctrl` is the Table control where the cell was selected.

`time` is the time when the event was generated. It is the number of seconds since the start of January 1, 2001 using Greenwich Mean Time. Note that it is possible for the time to jump a bit due to the clock being reset, either by the user or by synchronization with an external source.

`row` and `section` are the row and section number of the selected cell in the table, counting from 1.

---

**SUB didRotate (fromOrientation AS INTEGER, toOrientation AS INTEGER)**

Called when the device orientation changes in full screen mode, allowing the program to change the location and size of controls to match the new orientation. See `System.setAllowedOrientations` for a way to block or allow the various device orientations. `fromOrientation` is the orientation before the device was rotated.

`toOrientation` is the new orientation.

Note that this subroutine is not called unless `Graphics.showGraphics` has been used to set the graphics screen to full screen mode.

The values will be one of the following:

orientation	Description
1	The device is in portrait mode, with the device held upright and the home button on the bottom.
2	The device is in landscape mode with the home button on the left.
3	The device is in landscape mode with the home button on the right.
4	The device is in portrait mode, held upside down with the home button on the top.

The snippet shows a program that presents switches to set the allowed orientations.

Snippet

```
! Display the graphics screen.
System.showGraphics
```

```
! Set up some labels to say what the program does.
DIM label1 AS Label, label2 AS Label
label1 = Graphics.newLabel(10, 10, 300)
label1.setText("Try device orientations.")
label2 = Graphics.newLabel(10, 51, 300)
label2.setText("Allowed home button orientations:")
```



```

! Set up the switches used to turn the allowed orientations on or off.
DIM upSwitch AS Switch, downSwitch AS Switch, leftSwitch AS Switch,
rightSwitch AS Switch
upSwitch = Graphics.newSwitch(10, 82)
downSwitch = Graphics.newSwitch(10, 119)
rightSwitch = Graphics.newSwitch(10, 158)
leftSwitch = Graphics.newSwitch(10, 195)

upSwitch.setOn(1)
leftSwitch.setOn(1)
rightSwitch.setOn(1)
IF System.device = 1 THEN downSwitch.setOn(1)

! Label the switches.
DIM upLabel AS Label, downLabel AS Label, leftLabel AS Label, rightLabel
AS Label
upLabel = Graphics.newLabel(99, 85)
upLabel.setText("Up")
downLabel = Graphics.newLabel(99, 122, 100)
downLabel.setText("Down")
rightLabel = Graphics.newLabel(99, 161)
rightLabel.setText("Right")
leftLabel = Graphics.newLabel(99, 198)
leftLabel.setText("Left")

! Create some labels to tell the user what the orientation was before and
after it changes.
DIM fromLabel AS Label, toLabel AS Label
fromLabel = Graphics.newLabel(10, 242, 300)
toLabel = Graphics.newLabel(10, 273, 300)

! Create the Quit button.
DIM b AS Button
b = Graphics.newButton(10, 320)
b.setTitle("Quit")

! This will be called when the device changes orientation. It sets two
labels to show the orientation before and after the rotation.
SUB didRotate (fromOrientation AS INTEGER, toOrientation AS INTEGER)
PRINT $ s$ "Rotated from orientation "; fromOrientation;
fromLabel.setText(s$)
PRINT $ s$ "Rotated to orientation "; toOrientation;
toLabel.setText(s$)
END SUB

! Called when the Quit button is pressed, this exits the program.
SUB touchUpInside (ctrl AS Button, time AS DOUBLE)
STOP
END SUB

```

! Called when one of the switches changes, this sets the allowed orientations based on the current switch settings.

```
SUB valueChanged (ctrl AS Control, time AS DOUBLE)
  o% = 0
  IF upSwitch.isOn THEN o% = 1
  IF downSwitch.isOn THEN o% = o% BITOR 8
  IF rightSwitch.isOn THEN o% = o% BITOR 4
  IF leftSwitch.isOn THEN o% = o% BITOR 2
  System.setAllowedOrientations(o%)
END SUB
```

---

#### **SUB finishedLoad (ctrl AS Control, time AS DOUBLE, url AS STRING)**

The finished load event is generated when a `WebView` object completes loading a web page or document. **ctrl** is the `WebView` object that loaded the page.

**time** is the time when the event was generated. It is the number of seconds since the start of January 1, 2001 using Greenwich Mean Time. Note that it is possible for the time to jump a bit due to the clock being reset, either by the user or by synchronization with an external source.

**url** is the URL or path name of the web page or document that was loaded.

---

#### **SUB loadError (ctrl AS Control, time AS DOUBLE, message AS STRING, err AS LONG)**

The load error event is generated when a `WebView` object cannot load a web page or document. **ctrl** is the `WebView` object that attempted to load the page.

**time** is the time when the event was generated. It is the number of seconds since the start of January 1, 2001 using Greenwich Mean Time. Note that it is possible for the time to jump a bit due to the clock being reset, either by the user or by synchronization with an external source.

**message** is a human-readable description of the error that prevented the page from loading.

**err** is a system error number indicating the reason the load failed.

---

#### **SUB keyboardChanged (time AS DOUBLE, change AS INTEGER, beginX, beginY, beginWidth, beginHeight, endX, endY, endWidth, endHeight)**

The on-screen keyboard has changed by showing the keyboard, hiding the keyboard, or changing the frame of the keyboard. You can implement this event to respond to that change, shifting elements on your user interface so they are no longer under the keyboard.

**time** is the time when the event was generated. It is the number of seconds since the start of January 1, 2001 using Greenwich Mean Time. Note that it is possible for the time to jump a bit due to the clock being reset, either by the user or by synchronization with an external source.

**change** identifies the kind of keyboard event. It is one of the following:

Value	Description
0	The keyboard was shown.
1	The keyboard was hidden.
2	The keyboard's frame changed.

**beginX, beginY, beginWidth and beginHeight** are a rectangle indicating the size and location of the keyboard when the operation started. **endX, endY, endWidth and endHeight** give the size and location of the keyboard when the operation ended.

---

#### **SUB mapLocation (ctrl AS Control, time AS DOUBLE, latitude AS DOUBLE, longitude AS DOUBLE)**

A map location event is generated when the user taps on a `MapView` object.

**ctrl** is the `MapView` object where the tap occurred.

**time** is the time when the event was generated. It is the number of seconds since the start of January 1, 2001 using Greenwich Mean Time. Note that it is possible for the time to jump a bit due to the clock being reset, either by the user or by synchronization with an external source.

**latitude** and **longitude** are the map location where the tap occurred.

#### **SUB nullEvent (time AS DOUBLE)**

Null events are generated when there are no other events to process, but the program is not complete. This is the proper place to handle routine tasks that do not require user intervention, such as a few steps of a time-intensive task or updating a progress indicator. As with all event handlers, though, the program should not spend a great deal of time in this subroutine before returning to the caller. Doing so would cause the program to appear unresponsive to the user, who might justifiably cancel the program, thinking it is hung.

**time** is the time when the event was generated. It is the number of seconds since the start of January 1, 2001 using Greenwich Mean Time. Note that it is possible for the time to jump a bit due to the clock being reset, either by the user or by synchronization with an external source.

See the `Progress` class for a snippet that shows null events used to update a progress bar used as a 10 second timer.

See `System.setNullEventTime` for a way to change the frequency of calls to `nullEvent`.

#### **SUB readyToUpdateSubscribers (time AS DOUBLE)**

Called when a Bluetooth LE communications channel is available for transmission of data from a Bluetooth LE slave device to the central device.

**time** is the time when the event was generated. It is the number of seconds since the start of January 1, 2001 using Greenwich Mean Time. Note that it is possible for the time to jump a bit due to the clock being reset, either by the user or by synchronization with an external source.

When a call is made to the `BLE` class' `updateValue` method, it is possible for the communications channel to be busy from a previous call to `updateValue`. In that case, `updateValue` returns 0 rather than 1, indicating the data could not be updated. Once the channel is open a call is made to this subroutine to notify the program that the channel is ready for output. At that time, the data can be resent.

#### **SUB startedLoad (ctrl AS Control, time AS DOUBLE, url AS STRING)**

The started load event is generated when a `WebView` object begins loading a web page or document. **ctrl** is the `WebView` object that is loading the page.

**time** is the time when the event was generated. It is the number of seconds since the start of January 1, 2001 using Greenwich Mean Time. Note that it is possible for the time to jump a bit due to the clock being reset, either by the user or by synchronization with an external source.

**url** is the URL or path name of the web page or document to be loaded.

#### **SUB tap (e AS EVENT)**

Tap events are generated whenever the user taps on the graphics screen or on any control other than a `Button` or `MapView`. Both buttons and maps have other mechanisms for dealing with taps.

Depending on the program, taps could apply to something specific on the graphics screen like drawing a point in a paint program, or selecting a callout on a plot of a function. It can even be used for both. The program gets the tap event before any plots. If the plot should not see the tap event, use `e.setConsumed(1)` to consume the event before returning from the subroutine. If the tap should be passed on to any plot at the tap location, don't make the call to `setConsumed`, and the tap will create or hide callouts as it normally does on a plot. If the program does not have plots, don't worry about calling `setConsumed`, since it won't make any difference in how the tap is processed.

**e** is an `Event` object containing the event type, location, and time. See the `Event` class for a complete description of this object.

---

**SUB textChanged (ctrl AS Control, time AS DOUBLE)**

The text changed event is generated when the user makes any change to the text in a `TextField` or a `TextView`. `ctrl` is the `TextField` or `TextView` object where the text was changed.

`time` is the time when the event was generated. It is the number of seconds since the start of January 1, 2001 using Greenwich Mean Time. Note that it is possible for the time to jump a bit due to the clock being reset, either by the user or by synchronization with an external source.

---

**SUB touchesBegan (e AS EVENT)**

Touches began events are generated whenever the user touches the graphics screen while the program is running.

Depending on the program, touches could apply to something specific on the graphics screen like drawing a line in a paint program, or translating or rotating a plot of a function. It can even be used for both. The program gets the touch event before any plots. If the plot should not see the touch event, use `e.setConsumed(1)` to consume the event before returning from the subroutine. If the tap should be passed on to any plot for processing translations and rotations, don't make the call to `setConsumed`. If the program does not have plots, don't worry about calling `setConsumed`, since it won't make any difference in how the event is processed.

`e` is an Event object containing the event type, location, and time. See the Event class for a complete description of this object.

---

**SUB touchesCanceled (e AS EVENT)**

Touches canceled events are generated whenever a touch event is canceled while the program is running. Touches are typically canceled because another event, like an incoming phone call, interrupted the touch.

Depending on the program, touches could apply to something specific on the graphics screen like drawing a line in a paint program, or translating or rotating a plot of a function. It can even be used for both. The program gets the touch event before any plots. If the plot should not see the touch event, use `e.setConsumed(1)` to consume the event before returning from the subroutine. If the touch should be passed on to any plot for processing translations and rotations, don't make the call to `setConsumed`. If the program does not have plots, don't worry about calling `setConsumed`, since it won't make any difference in how the event is processed.

`e` is an Event object containing the event type, location, and time. See the Event class for a complete description of this object.

---

**SUB touchesEnded (e AS EVENT)**

Touches ended events are generated whenever a touch event completes because the user lifted his finger while the program is running.

Depending on the program, touches could apply to something specific on the graphics screen like drawing a line in a paint program, or translating or rotating a plot of a function. It can even be used for both. The program gets the touch event before any plots. If the plot should not see the touch event, use `e.setConsumed(1)` to consume the event before returning from the subroutine. If the touch should be passed on to any plot for processing translations and rotations, don't make the call to `setConsumed`. If the program does not have plots, don't worry about calling `setConsumed`, since it won't make any difference in how the event is processed.

`e` is an Event object containing the event type, location, and time. See the Event class for a complete description of this object.

---

**SUB touchesMoved (e AS EVENT)**

Touches moved events are generated whenever the user moves his finger after a touches began event and before the matching touches ended event.

Depending on the program, touches could apply to something specific on the graphics screen like drawing a line in a paint program, or translating or rotating a plot of a function. It can even be used for both. The program gets the touch event before any plots. If the plot should not see the touch event, use `e.setConsumed(1)` to consume the event before returning from the subroutine. If the touch should be passed on to any plot for processing translations and rotations, don't make the call to `setConsumed`. If the program does not have plots, don't worry about calling `setConsumed`, since it won't make any difference in how the event is processed.

**e** is an Event object containing the event type, location, and time. See the Event class for a complete description of this object.

---

**SUB touchUpInside (ctrl AS Button, time AS DOUBLE)**

Touch up inside events are generated whenever the user lifts a finger while it is touching a `Button`. This is the proper place to handle any action associated with tapping the button.

**ctrl** is the `Button` object where the text was changed.

**time** is the time when the event was generated. It is the number of seconds since the start of January 1, 2001 using Greenwich Mean Time. Note that it is possible for the time to jump a bit due to the clock being reset, either by the user or by synchronization with an external source.

See the `Button` class for numerous samples that implement `touchUpInside`.

---

**SUB valueChanged (ctrl AS Control, time AS DOUBLE)**

Value changed events are generated whenever the value of a control changes.

**ctrl** is the `Button` object where the text was changed.

**time** is the time when the event was generated. It is the number of seconds since the start of January 1, 2001 using Greenwich Mean Time. Note that it is possible for the time to jump a bit due to the clock being reset, either by the user or by synchronization with an external source.

The controls that can generate value changed events are:

Control	Description
<code>ColorPicker</code>	Generated when a new color is selected.
<code>DatePicker</code>	Generated when the date displayed by the date picker is changed by the user.
<code>MapView</code>	Generated when an annotation is selected or deselected, when the map view changes the region being displayed in response to a swipe or pinch, and when the user location is updated. Use the various methods in the <code>MapView</code> class to determine the new status of the annotations, the displayed map region, and the user location.
<code>Picker</code>	Generated when any of the wheels displayed by the picker is changed by the user.
<code>SegmentedControl</code>	Generated when the selected button in the segmented control is changed due to user action.
<code>Slider</code>	Generated when the user slides the thumb on the slider.
<code>Stepper</code>	Generated when the stepper value is changed due to tapping or holding by the user.
<code>Switch</code>	Generated when the switch changed from on to off or off to on due to user action.
<code>TextField</code>	Generated when the user presses the return key (or its equivalent) while editing text.

See any of the classes that generate value-changed events for sample programs that implement `valueChanged`.



## Chapter 14 – System Classes

System classes return general information not associated with the graphical user interface (GUI) of a program or with plots.

---

### Date

The `Date` class stores a date and time, and provides methods to access the date and time in various forms. For example, the program

```
DIM d as Date
d = system.date

PRINT "The long date "; d.longDate
PRINT "The short date "; d.shortDate
PRINT "The long time "; d.longTime
PRINT "The short time "; d.shortTime
PRINT "The year "; d.year
PRINT "The month "; d.month
PRINT "The day is "; d.day
PRINT "The hour "; d.hour
PRINT "The minute "; d.minute
PRINT "The second "; d.second
PRINT "The day of the week "; d.dayOfWeek
PRINT "The day of the year "; d.dayOfYear
```

prints the various date fields. The date and time is the current date and time on your computer, as represented by your computer's clock. An example of the output is

```
The long date September 13, 2011
The short date 09/13/11
The long time 08:49:19 AM MDT
The short time 08:39 AM
The year 2011
The month 8
The day is 13
The hour 8
The minute 49
The second 19
The day of the week 3
The day of the year 256
```

Note that the month is September, but the numeric value is 8, not 9. Months are numbered from 0 to 11. The specific format for the date and time will vary from one local to the next, using the current user settings. The values shown are for dates and times in the United States.

---

#### **FUNCTION day AS INTEGER**

Returns the day of the month. This is a number from 1 to 31.

---

#### **FUNCTION dayOfWeek AS INTEGER**

Returns the day of the week. This is a number from 1 to 7, where Sunday is 1, Monday is 2, and so forth.

**FUNCTION dayOfYear AS INTEGER**

Returns the day of the year. This is a number from 1 to 366, where January 1<sup>st</sup> is 1, February 1<sup>st</sup> is 32, and so forth.

**FUNCTION hour AS INTEGER**

Returns the hour of the day. This is the hour on a 24 hour clock, so 9:00 AM is 9, while 9:00 PM is 21.

**FUNCTION longDate AS STRING**

Returns the date in a long format. The format depends on the local operating system. On iOS in the United States, the format is mm/dd/yyMonth day, year, where mm Month is the month numbername, dd day is the day number, and yy year is the last two digits of the year.

**FUNCTION longTime AS STRING**

Returns the time in a long format. The format depends on the local operating system. On iOS in the United States, the format is hh:mm:ss ap zzz, where hh is the hour on a 24 12 hour clock, mm is the minute, and ss is the seconds, ap is AM or PM and zzz is the time zone.

**FUNCTION minute AS INTEGER**

Returns the minute. This is a number from 0 to 59.

**FUNCTION month AS INTEGER**

Returns the month number. Months are numbered from 1 to 12, so 1 is January and 12 is December.

**FUNCTION second AS INTEGER**

Returns the second. This is a number from 0 to 59.

**FUNCTION shortDate AS STRING**

Returns the date in a short format. The format depends on the local operating system. On iOS in the United States, the format is mm/dd/yy, where mm is the month number, dd is the day number, and yy is the last two digits of the year.

**FUNCTION shortTime AS STRING**

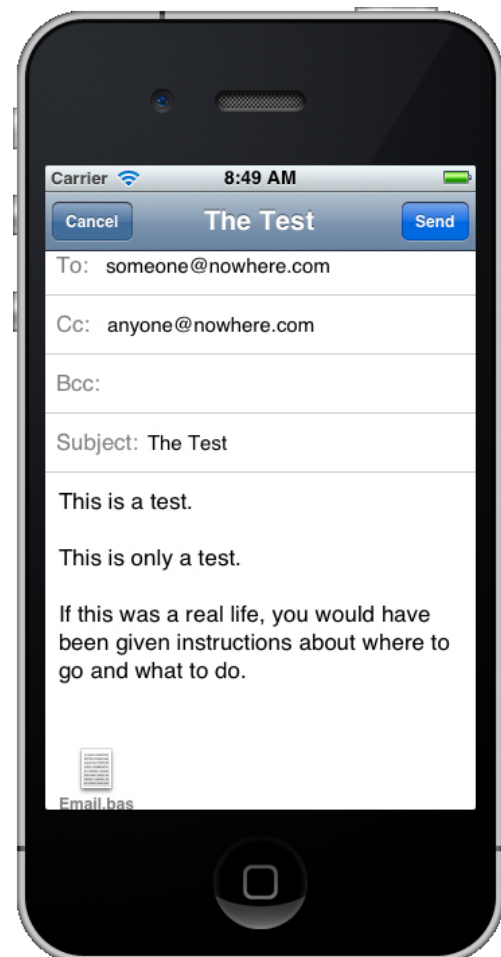
Returns the time in a short format. The format depends on the local operating system. On iOS in the United States, the format is hh:mm ap:ss, where hh is the hour on a 24 12 hour clock, mm is the minute, and ss ap is the secondsAM or PM.

**FUNCTION year AS INTEGER**

Returns the year, for example, 2011.

## Email

The `Email` class is used for sending emails under program control. To protect privacy and prevent abuse, iOS limits this feature by requiring all email messages to be approved by the user before being sent. This means the `Email` class sends the email by passing it on to the iOS email client, and the user has the option of modifying the email before it is sent or canceling the email completely.





Email messages are initially created by a call to the `newEmail` method in the `System` class. Once an `Email` object is created, it can be used multiple times to create and send email messages.

The following sample shows a short program that creates an email with an attachment and prepares it for sending. If the program is named `Email`, the attachment is the source code for the program itself. Assuming the email addresses were valid, all the user has to do is tap `Send`.

#### Snippet

```
DIM em AS Email
msg$ = "This is a test." & CHR(10)
msg$ = msg$ & CHR(10) & "This is only a test." & CHR(10)
msg$ = msg$ & CHR(10) & "If this was a real life, you would have been
given instructions about where to go and what to do."

em = System.newEmail("someone@nowhere.com", "The Test", msg$)
em.addCC("anyone@nowhere.com")
em.addAttachment("Email.bas", "text/plain")

em.send
```

---

#### **SUB addAttachment (path AS STRING, mimeType AS STRING)**

Adds an attachment to the email message.

The name of the file is passed using the **path** parameter. The mime type for the file must also be specified; it is passed in the **mimeType** parameter. A list of valid mime types, along with background information, is available at <http://www.iana.org/assignments/media-types/index.html>. A few common mime types are:

Mime Type	File Description
image/jpeg	JPEG image file
image/png	PNG image file
text/plain	ASCII text, including techBASIC source files

See the snippet at the start of this class for a sample program that uses the `addAttachment` method.

---

#### **SUB addBCC (address AS STRING)**

Adds a blind copy address to the email. Blind copy recipients will get the email, but their address will not be visible to others who also get the email.

The address must be a valid email address.

Multiple blind copy addresses may be added by making multiple calls to `addBCC`.

---

#### **SUB addCC (address AS STRING)**

Adds a copy address to the email.

The address must be a valid email address.

Multiple copy addresses may be added by making multiple calls to `addCC`.

See the snippet at the start of this class for a sample program that uses the `addCC` method.

---

#### **SUB addTo (address AS STRING)**

Adds an address to the email. The email must have at least one address to be sent.

The address must be a valid email address.

Multiple addresses may be added by making multiple calls to `addTo`.

See the snippet at the start of this class for a sample program that uses the `addTo` method.

---

**FUNCTION canSendMail AS INTEGER**

Returns one if the current device can send emails, and zero if not. Use this function if your program will run on multiple devices, and you need to find out in advance if sending the email is possible.

---

**SUB clearAttachments**

Removes all attachments from the email.

---

**SUB clearBCC**

Removes all blind copy addresses from the BCC field of the email. Use this subroutine to change the blind copy address, allowing a single email object to be used to send several emails.

---

**SUB clearCC**

Removes all copy addresses from the CC field of the email. Use this subroutine to change the copy address, allowing a single email object to be used to send several emails.

---

**SUB clearTo**

Removes all addresses from the to field of the email. Use this subroutine to change the address, allowing a single email object to be used to send several emails.

---

**SUB send**

Send the email. This subroutine sends the email to the email application, where the user can modify it, send it, or cancel it.

See the snippet at the start of this class for a sample program that uses the `send` method.

---

**SUB setMessage (message AS STRING)**

Sets the message for the email. This replaces any previous message, allowing the same email object to be used to send several emails.

Use `CHR(10)` characters to create new lines within an email.

See the snippet at the start of this class for a sample program that uses the `setMessage` method.

---

**SUB setSubject (subject AS STRING)**

Sets the subject for the email. This replaces any previous subject, allowing the same email object to be used to send several emails.

See the snippet at the start of this class for a sample program that uses the `setSubject` method.

---

**FUNCTION succeeded AS INTEGER**

Returns one if the most recent email was successfully sent or canceled by the user, and zero if there was an error.

---

## Err

The `Err` class is used for tracking errors. A global variable by the same name is predefined in every BASIC program. When an error is encountered, the predefined `Err` object is filled in with information about the error. `ON ERROR` handlers can interrogate the `Err` variable to determine the cause of the error.

Snippet

```
ON ERROR GOTO 1
foo: i = 1e20
i% = i
END
```

1

```
PRINT err.description
PRINT err.label
PRINT err.erl
PRINT err.number
```

The snippet will print:

```
Runtime error: Integer overflow
FOO
-1
2
```

---

#### **FUNCTION description AS STRING**

Returns a text description of the error. This is the same text that would be printed as the run time error message if an `ON ERROR` handler were not used.

The various error messages and the associated error numbers are listed in Appendix A.

---

#### **FUNCTION label AS STRING**

Returns the name of the label most recently encountered before the error. The label name is converted to all uppercase letters.

If the most recent label was numeric, the number is converted to a string.

If no label has been encountered when the error occurs, the label will be the empty string.

---

#### **FUNCTION erl AS INTEGER**

Returns the line number most recently encountered before the error. If a label was encountered, the value returned is -1. If no label or line number has been encountered when the error occurs, the value returned is 0.

---

#### **FUNCTION number AS INTEGER**

Returns the error number for the most recent error. See *description*, above, for the possible error numbers.

---

## Event

Event objects are created and passed as a parameter for several of the subroutines that are called in response to events, including `tap`, `touchesBegan`, `touchesCanceled`, `touchesEnded` and `touchesMoved`. The events and the subroutines that handle them are described in Chapter 13.

The methods in this class allow extracting information from the event and, in the case of `setConsumed`, to prevent the event from being handled later in the event chain.

See *Event Driven Programs* in Chapter 13 for a sample program that uses tap and touch events to create a simple paint program.

---

#### **FUNCTION consumed AS INTEGER**

Returns one if the event was marked as consumed by a previous call to `setConsumed`, or zero if the event has not been marked as consumed.

---

#### **SUB setConsumed (consumed AS INTEGER)**

Sets the consumed flag. Pass zero if the event should not be consumed, or a non-zero value to consume the event.

Touch and tap events can be tracked by the program to respond to gestures on the graphics screen, or they can be used by plots to create or remove callouts, translate the plot, rotate the plot or resize the plot. Events are passed first to any event handling subroutine in the program, then to plots. If the program's event handling subroutine sets the consumed flag using this method, the event will not be passed on to the plot.

The consumed flag defaults to zero for all events, so if the program's event handling subroutines do not make this call, the event will be processed both by the program's event handler and by the plot.

---

**FUNCTION what AS INTEGER**

Returns the kind of the event.

<b>what</b>	Event Kind
1	tap
2	touches began
4	touches moved
8	touches canceled
16	touches ended

---

**FUNCTION when AS DOUBLE**

Returns the time when the event was generated. It is the number of seconds since the start of January 1, 2001 using Greenwich Mean Time. Note that it is possible for the time to jump a bit due to the clock being reset, either by the user or by synchronization with an external source.

---

**FUNCTION where AS DOUBLE (n, 2)**

Returns the current location of all touches associated with the event.

The number of touches is not limited to tracking a touch by a single finger. When multiple fingers are touching the screen, each of the touch locations is returned. The resulting array can have any number of entries. Each row in the array represents one of these touch events. In each row, the first value is the horizontal location of the touch, while the second is the vertical location where zero is at the top of the graphics screen.

Use UBOUND to determine the number of touches.

The snippet shows how to count the number of touches. It looks at the number of touches for touches down and touches moved events, displaying the number in a label. Touches ended events do still report the number of fingers when the gesture ended, but the program takes that opportunity to report that there are no touches.

Snippet

```
! Show the number of fingers touching the screen.
```

```
DIM message AS Label
message = Graphics.newLabel(10, 10, 300)
```

```
DIM quit AS Button
quit = Graphics.newButton(10, 40)
quit.setTitle("Quit")
```

```
System.showGraphics
END
```

```
SUB touchesBegan (e AS Event)
countTouches(e)
END SUB
```

```
SUB touchesMoved (e AS Event)
countTouches(e)
END SUB
```

```
SUB touchesEnded (e AS Event)
message.setText("0 touches")
END SUB
```

```

SUB touchUpInside (ctrl AS Button, when AS DOUBLE)
STOP
END SUB

SUB countTouches (e AS Event)
where = e.where
touches = UBOUND(where, 1)
IF touches = 1 THEN
    message.setText("1 touch")
ELSE
    message.setText(STR(touches) & " touches")
END IF
END SUB

```

---

## Math

The `Math` class is a collection of subroutines and functions that extend the capabilities normally found in a computer programming language to include numeric integration, regression, linear algebra and special functions.

A predefined object by the same name gives you access to the methods in this class. Multiple examples show how to get the most from the data.

---

### **FUNCTION erf (x AS DOUBLE) AS DOUBLE**

The error function, generally abbreviated as `erf`, is defined as

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

It is used in many fields, but particularly in probability and statistics, where it is closely related to the normal cumulative distribution function.

---

### **FUNCTION erfc (x AS DOUBLE) AS DOUBLE**

The inverse error function, generally abbreviated `erfc`, is defined as

$$\text{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt$$

This means

$$\text{erfc}(x) = 1 - \text{erf}(x)$$

The reason for a separate function is that for values of `erf(x)` that are close to 1, subtracting `erf(x)` from 1 is not particularly accurate. Using `erfc` gives better results for these cases.

---

### **FUNCTION gamma (x AS DOUBLE) AS DOUBLE**

The gamma function is defines as

$$\Gamma(n) = \int_0^\infty e^{-t} t^{n-1} dt$$

For integers, this is equivalent to

$$\Gamma(n) = (n - 1)!$$

Snippet

```
! Print the factorials from 1 to 10.
FOR i = 1 TO 10
  PRINT i; "! = "; Math.gamma(i + 1)
NEXT
```

**FUNCTION isInf (x AS DOUBLE) AS INTEGER**

Returns -1 if the argument is negative infinity, 0 if the argument is finite, and 1 if the argument is positive infinity. Since BASIC treats any non-zero value as true and 0 as false, this allows the function to serve a double purpose. The result can be used as a logical test in statements like IF, but it can also be used to indicate both whether the value is infinity and the sign of the infinity.

Snippet

```
! Print all of the factorials that a double will hold.
i = 1
f# = 1
WHILE NOT Math.isInf(f#)
  PRINT i; "! = "; f#
  i = i + 1
  f# = f#*i
WEND
```

**FUNCTION isNaN (x AS DOUBLE) AS INTEGER**

Returns 1 if the argument is not a number, and 0 if it is.

A value that is not a number, printed NaN, is returned when an operation is performed that does not result in a valid number, such as taking the square root of a negative number.

**FUNCTION logGamma (x AS DOUBLE) AS DOUBLE**

Returns the natural log of the gamma function.

The gamma function grows very large for fairly modest inputs. In many applications, the result of a gamma function is often divided by another gamma function. While each might individually overflow a double-precision value, the division does not. In these cases, taking the exponent of the difference of two logGamma functions is equivalent, but won't cause a numeric overflow.

Snippet

```
! Find the number of ways to choose three items from a group of 16
! where order does not matter.
PRINT binomial(16, 3)

FUNCTION binomial (n AS DOUBLE, r AS DOUBLE) AS DOUBLE
  binomial = EXP(Math.logGamma(n + 1) - Math.logGamma(r + 1) -
Math.logGamma(n - r + 1))
END FUNCTION
```

**FUNCTION LUBackSub (LU AS DOUBLE(), pivot AS INTEGER(), b AS DOUBLE()) AS DOUBLE()**

Given the result of a call to LUDComp, LUBackSub solves a series of linear equations.

Beginning with the linear equations

$$Ax = b$$

LUDComp converts  $A$  into two matrices,  $L$  and  $U$ , that are lower diagonal and upper diagonal. These are stored in a single matrix. LUDComp may swap rows while doing the conversion; it returns a pivot matrix, which gives the location of the original rows in the decomposed matrices. LUBackSub can then be called, passing the vector  $b$  and returning the solution vector  $x$ . Note that LUBackSub can be called multiple times to solve for multiple  $b$  vectors without redoing the decomposition.

See LUDComp for an example that used LUBackSub.

---

**FUNCTION LUDComp (a AS DOUBLE(), BYREF pivot AS INTEGER(), BYREF sign AS DOUBLE) AS DOUBLE(),**

Creates the LU decomposition of a square matrix. This is a workhorse routine used for many problems in linear algebra. One use is to solve a system of linear equations multiple times.

Begin a system of equations represented by the matrix equation

$$Ax = b$$

where we want to solve for  $x$  given  $A$  and  $b$ . In fact, it is common to want to solve for multiple  $x$  matrices given multiple  $b$  matrices, all for a single  $A$  matrix. LU Decomposition expresses the matrix  $A$  as two matrices, one upper diagonal and one lower diagonal, like this:

$$A = LU$$

or

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

We can now solve the original linear equation using an intermediate, first solving

$$Ly = b$$

for  $y$ , then solving

$$Ux = y$$

for  $x$ . This is much simpler than the original problem, since the  $U$  and  $L$  matrices are triangular, and easily solved by back substitution. In fact, the LUBackSub call in this class does just that.

In practice, LU decomposition works best (and in some forms, only) with partial pivoting, where rows may be interchanged. The pivot parameter is a vector recording the positions of the pivoted rows. For example, if no pivoting was needed, the array would contain the values 1 to  $n$ , in order, while if the rows were exactly reversed, the values would be  $n$  to 1. The pivot matrix is one of the inputs to LUBackSub. Note that it is passed by reference, so the values in the original matrix are replaced by the pivot values.

The last parameter is set to 1 if the number of rows interchanged was even, and -1 if the number of rows is odd. This value is not needed by LUBackSub, but is used for some operations, such as finding the determinant of the matrix.

It is fair to ask why this is better than finding the inverse of  $A$  and multiplying it by  $b$ . For many problems, this will work just fine, but for a wide range of interesting problems, the inverse of  $A$  is difficult to find accurately enough to provide good results. LUDComp and LUBackSub are more numerically stable, and more likely to yield useable results.

## Chapter 14: System Classes

### Snippet

```
! Solve the linear equation Ax=b for x.
DIM LU(1, 1) as double, A(1, 1) as double
DIM P(1) as integer, d as integer, b(3) as double
A = [[1, 2, -1],
     [4, 3, 1],
     [2, 2, 3]]
b = [2, 3, 5]
LU = Math.LUDComp(A, P, d)
x = Math.LUBackSub(LU, P, b)

PRINT "Solves Ax = b for x."
PRINT
PRINT "A = "
PRINT A
PRINT "b = "; b
PRINT "x = "; x
```

---

### FUNCTION mean (x AS DOUBLE()) AS DOUBLE

Find the mean (average) of a list of numbers.

### Snippet

```
! Find the GPA for three A's and one C.
PRINT Math.mean([4, 4, 4, 2])
```

---

### FUNCTION normal () AS DOUBLE

Generates a pseudo random number. The numbers generated are normally distributed with a mean of zero and standard deviation of one.

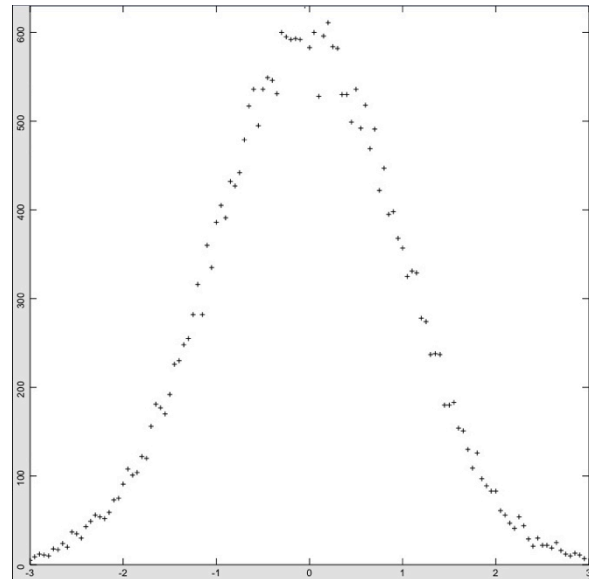
This function is an implementation of the Mersenne Twister random number generator, generally regarded as a good random number generator for most simulation use.

Use the `setSeed` call in this class to set the random number seed. Resetting the seed using `setSeed` with the same seed will regenerate the same sequence of random numbers. If a call to `normal` is made before calling `setSeed`, the seed is initialized from the system clock.

`normal` shares the same underlying random number generator as `rand`, another call in this class.

### Snippet

```
! Collect random numbers into bins,
! plotting the result to show how
! numbers are distributed in a normal
! distribution.
DIM bins(-60 TO 60, 2)
FOR i = 1 TO 30000
    j = Math.normal*20
    IF j >= -60 AND j <= 60 THEN
        bins(j, 2) = bins(j, 2) + 1
    END IF
NEXT
max = 0
FOR i = -60 TO 60
    bins(i, 1) = i/20
    IF bins(i, 2) > max THEN
        max = bins(i, 2)
    END IF
NEXT
```





```

DIM p AS PLOT, normal AS PlotPoint
p = Graphics.newPlot
normal = p.newPlot(bins)
normal.setStyle(0)
normal.setPointStyle(2)
p.setView(-3, 0, 3, max, 0)

System.showGraphics

```

---

**FUNCTION packDbl (value AS DOUBLE, bigEndian AS INTEGER = 0) AS INTEGER ()**

Returns an eight-element array with the value of the double-precision number converted to eight bytes. If the value passed is not a double-precision number, it is converted to a double-precision number using the normal unary conversion rules.

Double-precision numbers are stored internally as an eight-byte IEEE format number. This method will convert a double-precision number into eight individual bytes stored in an eight-element array. By default, the first byte will be the least significant byte, while the last byte will be the most significant byte; this is known as little-endian order. This is the most common byte order for Intel chips and iOS. Some micro-controllers and most network connections reverse the byte order, known as big-endian. `packDbl` returns the bytes in big-endian order if the **bigEndian** parameter to a non-zero value.

For example,

```
bytes = Math.packDbl(4)
```

returns an array with the values \$40, \$10, \$00, \$00, \$00, \$00, \$00 and \$00, while

```
bytes = Math.packDbl(4, 1)
```

returns an array with the values \$00, \$00, \$00, \$00, \$00, \$00, \$10 and \$40.

---

**FUNCTION packInt (value AS INTEGER, bigEndian AS INTEGER = 0) AS INTEGER ()**

Returns a two-element array with the value of the integer converted to two bytes. If the value passed is not an integer, it is converted to an integer using the normal unary conversion rules.

Integers are stored internally as a two-byte two's complement number. This method will convert an integer into two individual bytes stored in a two-element array. By default, the first byte will be the least significant byte, while the last byte will be the most significant byte; this is known as little-endian order. This is the most common byte order for Intel chips and iOS. Some micro-controllers and most network connections reverse the byte order, known as big-endian. `packInt` returns the bytes in big-endian order if the **bigEndian** parameter to a non-zero value.

For example,

```
bytes = Math.packInt(4)
```

returns an array with the values \$04 and \$00, while

```
bytes = Math.packInt(-4)
```

returns an array with the values \$FC and \$FF.

```
bytes = Math.packInt(4, 1)
```

returns an array with the values \$00 and \$04.

---

**FUNCTION packLng (value AS LONG, bigEndian AS INTEGER = 0) AS INTEGER ()**

Returns a four-element array with the value of the long integer converted to four bytes. If the value passed is not a long integer, it is converted to a long integer using the normal unary conversion rules.

Long integers are stored internally as a four-byte two's complement number. This method will convert a long integer into four individual bytes stored in a four-element array. By default, the first byte will be the least significant byte, while the last byte will be the most significant byte; this is known as little-endian order. This is the most common byte order for Intel chips and iOS. Some micro-controllers and most network connections reverse the byte order, known as big-endian. `packLng` returns the bytes in big-endian order if the **bigEndian** parameter to a non-zero value.

For example,

```
| bytes = Math.packLng(4)
```

returns an array with the values \$04, \$00, \$00 and \$00, while

```
| bytes = Math.packLng(-4)
```

returns an array with the values \$FC, \$FF, \$FF and \$FF.

```
| bytes = Math.packLng(4, 1)
```

returns an array with the values \$00, \$00, \$00 and \$04.

---

**FUNCTION packSng (value, bigEndian AS INTEGER = 0) AS INTEGER ()**

Returns a four-element array with the value of the single-precision number converted to four bytes. If the value passed is not a SINGLE number, it is converted to a single-precision number using the normal unary conversion rules.

Single-precision numbers are stored internally as a four-byte IEEE format number. This method will convert a single-precision number into four individual bytes stored in a four-element array. By default, the first byte will be the least significant byte, while the last byte will be the most significant byte; this is known as little-endian order. This is the most common byte order for Intel chips and iOS. Some micro-controllers and most network connections reverse the byte order, known as big-endian. `packSng` returns the bytes in big-endian order if the **bigEndian** parameter to a non-zero value.

For example,

```
| bytes = Math.packSng(4)
```

returns an array with the values \$40, \$80, \$00, and \$00, while

```
| bytes = Math.packSng(4, 1)
```

returns an array with the values \$00, \$00, \$80 and \$40.

---

**FUNCTION PI () AS DOUBLE**

Returns the value of  $\pi$  in double precision.

Snippet

```
! Find the sine of 60 degrees.
PRINT SIN(Math.PI/3)
```

---

**FUNCTION poly (coef AS DOUBLE(), x AS DOUBLE) AS DOUBLE**

Evaluate a polynomial. The `coef` array contains the polynomial coefficients in increasing order. For example,

```
c = [1, 2, 3]
y = poly(c, x)
```

is equivalent to

```
y = c(1) + c(2)*x + c(3)*x*x
```

although the call to `poly` uses a more efficient way to evaluate the polynomial.

See `polyFit` for an example that uses `poly`.

**FUNCTION polyFit (x AS DOUBLE(), y AS DOUBLE(), order AS INTEGER = 1) AS DOUBLE()**

Performs a least-squares regression on the X and Y pairs passed in the two arrays. The fit can be any order of polynomial, beginning with a linear fit (first order polynomial). The coefficients for the polynomial are returned, with the lower powers appearing first.

For example, for the call

```
coefs = Math.polyFit(x, y, 2)
```

a vector with three elements is returned. One way to find the fitted value at  $x=4$  is

```
y = coefs(1) + coefs(2)*x + coefs(3)*x*x
```

A better way to find the fit is using the `poly` method from this class.

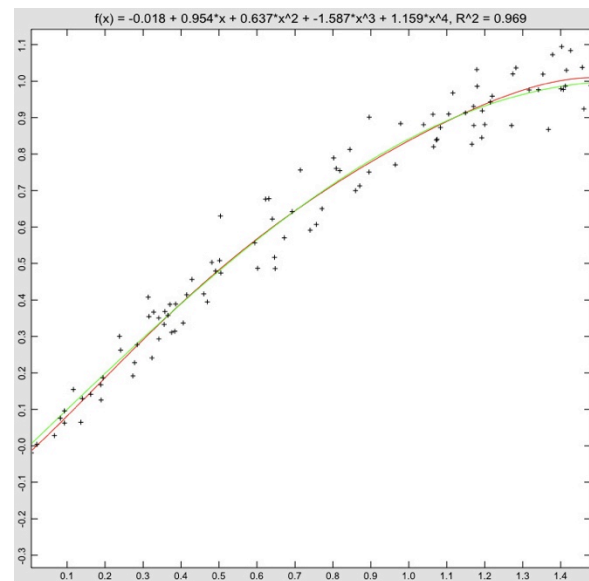
```
y = Math.poly(coefs, 4)
```

Use the `rSquare` method from this class to find the goodness of fit.

#### Snippet

```
! Generate data points for a
! regression using normally
! distributed variation from f(x)
! over the range 0 to 1.5.
Math.setSeed(42)
points = 100
DIM x(points), y(points)
DIM pairs(points, 2)
FOR i = 1 TO points
  x(i) = Math.rand*1.5
  y(i) = Math.normal/20 + f(x(i))
  pairs(i, 1) = x(i)
  pairs(i, 2) = y(i)
NEXT

! Do the regression.
order = 5
coef = Math.polyfit(x, y, order)
```



## Chapter 14: System Classes

```
! Draw the random points and the fit. Also draw the function the
! points were drawn from.
DIM p AS Plot
p = Graphics.newPlot

DIM scatter AS PlotPoint
scatter = p.newPlot(pairs)
scatter.setStyle(0)
scatter.setPointStyle(2)

DIM fitPlot AS PlotFunction
fitPlot = p.newFunction(function fit)
fitPlot.setColor(1, 0, 0)

DIM fPlot AS PlotFunction
fPlot = p.newFunction(FUNCTION f)
fPlot.setColor(0, 1, 0)

title$ = "f(x) = " & format(coef(1))
FOR i = 2 TO order
    title$ = title$ & " + " & format(coef(i))
    IF i = 2 THEN
        title$ = title$ & "*x"
    ELSE
        title$ = title$ & "*x^" & STR(i - 1)
    END IF
NEXT
title$ = title$ & ", R^2 = " & format(Math.rSquare(x, y, coef))
p.setTitle(title$)

System.showGraphics

! This function evaluates the fit. It is used to plot the
! fit for comparison with the points and the actual function.
FUNCTION fit (x)
    fit = Math.poly(coef, x)
END FUNCTION

! This is the function used to generate the random points
! the program fits. It is also used to show the actual line
! the fit should reproduce.
FUNCTION f(x)
    f = sin(x)
END FUNCTION
```

```

! Format a value with three decimal digits.
FUNCTION format (x AS DOUBLE) AS STRING
sign$ = ""
IF x < 0 THEN
    x = -x
    sign$ = "-"
END IF
x$ = STR(INT(x*1000))
WHILE LEN(x$) < 4
    x$ = "0" & x$
WEND
format = sign$ & LEFT(x$, LEN(x$) - 3) & "." & RIGHT(x$, 3)
END FUNCTION

```

---

**FUNCTION rand AS DOUBLE**

Generates a pseudo random number in the range zero to one, inclusive of zero. Due to the way the generator works, there is a possibility a number very close to one will be rounded to one, so guard against this possibility in code using this call.

This function is similar to the RAND function built into BASIC, but the two random number generators have different properties. The RAND function built into BASIC is fast and reasonably good for general use. This function is an implementation of the Mersenne Twister random number generator, generally regarded as a good random number generator for most simulation use.

Use the `setSeed` call in this class to set the random number seed. Resetting the seed using `setSeed` with the same seed will regenerate the same sequence of random numbers. If a call to `rand` is made before calling `setSeed`, the seed is initialized from the system clock.

`rand` shares the same underlying random number generator as `normal`, another call in this class.

**Snippet**

```

! Shuffle a deck of cards, represented by an array containing the
! ordinal number of each card.
DIM cards(52) AS INTEGER
FOR i = 1 TO 52
    cards(i) = i
NEXT

FOR i = 1 TO 52
    j = 0
    WHILE j < 1 OR j > 52
        j = 1 + Math.rand*52
    WEND
    temp = cards(i)
    cards(i) = cards(j)
    cards(j) = temp
NEXT

FOR i = 1 TO 52
    PRINT cards(i)
NEXT

```

---

**FUNCTION romb (FUNCTION f, a AS DOUBLE, b AS DOUBLE, error AS DOUBLE = 1.0e-6, steps AS INTEGER = 16, order AS INTEGER = 5) AS DOUBLE**

`romb` is an implementation of Romberg integration. Romberg integration allows for more rapid solutions to integrals that are sufficiently smooth. It is the method of choice for programs that must evaluate an integral repeatedly, and the function being evaluated is known to be smooth enough for Romberg integration. To get an idea how much more efficient it can be, for the function

```
function f (x as double) as double
f = x*x*x*x*log(x + sqr(x*x + 1))
end function
```

Romberg integration returns a result accurate to six digits after only 33 evaluations of the function, while trapezoidal integration requires 4097 function evaluations for the same accuracy.

The first input is the function to integrate. a and b are the limits of integration.

Romberg integration works by successively dividing the integration interval in half. On the first iteration, it simply evaluates the endpoints. The second iteration adds the center point. The third iteration adds two more points, and so forth. The number of function evaluations required for a specific number of these steps is  $1+2^{(n-1)}$ , where n is the number of steps performed. The `steps` parameter sets the maximum number of iterations allowed before the algorithm gives up and reports a run time error that the desired accuracy could not be achieved, preventing an infinite loop while trying to achieve an accuracy that cannot be reached. After the default 16 steps, the function will have been evaluated 32,769 times. For most functions evaluated to an error of  $1e-6$ , the desired accuracy is reached much, much sooner. Normally, Romberg integration stops with a solution when two successive evaluations return results that differ only after the first six significant digits match.

Unlike trapezoidal integration, implemented in this class using the `trap` call, Romberg integration essentially fits a curve to the points rather than a straight line. The `order` parameter specifies the order of the curve. The order must be at least one greater than the maximum number of allowed steps to allow for enough points to evaluate the specified order polynomial. An order of 2 is equivalent to integration using Simpson's Rule.

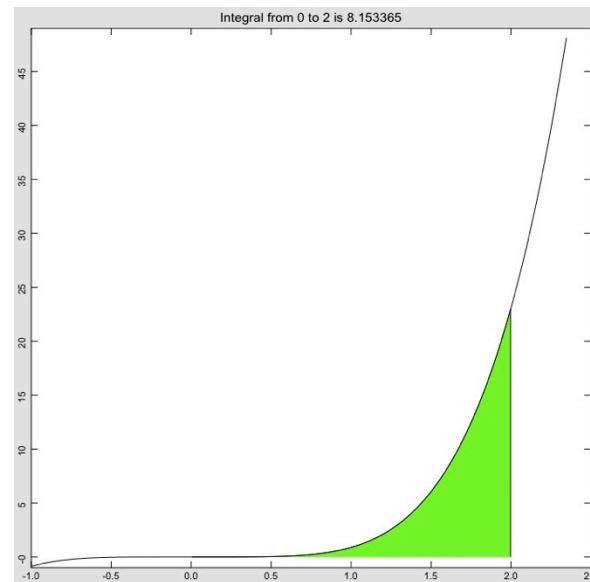
See `trap` and `trapF` for alternatives to `romb` for numeric integration.

#### Snippet

```
! Evaluate the integral of
!   x*x*x*x*log(x + sqr(x*x + 1))
! over the interval 0 to 2.
intfx = math.romb(function f, 0, 2)

DIM p AS Plot, pf AS PlotFunction,
pfi AS PlotFunction
p = Graphics.newPlot
pf = p.newFunction(FUNCTION f)
pfi = p.newFunction(FUNCTION f)
pfi.setDomain(0, 2)
pfi.setFillColor(0, 1, 0)
p.setView(-1, -1, 2.5, 49, 0)
p.setTitle("Integral from 0 to 2 is "
& STR(intfx))
System.showGraphics

function f (x as double) as double
f = x*x*x*x*log(x + sqr(x*x + 1))
end function
```



#### **FUNCTION rSquare (x AS DOUBLE(), y AS DOUBLE(), p AS DOUBLE()) AS DOUBLE**

Calculates the goodness of fit for a polynomial fit to a set of points contained in the `X` and `Y` arrays. Values close to 1 indicate a good fit, while values close to 0 indicate a poor fit.

The actual method used is

$$R^2 = \frac{\sum_{i=1}^n (y_i - f(x_i))^2}{\sum_{i=1}^n (y_i - \bar{y})^2}$$

where  $y_i$  is one of the y values passed as input,  $f(x_i)$  is the polynomial evaluated at the x value, and  $\bar{y}$  is the mean value for all of the y values passed.

See `polyFit` for a way to generate a fit to a set of data, as well as for an example of `rSquare` in use.

---

**SUB setSeed (x AS LONG)**

Sets the seed for the pseudo random number generator used to generate numbers for the `rand` and `normal` calls in this class.

The best random number seeds will be fairly random themselves, containing roughly the same number of zero and one bits distributed across the entire range of the 32 bit long value passed as the parameter. If a poor seed is used, generate a few dozen to a few hundred random numbers by calling `rand`, and discard these.

Resetting the seed using `setSeed` with the same seed will regenerate the same sequence of random numbers. If a call to `rand` or `normal` is made before calling `setSeed`, the seed is initialized from the system clock.

**Snippet**

```
! Generate the same 10 pseudo random numbers twice by resetting the seed.
Math.setSeed(-459908563)
FOR i = 1 TO 10
    PRINT Math.rand
NEXT

Math.setSeed(-459908563)
FOR i = 1 TO 10
    PRINT Math.rand
NEXT
```

---

**FUNCTION stDev (x AS DOUBLE()) AS DOUBLE**

Returns the sample standard deviation of a list of numbers. The sample standard deviation is defined as

$$s = \sqrt{\frac{1}{N-1} \sum_{i=1}^N (x_i - \bar{x})^2}$$

The sample standard deviation is used when the values are draws from a larger population. For the population standard deviation, multiply the result by

$$\sqrt{\frac{N-1}{N}}$$

**Snippet**

```
! Find the standard deviation in the height of a group of people.
PRINT Math.stDev([76, 75, 68, 71, 69])
```

---

**FUNCTION trap (FUNCTION f, a AS DOUBLE, b AS DOUBLE, error AS DOUBLE = 1.0e-6, steps AS INTEGER = 16) AS DOUBLE**

The `trap` function provides a workhorse call for numeric integration. For one-time use or when the characteristics of a function are not well understood, this is the method of choice for numeric integration.

The first input is the function to integrate. `a` and `b` are the limits of integration. The desired error and number of steps are optional parameters. The error should not be smaller than `1e-6` for single-precision functions, or smaller than about `1e-14` for double-precision functions.

The trapezoid rule works by successively dividing the integration interval in half. On the first iteration, it simply evaluates the endpoints. The second iteration adds the center point. The third iteration adds two more points, and so forth. The number of function evaluations required for a specific number of these steps is  $1+2^{(n-1)}$ , where `n` is the number of steps performed. The `steps` parameter sets the maximum number of iterations allowed before the algorithm gives up and reports a run time error that the desired accuracy could not be achieved, preventing an

infinite loop while trying to achieve an accuracy that cannot be reached. After the default 16 steps, the function will have been evaluated 32,769 times. For most functions evaluated to an error of  $1e-6$ , the desired accuracy is reached much, much sooner. Normally, the trapezoid rule stops with a solution when two successive evaluations return results that differ only after the first six significant digits match.

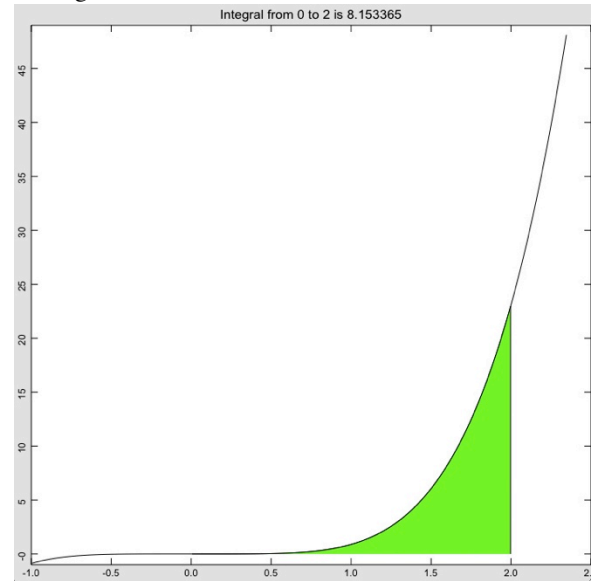
See `trapF` and `romb` for alternatives to `trap` for numeric integration.

#### Snippet

```
! Evaluate the integral of
!   x*x*x*x*log(x + sqr(x*x + 1))
! over the interval 0 to 2.
intfx = math.trap(function f, 0, 2)

DIM p AS Plot, pf AS PlotFunction,
pfi AS PlotFunction
p = Graphics.newPlot
pf = p.newFunction(FUNCTION f)
pfi = p.newFunction(FUNCTION f)
pfi.setDomain(0, 2)
pfi.setFillColor(0, 1, 0)
p.setView(-1, -1, 2.5, 49, 0)
p.setTitle("Integral from 0 to 2 is ")
& STR(intfx))
System.showGraphics

function f (x as double) as double
f = x*x*x*x*log(x + sqr(x*x + 1))
end function
```




---

#### **FUNCTION trapF (FUNCTION f, a AS DOUBLE, b AS DOUBLE, steps AS INTEGER = 10) AS DOUBLE**

`trapF` is an implementation of trapezoidal integration that evaluates an integral for a fixed number of steps. It is useful for some periodic functions that may report the same value as the integration interval is halved, resulting in an early exit from `trap` with an erroneous result.

The first input is the function to integrate. `a` and `b` are the limits of integration. The number of steps is an optional parameter.

The trapezoid rule works by successively dividing the integration interval in half. On the first iteration, it simply evaluates the endpoints. The second iteration adds the center point. The third iteration adds two more points, and so forth. The number of function evaluations required for a specific number of these steps is  $1+2^{(n-1)}$ , where  $n$  is the number of steps performed. The `steps` parameter sets the number of iterations. After the default 10 steps, the function will have been evaluated 513 times.

See `trap` and `romb` for alternatives to `trapF` for numeric integration.

#### Snippet

```
! Evaluate the integral of
!   x*x*x*x*log(x + sqr(x*x + 1))
! over the interval 0 to 2.
print math.trapF(function f, 0, 2)

function f (x as double) as double
f = x*x*x*x*log(x + sqr(x*x + 1))
end function
```

---

#### **FUNCTION unpackDb1 (value() AS INTEGER, index AS INTEGER = 1, bigEndian AS INTEGER = 0) AS DOUBLE**

Converts eight elements from an array into bytes, forming a double-precision floating-point number from the resulting bytes.



Double-precision values are stored internally as an eight-byte IEEE format number. This method will convert an IEEE number from eight individual bytes stored in an array into a double-precision value.

The conversion normally begins with the array element whose subscript is 1. Pass a value for **index** other than 1 to extract the value from a longer array containing multiple values.

By default, the first byte will be the least significant byte, while the last byte will be the most significant byte; this is known as little-endian order. This is the most common byte order for Intel chips and iOS. Some micro-controllers and most network connections reverse the byte order, known as big-endian. `unpackDbl` expects the bytes in big-endian order if the **bigEndian** parameter to a non-zero value.

For example,

```
n# = Math.unpackDbl([$00, $00, $00, $00, $00, $00, $10, $40])
```

sets `n#` to 4.0, as does

```
n# = Math.unpackDbl([$00, $40, $10, $00, $00, $00, $00, $00], 2, 1)
```

---

**FUNCTION unpackInt (value() AS INTEGER, index AS INTEGER = 1, bigEndian AS INTEGER = 0) AS INTEGER**

Converts two elements from an array into bytes, forming an integer from the resulting bytes.

Integers are stored internally as a two-byte two's complement number. This method will convert an integer from two individual bytes stored in an array into an integer value.

The conversion normally begins with the array element whose subscript is 1. Pass a value for **index** other than 1 to extract a two-byte integer from a longer array containing multiple values.

By default, the first byte will be the least significant byte, while the last byte will be the most significant byte; this is known as little-endian order. This is the most common byte order for Intel chips and iOS. Some micro-controllers and most network connections reverse the byte order, known as big-endian. `unpackInt` expects the bytes in big-endian order if the **bigEndian** parameter to a non-zero value.

For example,

```
i% = Math.unpackInt([4, 0])
```

sets `i%` to 4, while

```
i% = Math.unpackInt([%FC, $FF])
```

sets `i%` to -4.

```
bytes = Math.packInt([0, 0, $10], 2, 1)
```

sets `i%` to 16.

---

**FUNCTION unpackLng (value() AS INTEGER, index AS INTEGER = 1, bigEndian AS INTEGER = 0) AS LONG**

Converts four elements from an array into bytes, forming a long integer from the resulting bytes.

Long integers are stored internally as a four-byte two's complement number. This method will convert a long integer from four individual bytes stored in an array into a long integer value.

The conversion normally begins with the array element whose subscript is 1. Pass a value for **index** other than 1 to extract a four-byte integer from a longer array containing multiple values.

By default, the first byte will be the least significant byte, while the last byte will be the most significant byte; this is known as little-endian order. This is the most common byte order for Intel chips and iOS. Some micro-

controllers and most network connections reverse the byte order, known as big-endian. `unpackLng` expects the bytes in big-endian order if the **bigEndian** parameter to a non-zero value.

For example,

```
| L& = Math.unpackLng([4, 0, 0, 0])
```

sets `L&` to 4, while

```
| L& = Math.unpackLng([%FC, $FF, $FF, $FF])
```

sets `L&` to -4.

```
| bytes = Math.packLng([0, 0, 0, 0, $10], 2, 1)
```

sets `L&` to 16.

---

**FUNCTION unpackSng (value() AS INTEGER, index AS INTEGER = 1, bigEndian AS INTEGER = 0)**

Converts four elements from an array into bytes, forming a single-precision floating-point number from the resulting bytes.

Single-precision values are stored internally as a four-byte IEEE format number. This method will convert an IEEE number from four individual bytes stored in an array into a single-precision value.

The conversion normally begins with the array element whose subscript is 1. Pass a value for **index** other than 1 to extract the value from a longer array containing multiple values.

By default, the first byte will be the least significant byte, while the last byte will be the most significant byte; this is known as little-endian order. This is the most common byte order for Intel chips and iOS. Some micro-controllers and most network connections reverse the byte order, known as big-endian. `unpackSng` expects the bytes in big-endian order if the **bigEndian** parameter to a non-zero value.

For example,

```
| n = Math.unpackSng([$00, $00, $80, $40])
```

sets `n` to 4.0, as does

```
| n = Math.unpackSng([$00, $40, $80, $00, $00, $00], 2, 1)
```

---

**FUNCTION var (x AS DOUBLE()) AS DOUBLE**

Returns the sample variance of a list of numbers. The sample variance is defined as

$$s = \frac{1}{N-1} \sum_{i=1}^N (x_i - \bar{x})^2$$

The sample variance is used when the values are draws from a larger population. For the population variance, multiply the result by

$$\frac{N-1}{N}$$

Snippet

```
! Find the variance in the height of a group of people.
PRINT Math.var([76, 75, 68, 71, 69])
```

---

**FUNCTION zero (FUNCTION f, a AS DOUBLE, b AS DOUBLE, error = 1.0e-6) AS DOUBLE**

Find the location where the function crosses the X axis between two points.

The first parameter is the function to evaluate, while the next two are values that, when evaluated, lie on opposite sides of the X axis. This means one must evaluate to a positive value while the other must evaluate to a negative value. The error parameter specifies the allowed error in the result. The result will be the location where the function crosses the axis.

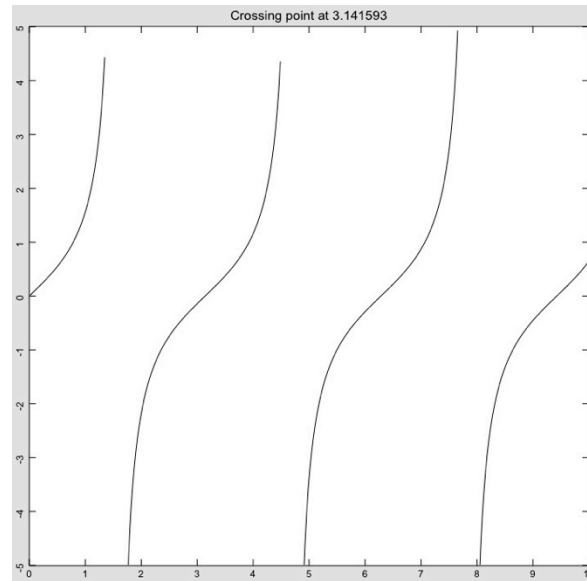
The slope-intercept method is used to find the crossing point. This guarantees finding an answer if one exists. It is always a good idea to plot the function to understand the nature of the solution.

**Snippet**

```
! Find x such that tan(x)=0 for
! a value of x between 2 and 4.
fx = Math.zero(function f, 2, 4)

DIM p AS Plot, pf AS PlotFunction
p = Graphics.newPlot
pf = p.newFunction(FUNCTION f)
p.setTitle("Crossing point at " &
STR(fx))
System.showGraphics

FUNCTION f (x AS DOUBLE) AS DOUBLE
f = TAN(x)
END FUNCTION
```




---

## System

The `System` class gives your programs access to various information about your device and techBASIC, such as version information and the current date. It also includes utilities.

A predefined object by the same name gives you access to the methods in this class. Multiple examples show how to get the most from the data.

---

**SUB clearConsole**

Clears all text from the console view.

**Snippet**

```
! Clear the console
system.clearConsole
```

---

**FUNCTION date AS Date**

Returns a date object set to the current date and time. See the description of the `Date` class for details on accessing the fields, as well as an example.

---

**FUNCTION device AS INTEGER**

Returns a 0 if techBASIC is running on an iPhone or iPod, and 1 if it is running on an iPad.

When running from techBASIC App Builder from the Macintosh, device returns 256 when running under the iPhone simulator, and 257 when running from the iPad simulator.

---

**FUNCTION newEmail AS Email**

Creates and returns a new `Email` object. The `Email` object can be used to create, edit and send email messages using the device's mail client. See the description of the `Email` class for information on creating and sending emails.

---

**FUNCTION newImage AS Image**

Creates and returns a new `Image` object. The `Image` object can load images from disk, take pictures, or grab images from the device's photo library. These images can be interrogated pixel by pixel, used to create stylish buttons, or displayed as an image on the graphics screen.

See the description of the `Image` class for information on creating and using images.

---

**FUNCTION orientation AS INTEGER**

Returns a value that indicates the current display orientation of the device.

<b>orientation</b>	<b>Description</b>
1	The device is in portrait mode, with the device held upright and the home button on the bottom.
2	The device is in landscape mode with the home button on the left.
3	The device is in landscape mode with the home button on the right.
0	The device is in portrait mode, held upside down with the home button on the top.

---

**FUNCTION osVersion AS STRING**

Returns a string with the version of the operating system in use.

The string will start with `iPhone` or `iPad`. iPods will return `iPhone`; from the standpoint of techBASIC, there is no difference between the two. The device name will be followed by the version string returned by the operating system.

For example, during development, the program

```
PRINT system.osVersion
```

printed

```
iPad Version 5.0 (Build 9A5313e)
```

and the iPhone printed

```
iPhone Version 5.0 (Build 9A5313e)
```

The specific build number and operating system will vary as Apple updates iOS.

---

**SUB setAllowedOrientations (value AS INTEGER)**

Sets the allowed orientations for the device when a program is running.

While the call is always allowed, it will have no effect unless the program is in full-screen mode. See `System.showGraphics` for a way to change the program to display the full screen.

The device can be held in four discernable orientations. The orientation of the home button (the round button used to exit a program, returning to the home screen) is used to describe the orientation; it is either down, left, right or up as the screen is viewed. As the device is moved, the operating system checks to see if the orientation of the home button is allowed. If it is, the program is notified of the new orientation by calling the `didRotate` subroutine, allowing the program to change the location and size of the controls and images that appear on the graphics screen to match the new orientation. The display is not changed if the orientation is not allowed.

The value passed as the parameter is a bit mask of up to four bits describing which orientations are allowed. The bits are:

Bit	Description
\$0001	The device is in portrait mode, with the device held upright and the home button on the bottom.
\$0002	The device is in landscape mode with the home button on the left.
\$0004	The device is in landscape mode with the home button on the right.
\$0008	The device is in portrait mode, held upside down with the home button on the top.

At least one orientation must be specified. Passing 0 is equivalent to passing 1.

The allowed values and the orientations the values will turn on are:

value	Down allowed	Left allowed	Right allowed	Up allowed
1	Yes	No	No	No
2	No	Yes	No	No
3	Yes	Yes	No	No
4	No	No	Yes	No
5	Yes	No	Yes	No
6	No	Yes	Yes	No
7	Yes	Yes	Yes	No
8	No	No	No	Yes
9	Yes	No	No	Yes
10	No	Yes	No	Yes
11	Yes	Yes	No	Yes
12	No	No	Yes	Yes
13	Yes	No	Yes	Yes
14	No	Yes	Yes	Yes
15	Yes	Yes	Yes	Yes

The default value is 15 (all orientations allowed) on the iPad, and 7 (all orientations except the home button up) on the iPhone and iPod.

Changing the allowed orientations does not change the orientation if the device is currently held in an orientation that is no longer supported. The device must be rotated to a supported orientation before the change takes effect. For example, making the call

```
System.setAllowedOrientations(8)
```

while the home button is down will do nothing until the device is rotated so the home button is up. Once that is done, the screen will not change as the device is rotated, since no other orientation is allowed. As a result, it is best to set the allowed orientations as soon as the program starts. If the orientation must be restricted later in the program, check to make sure the current orientation is allowed. If not, bring up an alert asking the user to rotate the device to the proper orientation.

See the description of the `didRotate` event for a sample program that uses this method.

---

#### **SUB setKeyboardAppearance (value AS INTEGER)**

Sets the appearance of the keyboard used to enter text in the console. This keyboard is displayed when the INPUT statement is used to read text.

Pass 2 to the method to set the keyboard to an alert-style keyboard, or 1 to use the default keyboard.

---

#### **SUB setKeyboardType (value AS INTEGER)**

Sets the type of the keyboard used to enter text in the console. This keyboard is displayed when the INPUT statement is used to read text. The allowed parameter values are:

Value	Keyboard Style
1	Default
2	ASCII Capable
3	Numbers and Punctuation
4	URL
5	Number Pad
6	Name and Phone Pad
7	Email Address
8	Decimal Pad
9	Twitter
10	Alphabet

---

**SUB setNullEventTime (deltaTime)**

Sets the minimum time between null events.

Null event calls are generated in event driven programs when no other events are being processed. For example, if a program is waiting on a user interaction, techBASIC will constantly be calling the `nullEvent` subroutine, assuming there is one. When a button is tapped, if the program takes 0.5 seconds to process the button press, there will be a 0.5 second gap when null events are not generated. By default, `nullEvent` are generated no more often than 0.05 seconds to allow the iOS user interface and background tasks plenty of time to do other things.

In some programs, it may be advantageous to make this time slower or faster. This call sets the actual delay time. To get null events about 100 times per second, for example, make the call:

```
System.setNullEventTime(0.01)
```

Assuming nothing else is chewing up much CPU time, `nullEvent` will be called slightly less than once every 0.01 seconds.

If `nullEvent` calls are made too frequently, other UI functions and background tasks may become sluggish. Test your program carefully if you set the delay to a very low value. This will vary among iPhone and iPad models, so be particularly careful of lowering the delay time if your program will be released on the App Store, where it may run on machines you have not tested.

---

**SUB showConsole**

Switches the view to the console.

**Snippet**

```
! Switch to the console to show the
results.
PRINT matrix
system.showConsole
```

---

**SUB showGraphics (fullScreen AS INTEGER = 0,  
hideStatus AS INTEGER = 0)**

Switches the view to the graphics display.

If the **fullScreen** parameter is set to 1, the graphics display will fill the entire screen, hiding the normal techBASIC controls. The tools control will still be visible unless it is hidden separately using `Graphics.setToolsHidden(1)`. While in full screen mode, the tools control provides access to the normal debugger controls for stepping through a program or stopping the program. There is also a button available to switch to debug mode, which will exit full screen



mode and display the source view. Switching back to the graphics view restores full screen mode.

If both the **fullScreen** parameter and the **hideStatus** parameter are set to 1, the iOS status bar showing battery level and connection status is also hidden.

The snippet shows a program that draws a plot, showing it in full screen mode. The image is a screen capture of the running program in an iPhone after tapping the tools control to access the various debugging controls.

#### Snippet

```
System.showGraphics(1)

DIM p AS Plot
p = graphics.newPlot
p.setBorderColor(1, 1, 1)
p.setSurfaceStyle(3)

DIM func AS PlotFunction
func = p.newSpherical(FUNCTION f)

p.setTranslation3D(0, 0, -5)
p.setTranslation(0, 1.5)

DIM quit AS Button
quit = Graphics.newButton(10, 10)
quit.setTitle("Quit")

FUNCTION f(theta, phi)
f = 5
END FUNCTION

SUB touchUpInside (ctrl AS Button, time AS DOUBLE)
STOP
END SUB
```

---

#### **SUB showSource**

Switches the view to the source view. This is useful during development; just before a program exits, use this command to switch back to the source view to immediately edit the program.

#### Snippet

```
! Switch to the source view.
PRINT "In progress"
system.showSource
```

---

#### **FUNCTION ticks AS DOUBLE**

Returns the elapsed time in seconds since January 1, 2001 at 00:00:00 GMT. This can be used to time sequences, as in

```
DIM time AS DOUBLE
time = system.ticks
longCalculation
PRINT "The calculation took "; system.ticks - time; " seconds."
```

The precision varies by device, but testing seems to indicate millisecond times are reliable. The time can jump backward, though, due to user changes to the clock or to the device synchronizing the system clock to an external data source.

The time is too large to see small changes when stored in a **FLOAT** variable, so be sure to use a **DOUBLE** variable to store the value, and double-precision math to perform operations.

---

### **FUNCTION version**

Returns the version of techBASIC as a floating point number. The major version is to the left of the decimal point, the minor version in the next two digits, and the bug fix version in the last two digits. For example, if you were using version 1.2.3 of techBASIC, the value returned would be 1.0203.

#### Snippet

```
! Print the techBASIC version.  
PRINT "Version "; system.version; " of techBASIC."
```

---

### **SUB vibrate**

On devices that support vibration, the device will vibrate for 0.4 seconds.

Vibrate on Ring must be enabled in the Settings App, Sounds panel.

#### Snippet

```
! Vibrate the device.  
System.vibrate  
system.wait(0.5)
```

---

### **SUB wait (seconds)**

Causes the program to wait for the specified number of seconds.

If you enter a number that is a bit longer than you are really wanting to wait, keep in mind that the Stop button can be used to halt the program.

#### Snippet

```
! Pause for half a second before continuing.  
system.wait(0.5)
```



## Chapter 15 – Sensor and Communication Classes

Sensor classes are used to return data from the internal sensors built into the iPhone or iPad, as well as external sensors such as HiJack. HiJack is also a communication class, supporting input from the HiJack device. Several classes are used to support Bluetooth Low Energy (BLE), also known as Bluetooth 4.0. These classes begin with BLE.

The camera can also be considered a sensor, but it is covered separately in the Image class, described in Chapter 14.

---

### Audio

The Audio class is used for sound input and output via the internal microphone jack, speaker and microphone. The device itself automatically switches between the headphone jack and the speaker and microphone as a plug is inserted or removed from the headphone jack.

---

#### **SUB detectPulses (SUB callback, trigger, reset, rate AS LONG = 44100)**

This method watches the sound input source for pulses of sound, calling a subroutine you designate with the system time when the pulse was detected. This call is designed to work with TTL devices that pulse the headphone port, reporting each pulse generated by the TTL device.

Pulses are separated from the surrounding sound using the trigger and reset values. Assuming the trigger is positive, a pulse is generated when the sound volume exceeds the trigger value. Since there may be noise in the input that could rapidly bounce up and down near the trigger value, generating multiple pulses when only one is desired, a reset value is also used. The pulse does not end until the sound volume drops below the reset value. The pulse must rise above the trigger value again to start a new pulse.

If the trigger value is negative, the pulse detector works essentially the same way, but the volume must fall below the trigger to start a pulse, and rise above the reset value to end a pulse.

The callback subroutine must have a single numeric parameter, which should be a double value. It is called with the system time when the trigger value was exceeded at the start of the pulse.

While the system will buffer pulses, you need to handle the pulses fast and return, or pulses may be lost. The number of pulses per second you can process depends on how much processing you do in the callback method, but is rarely more than 100 pulses per second.

The optional **rate** parameter sets the audio sample rate. It defaults to 44,100 Hz, but keep in mind that this is a request to the operating system. Some sound input devices may not support the rate you request. You can find the actual rate using the `sampleRate` method.

The pulse detection method will stop when the program stops. If your program does not have a graphical user interface, add an empty `nullEvent` subroutine to keep the program running.

The audio input system is designed with the idea that only one of `detectPulses`, `pulseHistogram` or `startSoundInput` will be used at a time, but it is actually possible to use them in combination, with some limitations. The first method called will set the sample rate for all three calls. The last method called sets the trigger and reset values. Stopping sound input with the `stopSoundInput` method stops all of the sound input routines.

The snippet shows a very simple pulse detector that watches the sound port for pulses. It prints the time of any pulse found to the console. It runs until you manually stop the program.

#### Snippet

```
Audio.detectPulses(SUB pulse, 0.75, 0.25)

SUB pulse (time AS DOUBLE)
  PRINT time
END SUB
```

```
SUB nullEvent (time AS DOUBLE)
END SUB
```

---

**SUB pulseHistogram (SUB callback, trigger, reset, width, rate AS LONG = 44100)**

This method watches the sound input source for pulses of sound, binning the pulses based on a supplied time. When the time for a particular bin expires, it calls your subroutine to report when the bin started and how many pulses were detected during that time slice. This call is designed to work with TTL devices that pulse the headphone port, reporting each pulse generated by the TTL device.

Pulses are separated from the surrounding sound using the trigger and reset values. Assuming the trigger is positive, a pulse is generated when the sound volume exceeds the trigger value. Since there may be noise in the input that could rapidly bounce up and down near the trigger value, generating multiple pulses when only one is desired, a reset value is also used. The pulse does not end until the sound volume drops below the reset value. The pulse must rise above the trigger value again to start a new pulse.

If the trigger value is negative, the pulse detector works essentially the same way, but the volume must fall below the trigger to start a pulse, and rise above the reset value to end a pulse.

**width** is the width of each bin, in seconds. Pulses are counted for the time indicated, and the final pulse count passed to your subroutine. Very small width values may make it difficult to process the data and return quickly enough to avoid data loss. Bin widths on the order of 0.1 seconds or longer are quite easy to handle.

The callback subroutine must have two numeric parameters. The first is the time at the start of the bin; you should use a `DOUBLE` value for this parameter. The second parameter is the number of pulses counted in the bin. While an `INTEGER` value is probably sufficient, a `LONG` is recommended for this parameter.

The optional **rate** parameter sets the audio sample rate. It defaults to 44,100 Hz, but keep in mind that this is a request to the operating system. Some sound input devices may not support the rate you request. You can find the actual rate using the `sampleRate` method.

The pulse detection method will stop when the program stops. If your program does not have a graphical user interface, add an empty `nullEvent` subroutine to keep the program running.

The audio input system is designed with the idea that only one of `detectPulses`, `pulseHistogram` or `startSoundInput` will be used at a time, but it is actually possible to use them in combination, with some limitations. The first method called will set the sample rate for all three calls. The last method called sets the trigger and reset values. Stopping sound input with the `stopSoundInput` method stops all of the sound input routines.

See the Geiger Counter sample for an example of this method in use. This sample is designed for use with the Images SI Inc. Analog Geiger Counter, converting the instrument into a digital Geiger counter. The program should work equally well to count pulses from any TTL device, though.

---

**FUNCTION sampleRate AS LONG**

Returns the actual sampling rate for `detectPulses`, `pulseHistogram` or `startSoundInput`.

The sample rate passed by these three methods is a request. `sampleRate` returns the sample rate the audio system in use was actually able to accommodate.

Stop the sound input using `stopSoundInput` and restart it again to change the sample rate.

---

**SUB startSoundInput (SUB callback, rate AS LONG = 44100)**

This method collects sound data from the sound input device, passing it to your subroutine as an array of volume values.

The optional **rate** parameter sets the audio sample rate. It defaults to 44,100 Hz, but keep in mind that this is a request to the operating system. Some sound input devices may not support the rate you request. You can find the actual rate using the `sampleRate` method.

The callback subroutine must have two parameters. The first is the time at the start of the sound; you should use a `DOUBLE` value for this parameter. The second parameter is an array of `SINGLE` values. Each value returned is the sound volume at a particular sound sample. Each sound sample is  $1/\text{sampleRate}$  seconds after the previous sound sample.

The sound will continue to build up as your program runs. Attempts to do large amounts of sound processing while sound data builds up will usually result in an out of memory error. The best strategy for using this method is to turn sound collection on for a short period of time, collecting the sound data in a local buffer, then turn it off while you process the data. This is the method used in the Oscilloscope sample, which plots input data from the sound port like a classic oscilloscope.

The sound input method will stop when the program stops. If your program does not have a graphical user interface, add an empty `nullEvent` subroutine to keep the program running.

The audio input system is designed with the idea that only one of `detectPulses`, `pulseHistogram` or `startSoundInput` will be used at a time, but it is actually possible to use them in combination, with some limitations. The first method called will set the sample rate for all three calls. The last method called sets the trigger and reset values. Stopping sound input with the `stopSoundInput` method stops all of the sound input routines.

---

#### **SUB stopSoundInput**

Stops sound input for `detectPulses`, `pulseHistogram` and `startSoundInput`. Sound can be subsequently restarted with another call to any of these methods.

---

## **BLE**

The BLE class is used to start and stop Bluetooth LE services, as well as to scan for peripherals and do other housekeeping functions.

Bluetooth LE devices follow the Bluetooth 4.0 standard. They are typically used for small, battery driven devices that send relatively small amounts of information, or send it infrequently. Examples are medical devices for monitoring heart rates, sports devices for tracking steps, or instrumentation that needs to supply data wirelessly. iOS devices beginning with the iPhone 4s and iPad 3<sup>rd</sup> generation have support for Bluetooth LE. While older iOS devices support Bluetooth, they do not work with Bluetooth LE, which uses a physically different system for communicating with devices.

Bluetooth LE works in two different modes, called slave and central. The central device would be called the client in a typical client-server relationship; it is the consumer of data. The slave device corresponds to the server, sending information to the central device. iOS 5 supports BLE central communications; iOS 6 supports both central and slave modes. The mapping of central and slave devices to client and server devices is not exact, but it is a useful way to get a basic understanding of the role of the two devices.

---

### **BLE Central Communications**

Getting and sending data as a central device to Bluetooth LE slave devices begins with a scan to see what devices are available. This is done with the `startScan` method. Like almost all Bluetooth LE calls, this is done asynchronously, since it can take time and since there is no guarantee that one, and only one, response will be received. A program implements the `BLEDiscoveredPeripheral` subroutine, which is called as peripherals are found. As with all subroutines called in response to Bluetooth LE actions, `BLEDiscoveredPeripheral` is documented in Chapter 13 in the section titled *Handling Events*.

Once a suitable device is found, the program can connect to the device with a call to `connect`. Once again, the connection may take time, so the program monitors the connection status using a subroutine called `BLEPeripheralInfo`.

Bluetooth LE devices are organized around services. Services are groups of functional information, like the battery service, which reports on the power level for the device's battery. The services that are available vary from device to device. In general, the program knows what services it is interested in—say, temperature—and scans for nearby devices that provide a temperature service. The services are distinguished by a UUID. Some of these are standardized across all Bluetooth LE devices, like the battery power level, which has a UUID of 180F. These standardized values are 16 bit values. For a current list, see <http://developer.bluetooth.org/gatt/services/Pages/ServicesHome.aspx>. Individual device manufacturers assign others, which are typically 128 bits long. Once a device is found, use the `BLEPeripheral`

`discoverServices` call to get a list of the available services, then monitor the results returned by the device with a `BLEPeripheralInfo` subroutine.

Services may contain other imbedded services or characteristics. Characteristics bundle individual pieces of information that can be read or written. For example, the battery service has a required characteristic with a UUID of 2A19 that returns the current battery level as a one-byte integer value ranging from 0 to 100. Use the `BLEPeripheral` `discoverCharacteristics` call to get a list of the available characteristics, then monitor the results returned by the device with a `BLEServiceInfo` subroutine.

Once a characteristic has been passed to the `BLEServiceInfo` subroutine, use the `BLEPeripheral` `readCharacteristic` call to pull data from the device, like the battery level, or the `BLEPeripheral` `setNotify` call to tell the characteristic to notify the program when new data is available, like a new heart beat rate on a heart monitor. Either way, when data is returned from the device, the program is informed the data is ready by a call to the `BLECharacteristicInfo` subroutine. The program can then read the data using the `BLECharacteristic` `value` call. To write data, use the `BLEPeripheral` `writeCharacteristic` call.

Characteristics may also include descriptors, which, like the services, can be used to read and write data. They function very much like characteristics, with their own set of read and write calls.

This overview gives a global look at the way Bluetooth LE devices are accessed, but there are many more calls and details that may prove useful with specific devices. For more information about Bluetooth LE, visit <http://developer.bluetooth.org>. For samples of Bluetooth LE access from techBASIC, look at the various code snippets that appear with the command descriptions, especially in the `BLEPeripheral` class. There is also a tutorial introduction to Bluetooth LE based on the TI LBE CC2540 Mini Development Kit on the Byte Works web site. Look in the Blogs section for this and other complete working programs.

---

## BLE Slave Communications

A slave device begins by creating a map of one or more services, created using `BLEMutableService` and `BLEMutableCharacteristic` objects. These are advertised using the `startAdvertising` method of the `BLE` class. Once a connection is established, the device can serve information on request through the `setValue` call or, if the central device asks for information using `setNotify`, by sending the information as available. Either way, the information is sent to the central device using the `updateValue` method of the `BLE` class.

---

### FUNCTION BLEAvailable () AS INTEGER

Checks to see if Bluetooth LE is available on the current device. One of the following values is returned:

Value	Meaning
2	The device does not support Bluetooth LE. iPhones support Bluetooth LE beginning with the iPhone 4s. iPads support Bluetooth LE beginning with the third generation iPad.
3	The application is not authorized to use Bluetooth LE.
4	Bluetooth LE is powered off. Use the Settings app, General tab, Bluetooth tab to enable Bluetooth.
5	Bluetooth LE is ready for use.

#### Snippet

```
SELECT CASE BLE.BLEAvailable
CASE 2: PRINT "BLE is not supported."
CASE 3: PRINT "Not authorized for BLE."
CASE 4: PRINT "BLE is powered off."
CASE 5: PRINT "BLE is ready for use."
END SELECT
```

---

### SUB connect (peripheral AS BLEPeripheral)

Attempts to establish a connection with the given Bluetooth LE device.

The program's `BLEPeripheralInfo` subroutine is called once the attempt to connect completes. The `BLEPeripheralInfo` subroutine is the normal place to begin working with a device, since the connection must

be established before most communication with the device can occur. If the connection completed normally, `BLEPeripheralInfo` is called with a kind of 1 and an error code of 0. If there was an error connecting to the device, `BLEPeripheralInfo` is called with a kind of 2 and an error code indicating the problem.

The snippet shows a program that attempts to connect with the first nearby peripheral it finds, then displays the connection status.

#### Snippet

```
DIM somePeripheral AS BLEPeripheral
discoveredPeripheral = 0
BLE.startBLE
DIM uuid(0) AS STRING
BLE.startScan(uuid)

SUB BLEDiscoveredPeripheral(time AS DOUBLE, peripheral AS BLEPeripheral,
services() AS STRING, advertisements(,) AS STRING, rssi AS SINGLE)
IF NOT discoveredPeripheral THEN
    discoveredPeripheral = 1
    somePeripheral = peripheral
    BLE.connect(somePeripheral)
    BLE.stopScan
END IF
END SUB

SUB BLEPeripheralInfo (time AS DOUBLE, peripheral AS BLEPeripheral, kind
AS INTEGER, msg AS STRING, err AS LONG)
SELECT CASE kind
CASE 1
    PRINT "Connected to "; peripheral.bleName
    BLE.disconnect(peripheral)

CASE 2
    PRINT "Failed to connect to "; peripheral.bleName; ": "; err

CASE 3
    PRINT "Lost connection to "; peripheral.bleName; ": "; err
END SELECT
END SUB
```

---

#### **SUB disconnect (peripheral AS BLEPeripheral)**

Disconnects from a peripheral.

See connect for a snippet that shows how to disconnect from a peripheral.

---

#### **FUNCTION newBLEPeripheralManager AS BLEPeripheralManager**

Use this method to create a peripheral manager that can be used to create services and characteristics for a Bluetooth LE slave device, and then advertise these characteristics so other devices can subscribe to the services.

See the `BLEPeripheralManager` class' `addService` call for an example of typical workflow using this class.

---

#### **SUB retrieveConnectedPeripherals**

Requests a list of all of the peripherals connected to this device. Once collected, the list is returned via a call to the `BLERetrievedPeripherals` call.

This call has been deprecated in iOS. While the call still exists for backward compatibility, it no longer does anything. The snippet shows a program that returns all connected peripherals.

---

**SUB retrievePeripherals (uuid() AS STRING)**

Requests a list of all of the peripherals known to this device, whether they are currently connected or not. Once collected, the list is returned via a call to the `BLERetrievedPeripherals` call.

This call has been deprecated in iOS. While the call still exists for backward compatibility, it no longer does anything.

---

**SUB startBLE**

Starts Bluetooth LE. Use this call or `BLEAvailable` before using any of the other Bluetooth LE calls.

See `connect`, among other calls, for a snippet showing the `startBLE` call.

---

**SUB startScan (uuid() AS STRING)**

Begins a scan for Bluetooth LE devices. Calls are made to `BLEDiscoveredPeripheral` as Bluetooth LE devices are found.

The `uuid` parameter is an array of strings containing the desired services. Only peripherals that advertise one of the services are sent to `BLEDiscoveredPeripheral` for further processing. Pass an empty array to get all available devices.

See `connect` for a snippet that uses this call.

---

**SUB stopBLE**

Stops Bluetooth LE. Use this call if Bluetooth LE services are no longer needed. This will reduce battery drain.

This call is only needed if the program will continue to run, since `techBASIC` will stop Bluetooth LE services as soon as the program ends. For example, a program that occasionally collects weather information, but remains running to show historical data, should make this call after collecting the data to reduce battery drain while historical data is viewed.

---

**SUB stopScan**

Stops scanning for Bluetooth LE devices. Use this call once the Bluetooth LE devices are found to reduce battery drain.

See `connect` for a snippet that uses this call.

---

## BLEATTRequest

`BLEATTRequest` is a class used to communicate with Bluetooth LE devices. See the description of the `BLE` class for an overview of the classes used to access Bluetooth LE devices.

This class is used when implementing a Bluetooth LE slave device. When the master device requests a read or write, it notifies the slave program through a call to `BLEPeripheralManagerInfo` with a kind flag of 4 or 5. See the `BLEPeripheralManagerInfo` event for details.

---

**FUNCTION characteristic AS BLECharacteristic**

This is the characteristic to read or write.

The `BLEATTRequest` class is used when a master device requests information from a slave device. See the `BLEPeripheralManagerInfo` event for details.

---

**FUNCTION offset AS INTEGER**

This is the characteristic to read or write.

The `BLEATTRequest` class is used when a master device requests information from a slave device. See the `BLEPeripheralManagerInfo` event for details.

This method returns the offset for a read or write operation. It allows the read or write request to start at a byte other than the first byte of the data.

**SUB setValue (value() AS INTEGER)**

Set the value for a read operation.

The `BLEATTRequest` class is used when a master device requests information from a slave device. A slave device use this method to set the data to return for a read request from a master device. The method is not used for write requests. See the `BLEPeripheralManagerInfo` event for details.

**FUNCTION value () () AS INTEGER**

Get the value for a write operation.

The `BLEATTRequest` class is used when a master device requests information from a slave device. A slave device uses this call to retrieve any information send by the master device for a write operation. This call is not used for read operations. See the `BLEPeripheralManagerInfo` event for details.

---

## BLECharacteristic

`BLECharacteristic` is a class used to communicate with Bluetooth LE devices. See the description of the `BLE` class for an overview of the classes used to access Bluetooth LE devices.

**FUNCTION descriptors () () AS BLEDescriptor**

Characteristics may contain descriptors, which provide additional information. This call returns the known descriptors for this characteristic.

Descriptors are not immediately available. Use the `BLEPeripheral` class' `discoverDescriptors` call to ask the device for a list of the descriptors. Once the descriptors are collected, the program is notified by a call to the `BLECharacteristicInfo` subroutine. After this call, this method will return the descriptors for this characteristic.

See the `discoverDescriptors` call in the `BLEPeripheral` class for a snippet that uses the `descriptors` method.

**FUNCTION isBroadcasted AS INTEGER**

Returns zero if the characteristic is broadcasted, and a non-zero value if not.

See Chapter 13, Handling Events, the `BLEServiceInfo` subroutine for a snippet that uses the `isBroadcasted` method.

**FUNCTION isNotifying AS INTEGER**

Returns zero if the characteristic is notifying, and a non-zero value if not.

Characteristics can supply values by push or pull. For characteristics that use push notification, the characteristic notifies the program a new value is available through a call to `BLECharacteristicInfo`. These notifications are only sent if the characteristic is directed to send updates using the `BLEPeripheral` class' `setNotify` method. The `isNotifying` method is used to detect whether the device is currently sending push notifications.

See Chapter 13, Handling Events, the `BLEServiceInfo` subroutine for a snippet that uses the `isNotifying` method.

**FUNCTION properties AS INTEGER**

Returns a bit mapped value indicating the properties for this characteristic. The bits used are:

Bit	Use
\$0001	The characteristic can be broadcasted.
\$0002	The characteristic can be read.
\$0004	The characteristic can be written to without the need to send back a response.
\$0008	The characteristic can be written.

- |        |  |
|--------|--|
| \$0010 | The characteristic can provide notifications when a new value is available. This is also known as push notification. |
| \$0020 | The characteristic permits indications of the characteristic value with acknowledgement.                             |
| \$0040 | The characteristic supports authenticated writes.  |
| \$0080 | Indicates that an extended property descriptor exists.   |

The various values from the table are combined using a logical OR to indicate the available properties. For example, a characteristic that can be read or written returns the value \$000A, or decimal 10.

See section 3.3.1.1 of *Specification of the Bluetooth System*, 30 June 2010 for details regarding these properties.

---

**FUNCTION uuid AS STRING**

Returns the UUID for this characteristic.

Each characteristic has a UUID that identifies that specific characteristic. This call returns the UUID for the current Bluetooth LE characteristic.

See the `discoverCharacteristics` call in the `BLEPeripheral` class for a snippet that uses the `uuid` method.

---

**FUNCTION value () () AS INTEGER**

Returns the last known value for the characteristic.

Characteristics are frequently used to supply values, whether that is by push or pull. The program is notified when a new value is available through a call to `BLECharacteristicInfo`. Once this notification is received, use the `value` method to read the value of the characteristic.

Values can be practically anything, from a single byte integer to a string to an encoded image. The value is returned as an array of integers, with each integer in the array holding one byte of the value. It is up to the program to know what the value means, and to convert the value to a more familiar form like a string if needed.

See the `discoverCharacteristics` call in the `BLEPeripheral` class for a snippet that uses the `value` method.

---

## BLEDescriptor

`BLEDescriptor` is a class used to communicate with Bluetooth LE devices. See the description of the `BLE` class for an overview of the classes used to access Bluetooth LE devices.

---

**FUNCTION uuid AS STRING**

Returns the UUID for this descriptor.

Each descriptor has a UUID that identifies that specific descriptor. This call returns the UUID for the current Bluetooth LE descriptor.

See the `discoverDescriptors` call in the `BLEPeripheral` class for a snippet that uses the `uuid` method.

---

**FUNCTION value () () AS INTEGER**

Returns the last known value for the descriptor.

The value for a descriptor is not automatically available. Use the `BLEPeripheral` class' `readDescriptor` call to ask the device for the current value. Once the descriptor is read, the program is notified by a call to the `BLEDescriptorInfo` subroutine. After this call, this method will return the value for this descriptor.

Descriptors can be practically anything, from a single byte integer to a string. The value is returned as an array of integers, with each integer in the array holding one byte of the value. It is up to the program to know what the value means, and to convert the value to a more familiar form like a string if needed.

See the `discoverDescriptors` call in the `BLEPeripheral` class for a snippet that uses the `value` method.



## BLEMutableCharacteristic

BLEMutableCharacteristic is a class used to set up characteristics for Bluetooth LE slave devices. See the description of the BLE class for an overview of the classes used to access Bluetooth LE devices.

---

### SUB addPresentationFormat (description() AS INTEGER)

Creates a new user description descriptor for this characteristic.

The user description is an array of bytes forming a presentation format that can be read by BLE controller devices after BLEPeripheral.discoverDescriptors call. The UUID for the user description of the characteristic is \$2904. See the following link for a current description of this format.

[http://developer.bluetooth.org/gatt/descriptors/Pages/DescriptorViewer.aspx?u=org.bluetooth.descriptor.gatt.characteristic\\_presentation\\_format.xml](http://developer.bluetooth.org/gatt/descriptors/Pages/DescriptorViewer.aspx?u=org.bluetooth.descriptor.gatt.characteristic_presentation_format.xml).

This is one of two descriptors currently supported by iOS. For the other, see addUserDescription.

---

### SUB addUserDescription (description AS STRING)

Creates a new user description descriptor for this characteristic.

The user description is a string that can be read by BLE controller devices after BLEPeripheral.discoverDescriptors call. The UUID for the user description of the characteristic is \$2901.

This is one of two descriptors currently supported by iOS. For the other, see addPresentationFormat.

---

### FUNCTION isBroadcasted AS INTEGER

Returns zero if the characteristic is broadcasted, and a non-zero value if not.

See Chapter 13, Handling Events, the BLEServiceInfo subroutine for a snippet that uses the isBroadcasted method.

---

### FUNCTION isNotifying AS INTEGER

Returns zero if the characteristic is notifying, and a non-zero value if not.

Characteristics can supply values by push or pull. For characteristics that use push notification, the characteristic notifies the program a new value is available through a call to BLECharacteristicInfo. These notifications are only sent if the characteristic is directed to send updates using the BLEPeripheral class' setNotify method. The isNotifying method is used to detect whether the device is currently sending push notifications.

See Chapter 13, Handling Events, the BLEServiceInfo subroutine for a snippet that uses the isNotifying method.

---

### FUNCTION permissions AS INTEGER

Returns a bit mapped value indicating the permissions for this characteristic. The bits used are:

Bit	Use
\$0001	The characteristic's value can be read.
\$0002	The characteristic can be written.
\$0004	The characteristic can only be read by a trusted device. This means encryption is required.
\$0008	The characteristic can only be written by a trusted device. This means encryption is required.

The various values from the table are combined using a logical OR to indicate the available properties. For example, a characteristic that can be read or written returns the value \$0003, or decimal 3.

See setPermissions for a discussion of these bits.

**FUNCTION properties AS INTEGER**

Returns a bit mapped value indicating the properties for this characteristic. The bits used are:

Bit	Use
\$0001	The characteristic can be broadcasted.
\$0002	The characteristic can be read.
\$0004	The characteristic can be written to without the need to send back a response.
\$0008	The characteristic can be written.
\$0010	The characteristic can provide notifications when a new value is available. This is also known as push notification.
\$0020	The characteristic permits indications of the characteristic value with acknowledgement.
\$0040	The characteristic supports authenticated writes.
\$0080	Indicates that an extended property descriptor exists.

The various values from the table are combined using a logical OR to indicate the available properties. For example, a characteristic that can be read or written returns the value \$000A, or decimal 10.

See section 3.3.1.1 of *Specification of the Bluetooth System*, 30 June 2010 for details regarding these properties.

**SUB setPermissions (permissions AS INTEGER)**

Sets the permissions for this characteristic using a bitmapped value. The bits used are:

Bit	Use
\$0001	The characteristic's value can be read.
\$0002	The characteristic can be written.
\$0004	The characteristic can only be read by a trusted device. This means encryption is required.
\$0008	The characteristic can only be written by a trusted device. This means encryption is required.

The various values from the table are combined using a logical OR to indicate the available properties. For example, to set the permissions to read and write, pass the value \$0003, or decimal 3.

Setting the permissions is distinct from setting the properties. Setting the properties sets flags that tell the client (central device) which services are available. Setting the permissions actually enables or disables those abilities, and specifies if they are restricted to encrypted links.

**SUB setProperties (properties AS INTEGER)**

Sets the properties for this characteristic using a bitmapped value. The bits used are:

Bit	Use
\$0001	The characteristic can be broadcasted.
\$0002	The characteristic can be read.
\$0004	The characteristic can be written to without the need to send back a response.
\$0008	The characteristic can be written.
\$0010	The characteristic can provide notifications when a new value is available. This is also known as push notification.
\$0020	The characteristic permits indications of the characteristic value with acknowledgement.
\$0040	The characteristic supports authenticated writes.
\$0080	Indicates that an extended property descriptor exists.

The various values from the table are combined using a logical OR to indicate the available properties. For example, to set the properties to read and write, pass the value \$000A, or decimal 10.

See also setPermissions.

---

**SUB setValue (value() AS INTEGER)**

Sets the value of the characteristic. The value can then be read by a Bluetooth LE central device.

---

**FUNCTION uuid AS STRING**

Returns the UUID for this characteristic.

Each characteristic has a UUID that identifies that specific characteristic, set when the characteristic is created. This call returns the UUID for the current Bluetooth LE characteristic.

---

**FUNCTION value () () AS INTEGER**

Returns the last known value for the characteristic.

Characteristics are frequently used to supply values, whether that is by push or pull. The program is notified when a new value is available through a call to `BLECharacteristicInfo`. Once this notification is received, use the `value` method to read the value of the characteristic.

Values can be practically anything, from a single byte integer to a string to an encoded image. The value is returned as an array of integers, with each integer in the array holding one byte of the value. It is up to the program to know what the value means, and to convert the value to a more familiar form like a string if needed.

---

## BLEMutableService

`BLEMutableService` is a class used to set up services for Bluetooth LE slave devices. See the description of the BLE class for an overview of the classes used to access Bluetooth LE devices.

---

**FUNCTION characteristics () () AS BLECharacteristics**

Returns the known characteristics for this service.

---

**FUNCTION includedServices () () AS BLEService**

Services may contain other services, which are called included services. This call returns the known included services for this service.

---

**FUNCTION isPrimary AS INTEGER**

Returns a nonzero value if this is a primary service, and a zero value if it is a secondary service. The value is set when the service is created.

---

**FUNCTION newCharacteristic (UUID AS String, properties AS INTEGER, permissions AS INTEGER) AS BLEMutableService**

Creates and returns a new mutable characteristic attaches to this service.

**UUID** is the UUID for the new characteristic.

**properties** is a bit map of the properties of the new characteristic. The various supported properties are:

---

Bit	Use
\$0001	The characteristic can be broadcasted.
\$0002	The characteristic can be read.
\$0004	The characteristic can be written to without the need to send back a response.
\$0008	The characteristic can be written.
\$0010	The characteristic can provide notifications when a new value is available. This is also known as push notification.
\$0020	The characteristic permits indications of the characteristic value with acknowledgement.
\$0040	The characteristic supports authenticated writes.
\$0080	Indicates that an extended property descriptor exists.

---

**permissions** is a bit map of the permissions for the characteristic. The permissions tell the characteristic which properties it should support; the property tells the central device that subscribes to the service what properties are supported. In general, these will be the same. The supported values are:

Bit	Use
\$0001	The characteristic can be broadcasted.
\$0002	The characteristic can be read.

See the `addService` call in the BLE class for a snippet that uses the `uuid` method.

---

**FUNCTION newConstantCharacteristic (UUID AS String, properties AS INTEGER, permissions AS INTEGER, value() AS INTEGER) AS BLEMutableService**

Creates and returns a new mutable characteristic attaches to this service. This is generally the same as the `newCharacteristic` call, but this call sets up a characteristic with a constant, predefined value.

**UUID** is the UUID for the new characteristic.

**properties** is a bit map of the properties of the new characteristic. The various supported properties are:

Bit	Use
\$0001	The characteristic can be broadcasted.
\$0002	The characteristic can be read.
\$0004	The characteristic can be written to without the need to send back a response.
\$0008	The characteristic can be written.
\$0010	The characteristic can provide notifications when a new value is available. This is also known as push notification.
\$0020	The characteristic permits indications of the characteristic value with acknowledgement.
\$0040	The characteristic supports authenticated writes.
\$0080	Indicates that an extended property descriptor exists.

**permissions** is a bit map of the permissions for the characteristic. The permissions tell the characteristic which properties it should support; the property tells the central device that subscribes to the service what properties are supported. In general, these will be the same. The supported values are:

Bit	Use
\$0001	The characteristic can be broadcasted.
\$0002	The characteristic can be read.

**value** is the value of the characteristic. The value is up to 20 bytes stored in an integer array, one byte per array element.

---

**FUNCTION uuid AS STRING**

Returns the UUID for this service.

Each service has a UUID that identifies that specific service, set when the service is created. This call returns the UUID for the current Bluetooth LE service.

---

## BLEPeripheral

BLEPeripheral is a class used to communicate with Bluetooth LE devices. See the description of the BLE class for an overview of the classes used to access Bluetooth LE devices.

---

**FUNCTION bleName AS STRING**

Returns the name of the Bluetooth LE device.

The snippet shows a program that will connect to the first Bluetooth LE device found, then print the name of the device. If the device is the key fob from the TI CC2540 Mini Development kit, the program will print Keyfobdemo.

#### Snippet

```

BLE.startBLE
DIM uuid(0) AS STRING
BLE.startScan(uuid)

SUB BLEDiscoveredPeripheral(time AS DOUBLE, peripheral AS BLEPeripheral,
services() AS STRING, advertisements(,) AS STRING, rssi AS SINGLE)
PRINT peripheral.bleName
BLE.stopScan
END SUB

```

---

#### **SUB discoverCharacteristics (uuid() AS STRING, service AS BLEService)**

Discover one or more services associated with a characteristic or this device.

Bluetooth LE devices have zero or more services, each of which can have other imbedded services. These services have characteristics. An example of a service is a battery level with a standard UUID of 180F. A characteristic of this service is the current power level, with a UUID of 2A19. This call is made once it is known that a specific service exists on the device. It asks for one or more characteristics associated with that service. If any are found, a call is made to BLEServiceInfo.

**uuid** is an array of the service UUIDs desired. Pass an empty array to discover all services for the characteristic.

See discoverDescriptors for a snippet that uses this call.

---

#### **SUB discoverDescriptors (characteristic AS BLECharacteristic)**

Discover any descriptors associated with a characteristic or this device.

Bluetooth LE devices have zero or more services, each of which can have other imbedded services. These services have characteristics. An example of a service is a battery level with a standard UUID of 180F. A characteristic of this service is the current power level, with a UUID of 2A19. Characteristics may have descriptors that provide additional information about the characteristic. This call asks the device for any available descriptors. If any are found, a call is made to BLECharacteristicsDiscovery, where the characteristic can be asked for any descriptors using the characteristic's descriptors method.

The snippet shows a program that finds the first Bluetooth LE device available and prints and descriptors found on that device.

#### Snippet

```

DIM somePeripheral AS BLEPeripheral
BLE.startBLE
DIM uuid(0) AS STRING
BLE.startScan(uuid)

SUB BLEDiscoveredPeripheral(time AS DOUBLE, peripheral AS BLEPeripheral,
services() AS STRING, advertisements(,) AS STRING, rssi AS SINGLE)
somePeripheral = peripheral
BLE.connect(somePeripheral)
BLE.stopScan
END SUB

SUB BLEPeripheralInfo (time AS DOUBLE, peripheral AS BLEPeripheral, kind
AS INTEGER, msg AS STRING, err AS LONG)
IF kind = 1 THEN
PRINT "Connected to "; peripheral.bleName
peripheral.discoverServices(uuid)
ELSE IF kind = 4 THEN

```

```

    DIM availableServices(1) AS BLEService
    availableServices = peripheral.services
    FOR s = 1 TO UBOUND(availableServices, 1)
        peripheral.discoverCharacteristics(uuid, availableServices(s))
    NEXT
END IF
END SUB

SUB BLEServiceInfo (time AS DOUBLE, peripheral AS BLEPeripheral, service
AS BLEService, kind AS INTEGER, msg AS STRING, err AS LONG)
IF kind = 1 THEN
    DIM characteristics(1) AS BLECharacteristic
    characteristics = service.characteristics
    FOR i = 1 TO UBOUND(characteristics, 1)
        peripheral.discoverDescriptors(characteristics(i))
    NEXT
END IF
END SUB

SUB BLECharacteristicInfo (time AS DOUBLE, peripheral AS BLEPeripheral,
characteristic AS BLECharacteristic, kind AS INTEGER, msg AS STRING, err AS
LONG)
IF kind = 1 THEN
    DIM descriptors(1) AS BLEDescriptor
    descriptors = characteristic.descriptors
    IF UBOUND(descriptors, 1) > 0 THEN
        PRINT "Found descriptors for characteristic "; characteristic.uuid;
": "
        FOR i = 1 TO UBOUND(descriptors, 1)
            peripheral.readDescriptor(descriptors(i))
        NEXT
    END IF
END IF
END SUB

SUB BLEDescriptorInfo (time AS DOUBLE, peripheral AS BLEPeripheral,
descriptor AS BLEDescriptor, kind AS INTEGER, msg AS STRING, err AS LONG)
PRINT "Value for descriptor "; descriptor.uuid; ": ";
DIM value(1) AS INTEGER
value = descriptor.value
FOR i = 1 TO UBOUND(value, 1)
    PRINT RIGHT(HEX(value(i)), 2);
NEXT
PRINT
END SUB

```

---

### **SUB discoverIncludedServices (uuid() AS STRING, service AS BLEService)**

Discover one or more services associated with a services or this device.

Bluetooth LE devices have zero or more services, each of which can have other imbedded services. An example of a service is a battery level with a standard UUID of 180F. This call asks a peripheral for any services imbedded in another service. If any are found, a call is made to BLEServiceInfo.

**uuid** is an array of the service UUIDs desired. Pass an empty array to discover all imbedded services for a service.

The snippet for the `discoverServices` call shows a program that finds the first available Bluetooth LE peripheral and returns all of its available services, including imbedded services.

---

**SUB discoverServices (uuid() AS STRING)**

Discover one or more services associated with this device.

Bluetooth LE devices have zero or more services. An example of a service is a battery level with a standard UUID of 180F. This call asks a peripheral for a list of available services. If any are found, a call is made to `BLEServiceInfo`.

**uuid** is an array of the service UUIDs desired. Pass an empty array to discover all services for the device.

The snippet shows a program that finds the first available Bluetooth LE peripheral and returns all of its available services. Services can have other services imbedded within them; this program also returns the imbedded services.

**Snippet**

```

DIM somePeripheral AS BLEPeripheral, peripheralSet AS INTEGER
BLE.startBLE
DIM uuid() AS STRING
BLE.startScan(uuid)

SUB BLEDiscoveredPeripheral(time AS DOUBLE, peripheral AS BLEPeripheral,
services() AS STRING, advertisements(,) AS STRING, rssi AS SINGLE)
    somePeripheral = peripheral
    BLE.connect(somePeripheral)
    BLE.stopScan
END SUB

SUB BLEPeripheralInfo (time AS DOUBLE, peripheral AS BLEPeripheral, kind
AS INTEGER, msg AS STRING, err AS LONG)
    SELECT CASE kind
        CASE 1
            peripheral.discoverServices(uuid)

        CASE 4
            DIM services(1) AS BLEService, included(1) AS BLEService
            services = peripheral.services
            PRINT "Discovered services:"
            FOR i = 1 TO UBOUND(services, 1)
                PRINT "  services("; i; "): "; services(i).uuid
                peripheral.discoverIncludedServices(uuid, services(i))
            NEXT
        END SELECT
    END SUB

SUB BLEServiceInfo (time AS DOUBLE, peripheral AS BLEPeripheral, service
AS BLEService, kind AS INTEGER, msg AS STRING, err AS LONG)
    IF kind = 2 THEN
        DIM services(1) AS BLEService
        services = service.includedServices
        IF UBOUND(services, 1) > 0 THEN
            PRINT "Found included services for service "; service.uuid; ":"
            FOR i = 1 TO UBOUND(services, 1)
                PRINT "  included service "; i; ": "; services(i).uuid
            NEXT
        END IF
    END IF
END SUB

```

---

**FUNCTION isConnected AS INTEGER**

Returns 1 if the peripheral is currently connected, and 0 if not.

**SUB readCharacteristic (characteristic AS BLECharacteristic)**

Reads the value of a characteristic on the device.

Bluetooth LE devices have zero or more services. These services have characteristics. An example of a service is a battery level with a standard UUID of 180F. A characteristic of this service is the current power level, with a UUID of 2A19. This call asks the device for the value of a characteristic. Once read, the value of the characteristic is returned using a call to `BLECharacteristicsDiscovery`. The value of the characteristic is read using the characteristic's `value` method.

See the `discoverCharacteristics` method for a snippet that uses `readCharacteristic`.

**SUB readDescriptor (descriptor AS BLEDescriptor)**

Reads the value of a descriptor on the device.

Bluetooth LE devices have zero or more services. These services have characteristics. An example of a service is a battery level with a standard UUID of 180F. A characteristic of this service is the current power level, with a UUID of 2A19. Characteristics may have additional information in the form of descriptors. This call asks the device for the value of a descriptor. Once read, the value of the descriptor is returned using a call to `BLEDescriptorInfo`. The value of the descriptor is read using the descriptor's `value` method.

See the `discoverDescriptors` method for a snippet that uses `readDescriptor`.

**FUNCTION rssi**

Returns the current RSSI (Received Signal Strength Indicator) of the radio signal for the peripheral.

**FUNCTION services () () AS BLEService**

Returns an array containing the currently known services for the device. The array may be empty.

Bluetooth LE devices have zero or more services. An example of a service is a battery level with a standard UUID of 180F. This call returns all of the known services for a device, but this may not be all of the services offered by the device. Before making this call, start with a call to `discoverServices` to ask the device for a list of its services. Once the services are collected, `BLEPeripheralInfo` will be called to notify the program that the services are now available. At that point, use this call to determine what those services are.

See the `discoverServices` method for a snippet that uses this call.

**SUB setNotify (characteristic AS BLECharacteristic, flag AS INTEGER)**

Turns notification on or off for a service.

Bluetooth LE devices have zero or more services. Examples of a service are a battery level with a standard UUID of 180F, or a button on the TI key fob in the CC2540 Mini Development Kit. These services can be either push notifications or pull notifications. Pull notifications are used for characteristics that do not update automatically, or do so rarely, like the battery level. Push notifications are used for characteristics that change because of something that happens on the device, like a button being pressed on a key fob.

`setNotify` is used with push notifications. Passing a non-zero value as the **flag** parameter tells the device to start reading push notifications for the device. The `BLECharacteristicInfo` subroutine is called whenever the Bluetooth LE device has an update for this value, just as if the program had issued a `readCharacteristic` call. The value can then be read and processed.

Notifications require increased battery power both on the iOS device and on the Bluetooth LE device. Once the information is no longer needed, call `setNotify` again, but pass 0 for the **flag** parameter to turn push notifications off.

The snippet shows a portion of a program used to monitor the TI key fob from the CC2540 Mini Development Kit. It turns notifications on for the buttons and





accelerometer on the key fob. See the blog on the Byte Works web site for a complete listing and description of the program.

#### Snippet

```
SUB BLEServiceInfo (time AS DOUBLE, peripheral AS BLEPeripheral, service
AS BLEService, kind AS INTEGER, msg AS STRING, err AS LONG)
  IF kind = 1 THEN
    DIM characteristics(1) AS BLECharacteristic
    characteristics = service.characteristics
    FOR i = 1 TO UBOUND(characteristics, 1)
      IF service.uuid = "FFE0" AND characteristics(i).uuid = "FFE1" THEN
        peripheral.setNotify(characteristics(i), 1)
      ELSE IF service.uuid = "FFA0" THEN
        SELECT CASE characteristics(i).uuid
          CASE "FFA1"
            DIM value(1) as INTEGER
            value(1) = 1
            peripheral.writeCharacteristic(characteristics(i), value, 1)

          CASE "FFA3", "FFA4", "FFA5"
            peripheral.setNotify(characteristics(i), 1)
        END SELECT
      ELSE IF service.uuid = "180F" THEN
        batteryCharacteristic = characteristics(i)
        batteryFound = 1
      ELSE IF service.uuid = "1802" THEN
        buzzerCharacteristic = characteristics(i)
        buzzerFound = 1
      END IF
    NEXT
  END IF
END SUB
```

---

#### **FUNCTION uuid AS STRING**

Returns the UUID for this peripheral.

Each peripheral has a UUID that identifies that specific peripheral. This call returns the UUID for the current Bluetooth LE peripheral.

See the `discoverCharacteristics` call for a snippet that uses the `uuid` method.

---

#### **SUB writeCharacteristic (characteristic AS BLECharacteristic, value() AS INTEGER, response AS INTEGER = 0)**

Writes a value for a characteristic to the device.

Bluetooth LE devices have zero or more services. These services have characteristics. An example of a service is an alarm on a Bluetooth LE device that can be triggered remotely. A characteristic of this service is the value used to trigger the alarm, or the current state of the alarm. The value would be a characteristic that is write-enabled. This call sends a new value to a write-enabled characteristic.

The value for a characteristic can take on virtually any form and any length, so it is passed as an array of integers, each of which represents a single byte from the value.

Some characteristics can be written without asking for a response, and others require one. This may be known from the specification of a device; if not, see the `properties` method in the `BLECharacteristic` class for a flag that tells whether a response is required. To get a notification back when the Bluetooth LE device gets the value, including

an error code indicating if the write was successful, pass a non-zero value for the response parameter. A call will be made to `BLECharacteristicInfo` once the write is complete.

The snippet shows a portion of a program used to monitor the TI key fob from the CC2540 Mini Development Kit. The key fob flashes an LED and sounds an audible alarm when the Buzzer button is pressed on the app. See the blog on the Byte Works web site for a complete listing and description of the program.

### Snippet

```
SUB touchUpInside (ctrl AS Button, time AS DOUBLE)
  IF ctrl = soundBuzzer THEN
    IF buzzerFound THEN
      DIM value(1) AS INTEGER
      value = [2]
      keyfob.writeCharacteristic(buzzerCharacteristic, value)
    END IF
  ELSE IF ctrl = quit THEN
    STOP
  END IF
END SUB
```

---

### **SUB writeDescriptor (descriptor AS BLEDescriptor, value() AS INTEGER)**

Writes a value to a descriptor on the device.

Bluetooth LE devices have zero or more services. These services have characteristics. An example of a service is a battery level with a standard UUID of 180F. A characteristic of this service is the current power level, with a UUID of 2A19. Characteristics may have additional information in the form of descriptors. This call writes a value to one of these descriptors.

The value for a descriptor can take on virtually any form and any length, so it is passed as an array of integers, each of which represents a single byte from the value.

A call will be made to `BLEDeviceDiscovery` once the write is complete.

---

## BLEPeripheralManager

`BLEPeripheralManager` is used to create and manage Bluetooth LE slave communications. It is instantiated using the `BLE` class' `newBLEPeripheralManager` call. Once instantiated, it is used to create and advertise services created on the local device.

---

### **SUB addService (service AS BLEMutableService)**

Adds a service to the list of services available when the device advertises its status as a Bluetooth LE slave device. The service and all characteristics within the service must be set up before this call is made; once made, the services and characteristics cannot be modified.

The snippet shows a subroutine from the BLEChat A sample in techBASIC that sets up a service with a single characteristic, then advertises this service. The complete program can be found in the O'Reilly Books folder in techBASIC.

### Snippet

```
DIM localTextCharacteristic AS BLEMutableCharacteristic
DIM localReadyCharacteristic AS BLEMutableCharacteristic
localName$ = "BLEChatA"
localServiceUUID$ = "01A00C7B-153C-45F9-B083-FE135E4E5CA0"
localCharacteristicUUID$ = "01A10C7B-153C-45F9-B083-FE135E4E5CA0"
DIM peripheralManager AS BLEPeripheralManager

SUB advertise
```

```

peripheralManager = BLE.newBLEPeripheralManager

DIM service AS BLEMutableService
service = peripheralManager.newService(localServiceUUID$, 1)

localTextCharacteristic =
service.newCharacteristic(localTextCharacteristicUUID$, $0012, $0001)
localReadyCharacteristic =
service.newCharacteristic(localReadyCharacteristicUUID$, $0006, $0003)

peripheralManager.addService(service)
peripheralManager.startAdvertising(localName$)
END SUB

```

---

**FUNCTION newService (UUID AS String, isPrimary AS INTEGER) AS BLEMutableService**

Creates and returns a new service for a Bluetooth LE slave device.

**UUID** is the UUID for the new service, while **isPrimary** designates whether this is a primary service.

A primary service is a service for a primary function of a device, such as acceleration for an accelerometer. A secondary service is one that is only relevant in the context of another service, such as the battery level for the device. Pass a non-zero value for a primary service, or a zero value for a secondary service.

The typical workflow is to create a service using this call, then add one or more characteristics using the `newCharacteristic` call from the `BLEMutableService` class. Once all characteristics have been created, the service is added to the list of services to be offered using a call to the `addService` method.

See the `addService` method for a snippet showing this call in use.

---

**SUB respondToRequest (request AS BLEATTRequest, error AS INTEGER)**

A slave device uses this method to respond to a read or write request from a master device.

**request** is the read or write request this call is responding to.

Read requests are passed to the slave device by a call to `BLEPeripheralManagerInfo` with a kind flag of 4. The event gets a single `BLEATTRequest` object. The slave device fills in the value and passes it back to the master device with this call.

Write requests are also passed to the slave device by a call to `BLEPeripheralManagerInfo`, this time with a kind flag of 5. The event gets an array of one or more `BLEATTRequest` objects. If all of the write requests can be fulfilled, the slave device does the writes and responds to the master with the `respondToRequest` method, passing back the first of the `BLEATTRequest` objects from the array provided to `BLEPeripheralManagerInfo`. If any of the write requests cannot be performed, the slave device skips all of them and returns an appropriate error code.

**error** is an error code, or 0 if there was no error. It can be any of the following values. See the Bluetooth 4.0 specification, Volume 3, Part F, Section 3.4.1.1 for details.

Error	Meaning
0	The read or write was completed successfully.
1	The attribute handle is invalid on this peripheral.
2	The attribute's value cannot be read.
3	The attribute's value cannot be written.
4	The attributes protocol data unit (the message) is invalid.
5	The attribute cannot be read or written without authentication.
6	The slave device does not support the request.
7	The offset was past the end of the attribute's value.
8	The attribute cannot be read or written without authorization.

- 9 Too many writes have been requested; the queue is full.
- 10 The attribute does not exist on the slave device.
- 11 The attribute cannot be read using the ATT read blob request.
- 12 The encryption key used for the encryption length is too short.
- 13 The length of the attribute's value is too short for the intended operation.
- 14 The ATT request has encountered an internal error and cannot complete the operation.
- 15 The attribute requires encryption before the value can be read or written.
- 16 The attribute type is not a supported grouping attribute.
- 17 There are not enough resources to support the request.

---

**SUB removeService (service AS BLEMutableService)**

Removes a service to the list of services for the device.

Any devices connected to the service will be notified of the change, and must resubscribe to all services.

---

**SUB startAdvertising (name AS STRING, services() AS STRING)**

Begins advertising services for a Bluetooth LE slave device.

**name** is the name of the device. It can be any convenient string.

**services** is an array of the available services the device offers.

See `addService` for a snippet showing the `startAdvertising` call.

---

**FUNCTION state AS INTEGER**

Returns the current state of the manager.

When the peripheral manager is created, the state will be 0. Each time the state changes, a call is made to `BLEPeripheralManagerInfo` with the `kind` parameter set to 2, indicating a new state value is available. Use this method to read the new state.

The state may be any of the following.

Value	State
0	The state is unknown. This is the initial value; it will be updated quickly to a new state.
1	The manager is resetting after losing a connection. Another update to the state flag will follow quickly.
2	The platform does not support the role of a BLE server.
3	The application is not authorized to use the BLE server role.
4	Bluetooth is currently powered off.
5	Bluetooth is powered on and available for use.

---

**FUNCTION updateValue (BLEMutableCharacteristic characteristic, value() AS INTEGER) AS INTEGER**

Called when a BLE slave needs to send a value to a central device, `updateValue` sends a new value for a single characteristic. It returns 1 if the value was successfully sent to the central device, and 0 if the value could not be sent because the communications channel was busy.

**characteristic** is the characteristic whose value is being changed.

**value** is the new value. The value is an array of bytes stored in integers. The number of bytes can vary up to 20 bytes, so long as both the slave and central device agree on the number or meaning of the bytes. Bluetooth LE is designed for sending short packets of information, and imposes a 20-byte limit on a single data packet.

To send a longer stream of data, set up a handshaking arrangement using a second characteristic that can be both read and written. Set the characteristic from the central device when a packet of information has been processed. The slave device writes another packet of data and clears the characteristic.

The snippet shows this idea in action from the Bluetooth LE slave device. It is from BLE Chat A, a sample program found in the O'Reilly Books folder of techBASIC. The sample shows a complete program that, if executed on two different iOS devices, implements a simple Bluetooth LE based peer-to-peer texting program.

#### Snippet

```
SUB sendText
! Make sure the central device is ready for more data. If not wait, but
! time out after 0.5 seconds.
time# = System.ticks
DIM value2(1) AS INTEGER
DO
    value2 = localReadyCharacteristic.value
    done = (UBOUND(value2, 1) = 1) AND (value2(1) = 1)
LOOP WHILE (NOT done) AND (System.ticks - time# < 0.5)

! Place up to 20 bytes into a value array.
IF LEN(line$) > 20 THEN
    length% = 20
ELSE
    length% = LEN(line$)
END IF
DIM value(length%) AS INTEGER
FOR i% = 1 TO length%
    value(i%) = ASC(MID(line$, i%, 1))
NEXT

! Set our handshaking value to 0. The central will set it to 1 after
! receiving the data.
value2 = [0]
localReadyCharacteristic.setValue(value)

! Send the data to the central device.
result% = peripheralManager.updateValue(localTextCharacteristic, value)

! If the send was successful, remove the bytes from the line.
IF result% THEN
    console.setText(console.getText & LEFT(line$, 20))
    IF LEN(line$) > 20 THEN
        line$ = RIGHT(line$, LEN(line$) - 20)
        sendText
    ELSE
        line$ = ""
    END IF
END IF
END IF
END SUB
```

---

## BLEService

BLEService is a class used to communicate with Bluetooth LE devices. See the description of the BLE class for an overview of the classes used to access Bluetooth LE devices.

---

### **FUNCTION characteristics () () AS BLECharacteristics**

Returns the known characteristics for this service.

Characteristics are not immediately available. Use the BLEPeripheral class' discoverCharacteristics call to ask the device for a list of the services. Once the characteristics are

collected, the program is notified by a call to the `BLEServiceInfo` subroutine. After this call, this method will return the characteristics for this service.

See the `discoverCharacteristics` call in the `BLEPeripheral` class for a snippet that uses the `characteristics` method.

---

**FUNCTION `includedServices` ( ) ( ) AS `BLEService`**

Services may contain other services, which are called included services. This call returns the known included services for this service.

Included services are not immediately available. Use the `BLEPeripheral` class' `discoverIncludedServices` call to ask the device for a list of the included services. Once the included services are collected, the program is notified by a call to the `BLEServiceInfo` subroutine. After this call, this method will return the included services for this service.

See the `discoverServices` call in the `BLEPeripheral` class for a snippet that uses the `includedServices` method.

---

**FUNCTION `isPrimary` AS `INTEGER`**

Returns a nonzero value if this is a primary service, and a zero value if it is a secondary service.

A primary service is a service for a primary function of a device, such as acceleration for an accelerometer. A secondary service is one that is only relevant in the context of another service, such as the battery level for the device.

---

**FUNCTION `uuid` AS `STRING`**

Returns the UUID for this service.

Each service has a UUID that identifies that specific service. This call returns the UUID for the current Bluetooth LE service.

See the `discoverServices` call in the `BLEPeripheral` class for a snippet that uses the `uuid` method.

---

## Comm

The `Comm` class supports various forms of network communication, including TCP/IP, UDP, HTTP and FTP. The `Comm` class is a predefined class.

For the most part, these classes are designed to use the existing BASIC methodologies for stream input and output, making them extremely easy to use. The normal way to use one of these networking technologies is to open a stream using one of the calls in this class, then read and write to the stream using standard BASIC input and output statements like `PRINT`, `LINE INPUT`, `GET` and `PUT`. See the snippets for examples.

---

**FUNCTION `dirFTP` (`url` AS `STRING` = "", `username` AS `STRING` = "", `password` AS `STRING` = "") AS `STRING`**

Functions just like the `DIR` function from BASIC, reading entries in a directory, but in this case the directory is on an FTP site. The first call should specify the FTP directory as a fully qualified path name. It will return the name of the first file in the directory, or an empty string if there are no files in the directory. Subsequent calls should not use any parameters, and will return the remaining directory entries in sequence. The empty string is returned once all directory entries have been processed.

`url` is the fully qualified name of the FTP site, as in `"ftp://www.myftp.com/ftpfolder/"`.

`username` is the user name. It can be omitted for anonymous FTP sites. `password` is the password, and, like `username`, can be omitted for anonymous sites.

The snippet shows a program that prints the entries in a directory from a hypothetical FTP site.

Snippet

```

url$ = "ftp://someserver.somesite.net/myftp/"
username$ = "MyUserName"
password$ = "MyPassWord"
name$ = Comm.dirFTP(url$, username$, password$)
WHILE name$ <> ""
    PRINT name$
    name$ = Comm.dirFTP
WEND

```

---

**FUNCTION existsFTP (url AS STRING, username AS STRING = "", password AS STRING = "") AS INTEGER**

Functions just like the EXISTS function from BASIC, returning zero if a file does not exist and a non-zero value if it does, but in this case the file is on an FTP site.

Directories are treated as files, so this call can also be used to determine if a directory exists. Use isDirFTP to determine if a particular entry is a directory or a file.

This call works well for checking to see if a file exists before performing some action, but it is not an efficient way to check for a large number of files. When a large number of files need to be checked, use dirFTP to get a list of the files and check the resulting list.

**url** is the fully qualified name of a file on the FTP site, as in "ftp://www.myftp.com/ftpfolder/myfile".

**username** is the user name. It can be omitted for anonymous FTP sites. **password** is the password, and, like **username**, can be omitted for anonymous sites.

The snippet shows a program that determines if a particular file exists.

Snippet

```

url$ = "ftp://someserver.somesite.net/myftp/myfile.txt"
username$ = "MyUserName"
password$ = "MyPassWord"
IF Comm.existsFTP(url$, username$, password$) THEN
    PRINT url$; " exists."
ELSE
    PRINT url$; " does not exist."
END IF

```

---

**FUNCTION isDirFTP (url AS STRING, username AS STRING = "", password AS STRING = "") AS INTEGER**

Functions just like the ISDIR function from BASIC, returning zero if a file is not a directory and a non-zero value if it is, but in this case the file is on an FTP site.

**url** is the fully qualified name of a file on the FTP site, as in "ftp://www.myftp.com/ftpfolder/somefile".

**username** is the user name. It can be omitted for anonymous FTP sites. **password** is the password, and, like **username**, can be omitted for anonymous sites.

The snippet shows a program that determines if a hypothetical file is a folder.

Snippet

```

url$ = "ftp://someserver.somesite.net/myftp/somefile"
username$ = "MyUserName"
password$ = "MyPassWord"
IF Comm.isDirFTP(url$, username$, password$) THEN
    PRINT url$; " is a directory."
ELSE
    PRINT url$; " is a file."
END IF

```

**FUNCTION isReadyForOutput (channel AS INTEGER) AS INTEGER**

Checks to see if a TCP/IP, HTTP or FTP stream is ready to receive output without blocking.

**channel** is the number of an open stream.

The operating system may allow opening of an external stream even if it will not accept output, or a stream may temporarily or permanently block output for any number of reasons, such as a lost connection or disk full error. If a program writes to a stream that is not ready to receive output, the stream will block further execution until the output can be written—which may never happen. This call lets a program check to see if a stream is ready to receive output. The program can implement a timeout if the stream doesn't respond after a specific amount of time, or do other tasks while waiting for the stream to be ready to receive more output.

The snippet shows a program that tries to open a fictitious TCP/IP channel. After waiting for one second, it stops with a timeout message. Of course, if the TCP/IP channel really exists, the program will exit without printing the error.

Snippet

```

DIM url AS STRING, t AS DOUBLE
url = "no_such_address.local"
Comm.openTCPIP(1, url, 5002)
t = System.ticks
found = 0
WHILE System.ticks - t < 1.0
    IF Comm.isreadyforoutput(1) THEN
        found = 1
        t = t - 1.0
    END IF
WEND
IF NOT found THEN
    PRINT "The connection to "; url; " failed."
END IF

```

**SUB killFTP (url AS STRING, username AS STRING = "", password AS STRING = "")**

Functions just like the KILL function from BASIC, deleting a file or directory, but in this case the file is on an FTP site.

It is not an error to attempt to delete a file that does not exist. Directories can be deleted with killFTP, but there cannot be any files in the directory.

**url** is the fully qualified name of a file on the FTP site, as in "ftp://www.myftp.com/ftpfolder/somefile".

**username** is the user name. It can be omitted for anonymous FTP sites. **password** is the password, and, like **username**, can be omitted for anonymous sites.

The snippet shows a program that deletes a hypothetical file.



Snippet

```
url$ = "ftp://someserver.somesite.net/myftp/somefile"
username$ = "MyUserName"
password$ = "MyPassWord"
Comm.killFTP(url$, username$, password$)
```

---

```
SUB mkDirFTP (url AS STRING, username AS STRING = "", password AS STRING =
    "")
```

Functions just like the MKDIR function from BASIC, creating a directory, but in this case the directory is on an FTP site.

**url** is the fully qualified name of the new directory, as in "ftp://www.myftp.com/ftpfolder/newfolder".

**username** is the user name. It can be omitted for anonymous FTP sites. **password** is the password, and, like **username**, can be omitted for anonymous sites.

The snippet shows a program that creates a new directory on a hypothetical FTP site.

Snippet

```
url$ = "ftp://someserver.somesite.net/myftp/newdirectory"
username$ = "MyUserName"
password$ = "MyPassWord"
Comm.mkDirFTP(url$, username$, password$)
```

---

```
SUB openTCPIP (channel AS INTEGER, host AS STRING, port AS LONG, bigEndian AS
    INTEGER = 1)
```

Opens a TCP/IP stream for input and output.

**channel** is the number used to open the stream; it will also be used in input and output commands used to read the stream. **host** is the name of the host to open; this can be an IP address or, if the operating system has a lookup service available, the URL of the host. **port** is the number of the port to use for communication.

The BASIC GET and PUT commands can be used to read and write binary values to the port. By default, INTEGER and LONG values are written and read with the most significant byte first, which is the normal format for Internet communication, but not the natural byte order for iOS and most Intel chips. Pass 0 for **bigEndian** to reverse the byte order for INTEGER and LONG values.

For the most part, all standard commands used to read and write files can also be used to read and write TCP/IP streams. Here is a list of the available commands and any changes in the way they function.

Command	Notes
CLOSE	Works just like file I/O.
EOF	There is no end of file for a TCP/IP port, only a situation where no more bytes are currently available. EOF returns true (a non-zero value) if there are bytes currently available for input, and false (zero) if there are no bytes currently available.
GET	Works just like file I/O.
INPUT	Works just like file I/O.
LINE INPUT	Works just like file I/O.
LOC	TCP/IP streams are not random access. This function results in a runtime error.
LOF	Returns the number of bytes returned from the port that have not yet been read.
PRINT	Works just like file I/O.
PUT	Works just like file I/O.
SEEK	TCP/IP streams are not random access. This function results in a runtime error.

While BASIC input commands are synchronous, the underlying implementation of TCP/IP streams is asynchronous. Use EOF or LOF to implement asynchronous access within BASIC. Here is a general outline of code that can be placed in, say, the `nullEvent` subroutine to handle information from a TCP/IP stream as it arrives.

```

IF NOT EOF(1) THEN
  processTCPIP
END IF

```

---

**SUB openUDP (channel AS INTEGER, host AS STRING, port AS LONG, bigEndian AS INTEGER = 1)**

Opens a UDP stream for input and output.

**channel** is the number used to open the stream; it will also be used in input and output commands used to read the stream. **host** is the name of the host to open; this can be an IP address or, if the operating system has a lookup service available, the URL of the host. **port** is the number of the port to use for communication.

The BASIC GET and PUT commands can be used to read and write binary values to the port. By default, INTEGER and LONG values are written and read with the most significant byte first, which is the normal format for Internet communication, but not the natural byte order for iOS and most Intel chips. Pass 0 for **bigEndian** to reverse the byte order for INTEGER and LONG values.

For the most part, all standard commands used to read and write files can also be used to read and write UDP streams. Here is a list of the available commands and any changes in the way they function.

Command	Notes
CLOSE	Works just like file I/O.
EOF	There is no end of file for a UDP port, only a situation where no more bytes are currently available. EOF returns true (a non-zero value) if there are bytes currently available for input, and false (zero) if there are no bytes currently available.
GET	Works just like file I/O.
INPUT	Works just like file I/O.
LINE INPUT	Works just like file I/O.
LOC	UDP streams are not random access. This function results in a runtime error.
LOF	Returns the number of bytes returned from the port that have not yet been read.
PRINT	Works just like file I/O.
PUT	Works just like file I/O.
SEEK	UDP streams are not random access. This function results in a runtime error.

Generally, bytes are written to BASIC streams one at a time. There are some situations in UDP communication where it is useful to send a block of bytes as a single transmission record. See `writeUDP` for a way to send a block of bytes.

BASIC input commands are synchronous, the underlying implementation of UDP streams is asynchronous. Use EOF or LOF to implement asynchronous access within BASIC. Here is a general outline of code that can be placed in, say, the `nullEvent` subroutine to handle information from a UDP stream as it arrives.

```

IF NOT EOF(1) THEN
  processUDP
END IF

```

The snippet shows a simple program that writes a line to a UDP port and waits for a line to be returned. If the UDP port is connected to, say, an XBee radio in transparent mode, the radio will send the information out to its serial port. If the radio is wired so the output is sent back to its serial input, the text will be printed in the techBASIC terminal. See the Byte Works web site for a blog that gives a full description of how to do this.

#### Snippet

```

out$ = "Hello, UDP!" & CHR(13)
DIM packet(LEN(out$)) AS BYTE
FOR i% = 1 TO LEN(out$)
  packet(i%) = ASC(MID(out$, i%, 1))
NEXT

```

```
Comm.openUDP(1, "10.0.1.58", 9750)
Comm.writeUDP(1, packet)
LINE INPUT #1, in$
PRINT in$
CLOSE #1
```

---

```
SUB readFTP (channel AS INTEGER, url AS STRING, username AS STRING = "",  

password AS STRING = "")
```

Opens an FTP stream for input.

**channel** is the number used to open the stream; it will also be used in input commands used to read the stream.

**url** is the fully qualified name of the file, as in "ftp://www.myftp.com/ftpfolder/myfile.txt".

**username** is the user name. It can be omitted for anonymous FTP sites. **password** is the password, and, like **username**, can be omitted for anonymous sites.

FTP directories can be read as if they were a text file. Be sure and place a closing / character after the name of a directory to distinguish it from a file.

For the most part, all standard commands used to read files can also be used to read FTP streams. Here is a list of the available commands and any changes in the way they function.

Command	Notes
CLOSE	Works just like file I/O.
EOF	Works just like file I/O.
GET	Works just like file I/O.
INPUT	Works just like file I/O.
LINE INPUT	Works just like file I/O.
LOC	FTP streams are not random access. This function results in a runtime error.
LOF	Returns the number of bytes that have currently been received from the web site that have not been read.
PRINT	readFTP opens a file for input, so this command results in a runtime error.
PUT	readFTP opens a file for input, so this command results in a runtime error.
SEEK	FTP streams are not random access. This function results in a runtime error.

While BASIC input commands are synchronous, the underlying implementation of FTP streams is asynchronous. Use LOF to implement asynchronous access within BASIC. LOF returns the number of available but unread bytes for FTP streams, so a BASIC program can check LOF to see if any bytes are available, proceeding to do other tasks if not. EOF will return a nonzero value when the entire page has been read. Putting these facts together, here is a general outline of code that can be placed in, say, the nullEvent subroutine to handle information from an FTP stream as it arrives.

```
IF NOT EOF(1) AND LOF(1) > 0 THEN
    processFTP
END IF
```

The snippet shows a simple synchronous program that reads and prints a text file from a hypothetical FTP site.

Snippet

```

url$ = "ftp://someserver.somesite.net/myftp/myfile.txt"
username$ = "MyUserName"
password$ = "MyPassWord"
Comm.readFTP(1, url$, username$, password$)
WHILE NOT EOF(1)
    LINE INPUT #1, a$
    PRINT a$
WEND

```

---

```

SUB readHTTP (channel AS INTEGER, url AS STRING, request AS STRING = "GET",
body AS STRING = "", header AS STRING = "", value AS STRING = "",
username AS STRING = "", password AS STRING = "", timeout = 5.0)

```

Opens an HTTP stream for input.

**channel** is the number used to open the stream; it will also be used in input commands used to read the stream. **url** is the Internet address of the page to open.

HTTP requests are driven by a command; the command is passed as the **request** parameter. For example, the GET command is used to read an HTTP page. Some requests, such as PUT, send data back to the web server. The **body** parameter is used to send data to the server. Some requests also use a header and header body; these are passed as the **header** and **value** parameters. The default parameters read the contents of a web page. See references on the HTTP protocol for a complete description of the various HTTP requests. For an overview, start with [http://en.wikipedia.org/wiki/List\\_of\\_HTTP\\_header\\_fields](http://en.wikipedia.org/wiki/List_of_HTTP_header_fields).

When used, the **username** and **password** fields are used for basic authentication. A typical call to get a web page from a site that requires authentication would look like this:

```

Comm.readHTTP(1, http://www.byteworks.us, "GET", "", "", "", "user name",
"password")

```

HTTP requests also return a status code. See the `statusHTTP` command for a way to read the status code.

For the most part, all standard commands used to read files can also be used to read HTTP streams. Here is a list of the available commands and any changes in the way they function.

Command	Notes
CLOSE	Works just like file I/O.
EOF	Works just like file I/O.
GET	Works just like file I/O.
INPUT	Works just like file I/O.
LINE INPUT	Works just like file I/O.
LOC	HTTP streams are not random access. This function results in a runtime error.
LOF	Returns the number of bytes that have currently been received from the web site that have not been read.
PRINT	HTTP is a read-only service, so this command results in a runtime error.
PUT	HTTP is a read-only service, so this command results in a runtime error.
SEEK	HTTP streams are not random access. This function results in a runtime error.

While BASIC input commands are synchronous, the underlying implementation of HTTP streams is asynchronous. Use `LOF` to implement asynchronous access within BASIC. `LOF` returns the number of available but unread bytes for HTTP streams, so a BASIC program can check `LOF` to see if any bytes are available, proceeding to do other tasks if not. `EOF` will return a nonzero value when the entire page has been read. Putting these facts together, here is a general outline of code that can be placed in, say, the `nullEvent` subroutine to handle information from an HTTP stream as it arrives.

```
IF NOT EOF(1) AND LOF(1) > 0 THEN
  processHTML
END IF
```

**timeout** is the number of seconds to wait for a reply before giving up and issuing error 55, "The network operation did not complete in the allowed time."

The snippet shows a simple synchronous program that prints the source code for the Byte Works landing page.

#### Snippet

```
Comm.readHTTP(1, "http://www.byteworks.us")
WHILE NOT EOF(1)
  LINE INPUT #1, a$
  PRINT a$
WEND
PRINT "Status code: "; Comm.statusHTTP(1)
CLOSE #1
```

---

#### FUNCTION statusHTTP (channel AS INTEGER) AS INTEGER

Returns the status code from an HTTP stream. **channel** is the channel number passed to the `readHTTP` command when the stream was opened.

The status code is generally set after all data has been passed back from the HTTP request. It must be read before the stream is closed. A normal status after reading an HTTP page is 200; other statuses are possible, even for a successful read. See <http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html> for a current list of status codes.

See the `readHTTP` command for a code sample that uses this call.

---

#### SUB writeFTP (channel AS INTEGER, url AS STRING, username AS STRING = "", password AS STRING = "")

Opens an FTP stream for output.

**channel** is the number used to open the stream; it will also be used in input commands used to write the stream.

**url** is the fully qualified name of the file, as in "ftp://www.myftp.com/ftpfolder/myfile.txt".

**username** is the user name. It can be omitted for anonymous FTP sites. **password** is the password, and, like **username**, can be omitted for anonymous sites.

For the most part, all standard commands used to write files can also be used to read FTP streams. Here is a list of the available commands and any changes in the way they function.

Command	Notes
CLOSE	Works just like file I/O.
EOF	Always returns 0.
GET	<code>writeFTP</code> opens a file for output, so this command returns a runtime error.
INPUT	<code>writeFTP</code> opens a file for output, so this command returns a runtime error.
LINE INPUT	<code>writeFTP</code> opens a file for output, so this command returns a runtime error.
LOC	Always returns 0.
LOF	Always returns 0.
PRINT	Works just like file I/O.
PUT	Works just like file I/O.
SEEK	FTP streams are not random access. This function results in a runtime error.

The snippet shows a simple program that writes a file to a hypothetical FTP site, then reads it and prints the file.

### Snippet

```
url$ = "ftp://someserver.somesite.net/myftp/myfile.txt"
username$ = "MyUserName"
password$ = "MyPassWord"

! Create and write the file.
Comm.writeFTP(1, url$, username$, password$)
PRINT #1, "Hello, FTP."
CLOSE #1

! Read and print the file.
Comm.readFTP(1, fileUrl$, username$, password$)
WHILE NOT EOF(1)
    LINE INPUT #1, line$
    PRINT line$
WEND
CLOSE #1
```

---

### **SUB writeUDP (channel AS INTEGER, data() AS INTEGER)**

Writes a block of data to a UDP stream.

**channel** is the channel number passed to the openUDP command when the stream was opened.

**data** is an array of the bytes to send to the stream. Each element is converted to an integer, after which the least significant 8 bits are converted to a byte. The result is sent to the UDP stream. Be sure to size the array so it holds exactly the number of bytes to write to the stream.

There are situations in UDP communication, such as sending a command to an XBee radio, where it is useful to send a block of data in a single transmission. BASIC normally writes bytes one at a time to the stream, though. This command gives you a way to bypass the normal output mechanism, sending the contents of an array in a single transmission.

See openUDP for a code snippet that shows how to use this method.

---

## HiJack

HiJack is a hardware device developed at the University of Michigan that plugs into the headphone jack on your iPad or iPhone. It's essentially an analog to digital converter that converts a 0-2.75 volt input into an 8 bit value ranging from 0 to 255. techBASIC supports HiJack, allowing you to easily develop programs to access any HiJack compatible sensors.

The HiJack class provides the link between the HiJack hardware and your program.

See <http://eecs.umich.edu/~prabal/projects/hijack/> for an introduction to the HiJack hardware.

Seed Studios sells the HiJack hardware. Start with <http://www.seeedstudio.com/depot/hijack-development-pack-p-865.html>, but be sure to check out the various modular components in the GROVE System, many of which work with the HiJack Development Kit. See [http://www.seeedstudio.com/wiki/GROVE\\_System](http://www.seeedstudio.com/wiki/GROVE_System).

There is a short introduction to HiJack showing how to connect a potentiometer to the HiJack hardware on the Byte Works web site at <http://www.byteworks.us>. This simple project gets you started with the HiJack hardware, preparing the way for more complex projects.

---

### **FUNCTION receive () () AS DOUBLE**

Returns the most recently read value from the HiJack hardware, along with a time stamp indicating when the value was read.

If the HiJack hardware has not been started, calling this function starts it. The returned value is a two-element array. The first element is the most recently read value from the HiJack hardware; it will be a value in the range 0 to 255. The second element of the array is a time stamp indicating when the value was read.

In normal use, **receive** is called to read a value. The program will then loop until the time stamp changes, processing a value from the hardware each time the time stamp updates.

The snippet collects a data point every 0.1 seconds, showing the most recent 100 points on a continuously updating plot. It ignores the time stamp, since the goal is to plot a point based on a fixed time scale, regardless of when the samples are actually collected. This gives an oscilloscope-like output that shows the raw data from any HiJack sensor. It's a perfect place to test hardware, as well as a good starting place for custom applications.

#### Snippet

```
! Shows a running plot of HiJack
! input for the last 10 seconds
! in 0.1 second intervals.
!
! Initialize the display with the
! value set to 0.
DIM value(100, 2)
FOR t = 1 TO 100
    value(t, 1) = (t - 100)/10.0
NEXT

! Initialize the plot and show
! it.
DIM p as Plot, ph as PlotPoint
p = Graphics.newPlot
p.setTitle("HiJack Raw Data")
p.setXAxisLabel("Time in Seconds")
p.setYAxisLabel("Value Read")
p.showGrid(1)
p.setGridColor(0.8, 0.8, 0.8)

ph = p.newPlot(value)
ph.setColor(1, 0, 0)
ph.setPointColor(1, 0, 0)

! Set the plot range and
! domain. This must be done
! after adding the first
! PlotPoint, since that also
! sets the range and domain.
p.setView(-10, 0, 0, 255, 0)

system.showGraphics

! Loop continuously, collecting
! HiJack data and updating the
! plot.
DIM time AS double
time = System.ticks - 10.0
WHILE 1
    ! Wait for 0.1 seconds to
    ! elapse.
    WHILE System.ticks < time + 10.1
        WEND
    time = time + 0.1
```

```

! Get and plot one data point.
h = HiJack.receive
FOR i = 1 TO 99
  value(i, 2) = value(i + 1, 2)
NEXT
value(100, 2) = h(1)
ph.setPoints(value)
Graphics.repaint
WEND

```

---

**SUB send (value AS INTEGER)**

Sends a value to the HiJack hardware. The value should be in the range 0 to 255.

The current version of the HiJack firmware uses any value sent by **send** to set the sample rate for the hardware. A value of 0 should not be sent—it can hang the program, forcing you to restart techBASIC. Values from 1 to 128 set the number of samples per second to approximately

$$f = 0.83 * \text{value} + 0.19$$

Values over 128 also set the sample rate, but the rate is set in a more-or-less random way, and the rate is always lower than the maximum sample rate set by passing 128.

Future versions of the HiJack firmware may use this value in a different way. If the HiJack firmware documentation gives contradictory information, trust the HiJack documentation—and check the Byte Works web site for an update to this manual!

---

**SUB setRate (rate)**

Sets the sample rate to **rate** hertz (samples per second, abbreviated Hz). Valid sample rates range from about 0.6 to 116 Hz. Values outside the supported range are pinned to the supported range. The default sample rate is 50, or about 42 Hz.

---

**SUB stopHiJack**

Stops sampling the HiJack hardware.

Getting data from the HiJack hardware draws power from the iPhone or iPad battery. If your program will process data for a significant amount of time after collecting it from the HiJack device, use this command to stop sampling the HiJack device and preserve battery power.

Sampling is turned off automatically when a program completes. Sampling is turned on automatically by a call to **receive**, **send** or **setRate**.

---

## Sensors

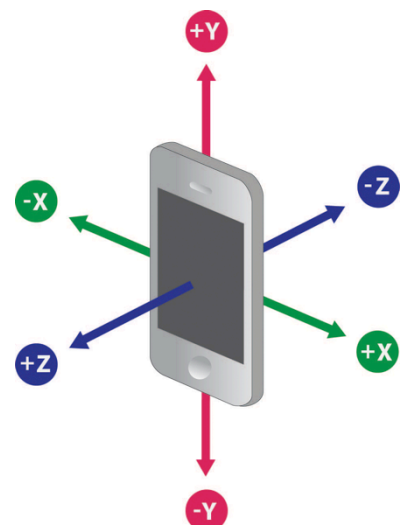
The **Sensors** class gives your programs access to the physical sensors built into the iPhone, iPod or iPad. You can sample physical data, save it, process it, and display results, all on your iOS device. This turns your iOS computer into a powerful physical data collection and processing tool.

A predefined object by the same name gives you access to the methods in this class. Multiple examples show how to get the most from the data.

---

**FUNCTION accel () (4) AS DOUBLE**

Returns the most recently sampled acceleration. The vector returned has four elements. The first three are the acceleration as a G force for the X, Y and Z axis, respectively. For iOS, one G is defined as 9.81 meters/second<sup>2</sup>. The accelerometer in current models of the iPhone and iPad are limited to a





maximum of about 2G. The orientation of the axis relative to the iPhone is shown in the illustration; the iPad is similar.

The fourth element is a time stamp in seconds. It is consistent with the other sensor time stamps.

Using this method starts the accelerometer if it is not already sampling. The default internal sample rate is 0.01 seconds per sample. Your program can adjust this rate using the `acclRate` function. Keep in mind that the sampling occurs on a completely separate thread. Your program can impact the rate samples are taken by using too much of the available computing power. It is also possible for your program to miss samples if too many calculations are done between samples. A sample rate of 0.01 seconds is generally about as fast as iOS can support as this is written; processing the data as it is collected makes it unlikely all points will be collected, but then, samples every 0.01 seconds may not be needed.

The accelerometer draws power, and draws more power for faster sampling rates. In addition to adjusting the sample rate to the lowest rate needed, your program should stop the accelerometer with `stopAccel` as soon as data collection completes. The accelerometer is automatically stopped when your program ends.

Acceleration can be sampled two ways. This method is a little slower, resulting in data that generally has less consistency in the time between samples, but allows the program to react to the data as it is sampled. See the `sample` method for a way to sample the data in blocks.

The snippet shows a simple program that displays the current acceleration as a continuously updating plot. This program is also a sample in the techBASIC app; the sample is called Accelerometer.

#### Snippet

```
! Shows a running plot of the acceleration for the last 10
! seconds in 0.1 second intervals.
!
! Initialize the display with acceleration set to 0 along all axis.
DIM ax(100, 2), ay(100, 2), az(100, 2)
FOR t = 1 TO 100
    ax(t, 1) = t/10.0
    ay(t, 1) = t/10.0
    az(t, 1) = t/10.0
NEXT

! Initialize the plot and show it.
DIM p as Plot, px as PlotPoint, py as PlotPoint, pz as PlotPoint
p = Graphics.newPlot
p.setTitle("Acceleration in Gravities")
p.setXAxisLabel("Time in Seconds")
p.setYAxisLabel("Acceleration: X: Green, Y: Red, Z: Blue")
p.showGrid(1)
p.setGridColor(0.8, 0.8, 0.8)

px = p.newPlot(ax)
px.setColor(0, 1, 0)
px.setPointColor(0, 1, 0)

py = p.newPlot(ay)
py.setColor(1, 0, 0)
py.setPointColor(1, 0, 0)

pz = p.newPlot(az)
pz.setColor(0, 0, 1)
pz.setPointColor(0, 0, 1)

! Set the plot range and domain. This must be done
! after adding the first PlotPoint, since that also
! sets the range and domain.
p.setView(0, -2, 10, 2, 0)
```

```

system.showGraphics

! Loop continuously, collecting accelerometer data
! and updating the plot.
sensors.setAccelRate(0.1)
t0 = -1
index = 1
WHILE (1)
    a = sensors.accel
    IF a(4) > t0 THEN
        ax(index, 2) = a(1)
        ay(index, 2) = a(2)
        az(index, 2) = a(3)
        px.setPoints(ax)
        py.setPoints(ay)
        pz.setPoints(az)
        index = index + 1
        IF index > 100 THEN index = 1
        Graphics.repaint
        t0 = a(4)
    END IF
WEND

```

**FUNCTION accelAvailable AS INTEGER**

Returns zero if no accelerometer is available, and a non-zero value if an accelerometer is available.  
All current iOS devices support an accelerometer.

**FUNCTION altimeterAvailable AS INTEGER**

Returns zero if no altimeter is available, and a non-zero value if an altimeter is available.  
iPhones beginning with the iPhone 6 have an altimeter.  
See **startAltimeter** for a description of how to use the altimeter.

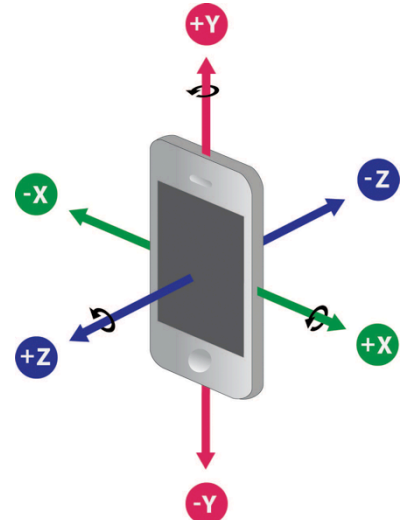
**FUNCTION gyro () (4) AS DOUBLE**

Returns the most recently sampled rotation rate. The vector returned has four elements. The first three are the rotation rate about the X, Y and Z axis, respectively, in radians per second. The illustration shows the direction for positive results; rotating in the other direction would return a negative value.

The fourth element is a time stamp in seconds. It is consistent with the other sensor time stamps.

Using this method starts the gyroscope if it is not already sampling. The default internal sample rate is 0.01 seconds per sample. Your program can adjust this rate using the `setGyroRate` function. Keep in mind that the sampling occurs on a completely separate thread. Your program can impact the rate samples are taken by using too much of the available computing power. It is also possible for your program to miss samples if too many calculations are done between samples. A sample rate of 0.01 seconds is generally about as fast as iOS can support as this is written; processing the data as it is collected makes it unlikely all points will be collected, but then, samples every 0.01 seconds may not be needed.

The gyroscope draws power, and draws more power for faster sampling rates. In addition to adjusting the sample rate to the lowest rate needed, your program should stop the gyroscope with `stopGyro` as soon as data collection completes. The gyroscope is automatically stopped when your program ends.



Rotation can be sampled two ways. This method is a little slower, resulting in data that generally has less consistency in the time between samples, but allows the program to react to the data as it is sampled. See the `sample` method for a way to sample the data in blocks.

The snippet shows a simple program that displays the current rotation rate on a continuously updating plot. This program is also a sample in the techBASIC app; the sample is called Gyroscope.

#### Snippet

```
! Shows a running plot of the gyroscope for the last 10
! seconds in 0.1 second intervals.
!
! Initialize the display with rotation set to 0 along all axis.
DIM rx(100, 2), ry(100, 2), rz(100, 2)
FOR t = 1 TO 100
    rx(t, 1) = (t - 100)/10.0
    ry(t, 1) = (t - 100)/10.0
    rz(t, 1) = (t - 100)/10.0
NEXT

! Initialize the plot and show it.
DIM p as Plot, px as PlotPoint, py as PlotPoint, pz as PlotPoint
p = Graphics.newPlot
p.setTitle("Rotation in Radians/Second")
p.setXAxisLabel("Time in Seconds")
p.setYAxisLabel("Rotation: X: Green, Y: Red, Z: Blue")
p.showGrid(1)
p.setGridColor(0.8, 0.8, 0.8)

px = p.newPlot(rx)
px.setColor(0, 1, 0)
px.setPointColor(0, 1, 0)

py = p.newPlot(ry)
py.setColor(1, 0, 0)
py.setPointColor(1, 0, 0)

pz = p.newPlot(rz)
pz.setColor(0, 0, 1)
pz.setPointColor(0, 0, 1)

! Set the plot range and domain. This must be done
! after adding the first PlotPoint, since that also
! sets the range and domain.
p.setView(-10, -10, 0, 10, 0)

system.showGraphics

! Loop continuously, collecting gyroscope data
! and updating the plot.
sensors.setGyroRate(0.1)
t0 = -1
WHILE (1)
    r = sensors.gyro
    IF r(4) > t0 THEN
```

```

FOR i = 1 TO 99
    rx(i, 2) = rx(i + 1, 2)
    ry(i, 2) = ry(i + 1, 2)
    rz(i, 2) = rz(i + 1, 2)
NEXT
rx(100, 2) = r(1)
ry(100, 2) = r(2)
rz(100, 2) = r(3)
px.setPoints(rx)
py.setPoints(ry)
pz.setPoints(rz)
Graphics.repaint
t0 = a(4)
END IF
WEND

```

---

**FUNCTION gyroAvailable AS INTEGER**

Returns zero if no gyroscope is available, and a non-zero value if a gyroscope is available.

---

**FUNCTION heading () (7) AS DOUBLE**

Returns the heading.

The heading is measured in degrees, as with a compass. Zero degrees is due north, 90 degrees is east, 180 degrees is south, and 270 degrees is west. The heading is measured from along the Y axis, which is up when looking at the device in portrait mode.

The returned array has seven values. The first is the true heading, as determined by the device using all available information. The second element of the array is the magnetic heading, which varies from the true heading depending on your location. The third element is the accuracy of the reading. If the accuracy is -1, the device needs to be recalibrated. An easy way to recalibrate the device is to use the direction feature in the Maps application, or on the iPhone, to use the compass. The fourth element of the array is the time stamp, which is consistent with the time stamps returned by the other sensor calls.

The remaining three elements of the array are the geomagnetic data for the X, Y and Z axes, respectively. These are given in microteslas. These values represent the deviation from the magnetic field lines being tracked by the device along each axis.

Finding the heading takes power, and once you find a heading, the device continues to update the information internally. Once all needed heading information is collected, use `stopHeading` to turn off the internal data collection.

The snippet shows a program that prints the current heading.

Snippet

```

heading = sensors.heading
PRINT USING "True heading      : #"; heading(1)
PRINT USING "Magnetic Heading: #"; heading(2)
PRINT USING "Error            : #"; heading(3)
PRINT      "Time stamp       : "; heading(4)
PRINT USING "X,Y,Z Deviation : #,#,#"; heading(5), heading(6), heading(7)

```

---

**FUNCTION headingAvailable AS INTEGER**

Returns zero if the heading is not available, and a non-zero value if the heading is available.



**FUNCTION location (wait) (8) AS DOUBLE**

Returns the current location.

Collecting the location can take several seconds, and to conserve power, the iOS device may not update it right away. The `location` method waits for the iOS device to update the location before returning. You can place an upper limit on the time the method will wait using the **wait** parameter, which specifies the wait time in seconds.

The first two elements of the array returned are the latitude and longitude. Next is the altitude in meters. The fourth and fifth elements of the array are the horizontal position error and vertical position error in meters. The sixth and seventh elements of the array are the speed in meters per second, and the heading as a compass heading, as with the `heading` method. These values will be -1 if the speed and heading cannot be determined. The last element of the array is the time stamp, which is consistent with the time stamps returned by the other sensor calls.

Finding the location takes power, and once you find a location, the device continues to update the information internally. Once all needed location information is collected, use `stopLocation` to turn off the internal data collection.

The snippet shows a program that prints the current location information.

Snippet

```
location = sensors.location(30)
PRINT USING "Latitude      : #####.###"; location(1)
PRINT USING "Longitude    : #####.###"; location(2)
PRINT USING "Altitude     : #####.###"; location(3)
PRINT USING "Horiz. Error: #####.###"; location(4)
PRINT USING "Vert. Error : #####.###"; location(5)
PRINT USING "Speed       : #####.###"; location(6)
PRINT USING "Direction   : #####"; location(7)
PRINT      "Time stamp   : "; location(8)
```

**FUNCTION locationAvailable AS INTEGER**

Returns zero if the location is not available, and a non-zero value if the location is available.

**FUNCTION mag () (4) AS DOUBLE**

Returns the most recently sampled magnetometer reading. The vector returned has four elements. The first three are the magnetic field strength in the X, Y and Z axis, respectively, measured in micro Teslas. The orientation of the axis relative to the iPhone is shown in the illustration; the iPad is similar.

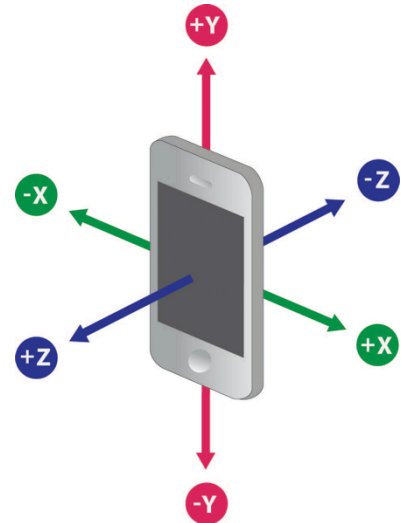
The fourth element is a time stamp in seconds. It is consistent with the other sensor time stamps.

Using this method starts the magnetometer if it is not already sampling. The default internal sample rate is 0.01 seconds per sample. Your program can adjust this rate using the `setMagRate` function. Keep in mind that the sampling occurs on a completely separate thread. Your program can impact the rate samples are taken by using too much of the available computing power. It is also possible for your program to miss samples if too many calculations are done between samples. A sample rate of 0.01 seconds is generally about as fast as iOS can support as this is written; processing the data as it is collected makes it unlikely all points will be collected, but then, samples every 0.01 seconds may not be needed.

The magnetometer draws power, and draws more power for faster sampling rates. In addition to adjusting the sample rate to the lowest rate needed, your program should stop the magnetometer with `stopMag` as soon as data collection completes. The magnetometer is automatically stopped when your program ends.

Magnetic field strength can be sampled two ways. This method is a little slower, resulting in data that generally has less consistency in the time between samples, but allows the program to react to the data as it is sampled. See the `sample` method for a way to sample the data in blocks.

The snippet shows a simple program that displays the current magnetic field strength on a continuously updating plot. This program is also a sample in the techBASIC app; the sample is called Magnetometer.



## Chapter 15: Sensor and Communication Classes

### Snippet

```
! Shows a running plot of the magnetic field for the last 10
! seconds in 0.1 second intervals.
!
! Initialize the display with field set to 0 along all axis.
DIM mx(100, 2), my(100, 2), mz(100, 2)
FOR t = 1 TO 100
    mx(t, 1) = (t - 100)/10.0
    my(t, 1) = (t - 100)/10.0
    mz(t, 1) = (t - 100)/10.0
NEXT

! Initialize the plot and show it.
DIM p as Plot, px as PlotPoint, py as PlotPoint, pz as PlotPoint
p = Graphics.newPlot
p.setTitle("Magnetic Field in Micro Teslas")
p.setXAxisLabel("Time in Seconds")
p.setYAxisLabel("Field: X: Green, Y: Red, Z: Blue")
p.showGrid(1)
p.setGridColor(0.8, 0.8, 0.8)

px = p.newPlot(mx)
px.setColor(0, 1, 0)
px.setPointColor(0, 1, 0)

py = p.newPlot(my)
py.setColor(1, 0, 0)
py.setPointColor(1, 0, 0)

pz = p.newPlot(mz)
pz.setColor(0, 0, 1)
pz.setPointColor(0, 0, 1)

! Set the plot range and domain. This must be done
! after adding the first PlotPoint, since that also
! sets the range and domain.
p.setView(-10, -10, 0, 10, 0)

system.showGraphics

! Loop continuously, collecting accelerometer data
! and updating the plot.
sensors.setMagRate(0.1)
t0 = -1
index = 1
WHILE (1)
    ! Get a new reading, then make sure it is a change since the
    ! last one.
    m = sensors.mag
    IF m(4) > t0 THEN
        ! Move all previous points one sample back.
        FOR i = 1 TO 99
            mx(i, 2) = mx(i + 1, 2)
            my(i, 2) = my(i + 1, 2)
            mz(i, 2) = mz(i + 1, 2)
        NEXT
        ! Add the new point
        mx(index, 2) = m(1)
        my(index, 2) = m(2)
        mz(index, 2) = m(3)
        index = index + 1
        t0 = t0 + 0.1
    END IF
END WHILE
```

```

! Place the new point at the right of the plot.
mx(100, 2) = m(1)
my(100, 2) = m(2)
mz(100, 2) = m(3)

! Update the plots.
px.setPoints(mx)
py.setPoints(my)
pz.setPoints(mz)

! Adjust the function range based on the maximum observed value.
max = 0
FOR i = 1 TO 100
    IF ABS(mx(i, 2)) > max THEN max = ABS(mx(i, 2))
    IF ABS(my(i, 2)) > max THEN max = ABS(my(i, 2))
    IF ABS(mz(i, 2)) > max THEN max = ABS(mz(i, 2))
NEXT
range = 10^(INT(LOG(max)/LOG(10)) + 1)
p.setView(-10, -range, 0, range, 0)

! Repaint the plots.
Graphics.repaint

! Reset the time to the new time.
t0 = m(4)
END IF
WEND

```

---

**FUNCTION magAvailable AS INTEGER**

Returns zero if no magnetometer is available, and a non-zero value if a magnetometer is available.

---

**FUNCTION sample (sensors AS INTEGER, rate, samples AS INTEGER) (n, 12) AS DOUBLE**

The `sample` method is used to sample any combination of the accelerometer, magnetometer and gyroscope. Unlike `accel`, `mag` and `gyro`, which sample the current values and return right away, `sample` collects the data at machine speeds and returns an array of the completed data. This allows the data to be collected without the overhead of a running program, generally resulting in faster collection rates and less likelihood of missed samples. The penalty paid for collecting data this way is that the program freezes until all of the data is collected, then returns it in a single array.

**sensors** is an integer mask that specifies which sensors to collect data from. It is a bitmap of 1 for the accelerometer, 2 for the magnetometer and 4 for the gyroscope. The possible values and which sensors are started is shown in the table below.

<b>sensors</b>	Sensors started
1	Accelerometer
2	Magnetometer
3	Accelerometer, Magnetometer
4	Gyroscope
5	Accelerometer, Gyroscope
6	Magnetometer, Gyroscope
7	Accelerometer, Magnetometer, Gyroscope

**rate** is the sample rate in seconds between samples. A sample rate of 0.01 (100 samples per second) is about as much as current iOS devices can handle. Faster sampling rates use more power than slower rates.

**samples** is the number of samples to collect. Sample collection will last for approximately **rate\*samples** seconds, and the samples returned will use about **samples\*96** bytes of memory. The sensors are stopped when

sample collection is complete, so there is no need to call `stopAccel`, `stopGyro` or `stopMag`, although making the calls will do no harm.

Each row of the returned matrix contains one sample point each for the accelerometer, gyroscope and magnetometer, in that order. If one or more of the instruments was not sampled, or if the sensor is not available on the device, the matrix elements for that sensor are set to 0. For each sensor, four data elements give the sensor reading about the X, Y and Z axis and a time stamp; refer to the illustration to see how the axis align with the device. Acceleration is returned as a G force, where one G is 9.81 meters/second<sup>2</sup>. The gyroscope measures rotation about the axis in radians per second, while the magnetometer returns the magnetic field strength in micro Teslas. For each sensor, the fourth element is the time stamp when the sensor reading was made. The sensors are independent, so the time tamps may not match exactly across a row. In fact, especially for faster sampling rates, they can vary by as much as the sampling rate itself.

The snippet illustrates these ideas with a program that collects data for 5 seconds, then shows the average and maximum values for each sensor over the sample period.

### Snippet

```
! Collect sensor samples for 5 seconds.
samples = 250
samp = Sensors.sample(7, 0.02, samples)

! Find the average and maximum values for each
! sensor.
FOR i = 1 TO samples
    ax = ax + samp(i, 1)
    ay = ay + samp(i, 2)
    az = az + samp(i, 3)

    IF ABS(max) < ABS(samp(i, 1)) THEN max = samp(i, 1)
    IF ABS(may) < ABS(samp(i, 2)) THEN may = samp(i, 2)
    IF ABS(maz) < ABS(samp(i, 3)) THEN maz = samp(i, 3)

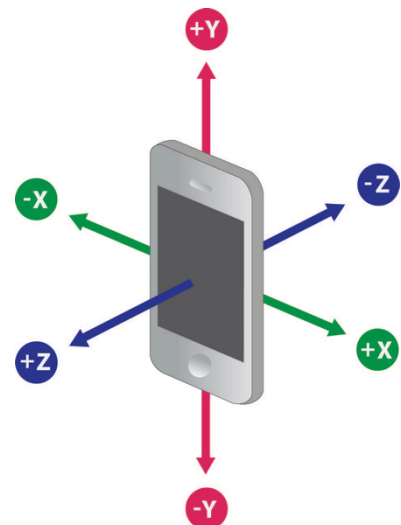
    gx = gx + samp(i, 5)
    gy = gy + samp(i, 6)
    gz = gz + samp(i, 7)

    IF ABS(mgx) < ABS(samp(i, 5)) THEN mgx = samp(i, 5)
    IF ABS(mgy) < ABS(samp(i, 6)) THEN mgy = samp(i, 6)
    IF ABS(mgz) < ABS(samp(i, 7)) THEN mgz = samp(i, 7)

    mx = mx + samp(i, 9)
    my = my + samp(i, 10)
    mz = mz + samp(i, 11)

    IF ABS(mmx) < ABS(samp(i, 9)) THEN mmx =
samp(i, 9)
    IF ABS(mmy) < ABS(samp(i, 10)) THEN mmy =
samp(i, 10)
    IF ABS(mmz) < ABS(samp(i, 11)) THEN mmz =
samp(i, 11)
NEXT

PRINT USING "Max acceleration: ##.##, ##.##,
##.##"; max, may, maz
PRINT USING "Max rotation: ###.##, ###.##,
###.##"; mgx, mgy, mgz
```





```

PRINT USING "Max magnetic field: ###.##, ###.##, ###.##"; mmx, mmy, mmz

PRINT USING "Average acceleration: ##.##, ##.##, ##.##"; ax/samples,
ay/samples, az/samples
PRINT USING "Average rotation: ###.##, ###.##, ###.##"; gx/samples,
gy/samples, gz/samples
PRINT USING "Average magnetic field: ###.##, ###.##, ###.##"; mx/samples,
my/samples, mz/samples

```

---

#### **SUB setAccelRate (rate)**

Sets the sampling rate used for the accelerometer.

The acceleration rate is approximate; the actual acceleration returned by the accelerometer may differ somewhat from the requested rate, and can also vary over time. Samples tend to be returned more often when the acceleration is changing rapidly. For situations where the exact time between samples is important, be sure to read the actual time from the returned sample.

See the `accel` method for a description of the accelerometer and an example that uses this method.

---

#### **SUB setGyroRate (rate)**

Sets the sampling rate used for the gyroscope.

The sample rate is approximate; the actual gyroscopic measurements may differ somewhat from the requested rate, and can also vary over time. For situations where the exact time between samples is important, be sure to read the actual time from the returned sample.

See the `gyro` method for a description of the gyroscope and an example that uses this method.

---

#### **SUB setMagRate (rate)**

Sets the sampling rate used for the magnetometer.

The sample rate is approximate; the actual magnetometer measurements may differ somewhat from the requested rate, and can also vary over time. For situations where the exact time between samples is important, be sure to read the actual time from the returned sample.

See the `mag` method for a description of the magnetometer and an example that uses this method.

---

#### **SUB startAltimeter (SUB Altimeter)**

Starts the altimeter service, which will call the passed subroutine with altimeter updates.

The altimeter uses a barometer to determine the change in altitude since data collection starts. It can also report the air pressure in kilopascals (kPa).

Updates occur as new pressure information is made available by the operating system. iOS does not provide any mechanism for setting the update rate. While this may vary by device and the version of the operating system, updates in iOS 8 running on an iPhone 6 occur roughly every 1.3 seconds.

When new information is available, the subroutine you pass is called. This subroutine must have three parameters. While you can use any numeric parameters, for best results, the first parameter should be a `DOUBLE` value, while the last two should be `SINGLE` values.

The first value is the time when the sample was collected, given as the elapsed time in seconds since January 1, 2001 at 00:00:00 GMT.

The second value is the altitude in meters relative to the first reported altitude. The first reported altitude is always 0. Subsequent readings will be positive if the device is above the original altitude and negative if the device is below the original altitude.

The last value passed is the barometric pressure in kilopascals. One kilopascal is one Newton per square meter. In terms of atmospheric pressure, the standard atmosphere at sea level is defined as 101.325 kPa.

Use `stopAltimeter` to stop altimeter updates. Stop and restart the altimeter to reset the first reported altitude.

The snippet shows a very simple program that prints the relative altitude and pressure for 15 seconds, then exits.

### Snippet

```
! Print altimeter information for 15 seconds.
IF Sensors.altimeterAvailable THEN
  Sensors.startAltimeter(SUB Altimeter)
  DIM t AS DOUBLE
  t = System.ticks
  WHILE System.ticks - t < 15.0
    WEND
  Sensors.stopAltimeter
ELSE
  PRINT "This device does not have a barometer."
END IF

SUB Altimeter (time AS DOUBLE, altitude, pressure)
PRINT time, altitude, pressure
END SUB
```

---

### **SUB stopAccel**

Stops sampling for the accelerometer.  
See `accel` for a description of the accelerometer and tips on when to use this method.

---

### **SUB stopAltimeter**

Stops sampling for the altimeter.  
See `startAltimeter` for a description of the altimeter and tips on when to use this method.

---

### **SUB stopGyro**

Stops sampling for the gyroscope.  
See `gyro` for a description of the gyroscope and tips on when to use this method.

---

### **SUB stopHeading**

Stops sampling for heading.  
See `heading` for a description of the heading reading and tips on when to use this method.

---

### **SUB stopLocation**

Stops sampling for location.  
See `location` for a description of the location reading and tips on when to use this method.

---

### **SUB stopMag**

Stops sampling for the magnetometer.  
See `mag` for a description of the magnetometer and tips on when to use this method.

## Chapter 16 – Graphics Classes

Graphics classes are used to draw images directly on the graphics view or to create and manipulate plots of functions and data.

### Callout

Callouts are used on plots to show the location of a point or short text messages at a location. The `newCallout` method of the `Plot` class creates textual callouts and adds them to a plot. That method returns a `Callout` object; this section describes the methods in the `Callout` class that can be used to manipulate the `Callout` object returned by `newCallout`.

The snippet illustrates the methods in this class. The plot created by the snippet is shown to the right.

Snippet

```
! Show the graphics screen.
system.showGraphics
```

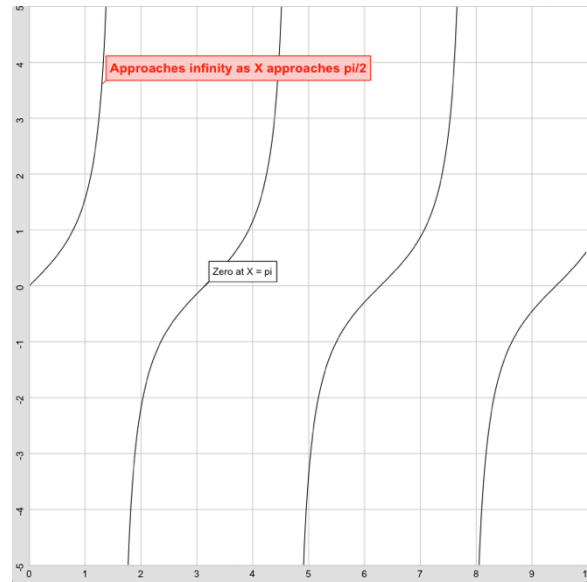
```
! Create the plot.
DIM p AS Plot
p = graphics.newPlot
p.showGrid(1)
p.setGridColor(0.8, 0.8, 0.8)
```

```
! Plot the function.
DIM m AS PlotFunction
m = p.newFunction(FUNCTION f)
```

```
! Add callouts for the first zero and infinity.
DIM c AS CALLOUT
c = p.newCallout("Approaches infinity as X approaches pi/2", 1.3,
TAN(1.3))
c.setBackgroundColor(1, 0.8, 0.8)
c.setFontColor(1, 0, 0)
c.setFont("Sans-serif", 16, 1)
```

```
c = p.newCallout("Zero at X = pi", 3.14159, 0)
END
```

```
FUNCTION f(x)
f = TAN(x)
END FUNCTION
```



#### **SUB setBackgroundColor (red, green, blue)**

Sets the background color for the callout. See the snippet in the class description, above, for an example. See `setColor` in the `Graphics` class for a discussion of RGB colors.

**SUB setFontColor (red, green, blue)**

Sets the font and box outline color for the callout. See the snippet in the class description, above, for an example. See `setColor` in the `Graphics` class for a discussion of RGB colors.

**SUB setFont (name AS STRING, size, style AS INTEGER)**

Sets the font for the callout. See the snippet in the class description, above, for an example. See `setFont` in the `Graphics` class for a discussion of fonts.

---

## Graphics

The `Graphics` class is used when drawing to the graphics view. A global variable by the same name is predefined in every BASIC program.

There are two fundamentally different ways to draw graphics. Most of the methods in this class are graphics primitives that can be used in a program to draw specific shapes, such as lines or ovals, to the graphics view. The other way to draw is create a plot of a function, or a graph of a set of data points. The `Graphics` class is still the starting point for these types of drawings, but the details and most of the methods used to manipulate plots and graphs are described with the `Plot` class, later in this chapter.

The drawing area is composed of pixels, using a coordinate system that places the origin at the top left of the screen, with positive X to the right and positive Y down. The number of pixels available on the drawing screen varies depending on the orientation of the device.

techBASIC supports both vector graphics and pixel graphics for the basic drawing commands. Pixel graphics draws to a paint layer with a one-to-one correspondence between the pixels you see and the pixels drawn by the drawing command. When the image is scaled, either through the `setScale` command or when exported using the Share button, the image can become pixelated. Vector graphics results in no loss of resolution due to pixilation when zoomed or shared, but drawing repeatedly to the screen gradually fills up memory and slows down refreshing of the graphics screen. Both methods have their uses, so both are available, and they can even be mixed. Drawing with the vector graphics commands draws on top of the pixel graphics layer. The default is pixel graphics. Use the `setPixelGraphics` method to switch between the two drawing modes, and `isPixelGraphics` to determine which drawing mode it is in use. The snippet shows a few of the capabilities of the graphics primitive commands.

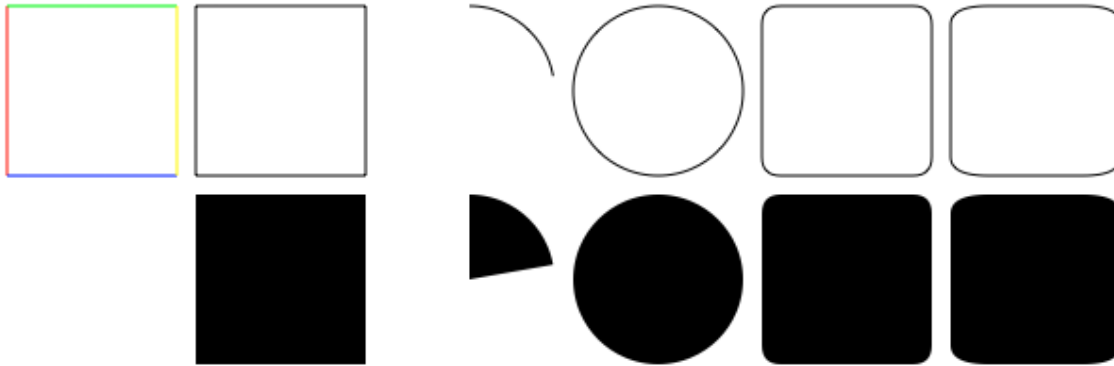
Snippet

```
Graphics.setColor(1, 0, 0)
Graphics.drawLine(10, 10, 10, 100)
Graphics.setColor(0, 1, 0)
Graphics.drawLine(10, 10, 100, 10)
Graphics.setColor(0, 0, 1)
Graphics.drawLine(100, 100, 10, 100)
Graphics.setColor(1, 1, 0)
Graphics.drawLine(100, 100, 100, 10)

Graphics.setColor(0, 0, 0)
Graphics.drawRect(110, 10, 90, 90)
Graphics.drawArc(210, 10, 90, 90, 10, 80)
Graphics.drawOval(310, 10, 90, 90)
Graphics.drawRoundRect(410, 10, 90, 90, 20, 20)
Graphics.drawRoundRect(510, 10, 90, 90, 40, 20)
Graphics.fillRect(110, 110, 90, 90)
Graphics.fillArc(210, 110, 90, 90, 10, 80)
Graphics.fillOval(310, 110, 90, 90)
Graphics.fillRoundRect(410, 110, 90, 90, 20, 20)
Graphics.fillRoundRect(510, 110, 90, 90, 40, 20)
```

```
s$ = "Hello, world."
Graphics.setFont("baskerville", 20, 3)
Graphics.drawText(10, 240, 0, s$)
```

The snippet will draw this test pattern on the graphics screen. Scroll the screen sideways or use a pinch gesture on the iPhone to see the entire drawing.



***Hello, world.***

---

#### **FUNCTION ascender**

Returns the font ascender for the current font in pixels. The ascender is the distance from the baseline of the font to the top of the highest character in the font.

##### Snippet

```
Graphics.drawText(10, 10 + Graphics.ascender, 0, "Hello, world.")
```

---

#### **FUNCTION descender**

Returns the font descender for the current font in pixels. The descender is the distance from the baseline of the font to the bottom of the lowest character in the font. The value is negative.

##### Snippet

```
Graphics.drawText(10, 10 - Graphics.descender, 0, "Hello, world.")
```

---

#### **SUB drawArc (x, y, width, height, startAngle, arcAngle)**

Draws the outline of an arc formed by drawing part of the oval that would fill the given rectangle. The color of the arc is set by the most recent call to `setColor`.

The **x**, **y**, **width** and **height** parameters define the size and location of the rectangle.

Angles are measured in degrees, starting from the 3 o'clock position. Positive angles rotate counterclockwise. **startAngle** is the angle where the arc will begin drawing, while **arcAngle** is the size of the arc.

Arcs can be drawn using pixel graphics or vector graphics. See `setPixelGraphics` for a method to switch between the drawing modes.

##### Snippet

```
! Draw an arc filling the top-left quarter of a rectangle.
Graphics.drawArc(100, 100, 100, 50, 90, 90)
```

---

#### **SUB drawImage (theImage AS Image, x, y [, width = 0 [, height = 0]])**

Draws an image to the pixel layer of the graphics view.

**theImage** is the image to draw. It is drawn at the location given by **x** and **y**. If the width and height are not given, or if they are less than 1, the image is drawn at its normal size. If the width and height are specified, the image is scaled to fit in the specified area.

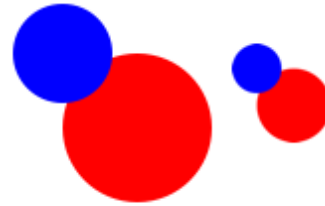
### Snippet

```
System.showGraphics
width = 100
height = 100

DIM img AS Image
img = System.newImage
img.newImage(width, height)

img.setColor(1, 0, 0)
img.fillOval(width/4, height/4, 3*width/4, 3*height/4)
img.setColor(0, 0, 1)
img.fillOval(0, 0, width/2, height/2)

Graphics.drawImage(img, 10, 10)
Graphics.drawImage(img, width + 20, 30, width/2, height/2)
```




---

### SUB drawLine (x1, y1, x2, y2)

Draws a line connecting two points. The color of the line is set by the most recent call to `setColor`.

**x1** and **y1** specify the first point, while **x2** and **y2** specify the second.

Lines can be drawn using pixel graphics or vector graphics. See `setPixelGraphics` for a method to switch between the drawing modes.

### Snippet

```
Graphics.setColor(255, 0, 0)
Graphics.drawLine(100, 100, 150, 150)
Graphics.drawLine(150, 150, 100, 200)
Graphics.drawLine(100, 200, 50, 150)
Graphics.drawLine(50, 150, 100, 100)
```

---

### SUB drawOval (x, y, width, height)

Draws the outline of an oval that fills the given rectangle. The color of the oval is set by the most recent call to `setColor`.

The **x**, **y**, **width** and **height** parameters define the size and location of the rectangle.

Ovals can be drawn using pixel graphics or vector graphics. See `setPixelGraphics` for a method to switch between the drawing modes.

### Snippet

```
! Draw 100 ovals with sizes ranging from 10 to 100 pixels
! using random colors.
FOR i = 1 TO 100
    r = RND(1.0)
    g = RND(1.0)
    b = RND(1.0)
    Graphics.setColor(r, g, b)

    w = 10 + RND(1.0)*90
    h = 10 + RND(1.0)*90
    x = RND(1.0)*(400 - w%)
    y = RND(1.0)*(400 - h%)
    Graphics.drawOval(x, y, w, h)
NEXT
```

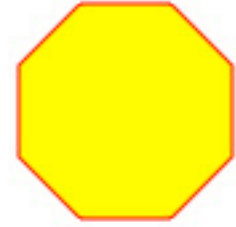
---

### SUB drawPoly (points(,))

Draws the outline of a polygon with the current color.

The polygon is formed from the points in the passed array. The first subscript of the array indexes over points, while the second indexes over X and Y. The number of elements in the second subscript must be two. The shape is outlined by moving from one point to the next, automatically closing the shape by returning to the first point if necessary.

The snippet shows an octagon, both outlined with `drawPoly` and filled with `fillPoly`.



#### Snippet

```
DIM poly(8, 2)
poly = [[10, 33],
        [10, 67],
        [33, 90],
        [67, 90],
        [90, 67],
        [90, 33],
        [67, 10],
        [33, 10]]
Graphics.setColor(1, 1, 0)
Graphics.fillPoly(poly)
Graphics.setColor(1, 0, 0)
Graphics.drawPoly(poly)

System.showGraphics
```

---

#### SUB drawRect (x, y, width, height)

Draws the outline of a rectangle. The color of the rectangle is set by the most recent call to `setColor`.

The **x**, **y**, **width** and **height** parameters define the size and location of the rectangle.

Rectangles can be drawn using pixel graphics or vector graphics. See `setPixelGraphics` for a method to switch between the drawing modes.

#### Snippet

```
Graphics.drawRect(10, 10, 100, 100)
```

---

#### SUB drawRoundRect (x, y, width, height, arcWidth, arcHeight)

Draws the outline of a rectangle with rounded corners. The color of the rounded rectangle is set by the most recent call to `setColor`.

The **x**, **y**, **width** and **height** parameters define the size and location of the rectangle.

The **arcWidth** and **arcHeight** parameters define the size of the rounded corners. Imagine an oval whose size is given by **arcWidth** and **arcHeight**, sliced into four parts to form four arcs. These arcs are then used as the rounded corners of the rectangle.

Rounded rectangles can be drawn using pixel graphics or vector graphics. See `setPixelGraphics` for a method to switch between the drawing modes.

#### Snippet

```
Graphics.drawRoundRect(10, 10, 100, 100, 30, 30)
```

---

#### SUB drawText (x, y, angle, str AS STRING)

Draws text. The color of the text is set by the most recent call to `setColor`, while the font is set by `setFont`.

The **x** and **y** parameters define the location of the text. The vertical position is to the baseline of the text, not to the top or bottom of a rectangle containing the text. Think of it as the location of the line on lined paper when writing text by hand. **angle** is the angle, in radians, of the text from the horizontal; positive angles rotate the text counter-clockwise. **str** is the text to draw.

Text can be drawn using pixel graphics or vector graphics. See `setPixelGraphics` for a method to switch between the drawing modes.

Snippet

```
Graphics.drawText(10, 20, 0, "Hello, world.")
```

**SUB fillArc (x, y, width, height, startAngle, arcAngle)**

Draws a solid arc formed by drawing part of the oval that would fill the given rectangle. The color of the arc is set by the most recent call to setColor.

The **x**, **y**, **width** and **height** parameters define the size and location of the rectangle.

Angles are measured in degrees, starting from the 3 o'clock position. Positive angles rotate counterclockwise. **startAngle** is the angle where the arc will begin drawing, while **arcAngle** is the size of the arc.

Arcs can be drawn using pixel graphics or vector graphics. See setPixelGraphics for a method to switch between the drawing modes.

The snippet shows the fillArc command in use, drawing the pie chart shown to the right.

Snippet

```
slices = [10, 10, 30, 40, 80]
drawPie(10, 10, 200, 200, slices())
END

SUB drawPie (x, y, width, height, slices())
! Find the size for all slices.
sum = 0.0
FOR i = 1 to UBOUND(slices, 1)
    sum = sum + slices(i)
NEXT

! Draw the slices.
start = 0
FOR i = 1 TO UBOUND(slices, 1)
    ! Set the color.
    IF r = 1 THEN
        r = 0
        IF b = 1 THEN
            b = 0
            g = 1
        ELSE
            b = 1
        END IF
    ELSE
        r = 1
    END IF
    Graphics.setColor(r, g, b)

    ! Draw one slice.
    IF i = UBOUND(slices, 1) THEN
        sliceSize = 360 - start
    ELSE
        sliceSize = 360*slices(i)/sum
    END IF
    Graphics.fillArc(x, y, width, height, start, sliceSize)
    start = start + sliceSize
NEXT
END SUB
```



**SUB fillOval (x, y, width, height)**

Draws a solid oval that fits in the given rectangle. The color of the oval is set by the most recent call to `setColor`.

The **x**, **y**, **width** and **height** parameters define the size and location of the rectangle.

Ovals can be drawn using pixel graphics or vector graphics. See `setPixelGraphics` for a method to switch between the drawing modes.

Snippet

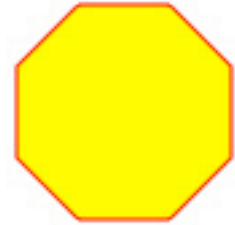
```
Graphics.fillOval(10, 10, 100, 100)
```

**SUB fillPoly (points(,))**

Fills a polygon with the current color.

The polygon is formed from the points in the passed array. The first subscript of the array indexes over points, while the second indexes over X and Y. The number of elements in the second subscript must be two. The shape is outlined by moving from one point to the next, automatically closing the shape by returning to the first point if necessary. The interior of the shape is then painted with the current color.

The snippet shows an octagon, both outlined with `drawPoly` and filled with `fillPoly`.

Snippet

```
DIM poly(8, 2)
poly = [[10, 33],
        [10, 67],
        [33, 90],
        [67, 90],
        [90, 67],
        [90, 33],
        [67, 10],
        [33, 10]]
Graphics.setColor(1, 1, 0)
Graphics.fillPoly(poly)
Graphics.setColor(1, 0, 0)
Graphics.drawPoly(poly)

System.showGraphics
```

**SUB fillRect (x, y, width, height)**

Draws a solid rectangle. The color of the rectangle is set by the most recent call to `setColor`.

The **x**, **y**, **width** and **height** parameters define the size and location of the rectangle.

Rectangles can be drawn using pixel graphics or vector graphics. See `setPixelGraphics` for a method to switch between the drawing modes.

Snippet

```
! Draw 100 rectangle with sizes ranging from 10 to 100 pixels
! using random colors.
FOR i = 1 TO 100
  r = RND(1.0)
  g = RND(1.0)
  b = RND(1.0)
  Graphics.setColor(r, g, b)
```

```

w = 10 + RND(1.0)*90
h = 10 + RND(1.0)*90
x = RND(1.0)*(400 - w)
y = RND(1.0)*(400 - h)
Graphics.fillRect(x, y, w, h)
NEXT

```

---

**SUB fillRoundRect (x, y, width, height, arcWidth, arcHeight)**

Draws a solid rectangle with rounded corners. The color of the rounded rectangle is set by the most recent call to `setColor`.

The **x**, **y**, **width** and **height** parameters define the size and location of the rectangle.

The **arcWidth** and **arcHeight** parameters define the size and location of the rounded corners. Imagine an oval whose size is given by **arcWidth** and **arcHeight**, sliced into four parts to form four arcs. These arcs are then used as the rounded corners of the rectangle.

Rounded rectangles can be drawn using pixel graphics or vector graphics. See `setPixelGraphics` for a method to switch between the drawing modes.

Snippet

```
Graphics.fillRoundRect(10, 10, 100, 100, 30, 30)
```

---

**FUNCTION font (index AS INTEGER) AS STRING**

Returns the name of one of the available fonts. **index** is the index of the font, counting from 1. Use `fontCount` to get the number of available fonts.

There are always at least three fonts, and they are predefined in techBASIC. These fonts are

Font Number	Font Name	iOS Font Used
1	Sans-serif	Arial
2	Serif	TimesNewRomanPSMT
3	Monospaced	Courier

These are not normal system fonts, but predefined fonts that give a reasonable font on all systems where techBASIC is implemented. Serif is a font with decorative elements in the letters, like the Times font used for the text in this manual. Sans-serif is a font with no decorative elements, like **Geneva**. Monospaced is a font where all of the characters have the same width, like `Courier New`, used for code samples in this manual.

The snippet shows a short program that lists the available fonts.

Snippet

```

FOR i = 1 TO Graphics.fontCount
  PRINT Graphics.font(i)
NEXT

```

---

**FUNCTION fontCount AS INTEGER**

Returns the number of available font names. See `font` for a full discussion and an example.

---

**FUNCTION fontName AS STRING**

Returns the name of the current font. See `setFont` for a discussion and example.

---

**FUNCTION fontSize AS INTEGER**

Returns the size of the current font. See `setFont` for a discussion and example.

---

**FUNCTION fontStyle AS INTEGER**

Returns the style of the current font. See `setFont` for a discussion and example.

---

**FUNCTION height**

Returns the current height of the visible portion of the graphics view. This is the height, in pixels, of what the user of the program can see. The height of the area available for drawing is not fixed; using the various swipe and pinch gestures described in Chapters 2 and 3 the person using the program can expand, shrink or pan the drawing area to see more or less of the image.

---

**FUNCTION isAntialiased AS INTEGER**

Returns 0 if drawing will be done without antialiasing, and 1 if drawing will be done using antialiasing. See `setAntialiased` for a description of antialiasing and a code sample.

---

**FUNCTION isPixelGraphics AS INTEGER**

Returns 1 if drawing will be done using pixel graphics, and 0 if it will be done using vector graphics. See `setPixelGraphics` for a comparison of pixel and vector graphics and a code sample.

---

**FUNCTION lineHeight AS INTEGER**

Returns the height of a line of text drawn with `drawText` using the current font set by `setFont`. See `setFont` for an example.

This height is not the same thing as the ascender – descender; in other words, it is not the height just of the visible text. This is the recommended total line height, including the leading that separates the lines of text. This is the value to use to decide how far apart to place lines of text when drawing multiple lines on the graphics view.

---

**FUNCTION newActivity (x, y, width = 20, height = 20) AS Activity**

Creates and returns a new Activity object. An Activity object is a control that indicates activity of an unpredictable duration. See the description of the Activity class for details.

The upper-left corner of the control will be placed at **x, y**. The initial width and height will be **width** and **height**.

Controls vanish as soon as the program completes execution. To see a control, be sure and add at least one application method to handle events. See Chapter 13 for a description of event handling.

---

**FUNCTION newButton (x, y, width = 72, height = 37) AS Button**

Creates and returns a new Button object. A Button object is a push button control that generally triggers some action when tapped. See the description of the Button class for details.

The upper-left corner of the control will be placed at **x, y**. The initial width and height will be **width** and **height**.

Controls vanish as soon as the program completes execution. To see a control, be sure and add at least one application method to handle events. See Chapter 13 for a description of event handling.

---

**FUNCTION newColorPicker (x, y, width = 120, height = 240) AS ColorPicker**

Creates and returns a new ColorPicker object. A ColorPicker object is a control used to select colors using touch and swipe events. See the description of the ColorPicker class for details.

The upper-left corner of the control will be placed at **x, y**. The initial width and height will be **width** and **height**. As a general rule, ColorPicker objects should be 20% wider than they are tall to allow room for the luminosity slider.

Controls vanish as soon as the program completes execution. To see a control, be sure and add at least one application method to handle events. See Chapter 13 for a description of event handling.

---

**FUNCTION newDatePicker (x, y, width = 330, height = 216) AS DatePicker**

Creates and returns a new DatePicker object. A DatePicker object is set of picker wheels used to select a date or time. See the description of the DatePicker class for details.

The upper-left corner of the control will be placed at **x, y**. The initial width and height will be **width** and **height**.

Controls vanish as soon as the program completes execution. To see a control, be sure and add at least one application method to handle events. See Chapter 13 for a description of event handling.

---

**FUNCTION `newImageView (x, y, width = 64, height = 64) AS ImageView`**

Creates and returns a new `ImageView` object. An `ImageView` object is used to display an image loaded from disk or contained in a `Image` object. See the description of the `ImageView` class for details.

The upper-left corner of the control will be placed at `x, y`. The initial width and height will be `width` and `height`.

Controls vanish as soon as the program completes execution. To see a control, be sure and add at least one application method to handle events. See Chapter 13 for a description of event handling.

---

**FUNCTION `newLabel (x, y, width = 42, height = 21) AS Label`**

Creates and returns a new `Label` object. A `Label` object is used to display text, usually as a visual queue to other elements of a user interface. See the description of the `Label` class for details.

The upper-left corner of the control will be placed at `x, y`. The initial width and height will be `width` and `height`.

Controls vanish as soon as the program completes execution. To see a control, be sure and add at least one application method to handle events. See Chapter 13 for a description of event handling.

---

**FUNCTION `newMapView (x, y, width = 120, height = 240) AS MapView`**

Creates and returns a new `MapView` object. A `MapView` object is used to display an interactive map. See the description of the `MapView` class for details.

The upper-left corner of the control will be placed at `x, y`. The initial width and height will be `width` and `height`.

Controls vanish as soon as the program completes execution. To see a control, be sure and add at least one application method to handle events. See Chapter 13 for a description of event handling.

---

**FUNCTION `newPicker (x, y, width = 330, height = 216) AS Picker`**

Creates and returns a new `Picker` object. A `Picker` object has one or more wheels that present selectable choices. See the description of the `Picker` class for details.

The upper-left corner of the control will be placed at `x, y`. The initial width and height will be `width` and `height`.

Controls vanish as soon as the program completes execution. To see a control, be sure and add at least one application method to handle events. See Chapter 13 for a description of event handling.

---

**FUNCTION `newPlot AS Plot`**

Creates and returns a new `Plot` object. See the description of the `Plot` class for details.

---

**FUNCTION `newProgress (x, y, width = 150, height = 9) AS Progress`**

Creates and returns a new `Progress` object. A `Progress` object is a control that indicates progress on an activity whose duration can be predicted. See the description of the `Progress` class for details.

The upper-left corner of the control will be placed at `x, y`. The initial width and height will be `width` and `height`.

Controls vanish as soon as the program completes execution. To see a control, be sure and add at least one application method to handle events. See Chapter 13 for a description of event handling.

---

**FUNCTION `newSegmentedControl (x, y, width = 207, height = 44) AS SegmentedControl`**

Creates and returns a new `SegmentedControl` object. A `SegmentedControl` object is a control that presents a series of connected buttons, only one of which can be selected at one time. See the description of the `SegmentedControl` class for details.

The upper-left corner of the control will be placed at **x, y**. The initial width and height will be **width** and **height**.

Controls vanish as soon as the program completes execution. To see a control, be sure and add at least one application method to handle events. See Chapter 13 for a description of event handling.

**FUNCTION newSlider (x, y, width = 118, height = 23) AS Slider**

Creates and returns a new Slider object. A Slider object is a control that can be slid to select a more or less continuous value. See the description of the `Slider` class for details.

The upper-left corner of the control will be placed at **x, y**. The initial width and height will be **width** and **height**.

Controls vanish as soon as the program completes execution. To see a control, be sure and add at least one application method to handle events. See Chapter 13 for a description of event handling.

**FUNCTION newStepper (x, y, width = 94, height = 27) AS Stepper**

Creates and returns a new Stepper object. A Stepper object is a control used to change a value. See the description of the `Stepper` class for details.

The upper-left corner of the control will be placed at **x, y**. The initial width and height will be **width** and **height**.

Controls vanish as soon as the program completes execution. To see a control, be sure and add at least one application method to handle events. See Chapter 13 for a description of event handling.

**FUNCTION newSwitch (x, y, width = 79, height = 27) AS Switch**

Creates and returns a new Switch object. A Switch object is a control used to present an on or off choice. See the description of the `Switch` class for details.

The upper-left corner of the control will be placed at **x, y**. The initial width and height will be **width** and **height**.

Controls vanish as soon as the program completes execution. To see a control, be sure and add at least one application method to handle events. See Chapter 13 for a description of event handling.

**FUNCTION newTable (x, y, width = 120, height = 240, style AS INTEGER = 1) AS Table**

Creates and returns a new Table object. A Table object is a control that displays multiple selectable cells, which can be divided into sections. See the description of the `Table` class for details.

The upper-left corner of the control will be placed at **x, y**. The initial width and height will be **width** and **height**.

Tables can be grouped, which means that the sections do not float. Pass 2 as the style for a grouped table, or 1 for an ungrouped table.

Controls vanish as soon as the program completes execution. To see a control, be sure and add at least one application method to handle events. See Chapter 13 for a description of event handling.

**FUNCTION newTextField (x, y, width = 97, height = 31) AS TextField**

Creates and returns a new TextField object. A TextField object is a control used to enter a single line of text. See the description of the `TextField` class for details.

The upper-left corner of the control will be placed at **x, y**. The initial width and height will be **width** and **height**.

Controls vanish as soon as the program completes execution. To see a control, be sure and add at least one application method to handle events. See Chapter 13 for a description of event handling.

**FUNCTION newTextView (x, y, width = 120, height = 240) AS TextView**

Creates and returns a new TextView object. A TextView object is a control used to enter or display multiple lines of text. See the description of the `TextView` class for details.

The upper-left corner of the control will be placed at **x, y**. The initial width and height will be **width** and **height**.

Controls vanish as soon as the program completes execution. To see a control, be sure and add at least one application method to handle events. See Chapter 13 for a description of event handling.

---

### **FUNCTION newWebView (x, y, width = 120, height = 240) AS WebView**

Creates and returns a new WebView object. A WebView object is a control used to display web pages or some kinds of documents. See the description of the `WebView` class for details.

The upper-left corner of the control will be placed at **x, y**. The initial width and height will be **width** and **height**.

Controls vanish as soon as the program completes execution. To see a control, be sure and add at least one application method to handle events. See Chapter 13 for a description of event handling.

---

### **SUB repaint**

Repaints the graphics screen.

This method is used when a plot has been added to the graphics screen using the `newPlot` command. Without this method, plots are drawn after program execution ends—perfectly appropriate for a program showing a mathematical function or data that will not change, since techBASIC can still evaluate the function after the program stops running to respond to touch and swipe events.

If the data changes as the program runs, however, the repaint method is needed to update the graphics screen for the new data. The snippet shows an example where acceleration is drawn as the program runs, providing a continuous output accelerometer that looks much like an oscilloscope. This snippet is also a sample called `Accelerometer`.

#### Snippet

```
! Shows a running plot of the acceleration for the last 10
! seconds in 0.1 second intervals.
!
! Initialize the display with acceleration set to 0 along all axis.
DIM ax(100, 2), ay(100, 2), az(100, 2)
FOR t = 1 TO 100
    ax(t, 1) = t/10.0
    ay(t, 1) = t/10.0
    az(t, 1) = t/10.0
NEXT

! Initialize the plot and show it.
DIM p as Plot, px as PlotPoint, py as PlotPoint, pz as PlotPoint
p = Graphics.newPlot
p.setTitle("Acceleration in Gravities")
p.setXAxisLabel("Time in Seconds")
p.setYAxisLabel("Acceleration: X: Green, Y: Red, Z: Blue")
p.showGrid(1)
p.setGridColor(0.8, 0.8, 0.8)

px = p.newPlot(ax)
px.setColor(0, 1, 0)
px.setPointColor(0, 1, 0)

py = p.newPlot(ay)
py.setColor(1, 0, 0)
py.setPointColor(1, 0, 0)
```

```

pz = p.newPlot(az)
pz.setColor(0, 0, 1)
pz.setPointColor(0, 0, 1)

! Set the plot range and domain. This must be done
! after adding the first PlotPoint, since that also
! sets the range and domain.
p.setView(0, -2, 10, 2, 0)

system.showGraphics

! Loop continuously, collecting accelerometer data
! and updating the plot.
sensors.setAccelRate(0.1)
t0 = -1
index = 1
WHILE (1)
  a = sensors.accel
  IF a(4) > t0 THEN
    ax(index, 2) = a(1)
    ay(index, 2) = a(2)
    az(index, 2) = a(3)
    px.setPoints(ax)
    py.setPoints(ay)
    pz.setPoints(az)
    index = index + 1
    IF index > 100 THEN index = 1
    Graphics.repaint
    t0 = a(4)
  END IF
WEND

```

---

### FUNCTION scale

Returns the current scale for the graphics view.

The scale starts at 1.0. As the image is expanded by zooming in, the scale gets larger. A scale of 2.0 indicates the apparent size of the drawing in the graphics view is twice the normal size. As the image is compressed by zooming out, the apparent size of the image shrinks. A scale of 0.5 indicates the image is half normal size.

The scale can be set manually using the pinch gestures described in Chapters 2 and 3, or under program control using the `setScale` call.

---

### SUB setAntialiased (flag AS INTEGER)

Sets the graphics view to display and draw using antialiased shapes. Pass 0 to turn antialiasing off, or any non-zero value to turn it on.

When antialiasing is turned off, lines and shapes are drawn to exact pixel boundaries. This results in pixilation, causing a jagged appearance at the edge of lines that are not perfectly horizontal or vertical. Turning antialiasing on blends the drawing at the edges, creating a smoother looking image. Unfortunately, this smoothing is not reversible, so, for example, drawing a line first in black and then in white on a white background leaves a shadow.

The `setAntialiased` command affects both drawing and text.

The snippet draws lines with antialiasing on and off. The illustration is greatly magnified to clearly show the difference



in the drawing methods. The leftmost line is drawn using antialiasing. The next line is drawn twice, both times with antialiasing. The first time the line is drawn it is drawn in black, then it is drawn again in yellow. Note this black shadow at the edge of the line. The third line is drawn in black with antialiasing off, showing distinct pixilation. The last line is drawn with antialiasing off, but like the second line, it is drawn twice, first in black and then in yellow. Notice there is no black shadow around the edge of the line.

#### Snippet

```
System.showGraphics

Graphics.drawLine(10, 10, 60, 60)
Graphics.drawLine(20, 10, 70, 60)
Graphics.setColor(1, 1, 0)
Graphics.drawLine(20, 10, 70, 60)
Graphics.setColor(0, 0, 0)
Graphics.setAntialiased(0)
Graphics.drawLine(30, 10, 80, 60)
Graphics.drawLine(40, 10, 90, 60)
Graphics.setColor(1, 1, 0)
Graphics.drawLine(40, 10, 90, 60)
```

---

#### SUB setColor (red, green, blue [, alpha = 1])

Set the color that will be used for the next drawing command.

As with most computer graphics systems, colors are additive colors. The color is specified as the quantity of the three primary colors, red, green and blue. Each of the primary colors has a range from 0.0 to 1.0, with 0.0 giving none of that color, 1.0 giving all of it, and values in between giving a proportional amount.

The alpha value specifies the transparency of the color, with 0.0 being totally transparent and 1.0 being totally opaque. This value may be omitted, in which case the color is opaque.

The following table shows some typical color values.

Red	Green	Blue	Color
0	0	0	Black
1	1	1	White
1	0	0	Red
1	0.6	0	Orange
1	1	0	Yellow
0	1	0	Green
0	0	1	Blue
0.8	0	0.8	Violet
1	0.625	0.625	Pink
0.8	0.55	0	Brown

---

#### SUB setFont (name AS STRING, size, style AS INTEGER)

Set the font that will be used with the drawText method.

**name** is the name of the font. This can either be a font family name, like Georgia, or a complete font name, such as Georgia-BoldItalic. **style** is a value that specifies whether the font should be plain (a value of 0), bold (a value of 1), italic (a value of 2) or both bold and italic (a value of 3). setFont selects the best available match to the specified name and style from the various fonts returned by font.

**size** is the font size in points. There are about 72 points per inch, so a font size of 72 gives characters that are about an inch high.

The snippet shows a program that sets the font, then prints the various characteristics of the font.



**Snippet**

```
Graphics.setFont("helvetica", 24, 3)
PRINT "The font name is "; Graphics.fontName
PRINT "The font size is "; Graphics.fontSize
PRINT "The font style is "; Graphics.fontStyle
PRINT "The ascender is "; Graphics.ascender
PRINT "The descender is "; Graphics.descender
PRINT "The height of one line is "; Graphics.lineHeight
PRINT "The width of the string "Hello, World." is ";
Graphics.stringWidth("Hello, world.")
```

This snippet prints

```
The font name is Helvetica
The font size is 24
The font style is 3
The ascender is 22
The descender is -5
The height of one line is 29
The width of the string "Hello, world." is 142
```

**SUB setPixelGraphics (flag AS INTEGER)**

Sets the drawing commands to draw using pixel graphics or vector graphics. Pass 0 for vector graphics, or any non-zero value for pixel graphics. The default is pixel graphics.

techBASIC supports both vector graphics and pixel graphics for the basic drawing commands. Pixel graphics draws to a paint layer with a one-to-one correspondence between the pixels you see and the pixels drawn by the drawing command. When the image is scaled, either through the `setScale` command or when exported using the Share button, the image can become pixelated. Vector graphics results in no loss of resolution due to pixilation when zoomed or shared, but drawing repeatedly to the screen gradually fills up memory and slows down refreshing of the graphics screen. Both methods have their uses, so both are available, and they can even be mixed. Drawing with the vector graphics commands draws on top of the pixel graphics layer.

All plots are drawn using vector graphics.

The methods that can draw in either mode are:

<code>drawArc</code>	<code>drawLine</code>	<code>drawOval</code>	<code>drawRect</code>
<code>drawRoundRect</code>	<code>drawText</code>	<code>fillArc</code>	<code>fillOval</code>
<code>fillRect</code>	<code>fillRoundRect</code>		

**SUB setScale (scale)**

Sets the current scale for the graphics view.

The scale starts at 1.0. Setting the scale to 2.0 sets the apparent size of the drawing in the graphics view to twice the normal size, while setting it to 0.5 sets the image to half normal size.

The scale can be set manually using the pinch gestures described in Chapters 2 and 3. Use the `scale` function to read the current scale.

**SUB setToolsHidden (flag AS INTEGER)**

Shows or hides the tools button.

The graphics screen shows a tools or pan/zoom toggle button in certain situations. On the iPhone, a wrench appears any time a program is running. Tapping the wrench brings up a dialog that can send a screen capture to the Photo application, and, if there is a 3D plot on the screen, switch between panning and zooming the function or axis. Debugger buttons also appear in full screen mode. On the iPad, the button appears if there is a 3D plot on the screen or when in full screen mode.

This command hides or shows the tools button.

Be careful! Hiding the button in full screen graphics mode leaves no way to force a program to stop using the debugger commands. Unless the program has a functioning Quit button or its equivalent, the only way to stop the program will be to completely shut down techBASIC.

---

**SUB setToolsLocation (x, y)**

Sets the location of the tools button.

The graphics screen shows a tools or pan/zoom toggle button in certain situations. On the iPhone, a wrench appears any time a program is running. Tapping the wrench brings up a dialog that can send a screen capture to the Photo application, and, if there is a 3D plot on the screen, switch between panning and zooming the function or axis. Debugger buttons also appear in full screen mode. On the iPad, the button appears if there is a 3D plot on the screen or when in full screen mode.

This command moves the button from its default location at the top right of the graphics screen to the location specified.

---

**SUB setTranslation (tx, ty)**

Sets the position of the graphics view so the point -tx, -ty is at the top left of the visible graphics view.

See `translationX` and `translationY` for methods that will read the current translation.

---

**SUB setUpdate (update AS INTEGER)**

Turns graphics screen updates on or off. Turning updates on forces the graphics screen to redraw.

Redrawing the graphics screen does not take a lot of time in an absolute sense, but when a program makes several dozen changes in a group, updating the graphics screen after each change can cause flicker. This method is used to turn automatic updates of the graphics screen after each drawing command off. Use it before a series of drawing commands to turn updates off, then turn them back on to view the result with a single update.

The following snippet draws 1000 small ovals to the iPad display. (Change 750 to 300 for the iPhone.) As shown, the ovals all appear at once. Comment out the calls to `setUpdate`, though, and it is clear the ovals being drawn one-by-one.

Snippet

```
System.showGraphics

Graphics.setUpdate(0)
FOR i = 1 TO 1000
  x = RND(1)*750
  y = RND(1)*750
  Graphics.setColor(RND(1), RND(1), RND(1))
  Graphics.fillOval(x, y, 10, 10)
NEXT
Graphics.setUpdate(1)
```

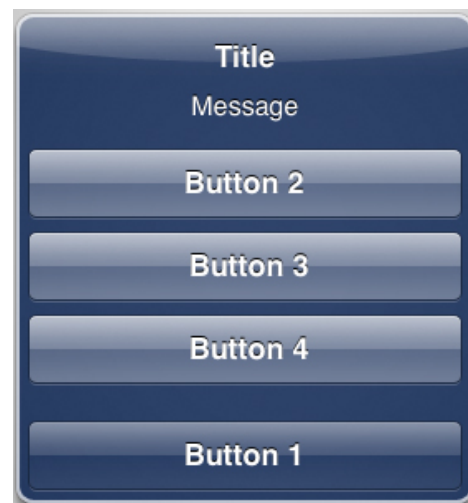
---

**FUNCTION showAlert (title AS STRING, message AS STRING, button1 AS STRING = "", button2 AS STRING = "", button3 AS STRING = "", button4 AS STRING = "") AS INTEGER**

Shows a modal alert. The user dismisses the alert by tapping one of the buttons on the alert. The number of the button pressed is returned, counting from 1.

While up to four buttons can be created, the call works even if no buttons are supplied. The alert shows a single button titled OK if no buttons are passed as parameters.

The snippet shows a short program that displays a button called Show Alert. When pressed, the alert shown is displayed.



Tapping one of the buttons prints the button number in a label that appears to the right of the Show Alert button.

Snippet

```
DIM alertButton AS Button, alertLabel AS Label
alertButton = Graphics.newButton(10, 10, 130)
alertButton.setTitle("Show Alert")

alertLabel = Graphics.newLabel(150, 15, 150)

DIM quitButton AS Button
quitButton = Graphics.newButton(10, 60, 130)
quitButton.setTitle("Quit")

System.showGraphics

SUB touchUpInside (ctrl AS Button, when AS DOUBLE) IF ctrl = quitButton
THEN
    STOP
ELSE IF ctrl = alertButton THEN
    i = Graphics.showAlert("Title", "Message", "Button 1", "Button 2", "
Button 3", " Button 4")
    alertLabel.setText("Button " & str(i) & " pressed.")
END IF
END SUB
```

---

**FUNCTION stringWidth (str AS STRING) AS INTEGER**

Returns the width of the characters in the string **str** when drawn with `drawText` using the current font set by `setFont`. See `setFont` for an example.

---

**FUNCTION translationX**

Returns the current horizontal translation of the graphics view. This is the number of pixels the origin appears to the right of the current left edge of the graphics view. If the value is negative, the origin appears to the left of the visible screen.

The translation can be set either by the `setTranslation` call or by using swipe gestures to manually pan the screen.

---

**FUNCTION translationY**

Returns the current vertical translation of the graphics view. This is the number of pixels the origin appears below the current top edge of the graphics view. If the value is negative, the origin appears to the above the visible screen.

The translation can be set either by the `setTranslation` call or by using swipe gestures to manually pan the screen.

---

**FUNCTION width**

Returns the current width of the visible portion of the graphics view. This is the width, in pixels, of what the user of the program can see. The width of the area available for drawing is not fixed; using the various swipe and pinch gestures described in Chapters 2 and 3 the person using the program can expand, shrink or pan the drawing area to see more or less of the image.

## Image

Images are used to capture images from the camera or photo library, load images from the sandbox, or draw images that can be used for buttons. These images can then be analyzed pixel by pixel, saved to disk, used as an image in a GUI program, or used as a button image in a Button control.

Images are created using the `newImage` call in the `System` class. Most methods in this class require that an image be present before the method can perform its function. Images are loaded using the `getCameraImage`, `getPhotoImage` or `load` methods. Images can be created from scratch using the `newImage` method in this class combined with the various drawing commands. The drawing commands can also be used to draw on top of an image loaded from another source.

The snippet shows a program that loads images from either the camera or photo library.

### Snippet

```

DIM picture AS ImageView, img AS Image
DIM take AS Button, getImage AS Button,
quit AS Button

picture = Graphics.newImageView(10, 10, 300, 250)

img = System.newImage

take = Graphics.newButton(10, 270, 145)
take.setTitle("Take Picture")

getImage = Graphics.newButton(165, 270, 145)
getImage.setTitle("Photo Library")

quit = Graphics.newButton(10, 310, 300)
quit.setTitle("Quit")

System.showGraphics

IF NOT img.hasCamera THEN
    pressed = Graphics.showAlert("No Camera", "This device does not have a
useable camera.")
    take.setHidden(1)
END IF
IF NOT img.hasPhotoLibrary THEN
    pressed = Graphics.showAlert("No Photo Library", "This device does not
have a useable photo library.")
    getImage.setHidden(1)
END IF
END IF

SUB touchUpInside (ctrl AS Button, when AS DOUBLE)
IF ctrl = quit THEN
    STOP

```



Take Picture

Photo Library

Quit

```

ELSE IF ctrl = take THEN
    img.getCameraImage
    IF img.succeeded THEN
        picture.setImage(img, 0)
    END IF
ELSE IF ctrl = getImage THEN
    img.getPhotoImage
    IF img.succeeded THEN
        picture.setImage(img, 0)
    END IF
END IF
END SUB

```

---

**FUNCTION ascender**

Returns the font ascender for the current font in pixels. The ascender is the distance from the baseline of the font to the top of the highest character in the font.

**Snippet**

```

DIM img AS Image
img = System.newImage
img.newImage(100, 50)
img.drawText(10, 10 + img.ascender, 0, "Hello, world.")

```

---

**SUB crop (x, y, width height)**

Extracts a portion of the existing image, setting this image to a smaller image.

**x** and **y** are the top left pixel to include in the new image, indexing from 0. **width** and **height** indicate the number of pixels in the new image.

---

**FUNCTION descender**

Returns the font descender for the current font in pixels. The descender is the distance from the baseline of the font to the bottom of the lowest character in the font. The value is negative.

**Snippet**

```

DIM img AS Image
img = System.newImage
img.newImage(100, 50)
img.drawText(10, 10 - img.descender, 0, "Hello, world.")

```

---

**SUB drawArc (x, y, width, height, startAngle, arcAngle)**

Draws the outline of an arc formed by drawing part of the oval that would fill the given rectangle. The color of the arc is set by the most recent call to `setColor`.

The **x**, **y**, **width** and **height** parameters define the size and location of the rectangle.

Angles are measured in degrees, starting from the 3 o'clock position. Positive angles rotate counterclockwise. **startAngle** is the angle where the arc will begin drawing, while **arcAngle** is the size of the arc.

**Snippet**

```

! Draw an arc filling the top-left quarter of an image.
DIM img AS Image
img = System.newImage
img.newImage(100, 50)
img.drawArc(0, 0, 100, 50, 90, 90)

```

---

**SUB drawLine (x1, y1, x2, y2)**

Draws a line connecting two points. The color of the line is set by the most recent call to `setColor`.

**x1** and **y1** specify the first point, while **x2** and **y2** specify the second.

### Snippet

```
DIM img AS Image
img = System.newImage
img.newImage(250, 250)
img.setColor(255, 0, 0)
img.drawLine(100, 100, 150, 150)
img.drawLine(150, 150, 100, 200)
img.drawLine(100, 200, 50, 150)
img.drawLine(50, 150, 100, 100)
```

---

### SUB drawOval (x, y, width, height)

Draws the outline of an oval that fills the given rectangle. The color of the oval is set by the most recent call to setColor.

The **x**, **y**, **width** and **height** parameters define the size and location of the rectangle.

### Snippet

```
! Draw 100 ovals with sizes ranging from 10 to 100 pixels
! using random colors.
DIM img AS Image
img = System.newImage
img.newImage(250, 250)
FOR i = 1 TO 100
    r = RND(1.0)
    g = RND(1.0)
    b = RND(1.0)
    img.setColor(r, g, b)

    w = 10 + RND(1.0)*90
    h = 10 + RND(1.0)*90
    x = RND(1.0)*(400 - w%)
    y = RND(1.0)*(400 - h%)
    img.drawOval(x, y, w, h)
NEXT
```

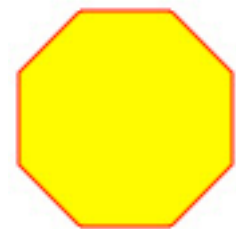
---

### SUB drawPoly (points(,))

Draws the outline of a polygon with the current color.

The polygon is formed from the points in the passed array. The first subscript of the array indexes over points, while the second indexes over X and Y. The number of elements in the second subscript must be two. The shape is outlined by moving from one point to the next, automatically closing the shape by returning to the first point if necessary.

The snippet shows an octagon, both outlined with drawPoly and filled with fillPoly.



### Snippet

```
DIM img AS Image
img = System.newImage
DIM poly(8, 2)
poly = [[10, 33],
        [10, 67],
        [33, 90],
        [67, 90],
        [90, 67],
        [90, 33],
        [67, 10],
        [33, 10]]
img.setColor(1, 1, 0)
img.fillPoly(poly)
```

```
img.setColor(1, 0, 0)
img.drawPoly(poly)
```

---

**SUB drawRect (x, y, width, height)**

Draws the outline of a rectangle. The color of the rectangle is set by the most recent call to `setColor`.

The **x**, **y**, **width** and **height** parameters define the size and location of the rectangle.

**Snippet**

```
DIM img AS Image
img = System.newImage
img.newImage(120, 120)
img.drawRect(10, 10, 100, 100)
```

---

**SUB drawRoundRect (x, y, width, height, arcWidth, arcHeight)**

Draws the outline of a rectangle with rounded corners. The color of the rounded rectangle is set by the most recent call to `setColor`.

The **x**, **y**, **width** and **height** parameters define the size and location of the rectangle.

The **arcWidth** and **arcHeight** parameters define the size of the rounded corners. Imagine an oval whose size is given by **arcWidth** and **arcHeight**, sliced into four parts to form four arcs. These arcs are then used as the rounded corners of the rectangle.

**Snippet**

```
DIM img AS Image
img = System.newImage
img.newImage(120, 120)
img.drawRoundRect(10, 10, 100, 100, 30, 30)
```

---

**SUB drawText (x, y, angle, str AS STRING)**

Draws text. The color of the text is set by the most recent call to `setColor`, while the font is set by `setFont`.

The **x** and **y** parameters define the location of the text. The vertical position is to the baseline of the text, not to the top or bottom of a rectangle containing the text. Think of it as the location of the line on lined paper when writing text by hand. **angle** is the angle, in radians, of the text from the horizontal; positive angles rotate the text counter-clockwise. **str** is the text to draw.

**Snippet**

```
DIM img AS Image
img = System.newImage
img.newImage(200, 30)
img.drawText(10, 20, 0, "Hello, world.")
```

---

**SUB fillArc (x, y, width, height, startAngle, arcAngle)**

Draws a solid arc formed by drawing part of the oval that would fill the given rectangle. The color of the arc is set by the most recent call to `setColor`.

The **x**, **y**, **width** and **height** parameters define the size and location of the rectangle.

Angles are measured in degrees, starting from the 3 o'clock position. Positive angles rotate counterclockwise. **startAngle** is the angle where the arc will begin drawing, while **arcAngle** is the size of the arc.

The snippet shows the `fillArc` command in use, drawing the pie chart shown to the right.



## Chapter 16: Graphics Classes

### Snippet

```
DIM img AS Image
slices = [10, 10, 30, 40, 80]
drawPie(10, 10, 200, 200, slices())
END

SUB drawPie (x, y, width, height, slices())
    ! Find the size for all slices.
    sum = 0.0
    FOR i = 1 TO UBOUND(slices, 1)
        sum = sum + slices(i)
    NEXT

    ! Create the image to draw to.
    img = System.newImage
    img.newImage(width, height)

    ! Draw the slices.
    start = 0
    FOR i = 1 TO UBOUND(slices, 1)
        ! Set the color.
        IF r = 1 THEN
            r = 0
            IF b = 1 THEN
                b = 0
                g = 1
            ELSE
                b = 1
            END IF
        ELSE
            r = 1
        END IF
        img.setColor(r, g, b)

        ! Draw one slice.
        IF i = UBOUND(slices, 1) THEN
            sliceSize = 360 - start
        ELSE
            sliceSize = 360*slices(i)/sum
        END IF
        img.fillArc(x, y, width, height, start, sliceSize)
        start = start + sliceSize
    NEXT
END SUB
```

---

### **SUB fillOval (x, y, width, height)**

Draws a solid oval that fits in the given rectangle. The color of the oval is set by the most recent call to setColor.

The **x,y,width** and **height** parameters define the size and location of the rectangle.

### Snippet

```
DIM img AS Image
img = System.newImage
img.newImage(120, 120)
img.fillOval(10, 10, 100, 100)
```

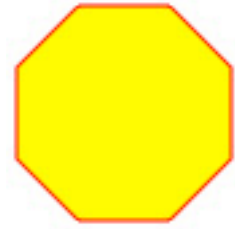


**SUB fillPoly (points(),)**

Fills a polygon with the current color.

The polygon is formed from the points in the passed array. The first subscript of the array indexes over points, while the second indexes over X and Y. The number of elements in the second subscript must be two. The shape is outlined by moving from one point to the next, automatically closing the shape by returning to the first point if necessary. The interior of the shape is then painted with the current color.

The snippet shows an octagon, both outlined with drawPoly and filled with fillPoly.

Snippet

```
DIM img AS Image
img = System.newImage
DIM poly(8, 2)
poly = [[10, 33],
        [10, 67],
        [33, 90],
        [67, 90],
        [90, 67],
        [90, 33],
        [67, 10],
        [33, 10]]
img.setColor(1, 1, 0)
img.fillPoly(poly)
img.setColor(1, 0, 0)
img.drawPoly(poly)
```

**SUB fillRect (x, y, width, height)**

Draws a solid rectangle. The color of the rectangle is set by the most recent call to setColor.

The **x**, **y**, **width** and **height** parameters define the size and location of the rectangle.

Snippet

```
! Draw 100 rectangle with sizes ranging from 10 to 100 pixels
! using random colors.
DIM img AS Image
img = System.newImage
img.newImage(400, 400)
FOR i = 1 TO 100
  r = RND(1.0)
  g = RND(1.0)
  b = RND(1.0)
  img.setColor(r, g, b)

  w = 10 + RND(1.0)*90
  h = 10 + RND(1.0)*90
  x = RND(1.0)*(400 - w)
  y = RND(1.0)*(400 - h)
  img.fillRect(x, y, w, h)
NEXT
```

**SUB fillRoundRect (x, y, width, height, arcWidth, arcHeight)**

Draws a solid rectangle with rounded corners. The color of the rounded rectangle is set by the most recent call to setColor.

The **x**, **y**, **width** and **height** parameters define the size and location of the rectangle.

The **arcWidth** and **arcHeight** parameters define the size and location of the rounded corners. Imagine an oval whose size is given by **arcWidth** and **arcHeight**, sliced into four parts to form four arcs. These arcs are then used as the rounded corners of the rectangle.

Snippet

```
Graphics.fillRoundRect(10, 10, 100, 100, 30, 30)
```

---

**FUNCTION font (index AS INTEGER) AS STRING**

Returns the name of one of the available fonts. **index** is the index of the font, counting from 1. Use **fontCount** to get the number of available fonts.

There are always at least three fonts, and they are predefined in techBASIC. These fonts are

Font Number	Font Name	iOS Font Used
1	Sans-serif	Arial
2	Serif	TimesNewRomanPSMT
3	Monospaced	Courier

These are not normal system fonts, but predefined fonts that give a reasonable font on all systems where techBASIC is implemented. Serif is a font with decorative elements in the letters, like the Times font used for the text in this manual. Sans-serif is a font with no decorative elements, like **Geneva**. Monospaced is a font where all of the characters have the same width, like **Courier New**, used for code samples in this manual.

The snippet shows a short program that lists the available fonts. The fonts returned by this method are exactly the same as those returned by the `Graphics.font` method.

Snippet

```
DIM img AS Image
img = System.newImage
img.newImage(120, 120)
FOR i = 1 TO img.fontCount
    PRINT img.font(i)
NEXT
```

---

**FUNCTION fontCount AS INTEGER**

Returns the number of available font names. See **font** for a full discussion and an example.

---

**FUNCTION fontName AS STRING**

Returns the name of the current font. See **setFont** for a discussion and example.

---

**FUNCTION fontSize AS INTEGER**

Returns the size of the current font. See **setFont** for a discussion and example.

---

**FUNCTION fontStyle AS INTEGER**

Returns the style of the current font. See **setFont** for a discussion and example.

---

**SUB getCameraImage (allowsEditing AS INTEGER = 0, frontCamera AS INTEGER = 0)**

Allows the user to take a picture, returning the picture as the new image.

The user can opt to cancel the operation. Use the **succeeded** method to see if a picture was taken, or the **hasImage** method to see if any image is loaded.

The **allowsEditing** parameter, if set, allows the user to pan and scale the photograph before returning it. This also limits the resolution of the image to the number of pixels seen in the preview. The default behavior is to return the entire image to the resolution of the camera used.

The **frontCamera** parameter, if set, tells the device to default to the front camera rather than the rear camera. The user can still manually change the camera. This parameter is ignored if the device does not have a front-facing camera.

Use the `hasCamera` method to determine if the device has a usable camera before making this call.

See the snippet at the beginning of this class for an example of the `getCameraImage` method.

---

**SUB getPhotoImage (allowsEditing AS INTEGER = 0)**

Allows the user to fetch an image from the photo library.

The user can opt to cancel the operation. Use the `succeeded` method to see if a photo was selected, or the `hasImage` method to see if any image is loaded.

The **allowsEditing** parameter, if set, allows the user to pan and scale the photo before returning it. This also limits the resolution of the image to the number of pixels seen in the preview. The default behavior is to return the entire photo to the resolution saved in the photo library.

Use the `hasPhotoLibrary` method to determine if the device has a usable photo library before making this call.

See the snippet at the beginning of this class for an example of the `getPhotoImage` method.

---

**FUNCTION hasCamera AS INTEGER**

Returns 1 if the device has a useable camera, and 0 if not.

Use this method before calling the `getCameraImage` method to make sure the device has a useable camera.

See the snippet at the beginning of this class for an example of the `hasCamera` method.

---

**FUNCTION hasImage AS INTEGER**

Returns 1 if the object has a useable image, and 0 if not.

---

**FUNCTION hasPhotoLibrary AS INTEGER**

Returns 1 if the device has useable camera, and 0 if not.

Use this method before calling the `getPhotoImage` method to make sure the device has a useable photo library.

See the snippet at the beginning of this class for an example of the `hasCamera` method.

---

**FUNCTION height**

Returns the height of the image in pixels.

---

**FUNCTION isAntialiased AS INTEGER**

Returns 0 if drawing will be done without antialiasing, and 1 if drawing will be done using antialiasing.

See `setAntialiased` for a description of antialiasing and a code sample.

---

**FUNCTION lineHeight AS INTEGER**

Returns the height of a line of text drawn with `drawText` using the current font set by `setFont`. See `setFont` for an example.

This height is not the same thing as the ascender – descender; in other words, it is not the height just of the visible text. This is the recommended total line height, including the leading that separates the lines of text. This is the value to use to decide how far apart to place lines of text when drawing multiple lines in an image.

---

**SUB newImage (width AS INTEGER, height AS INTEGER)**

Creates a new image, replacing any existing image in the process. The image has a size as dictated by the parameters, and is initially transparent.

The snippet shows a short program that illustrates the various drawing commands, creating a transparent button. The background is drawn with a checkerboard pattern to show the transparency of the button.

## Chapter 16: Graphics Classes

### Snippet

```
! Show the graphics screen.
System.showGraphics

! Set the button size.
width = 370
height = 130

! Draw the checkerboard background.
Graphics.setColor(0.95, 0.95, 0.95)
draw = 0
xOdd = 0
FOR x = 0 TO width + 10 STEP 10
    IF xOdd THEN
        draw = 1
        xOdd = 0
    ELSE
        draw = 0
        xOdd = 1
    END IF
    FOR y = 0 TO height + 20 STEP 10
        IF draw THEN
            Graphics.fillRect(x, y, 10, 10)
            draw = 0
        ELSE
            draw = 1
        END IF
    NEXT
NEXT

! Create the image.
DIM bi AS Image
bi = System.newImage
bi.newImage(width, height)

! Draw the image for the button.
bi.drawRect(0, 0, width, height)

bi.setColor(1, 0, 0)
bi.drawLine(10, 10, 60, 10)
bi.setColor(0, 1, 0)
bi.drawLine(60, 10, 60, 60)
bi.setColor(0, 0, 1)
bi.drawLine(10, 60, 60, 60)
bi.setColor(1, 1, 0)
bi.drawLine(10, 10, 10, 60)
```

```

bi.setColor(1, 0, 0)
bi.drawArc(70, 10, 50, 50, 22.5, 45)
bi.fillArc(70, 70, 50, 50, 22.5, 45)
bi.setColor(0, 1, 0)
bi.drawOval(130, 10, 50, 50)
bi.fillOval(130, 70, 50, 50)
bi.setColor(0, 0, 1)
bi.drawRect(190, 10, 50, 50)
bi.fillRect(190, 70, 50, 50)
bi.setColor(1, 0, 1)
bi.drawRoundRect(250, 10, 50, 50, 20, 20)
bi.fillRoundRect(250, 70, 50, 50, 20, 20)

bi.setFont("Baskerville", 20, 3)
bi.drawText(310, 60, Math.PI/8, "Hello")

bi.setColor(0, 0, 0)
bi.drawLine(10, 70, 50, 120)
bi.setAntialiased(0)
bi.drawLine(20, 70, 60, 120)

! Place the image on a button.
DIM btn AS button
btn = Graphics.newButton(10, 10, width, height)
btn.setImage(bi)

```

---

**SUB load (path AS STRING)**

Loads an existing image from disk.

The currently supported image formats are:

Extension	Format
.tiff, .tif	Tagged Image File Format (TIFF)
.jpg, .jpeg	Joint Photographic Experts Group (JPEG)
.gif	Graphic Interchange Format (GIF)
.png	Portable Network Graphic (PNG)
.bmp, .BMPf	Windows Bitmap Format (DIB)
.ico	Windows Icon Format
.cur	Windows Cursor
.xbm	XWindow bitmap

---

**FUNCTION pixel (x AS INTEGER, y AS INTEGER) AS LONG**

Returns a single pixel from the image.

**x** and **y** specify the pixel to return, indexed from 0.

The pixel value is returned packed in a long value. There are four components, the alpha (opacity), and the color intensities for red, green and blue. Each component is stored in one byte of the long value, with each value ranging from 0 to 255. For example, a pixel that is fully opaque (alpha of 255), with 0% red, 25% green and 50% blue would have a value of -1676704. Expressed as a hexadecimal value, this is 0xFF004080.

Snippet

```

! Insert this line in the sample at the start of the class to
! print a single pixel.
PRINT HEX(img.pixel(10, 10))

```

---

**FUNCTION pixels (x, y, width height) (,) AS LONG**

Returns a block of pixels from the image.

**x** and **y** specify the top left pixel to return, indexed from 0. The **width** and **height** parameters specify the size of the block of pixels to return.

The pixels are returned in a two-dimensional array where the first subscript indexes over the x coordinates, and the second over the y coordinates.

Each pixel value is packed in a long value. There are four components, the alpha (opacity), and the color intensities for red, green and blue. Each component is stored in one byte of the long value, with each value ranging from 0 to 255. For example, a pixel that is fully opaque (alpha of 255), with 0% red, 25% green and 50% blue would have a value of -1676704. Expressed as a hexadecimal value, this is 0xFF004080.

**Snippet**

```
! Insert these lines in the sample at the start of the class to
! print a small block of pixels.
DIM pixels (1, 0) AS LONG
Pixels = img.pixels(1, 1, 12, 12)
FOR y = 1 TO 12
  FOR x = 1 TO 12
    PRINT HEX(pixels(x, y)); " ";
  NEXT
  PRINT
NEXT
```

---

**SUB save (path AS STRING, format AS INTEGER = 1, compression = 1.0)**

Save the current image to disk.

The path name is passed as the **path** parameter. The file extension should be specified, and should be either .jpg or .png, depending on the format selected.

Two formats are available for saving images. The default is JPEG with as good an image quality as possible. Specify 2 for the **format** parameter to get a PNG format image.

JPEG files support lossy compression where image quality degrades as the compression increases. The **compression** parameter is used to specify the compression level for the JPEG image, where 1.0 is low compression and 0.0 is high compression. This parameter is ignored for PNG format files.

---

**SUB setAntialiased (flag AS INTEGER)**

Sets the graphics view to display and draw using antialiased shapes. Pass 0 to turn antialiasing off, or any non-zero value to turn it on.

When antialiasing is turned off, lines and shapes are drawn to exact pixel boundaries. This results in pixilation, causing a jagged appearance at the edge of lines that are not perfectly horizontal or vertical. Turning antialiasing on blends the drawing at the edges, creating a smoother looking image. Unfortunately, this smoothing is not reversible, so, for example, drawing a line first in black and then in white on a white background leaves a shadow.

The `setAntialiased` command affects both drawing and text.

---

**SUB setColor (red, green, blue [, alpha = 1])**

Set the color that will be used for the next drawing command.

As with most computer graphics systems, colors are additive colors. The color is specified as the quantity of the three primary colors, red, green and blue. Each of the primary colors has a range from 0.0 to 1.0, with 0.0 giving none of that color, 1.0 giving all of it, and values in between giving a proportional amount.

The alpha value specifies the transparency of the color, with 0.0 being totally transparent and 1.0 being totally opaque. This value may be omitted, in which case the color is opaque.

The following table shows some typical color values.

Red	Green	Blue	Color
0	0	0	Black
1	1	1	White
1	0	0	Red
1	0.6	0	Orange
1	1	0	Yellow
0	1	0	Green
0	0	1	Blue
0.8	0	0.8	Violet
1	0.625	0.625	Pink
0.8	0.55	0	Brown

---

**SUB setFont (name AS STRING, size, style AS INTEGER)**

Set the font that will be used with the drawText method.

**name** is the name of the font. This can either be a font family name, like Georgia, or a complete font name, such as Georgia-BoldItalic. **style** is a value that specifies whether the font should be plain (a value of 0), bold (a value of 1), italic (a value of 2) or both bold and italic (a value of 3). setFont selects the best available match to the specified name and style from the various fonts returned by font.

**size** is the font size in points. There are about 72 points per inch, so a font size of 72 gives characters that are about an inch high.

The snippet shows a program that sets the font, then prints the various characteristics of the font.

**Snippet**

```
DIM img AS Image
img = System.newImage
img.setFont("helvetica", 24, 3)
PRINT "The font name is "; img.fontName
PRINT "The font size is "; img.fontSize
PRINT "The font style is "; img.fontStyle
PRINT "The ascender is "; img.ascender
PRINT "The descender is "; img.descender
PRINT "The height of one line is "; img.lineHeight
PRINT "The width of the string ""Hello, World."" is ";
img.stringWidth("Hello, world.")
```

This snippet prints

```
The font name is Helvetica
The font size is 24
The font style is 3
The ascender is 22
The descender is -5
The height of one line is 29
The width of the string "Hello, world." is 142
```

---

**FUNCTION stringWidth (str AS STRING) AS INTEGER**

Returns the width of the characters in the string **str** when drawn with drawText using the current font set by setFont. See setFont for an example.

---

**FUNCTION succeeded AS INTEGER**

Returns 1 if the most recent call to getCameraImage, getPhotoImage, load or save was successful, and 0 if the call was canceled by the user or generated an error.

See the snippet at the beginning of this class for an example of the `succeeded` method.

---

### FUNCTION `width`

Returns the width of the image in pixels.

---

## Plot

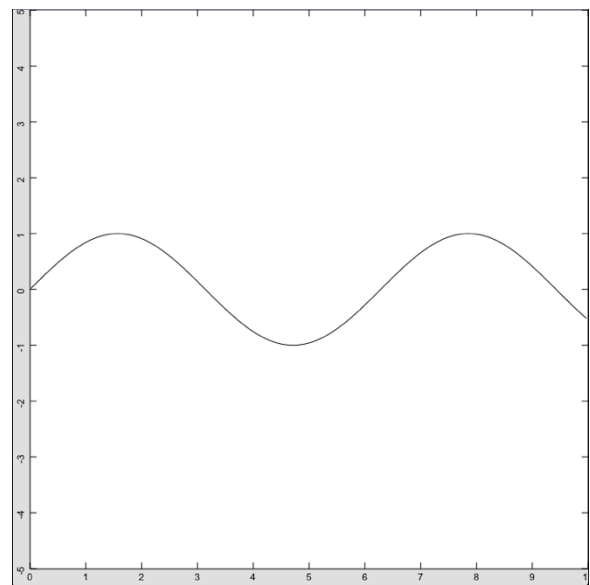
Plots are used to draw functions, plot point cloud data, or to connect point cloud data with lines or surfaces. They are created using the `newPlot` function in the predefined `graphics` object. The following short program creates the graph of the `SIN` function shown to the right.

```
DIM p AS Plot, fx AS PlotFunction

system.showGraphics
p = graphics.newPlot
fx = p.newFunction(FUNCTION f)
END

FUNCTION f(x)
f = SIN(x)
END FUNCTION
```

The first line isn't really required; `system.showGraphics` is a programmatic way to force the graphics view to display so the result of the program can be seen without manually tapping the button that displays the graphics view. The next line creates a new plot that displays information using a two-dimensional Cartesian coordinate system. Finally, a function is added to the plot. This one is a simple function that plots the built-in sine function.



There are lots of options that provide control over the specific nature of the plot. In addition to the two-dimensional Cartesian coordinate system shown, polar, three-dimensional Cartesian, cylindrical and spherical coordinate systems are available. It is easy to add additional functions to the plot, as well as point cloud data and error bars. The color, axis style and titles can all be manipulated with the subroutines in this class, too. It is also possible to create multiple plots, which are automatically tiled—but can be repositioned under program control if the default tiling is not what is desired. The methods in this class provide control over the overall plot. See `PlotFunction`, `PlotMesh` and `PlotPoint`, later in this chapter, for ways to control the individual functions that appear on a plot.

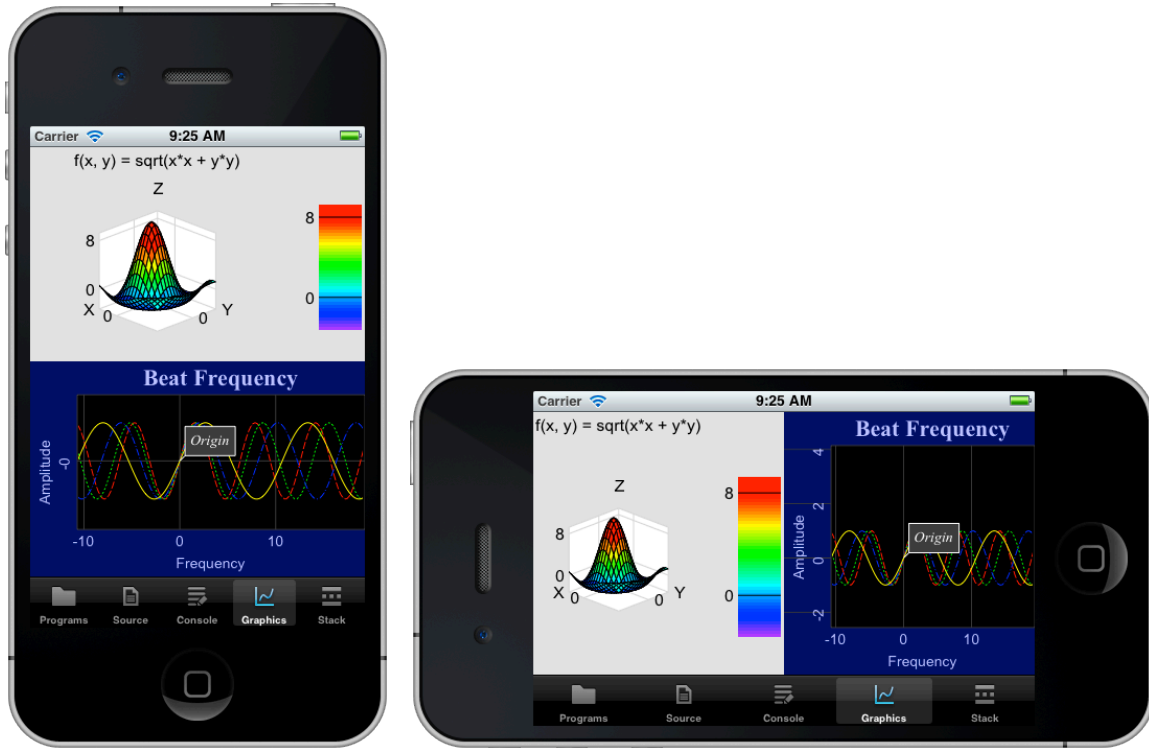
---

## Tiling Plots

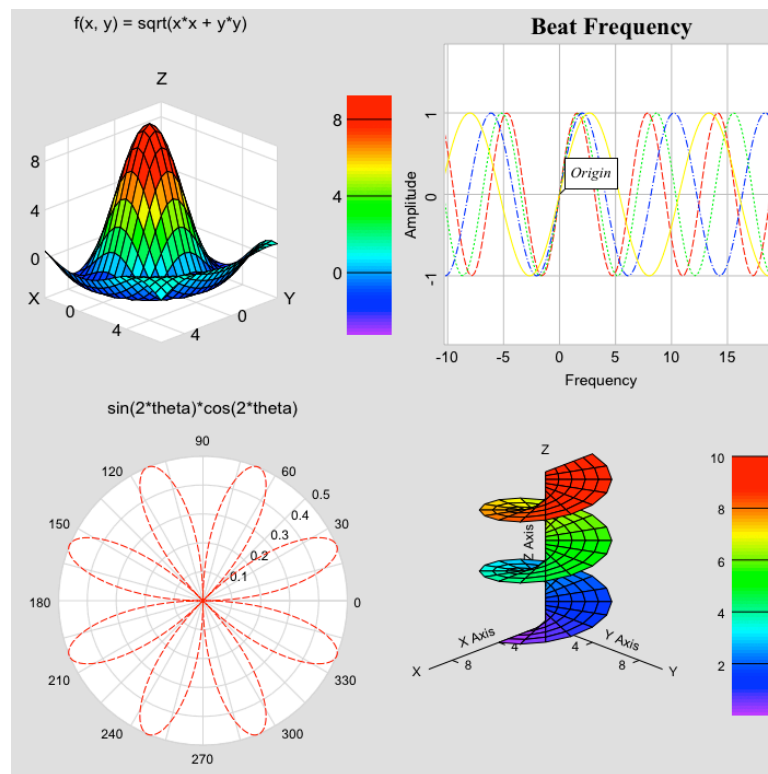
The first plot added to the graphics view with the `graphics.newPlot` command will fill the available graphics view in the current user orientation.

The second plot added will resize the original plot to use half of the available view. The new plot will use the other half. The way the graphics view is split depends on the orientation of the device—the view will be split to divide the long axis in half.





Additional plots continue to divide the screen along the longest uncut axis. While there is no theoretical limit on the number of plots you can add, space considerations make one, or perhaps two, the practical limit on the iPhone or iPod, while four or perhaps nine will work on the iPad.



---

## Colors

Many of the methods in this section set colors. Colors are set using the additive colors of red, green and blue. Each of these color values ranges from 0.0 (none of that color) to 1.0 (saturated with that color). The following table shows a few useful colors and their red, green, blue components.

Red	Green	Blue	Color
0	0	0	Black
1	1	1	White
1	0	0	Red
1	0.6	0	Orange
1	1	0	Yellow
0	1	0	Green
0	0	1	Blue
0.8	0	0.8	Violet
1	0.625	0.625	Pink
0.8	0.55	0	Brown

Some methods also accept an alpha parameter. This is the amount of transparency—a value of 1.0 gives a completely opaque color that completely covers anything beneath the color. A value of 0.0 is completely transparent; drawing with a transparent color does not change the image. Any value in between gives a translucent affect.

---

### FUNCTION `aspectY`

Returns the current aspect ratio between the values plotted along the Y axis and those plotted along the X axis. An aspect ratio greater than 1 indicates the Y axis is magnified compared to the X axis by the amount returned; a value less than 1 indicates the X axis is magnified by the reciprocal relative to the Y axis.

The aspect ratio can be changed using the `setAspect` or `setView` methods, or by manual pinches.

---

### SUB `logAxis (x AS INTEGER, y AS INTEGER, z AS INTEGER)`

Sets some combinations of the axis on a graph to be logarithmic. Pass a nonzero value for an axis to plot that axis as logarithmic, or 0 to plot the axis in the normal way.

---

### FUNCTION `maximumX`

Get the highest X value currently displayed on a Cartesian plot. Both two- and three-dimensional Cartesian plots are supported, but this method returns a meaningless value for polar plots.

The range of values displayed can be changed with the `setView` or `setView3D` method, the `setAspect` method, or manually by swipe and pinch gestures.

---

### FUNCTION `maximumY`

Get the highest Y value currently displayed on a Cartesian plot. Both two- and three-dimensional Cartesian plots are supported, but this method returns a meaningless value for polar plots.

The range of values displayed can be changed with the `setView` or `setView3D` method, the `setAspect` method, or manually by swipe and pinch gestures.

---

### FUNCTION `maximumZ`

Get the highest Z value currently displayed on a three-dimensional Cartesian plot. This method returns a meaningless value for two-dimensional Cartesian plots or polar plots.

The range of values displayed can be changed with the `setView3D` method.

**FUNCTION minimumX**

Get the lowest X value currently displayed on a Cartesian plot. Both two- and three-dimensional Cartesian plots are supported, but this method returns a meaningless value for polar plots.

The range of values displayed can be changed with the `setView` or `setView3D` method, the `setAspect` method, or manually by swipe and pinch gestures.

**FUNCTION minimumY**

Get the lowest Y value currently displayed on a Cartesian plot. Both two- and three-dimensional Cartesian plots are supported, but this method returns a meaningless value for polar plots.

The range of values displayed can be changed with the `setView` or `setView3D` method, the `setAspect` method, or manually by swipe and pinch gestures.

**FUNCTION minimumZ**

Get the lowest Z value currently displayed on a three-dimensional Cartesian plot. This method returns a meaningless value for two-dimensional Cartesian plots or polar plots.

The range of values displayed can be changed with the `setView3D` method.

**FUNCTION newCallout (str AS STRING, x, y, z = 0) AS Callout**

Creates a new callout.

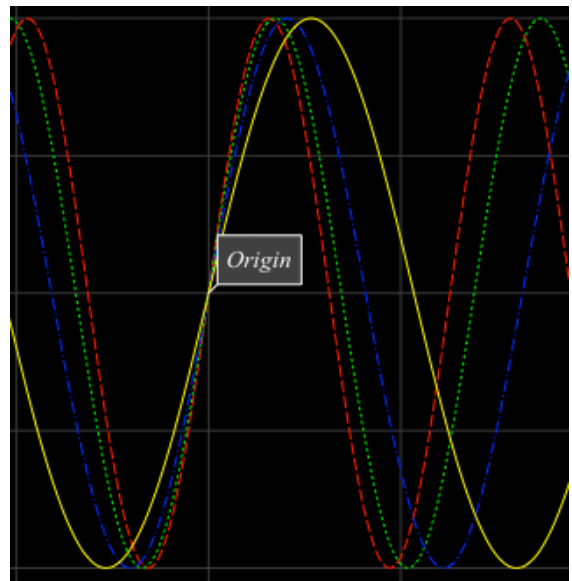
Callouts are text messages attached to values on the plot. Callouts can be created manually by tapping on the plot, but they can also be created using this method. The advantage of this method is that the callout message can be set, and the user does not need to tap on the screen to see the callout.

**str** specifies the text message that will appear in the callout. The **x**, **y** and **z** parameters specify the location on the plot to place the callout. The **z** parameter is optional; if omitted, **z** is set to 0.0. The callout is attached to the geometric position specified; this is not a screen coordinate. The callout will move with the data as the plot is panned and zoomed.

Unlike manual callouts, the user cannot dismiss a callout created with this method by tapping on the callout message.

Snippet

```
DIM c AS Callout
c = p.newCallout("Origin", 0, 0)
c.setBackgroundColor(0.25, 0.25, 0.25)
c.setFontColor(1, 1, 1)
c.setFont("Serif", 14, 2)
```

**FUNCTION newCylindrical (FUNCTION f) AS PlotFunction**

Create a new three-dimensional cylindrical coordinate function and add it to the current plot. The function is a function in theta and r, returning Z. The default domain for the function is 0.0 to  $2\pi$ . The domain can be set to other values using the `setDomain` method in the returned `PlotFunction` object, as seen in the snippet.

The function is added to the current plot. If this is the first function added, the plot is initialized using three-dimensional Cartesian coordinates. The background is set to white, and the border to gray.

The function will not be shown if the axis style is two-dimensional Cartesian or Polar. The axis style can be set manually with the `setAxisStyle` method.

The function defaults to back lines for the mesh that connects the sampled points, with white polygons filling the mesh. The sample shows the result of setting the mesh style to 3 (false color). This is just one of the many plot characteristics that can be set using the calls in this class.

## Chapter 16: Graphics Classes

The snippet shows a program that generates the screw shape seen here. This program is also a sample in the techBASIC app; the sample is called Screw.

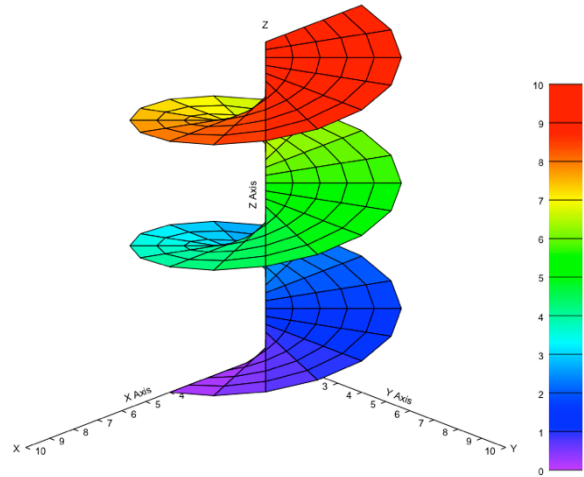
### Snippet

```
! Display the graphics view.
system.showGraphics

! Set up the plot, labeling the axis,
setting the axis
! style to show the color legend on a
standard axis,
! and changing the mesh size to
better display the entire
! range of the function.
DIM p AS Plot
p = graphics.newPlot
p.setSurfaceStyle(3)
p.setAxisStyle(4)
p.setBorderColor(1, 1, 1)
p.setMeshSize(5, 40)
p.setXAxisLabel("X Axis")
p.setYAxisLabel("Y Axis")
p.setZAxisLabel("Z Axis")

! Plot the function for a radius of 0 to 4 and theta
! from 0 to 5*pi.
DIM func AS PlotFunction
func = p.newCylindrical(FUNCTION f)
func.setDomain(0, 4, 0, 5*3.14159)
END

FUNCTION f(r, theta)
f = 2*theta/3.14159
END FUNCTION
```



---

### FUNCTION newFunction (FUNCTION f) AS PlotFunction

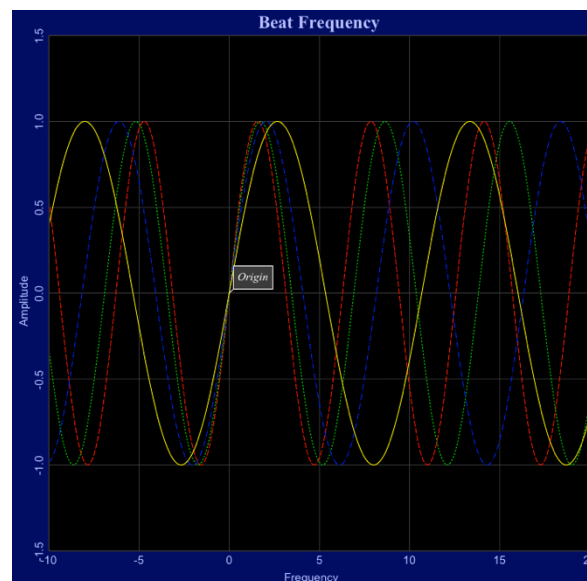
Create a new two- or three-dimensional function and add it to the current plot.

Functions of one variable are plotted on a two-dimensional Cartesian axis, while functions of two variables are plotted on a three-dimensional Cartesian axis. Functions of one variable are discussed first.

The function of one variable is added to the current plot. If it is the first function added, the plot is initialized using two-dimensional Cartesian coordinates and set to display approximately 0 to 10 along the X axis, and -5 to 5 along the Y axis. The Y values will be adjusted to maintain an aspect ratio of 1 and still fill the available space. The background is set to white, and the border to gray.

The function will not be shown if the axis style is three-dimensional Cartesian or Polar. The axis style can be set manually with the `setAxisStyle` method.

The function starts as a solid black line. The color, line type and valid domain for the function can be adjusted using the methods in the returned function object. See the



description of the `PlotFunction` class for more about these methods. Characteristics for the plot itself can be set using the calls in this class.

The snippet shows a program that generates the Beat Frequency plot seen here. This program is also a sample in the techBASIC app; the sample is called Beat Frequency.

Snippet

```
! Display the graphics view.
system.showGraphics

! Create the plot.
DIM p as Plot
p = graphics.newPlot

! Set the plot to a black background, dark blue border,
! light gray grid and light blue labels.
p.setBackgroundColor(0, 0, 0)
p.setBorderColor(0, 0, 0.4)
p.setGridColor(0.25, 0.25, 0.25)
p.setLabelColor(0.75, 0.75, 1)

! Set up the title and axis labels.
p.setTitle("Beat Frequency")
p.setTitleFont("Serif", 22, 1)
p.setYAxisLabel("Amplitude")
p.setXAxisLabel("Frequency")
p.setAxisFont("Sans-serif", 18, 2)
p.setLabelFont("Sans-serif", 14, 0)

! Use a grid instead of tick marks.
p.showGrid (1)

! Add a callout at the origin.
DIM c AS CallOut
c = p.newCallOut("Origin", 0, 0)
c.setBackgroundColor(0.25, 0.25, 0.25)
c.setFontColor(1, 1, 1)
c.setFont("Serif", 14, 2)

! Define four sine waves with varying frequencies.
DIM func AS PlotFunction
func = p.newFunction(FUNCTION f)
func.setColor(1, 0, 0)
func.setStyle(2)

DIM func2 as PlotFunction
func2 = p.newFunction(FUNCTION f2)
func2.setColor(0, 1, 0)
func2.setStyle(3)

DIM func3 as PlotFunction
func3 = p.newFunction(FUNCTION f3)
func3.setColor(0, 0, 1)
func3.setStyle(4)
```

## Chapter 16: Graphics Classes

```
DIM func4 as PlotFunction
func4 = p.newFunction(FUNCTION f4)
func4.setColor(1, 1, 0)
func4.setStyle(1)

! Set the view. Do this after adding functions,
! since adding a function resets the default view.
p.setView(-10, -1.5, 20, 1.5, 0)
END

FUNCTION f(x)
f = SIN(x)
END FUNCTION

FUNCTION f2(x)
f2 = SIN(x/1.1)
END FUNCTION

FUNCTION f3(x)
f3 = SIN(x/1.3)
END FUNCTION

FUNCTION f4(x)
f4 = SIN(x/1.7)
END FUNCTION
```

Functions of two variables are similar, but create a function in X and Y. If the function is the first one created in the plot, the axis is set to a three-dimensional Cartesian axis with the ranges for X, Y and Z set to 0 to 10. The function is sampled over a 21 by 21 grid in the X-Y plane; the resulting Z values are connected by a mesh for display. The mesh size can be changed with the `setMeshSize` method.

The function will not be shown if the axis style is two-dimensional Cartesian or Polar. The axis style can be set manually with the `setAxisStyle` method.

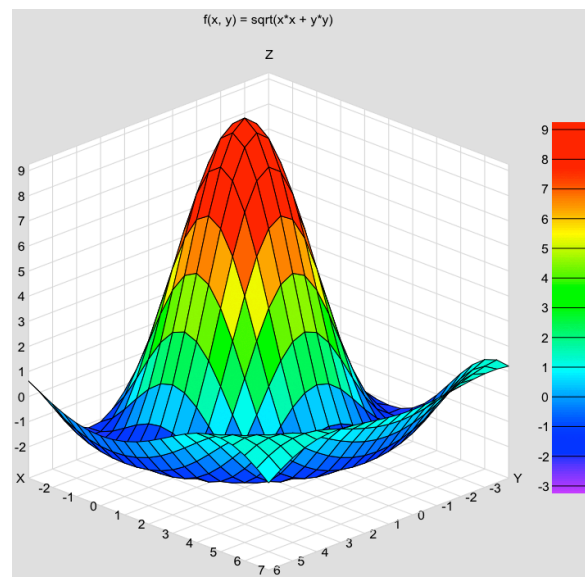
The snippet shows a function that rotates  $\sin(x)/x$  about the Z axis. The vertical dimension is expanded by a factor of 10 to make the initial plot more interesting. This program is also a sample in the techBASIC app; the sample is called `Sinx_x`.

### Snippet

```
! Display the graphics view.
system.showGraphics

! Set up the plot, label it, and
display the function
! with false color.
DIM p AS Plot
p = graphics.newPlot
p.setTitle("f(x, y) = sqrt(x*x + y*y)")
p.setGridColor(0.85, 0.85, 0.85)
p.setsurfacestyle(3)
p.setAxisStyle(5)

! Add the function.
DIM func AS PlotFunction
func = p.newFunction(FUNCTION f)
```



```

! Adjust the function so the portion under the X-Y
! plane is visible, and push it slightly off the Z
! axis so the beginning of the descending curve
! can be seen.
p.setTranslation3D(-4, -3, -2)
p.setScale3D(1, 1, .8)
END

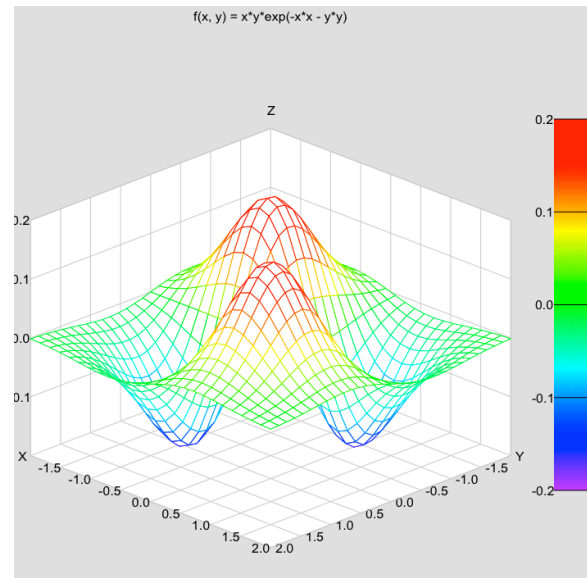
FUNCTION f(x, y)
d = SQR(x*x + y*y)
IF d = 0 THEN
  f = 10
ELSE
  f = 10*SIN(d)/d
END if
END FUNCTION

```

---

**FUNCTION newMesh (x(), y(), z()) AS PlotMesh**

A mesh is a collection of Z values spaced over a grid formed by intersecting values along the X and Y axis. The **x** vector gives a list of points along the X axis, while the **y** vector gives a list of points along the Y axis. The **z** matrix contains the Z values for each point. If the **x** vector has *m* elements, and the **y** vector has *n* elements, the **z** matrix must be an *m*×*n* matrix. The snippet shows how to build up a suitable matrix from a function, although if a function is available to supply Z values, `newFunction` is a better way to plot the function. The difference between a function plotted with `newFunction` and a mesh plotted with `newMesh` is that functions can be recalculated as you pan, zoom and rotate the plot to see different aspects of the surface, while meshes are fixed. If the visible part of a function is panned off of the plot, the formerly hidden portions are calculated, and that part of the function is plotted. If the mesh is panned off of the plot, there is no way to create additional points for display.



Meshes are similar to surfaces, created with the `newSurface` method, but there is a distinct difference. Meshes are functions of X and Y, while surfaces are general three-dimensional point clouds connected together to form a surface that is not, generally speaking, a function in Cartesian coordinates.

If the mesh is the first thing added to a plot, the axis is set to three-dimensional Cartesian coordinates. The default range for the axis are set to match the mesh, with some expansion possible to keep the aspect ratio between the various axis consistent. As shown in the snippet, the `setView3D` and `setScale3D` methods can be used to adjust this default.

The function will not be shown if the axis style is two-dimensional Cartesian or Polar. The axis style can be set manually with the `setAxisStyle` method.

The snippet shows the program used to create the illustration of the mesh. This program is also a sample in the `techBASIC` app; the sample is called `Exp`. While the sample shows coloring the mesh, this is just a stylistic choice—both functions created with `newFunction` and meshes created with `newMesh` can be set to display a colored mesh with white polygons, or colored polygons, as seen in the example for `newFunction`.

**Snippet**

```
! Display the graphics view.
```

```

system.showGraphics

! Create data for a mesh. Set the X axis to 30
! evenly distributed points, and Y to 25, then
! loop over the grid to set the Z values.
DIM x(31), y(26), z(31, 26)
FOR i = 1 TO 31
    x(i) = -2 + (i - 1)*4/30
NEXT
FOR j = 1 to 26
    y(j) = -2 + (j - 1)*4/25
NEXT
FOR i = 1 TO 31
    FOR j = 1 TO 26
        z(i, j) = f(x(i), y(j))
    NEXT
NEXT

! Set up the plot, label it, and display the mesh
! with false color.
DIM p AS Plot
p = graphics.newPlot
p.setGridColor(0.8, 0.8, 0.8)
p.setTitle("f(x, y) = x*y*exp(-x*x - y*y)")
p.setMeshStyle(3)
p.setAxisStyle(5)

! Add the mesh.
DIM m AS PlotMesh
m = p.newMesh(x, y, z)

! Adjust the function so the portion under the X-Y
! plane is visible, and push it off axis.
p.setView3D(-2, -2, -1.5, 2, 2, 1.5)
p.setScale3D(1, 1, 7.5)
END

FUNCTION f(x, y)
    f = x*y*exp(-x*x - y*y)
END FUNCTION

```

---

**FUNCTION newPlot (matrix(,)) AS PlotPoint**

A plot created with `newPlot` consists of point cloud data. Depending on the array passed, the result can be a two- or three-dimensional plot, with or without error bars.

The first subscript of the matrix passed indexes over the points to plot. The second subscript indexes over the coordinates and error bar values.

If the second index of the matrix has a size of two, the points are plotted on a two-dimensional Cartesian axis. The first value is the X coordinate of the point, while the second value is the Y coordinate. Each point is rendered as a small black dot. The points are connected with a black line. The range of the plot is set based on the values of the points passed, with the restriction that the aspect ratio is held at one. The various methods in this class can be used to change the characteristics of the axis, labels and background. The methods in the `PlotPoint` class manipulate the point data, allowing changes to the connecting line, the shape of the points, the colors used to draw the points and lines, and labels used for callouts on the points.



If the second index of the matrix has a size of four, the resulting plot is still two-dimensional plot, but it will also show error bars. The third and fourth value for each point are the error above and below the point, respectively. For example, if the point is the 7<sup>th</sup> data point, is located at (3.5, 0.9), and has an error range of 0.7 to 1.0, the array values passed would be:

Array Element	Value
matrix(7, 1)	3.5
matrix(7, 2)	0.9
matrix(7, 3)	0.1
matrix(7, 4)	0.2

In most cases, the error above and below the value are the same, but the flexibility exists to deal with asymmetric errors if they are present.

The first snippet shows a simple program that plots a parabola with asymmetric error bars.

#### Snippet

```
! Display the graphics view.
system.showGraphics

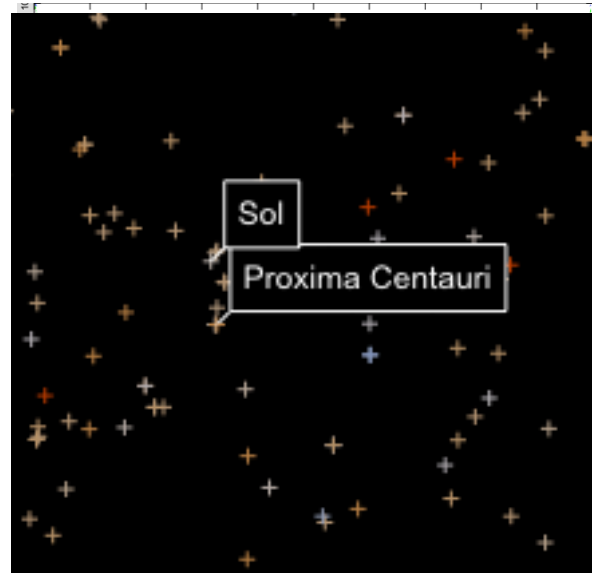
! Set up the point data to plot. This includes
! X and Y points, and errors in Y.
DIM a(0 TO 20, 4)
FOR i = 0 TO 20
  a(i, 1) = i/2
  a(i, 2) = 0.4*(i/2 - 5)*(i/2 - 5)
  a(i, 3) = 0.1
  a(i, 4) = 0.2
NEXT

! Create the plot.
DIM p AS Plot
p = Graphics.newPlot

! Plot the points. Use a green, dashed line with
! blue circles for points.
DIM plot AS PlotPoint
plot = p.newPlot(a())
plot.setColor(0, 1, 0)
plot.setStyle(4)
plot.setPointColor(0, 0, 1)
plot.setPointStyle(5)
END
```

If the second subscript has a size of three or five, the points are plotted on a three-dimensional Cartesian axis. The first three values for each point are the X, Y and Z coordinates, while the next two, if present, are again the error for each point above and below the given Z value.

The snippet shows one of the sample programs that comes with techBASIC. This one is called Stars; it plots 298 of our closest stellar neighbors with approximately the correct color for each star. These are the brightest stars within 10 parsecs of our sun. Brightness has been artificially increased to make the stars easier to see, and plus signs have been used instead of points for the same reason. Both are easily changed in the program. The image



has been zoomed considerably so the tags are easily visible on the printed page. These tags are attached to the point data, so tapping on a star shows the name for that star. A swipe gesture with two fingers twirls the star cloud about the origin, which is the location of our sun, Sol.

#### Snippet

```
! Create a table that maps star color indexes to RGB values.
colorMap = [[-0.4, $9b, $b2, $ff],
            [-0.2, $b2, $c5, $ff],
            [0.0, $d3, $dd, $ff],
            [0.2, $e9, $ec, $ff],
            [0.4, $fe, $f9, $ff],
            [0.6, $ff, $f3, $ea],
            [0.8, $ff, $eb, $d6],
            [1.0, $ff, $e5, $c6],
            [1.2, $ff, $df, $b8],
            [1.4, $ff, $d8, $a9],
            [1.6, $ff, $d0, $96],
            [1.8, $ff, $b7, $65],
            [2.0, $ff, $52, $00]]

! Open, read and process the star database. The first line in the file
! is the number of stars in the database, which lists the 298 brightest
! know stars within 10 parsecs of our sun. The lines that follow have
! comma separated valued containing the name of hte star, the x, y and
! z coordinates in parsecs, with the sun at the origin, the star's
! color index, and the brightness for the pixel, which has been mapped
! to a range of 0.7 to 1. This is brighter than realistic, but makes it
! easier to see the dim stars.
!
! For stars with no common name, the Gliese index of nearby stars is
! shown.
OPEN "Stars.txt" FOR INPUT AS #1
INPUT #1, count
DIM xyz(count, 3), names(count) AS STRING, colors(count, 3)
FOR i = 1 TO count

    ! Read one line from the star database.
    INPUT #1, names(i), xyz(i, 1), xyz(i, 2), xyz(i, 3), color, bright

    ! Create the color for the star's point.
    IF color < colorMap(1, 1) THEN
        colors(i, 1) = colorMap(1, 2)*bright/255
        colors(i, 2) = colorMap(1, 3)*bright/255
        colors(i, 3) = colorMap(1, 4)*bright/255
    ELSE
        FOR c% = UBOUND(colorMap, 1) TO 1 STEP -1
            IF color >= colorMap(c%, 1) THEN
                colors(i, 1) = colorMap(c%, 2)*bright/255
                colors(i, 2) = colorMap(c%, 3)*bright/255
                colors(i, 3) = colorMap(c%, 4)*bright/255
                GOTO out
            END IF
        NEXT
    END IF
NEXT
out:
END IF
NEXT
CLOSE #1
```

```

! Set up the plot on a black background with no axis. Set the label
! color to white so callouts will be visible on the black background.
DIM p AS Plot, s AS PlotPoint
p = graphics.newPlot
p.setBackgroundColor(0, 0, 0)
p.setBorderColor(0, 0, 0)
p.setLabelColor(1, 1, 1)
p.setAxisStyle(2)

! Set up the plot with the star locations for the point cloud data.
s = p.newPlot(xyz)

! Show the stars as a small + sign to make them easier to see.
! Comment out this line for point-like stars.
s.setPointStyle(2)

! Don't connect the stars with lines.
s.setStyle(0)

! Set up tags with the star names. Tapping a star will show a callout
! with the star name.
s.setTags(names)

! Set the star colors.
s.setColors(colors)

! Change the view to center it on our sun, and magnify it a bit.
p.setView3D(0, 0, 0, 10, 10, 10)
p.setTranslation(0, -2)
p.setScale(0.6, 0.6)

! Show the star cloud.
system.showGraphics

```

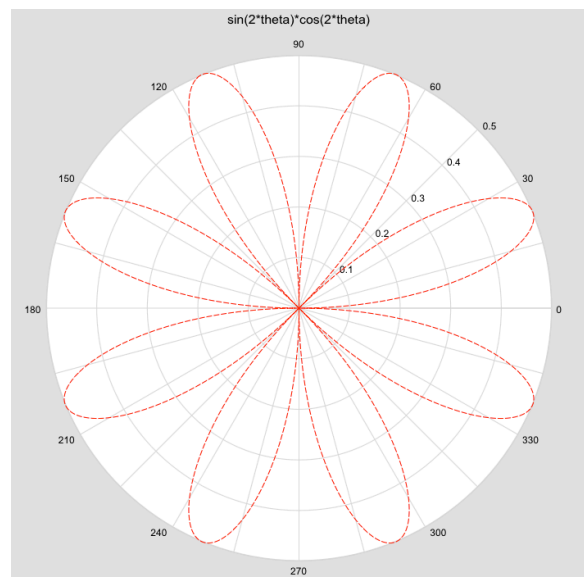
---

**FUNCTION newPolar (FUNCTION f) AS PlotFunction**

Create a new two-dimensional polar coordinate function and add it to the current plot. The function is a function in theta, returning the radius. The default domain for the function is 0.0 to  $2\pi$ . The domain can be set to other values using the `setDomain` method in the returned `PlotFunction` object.

The function is added to the current plot. If this is the first function added, the plot is initialized using Polar coordinates. The radius is established by sampling the function over the domain 0 to  $2\pi$  in 50 evenly distributed samples; the maximum radius encountered is used for the radius of the plot. If the maximum radius encountered is less than zero the radius of the plot is set to one; if it is greater than 10, the radius of the plot is set to 10. The radius can be changed under program control using the `setScale` method, or manually using pinch gestures. The background is set to white, and the border to gray.

The function will not be shown if the axis style is two- or three-dimensional Cartesian. The axis style can be set manually with the `setAxisStyle` method.



The function starts as a solid black line. The color, line type and valid domain for the function can be adjusted using the methods in the returned function object. See the description of the `PlotFunction` class for more about these methods. Characteristics for the plot itself can be set using the calls in this class.

The snippet shows a program that generates the rose pattern seen here. This program is also a sample in the techBASIC app; the sample is called Rose.

#### Snippet

```
! Display the graphics view.
system.showGraphics

! Set up the plot and set the plot title.
DIM p AS Plot
p = graphics.newPlot
p.setTitle("sin(2*theta)*cos(2*theta)")

! Define the function. Set the line color to red
! and the line style to a dashed line.
DIM func AS PlotFunction
func = p.newPolar(FUNCTION f)
func.setColor(1, 0, 0)
func.setStyle(2)
END

FUNCTION f(theta)
f = SIN(2*theta)*COS(2*theta)
END FUNCTION
```

---

#### **FUNCTION newSpherical (FUNCTION f) AS PlotFunction**

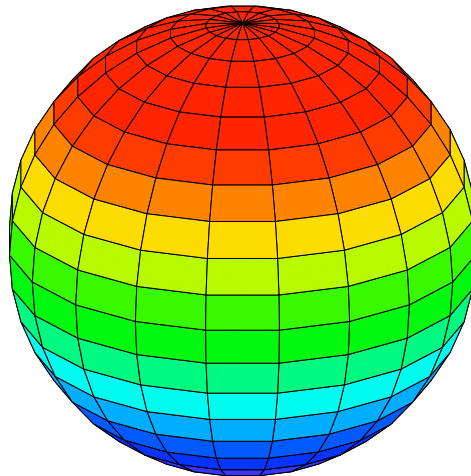
Create a new spherical coordinate function and add it to the current plot. The function is a function in theta and phi, returning the radius. Theta is measured in the X-Y plane, moving counterclockwise from the X axis. Phi is measured from the Z axis, rotating towards the -Z axis. The default domain for theta is 0.0 to  $2\pi$ , and 0 to  $\pi$  for phi. The domain can be set to other values using the `setDomain` method in the returned `PlotFunction` object.

The function is added to the current plot. If this is the first function added, the plot is initialized using three-dimensional Cartesian coordinates. Initially, the range for the axis will be 0 to 10 along each axis; this can be changed with the `setView3D` method. The background is set to white, and the border to gray.

The function will not be shown if the axis style is two-dimensional Cartesian or Polar. The axis style can be set manually with the `setAxisStyle` method.

The function defaults to back lines for the mesh that connects the sampled points, with white polygons filling the mesh. The sample shows the result of setting the mesh style to 3 (false color). This is just one of the many plot characteristics that can be set using the calls in this class.

The snippet shows a program that generates the globe seen here. This program is also a sample in the techBASIC app; the sample is called Ball.



Snippet

```

! Display the graphics view.
system.showGraphics

! Set up the plot. Turn off the axis, hide the border
! and use false color.
DIM p AS Plot
p = graphics.newPlot
p.setBorderColor(1, 1, 1)
p.setSurfaceStyle(3)
p.setAxisStyle(2)

! Plot the function.
DIM func AS PlotFunction
func = p.newSpherical(FUNCTION f)

p.setTranslation3D(0, 0, -5)
p.setTranslation(0, 1.5)
END

FUNCTION f(theta, phi)
f = 5
END FUNCTION

```

**FUNCTION newSurface (x(), y(), z()) AS PlotSurface**

A surface is a collection of points in three-space spread across a two-dimensional surface. The points can take any arbitrary shape.

The coordinates passed as arguments to `newSurface` consist of two-dimensional arrays, each of which must have the same number of subscripts as the other parameters passed. Adjacent points from the arrays are connected to form the surface. Surfaces are similar to meshes, created with the `newMesh` method, but there is a distinct difference. Meshes are functions of X and Y, while surfaces are general three-dimensional point clouds connected together to form a surface that is not, generally speaking, a function in Cartesian coordinates.

If the surface is the first thing added to a plot, the axis is set to three-dimensional Cartesian coordinates. The default range for the axis are set to match the surface, with some expansion possible to keep the aspect ratio between the various axis consistent.

The surface will not be shown if the axis style is two-dimensional Cartesian or Polar. The axis style can be set manually with the `setAxisStyle` method.

The snippet shows the program used to create the torus shown in the illustration.

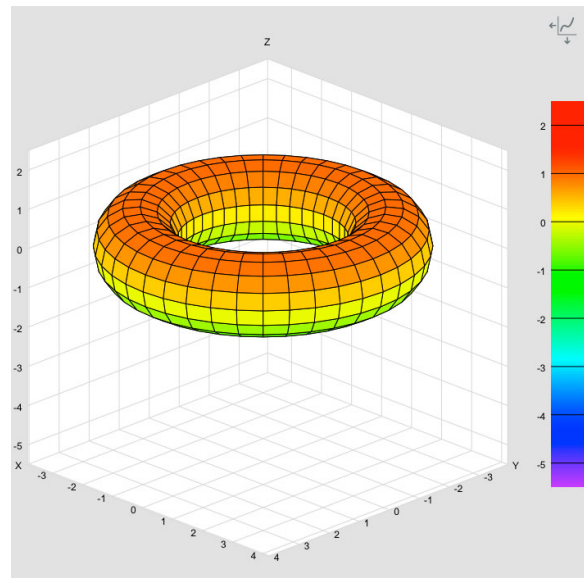
Snippet

```

System.showGraphics

DIM p AS Plot
p = Graphics.newPlot
p.setGridColor(0.85, 0.85, 0.85)
p.setSurfaceStyle(3)
p.setAxisStyle(5)

```



```

hSize% = 41
vSize% = 16
dim x(hSize%, vSize%), y(hSize%, vSize%), z(hSize%, vSize%)
R = 3
a = 1
PI = 3.1415926535

FOR u% = 1 TO hSize%
  u = u%*2*PI/(hSize% - 1)
  FOR v% = 1 TO vSize%
    v = v%*2*PI/(vSize% - 1)
    x(u%, v%) = (R + a*cos(v))*cos(u)
    y(u%, v%) = (R + a*cos(v))*sin(u)
    z(u%, v%) = a*sin(v)
  NEXT
NEXT

DIM surface AS PlotSurface
surface = p.newSurface(x(), y(), z())

```

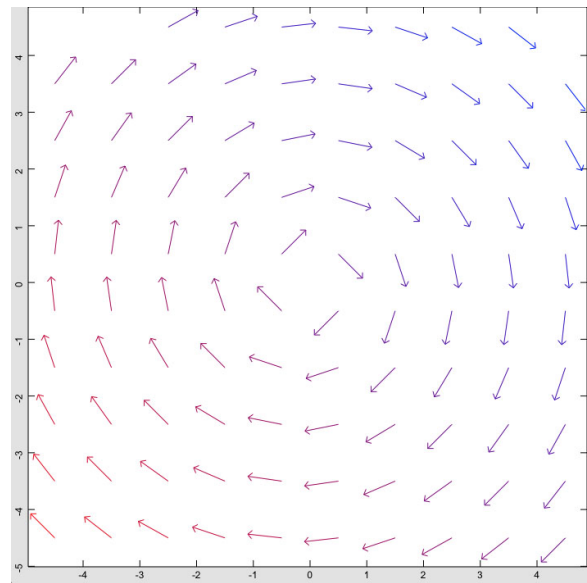
---

**FUNCTION newVectorPlot (matrix(,)) AS PlotVector**

A plot created with `newVectorPlot` consists of a point cloud of vectors representing a vector field. Depending on the array passed, the result can be a two- or three-dimensional plot.

The first subscript of the matrix passed indexes over the vectors to plot. The second subscript indexes over the coordinates.

If the second index of the matrix has a size of four, the vectors are plotted on a two-dimensional Cartesian axis. The first value is the X coordinate of the point, the second value is the Y coordinate, the third value is the X coordinate of the endpoint for the vector, and the fourth value is the Y coordinate of the endpoint for the vector. Each vector is rendered as an arrow with the base at the initial point and the tip of the arrow at the endpoint for the vector. The range of the plot is set based on the values of the vectors passed, with the restriction that the aspect ratio is held at one. The various methods in this class can be used to change the characteristics of the axis, labels and background. The methods in the `PlotVector` class manipulate the vector data, allowing changes to the colors used to draw the vectors and labels used for callouts on the vectors.



If the second subscript has a size of six, the vectors are plotted on a three-dimensional Cartesian axis. The first three values for each point are the X, Y and Z coordinates of the start point for the vector, while the last three are the X, Y and Z coordinates of the endpoint of the vector.

The snippet shows a vector field with directions perpendicular to rays from the center of the plot, much as would be found if plotting the magnetic field perpendicular to a wire running out of the view plane.

**Snippet**

```

DIM p AS Plot, v AS PlotVector
p = Graphics.newPlot

```

```

DIM m(100, 4), colors(100, 3)
FOR x = 1 TO 10
  FOR y = 1 TO 10
    i = (x - 1)*10 + y
    m(i, 1) = x - 5.5
    m(i, 2) = y - 5.5
    a = 0.6/SQR((y - 5.5)*(y - 5.5) + (x - 5.5)*(x - 5.5))
    m(i, 3) = m(i, 1) + a*(y - 5.5)
    m(i, 4) = m(i, 2) + -a*(x - 5.5)

    colors(i, 1) = 1 - x/20 - y/20
    colors(i, 2) = 0
    colors(i, 3) = x/20 + y/20
  NEXT y
NEXT x

v = p.newVectorPlot(m)
v.setColors(colors)

System.showGraphics

```

---

**SUB removeCallout (c AS Callout)**

Removes a callout from the plot.

The callout must have been added previously with `newCallout`. Pass the callout returned by `newCallout` as the parameter.

---

**SUB repaint**

Repaints the plot.

Call this subroutine in programs that use both plots and GUI controls whenever a function or data has changed and the plot needs to be redrawn to reflect the change. This method is never needed in programs that do not use GUI controls to update the plot, since the plot is drawn once after all programmatic changes have already been made.

The snippet shows a program with a slider control that adjusts the amplitude of a sine wave. The repaint command is used to update the plot after the user changes the amplitude using the slider.

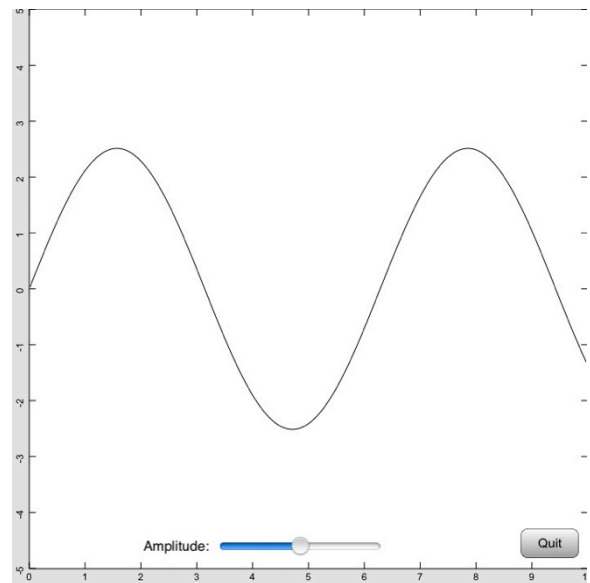
**Snippet**

```

DIM quit AS Button
quit =
Graphics.newButton(Graphics.width - 82,
Graphics.height - 70)
quit.setTitle("Quit")
quit.setBackgroundColor(1, 1, 1)
quit.setGradientColor(0.6, 0.6, 0.6)

DIM amp AS Slider
width = 200
h = (Graphics.width - width)/2
v = Graphics.height - 60
amp = Graphics.newSlider(h, v, width)
amp.setMinValue(0)
amp.setMaxValue(5)
amp.setValue(1)

```



```

DIM ampLabel as Label
ampLabel = Graphics.newLabel(h - 160, v, 150)
ampLabel.setText("Amplitude:")
ampLabel.setAlignment(3)

system.showGraphics

DIM p AS Plot
p = graphics.newPlot

DIM func AS PlotFunction
func = p.newFunction(FUNCTION f)

amplitude = 1.0
END

FUNCTION f(x)
f = amplitude*sin(x)
END FUNCTION

SUB touchUpInside (ctrl AS BUTTON, time AS DOUBLE)
STOP
END SUB

SUB valueChanged (ctrl AS Control, when AS DOUBLE)
IF ctrl = amp THEN
    amplitude = amp.value
    p.repaint
END IF
END SUB

```

---

### **SUB rotate (theta, nu, phi)**

Rotates any three-dimensional Cartesian plot about the axis origin. **Theta** is the angle about the X axis, **nu** about the Y axis, and **phi** about the Z axis. In all cases, the angle is given in radians and the rotation is counterclockwise about the axis when viewed from the positive side of the axis.

---

### **SUB setAllowedGestures (flags AS INTEGER)**

Sets the specific gestures allowed on a plot.

Plots normally allow a number of gestures to resize, rotate and translate the plot. While these gestures are useful in the majority of cases, there are some applications where they get in the way or could cause accidental issues, such as a slider control superimposed on a plot. Missing the slider slightly could move a plot that should remain in a fixed position.

`setAllowedGestures` uses a bit mapped value to set the specific gestures allowed for a plot. By default, all gestures are allowed. Pass a zero for a particular bit to disable the gesture, or a 1 to enable the gesture. For extra safety, pass one for all unused bits, so any future gestures added to the list will default to on unless specifically changed in the program.

For example, to enable panning and translation, but disable rotation and resizing, make the call

```
myPlot.setAllowedGestures($FF1F)
```



Bit	Description	Comments
\$0001	Translation in X	Allows movement of 2D and 3D plots along the X axis.
\$0002	Translation in Y	Allows movement of 2D and 3D plots along the Y axis.
\$0004	Translation in Z	Allows movement of 3D plots along the Z axis.
\$0008	Pan X	Allows panning a 3D plot horizontally across the screen. Ignored for 2D plots.
\$0010	Pan Y	Allows panning a 3D plot vertically across the screen. Ignored for 2D plots.
\$0020	Rotation	Allows rotation of 3D plots. Ignored for 2D plots.
\$0040	Resize Plot	Allows pinching to scale 2D or 3D plots.
\$0080	Resize Axis	Allows pinching to change the size of the axis for a 3D plot. Ignored for 2D plots.
\$0100	Callouts	Allows taps to create new callouts.

**SUB setAspectY (aspectY)**

Sets the current aspect ratio between the values plotted along the Y axis and those plotted along the X axis. An aspect ratio greater than 1 indicates the Y axis is magnified compared to the X axis by the amount returned; a value less than 1 indicates the X axis is magnified by the reciprocal relative to the Y axis.

The default aspect ratio is 1.

The aspect ratio can also be changed by the `setView` command or by manual pinches.

**SUB setAxisFont (name AS STRING, size, style AS INTEGER)**

Sets the font used to draw the axis titles.

See the description of `setFont` in the `Graphics` class for details about using fonts. The snippet for `newFunction` shows an example of this method.

**SUB setAxisKind (kind)**

Adding the first function to a plot automatically sets kind of the axis to the most appropriate one available, but for cases when a different axis is needed, this method allows it to be set manually. The parameter determines which axis kind is used. If a function or point cloud is not valid for the particular axis, it is not drawn.

See `setAxisStyle` for a visual look at these three axis kinds.

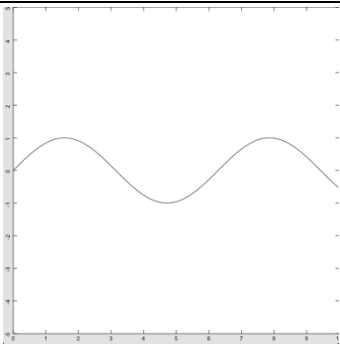
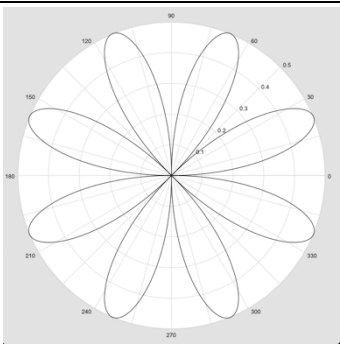
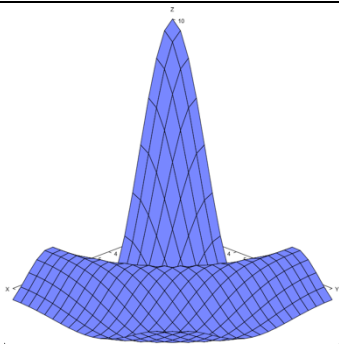
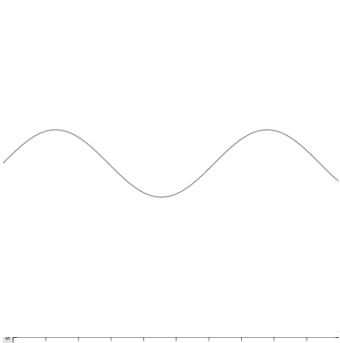
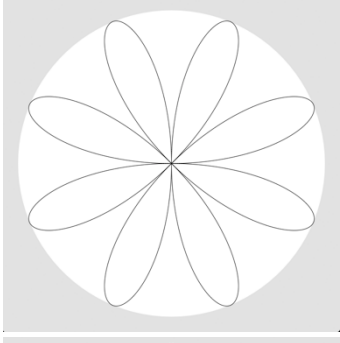
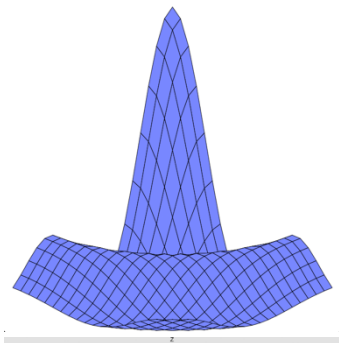
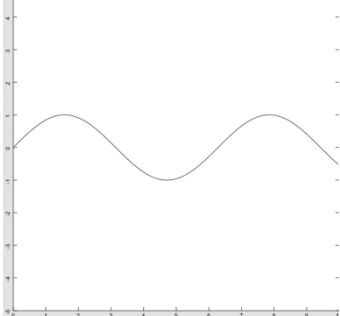
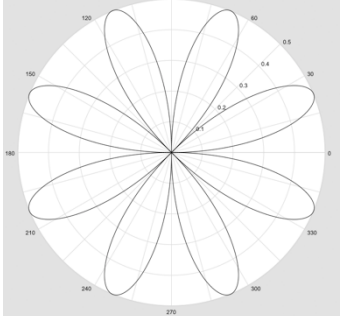
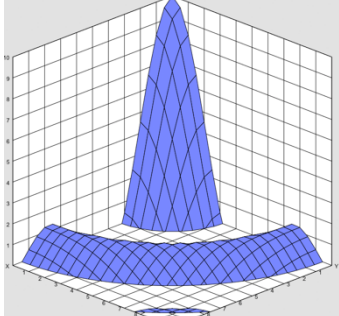
Kind	Description	Comments
1	2D Cartesian	Valid for functions added with <code>newFunction</code> (if they are functions of one variable), <code>newPlot</code> , <code>newPolar</code>
2	Polar	Valid for functions added with <code>newFunction</code> , <code>newPolar</code>
3	3D Cartesian	Valid for functions added with <code>newCylindrical</code> , <code>newFunction</code> (if they are functions of two variables), <code>newMesh</code> , <code>newSpherical</code>

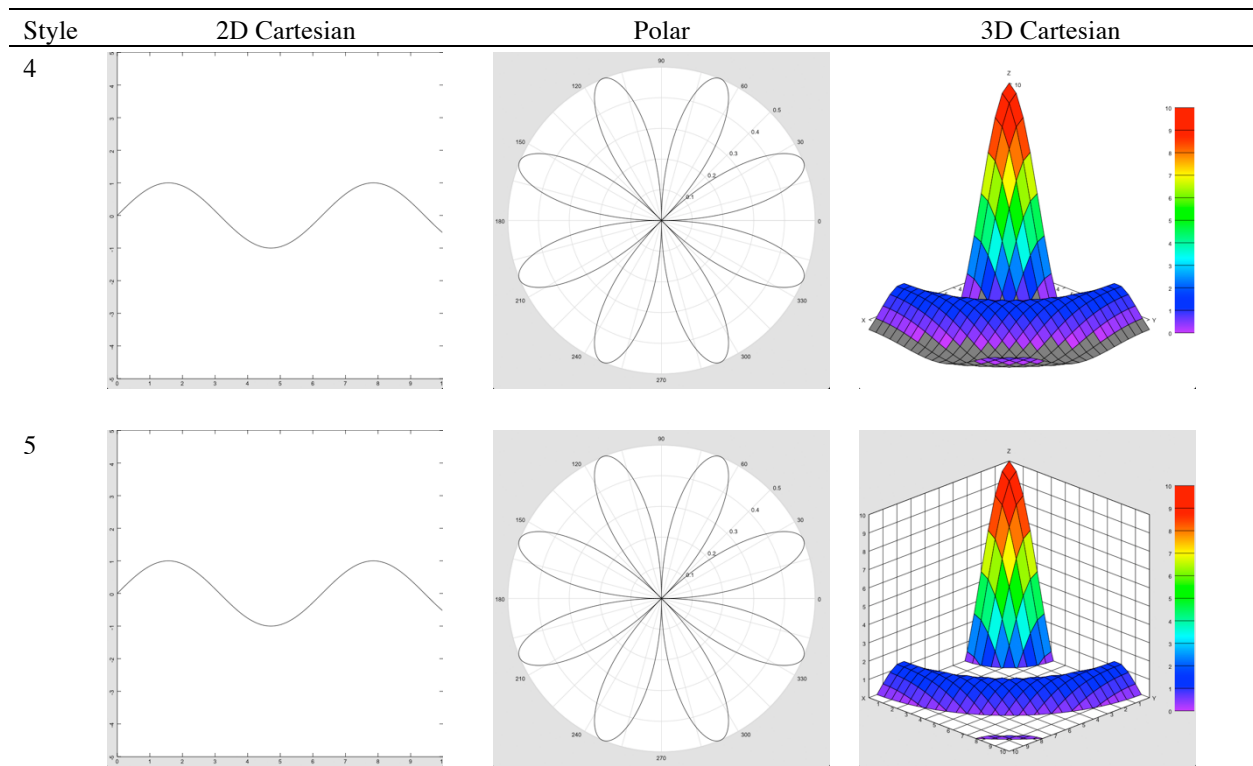
**SUB setAxisStyle (style)**

Sets the axis style to one of the five available styles.

Style	Description	Comments
1	Standard Cartesian or Polar axis.	
2	No axis is drawn.	
3	A box style axis, with solid planes forming three sides of a box.	Only valid for three-dimensional axis.
4	A standard axis with a color legend.	The color legend is only useful for three-dimensional plots where the grid or surface color style is set to 3 (false color).
5	A box axis with a color legend.	

The following table shows the plots that result from using each of these styles with a two-dimensional Cartesian plot created with `newFunction`, a polar coordinate plot created with `newPolar`, and a three-dimensional Cartesian plot created with `newFunction`. Cylindrical and spherical coordinate plots use the same axis shown for three-dimensional Cartesian plots. The border color has been set to white for styles 1, 2 and 4 for the three-dimensional Cartesian plots to emphasize that there is no box; you can also see the lack of a box by examining the behavior of the function as it dips below  $Z=0$ .

Style	2D Cartesian	Polar	3D Cartesian
1			
2			
3			




---

**SUB setBackgroundColor (red, green, blue, alpha = 1)**

Sets the color for the background. The background is the region behind the function, or the box for three-dimensional box-style axis. The default color is white.

See *Colors* at the beginning of the description of the Plot class for some suggested color values and an explanation of the parameters.

---

**SUB setBorderColor (red, green, blue)**

Sets the color for the border. The border is the region around the edge of the plot, behind the axis labels. It defaults to a light gray (0.886, 0.886, 0.886) for two-dimensional plots and three-dimensional plots with a box-style axis, and white for three-dimensional plots with a standard Cartesian axis.

See *Colors* at the beginning of the description of the Plot class for some suggested color values and an explanation of the parameters.

---

**SUB setColorMap (FUNCTION map)**

This method sets the false colors used to display meshes and surfaces in three-dimensional Cartesian plots. This has no effect unless the `setMeshStyle` or `setSurfaceStyle` methods have been used to set one or both of these to render using false color.

When these conditions are met, for each polygon, the renderer finds the smallest X, Y and Z value and passes that value to your custom color function. The function returns a vector with three or four elements; these are the red, green and blue color intensities and the alpha value, respectively. If the vector only has three elements, alpha is assumed to be 1. See *Colors* at the start of the description for this class for a discussion of RGBA colors. The returned color value is used to render the surface or mesh for the given polygon.

If the false color legend is displayed, it is shown for the range of Z values at X = 0, Y = 0. See `setAxisStyle` for a way to hide or display the false color legend.

## Chapter 16: Graphics Classes

The snippet shows a program that displays a function in varying shades of blue, using deep blue for low Z values and white for the largest Z values. The X and Y inputs are ignored. Notice how `minimumZ` and `maximumZ` are used to cause the colors to adjust to the range of displayed values as the plot is moved using swipe gestures.

### Snippet

```
! Display the graphics view.
system.showGraphics

! Set up the plot, label it, and display the function
! with false color.
DIM p AS Plot
p = Graphics.newPlot
p.setGridColor(0.85, 0.85, 0.85)
p.setsurfacestyle(3)
p.setAxisStyle(5)
p.setColorMap(FUNCTION map)

! Add the function.
DIM func AS PlotFunction
func = p.newFunction(FUNCTION f)

! Adjust the function so the portion under the X-Y
! plane is visible, and push it slightly off the Z
! axis so the beginning of the descending curve
! can be seen.
p.setTranslation3D(-4, -3, -2)
p.setScale3D(1, 1, .8)
end

! The function to plot.
FUNCTION f(x, y)
d = SQR(x*x + y*y)
IF d = 0 THEN
    f = 10
ELSE
    f = 10*SIN(d)/d
END IF
END FUNCTION

! Color map function that returns deep blue for low Z
! values, and white for high Z values.
FUNCTION map(x, y, z) (3)
DIM c
IF z < p.minimumZ OR z > p.maximumZ THEN
    c = 0
ELSE
    c = (z - p.minimumZ)/(p.maximumZ - p.minimumZ)
END IF
map = [c, c, 1]
END FUNCTION
```

---

### **SUB setGridColor (red, green, blue)**

Sets the color for the grid that is drawn behind plots.

For two-dimensional Cartesian plots, this is the color of the rectangle surrounding the plot, as well as the hash marks that appear along the axis, or the grid that appears behind the plot if `showGrid` has been used to turn it on. The default color is black.

For polar coordinates, this is the color of the concentric circles and radial lines behind the plot. The default color is a light gray; the actual color has red, green and blue components of 0.85, 0.85, 0.85.

For three-dimensional plots, this is the color of the axis and, if displayed by the selected axis style, the grid extending from the axis. The default color is black.

See *Colors* at the beginning of the description of the Plot class for some suggested color values and an explanation of the parameters.

---

#### **SUB setLabelColor (red, green, blue)**

Sets the color for the axis labels. This includes the numeric values the label the grid, the letters at the end of three-dimensional axis, and any axis labels added with the commands `setXAxisLabel`, `setYAxisLabel` or `setZAxisLabel`.

See *Colors* at the beginning of the description of the Plot class for some suggested color values and an explanation of the parameters.

---

#### **SUB setLabelFont (name AS STRING, size, style AS INTEGER)**

Sets the font used to draw the numbers, axis end cap names for axis.

See the description of `setFont` in the Graphics class for details about using fonts. The snippet for `newFunction` shows an example of this method.

---

#### **SUB setMeshColor (red, green, blue, alpha = 1)**

Sets the color for the mesh used to connect the surface of three-dimensional plots. The default color is black.

The surface mesh can be set to a varying color using `setMeshStyle`.

It is possible to turn the mesh off by setting alpha to 0.0, but a better way is to set the mesh style to transparent using the `setMeshStyle` method.

See *Colors* at the beginning of the description of the Plot class for some suggested color values and an explanation of the parameters.

---

#### **SUB setMeshSize (x, y)**

This method changes the number of polygons used to render surfaces of three-dimensional functions created with the `newFunction` method. The default is 20x20 polygons.

---

#### **SUB setMeshStyle (style)**

This method sets the color style used to render the mesh for three-dimensional functions created with the `newFunction` method and meshes created with the `newMesh` method.

Style	Description
1	Solid color
2	No mesh is drawn
3	Use false color

Solid color meshes are the default. The color defaults to black, but can be changed with the `setMeshColor` method.

Style 2 doesn't draw the mesh at all. The polygons are still drawn unless the `setSurfaceStyle` method is used to turn them off, too.

False color applies a color to the mesh based on the lowest X, Y and Z value for the polygon a mesh segment surrounds. The default is a rainbow hue mesh, as seen in the snippet for the `newMesh` method. This color can be replaced with a custom function using the `setColorMap` method.

---

#### **SUB setRect (x, y, width, height)**

Sets the area the plot occupies on the graphics view. This normally defaults to the entire view for a single plot, or tiled views for multiple plots, as described in *Tiling Plots*, earlier in this chapter. This method overrides the default position, causing the plot to occupy the exact coordinates specified.

---

**SUB setRotation (r(),)**

Sets the rotation matrix used to display three-dimensional Cartesian plots. The rotation rotates the axis itself. Unlike the rotate method, which adds any rotation to the existing rotation, this sets the rotation matrix to a specific value.

The parameter must be a 3x3 matrix.

See any graphics text or classical mechanics text for details on how rotation matrixes are built and used.

**Snippet**

```
! View a 3D plot from the top. Use this command after adding the plot!
p.setRotation([[ 0,  1,  0],
               [ 1,  0,  0],
               [ 0,  0, -1]])
```

---

**SUB setScale (x, y)**

Sets the overall scale for the plot. This is equivalent to a pinch gesture to change the size of the view area.

For two-dimensional Cartesian plots, this method changes the magnification independently in the X and Y directions, adjusting the range and domain of the plot.

For polar plots, the **x** parameter changes the magnification, in effect changing the radius of the plot. The **y** parameter is ignored.

For three-dimensional plots, this method changes the size of the axis, growing or shrinking the entire plot. See `setScale3D` for a way to change the visible part of the range and domain of the function.

---

**SUB setScale3D (x, y, z)**

For three-dimensional Cartesian plots, this method changes the magnification along each axis, in effect changing the visible part of the range and domain displayed along the axis. This method is ignored for two-dimensional Cartesian plots and polar plots.

---

**SUB setSurfaceColor (red, green, blue, alpha = 1.0)**

Sets the color for the polygons used to form the surface of three-dimensional plots. The default color is white.

The surface color can be set to a varying color using `setSurfaceStyle`.

It is possible to turn the surface off by setting alpha to 0.0, but a better way is to set the surface style to transparent using the `setSurfaceStyle` method.

See *Colors* at the beginning of the description of the Plot class for some suggested color values and an explanation of the parameters.

---

**SUB setSurfaceStyle (style)**

This method sets the color style used to render the polygons for three-dimensional functions created with the `newFunction` method and meshes created with the `newMesh` method.

Style	Description
1	Solid color
2	No mesh is drawn
3	Use false color

Solid color surfaces are the default. The color defaults to white, but can be changed with the `setSurfaceColor` method.

Style 2 doesn't draw the polygons at all. The mesh is still drawn unless the `setMeshStyle` method is used to turn them off, too. Using transparent surfaces reveals the mesh lines normally hidden by the frontmost polygons.

False color applies a color to the mesh based on the lowest X, Y and Z value for the polygon a mesh segment surrounds. The default is a rainbow hue, as seen in many snippets, including the one for the `newFunction` method. This color can be replaced with a custom function using the `setColorMap` method.

---

**SUB setTitle (title AS STRING)**

Sets the title displayed at the top of the plot. The default title is an empty string. See the snippet for `newFunction`, among others, for an example of this method.

---

**SUB setTitleFont (name AS STRING, size, style AS INTEGER)**

Sets the font used to draw the title.

See the description of `setFont` in the `Graphics` class for details about using fonts. The snippet for `newFunction` shows an example of this method.

---

**SUB setTranslation (x, y)**

Sets the overall location for the plot. This is equivalent to a swipe gesture to change the area viewed.

For two-dimensional Cartesian plots and polar plots, this method changes the location independently in the X and Y directions.

For three-dimensional plots, this method moves the axis. See `setTranslation3D` for a way to change the visible part of the range and domain of the function.

---

**SUB setTranslation3D (x, y, z)**

For three-dimensional Cartesian plots, this method translates the plot along each axis, in effect changing the visible part of the range and domain displayed along the axis. This method is ignored for two-dimensional Cartesian plots and polar plots.

---

**SUB setView (minX, minY, maxX, maxY, maintainAspect)**

Sets the minimum and maximum values displayed along a two-dimensional Cartesian axis. The default values are

Parameter	Default
<code>minX</code>	0.0
<code>minY</code>	-5.0
<code>maxX</code>	10.0
<code>maxY</code>	5.0

The view can also be changed using the `setAspect` method, by placing point cloud data in the plot as the first function, or manually using pinch and swipe gestures.

**maintainAspect** determines whether the Y axis will be adjusted to keep a one-to-one correspondence between the scale of the X and Y axis. Pass zero value to set the axis to the exact values specified, or a non-zero value to adjust the Y-axis to maintain an aspect ratio of 1.

This method also changes the view for polar coordinates, but in that case, **maxX** is the only value used. It sets the radius of the plot.

---

**SUB setView3D (minX, minY, minZ, maxX, maxY, maxZ)**

Sets the minimum and maximum values displayed along a three-dimensional Cartesian axis. The default values are 0.0 for each minimum and 10.0 for each maximum.

The view can also be changed by placing point cloud data in the plot as the first function, or manually using pinch and swipe gestures.

---

**SUB setXAxisLabel (label AS STRING)**

Sets the text displayed along the X axis. The default is the empty string, which displays no text.

See `setLabelColor` and `setAxisFont` for methods that change the appearance of the axis labels.

---

**SUB setYAxisLabel (label AS STRING)**

Sets the text displayed along the Y axis. The default is the empty string, which displays no text.

See `setLabelColor` and `setAxisFont` for methods that change the appearance of the axis labels.

---

**SUB `setZAxisLabel` (label AS STRING)**

Sets the text displayed along the Z axis. The default is the empty string, which displays no text.

This method has no effect if the plot is a two-dimensional Cartesian plot or a polar plot.

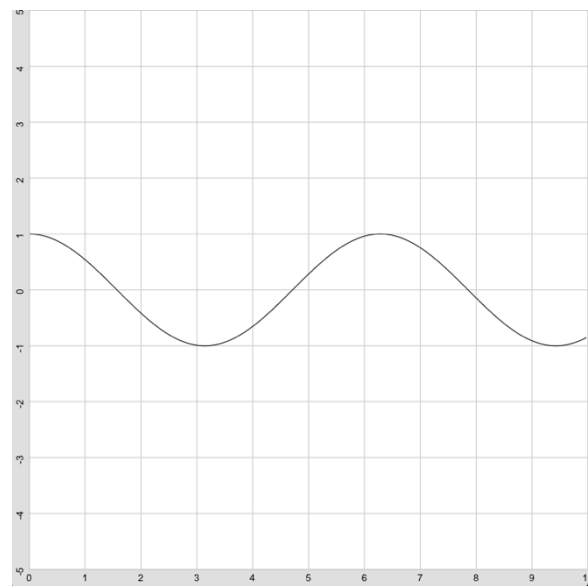
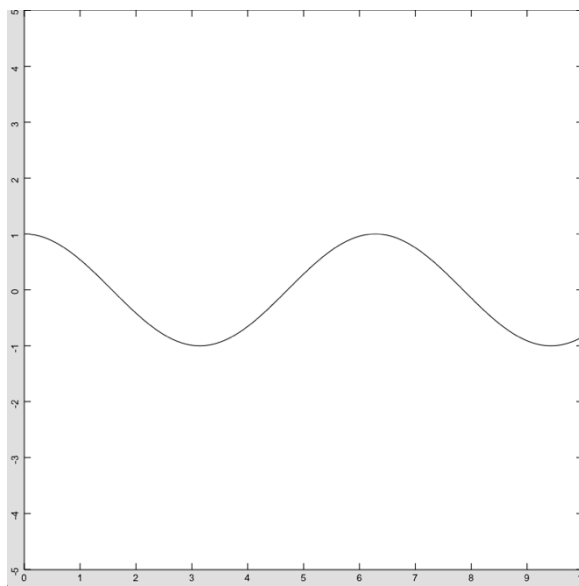
See `setLabelColor` and `setAxisFont` for methods that change the appearance of the axis labels.

---

**SUB `showGrid` (show AS INTEGER)**

The default axis for two- and three-dimensional Cartesian plots normally show a small line near each numeric value used to label an axis. This method can be used to turn on lines that extend across the background of the plot (pass a non-zero value) or to show just the sort guide lines (pass a zero).

The plot on the left is the default, while the image on the right shows the effect of passing a non-zero value to `showGrid`. In the example of the right, the grid color has also been changed to 80% gray using `setGridColor`.




---

**FUNCTION `translationX`**

Returns the current translation of the plot in the X direction. See `setTranslation` for a discussion of translation.

---

**FUNCTION `translationY`**

Returns the current translation of the plot in the Y direction. See `setTranslation` for a discussion of translation.

---

## PlotFunction

`PlotFunction` objects are returned by the `newFunction` method of the `Plot` class. The methods in this class are used to manipulate the function after it has been created. See the `newFunction` method for samples using this class.

---

**SUB `setColor` (red, green, blue)**

For two-dimensional plots and polar plots, this method sets the line color used to display the function. The default color is black. The `setMeshColor`, `setFalseColor`, `setMeshStyle`, `setSurfaceColor` and



`setSurfaceStyle` methods of the `Plot` function are used to set colors for three-dimensional Cartesian plots. For three-dimensional plots, the color set by this method is ignored.

See *Colors* at the start of the discussion of the `Plot` class for information on RGB colors.

#### **SUB setDomain (minX, maxX, minY = 0, maxY = 10)**

Sets the domain for functions. The domain is normally unlimited; the plot evaluates different parts of a function depending on what the plot is currently displaying. This can change as the plot is panned using swipe gestures. Some functions are not valid for all inputs, though. This method can be used to restrict the domain the plot will evaluate for display. Areas outside of the specified domain are not drawn.

The default domain for Cartesian plots is the range of floating point numbers, which is approximately  $-1e37$  to  $1e37$ . The default domain for polar functions is 0 to  $2\pi$ .

It is not necessary to set the domain to avoid math errors. If the result of evaluating the function is NaN (not a number), the value is also not plotted.

For functions of one variable drawn on either a two-dimensional Cartesian axis or a polar axis, **minX** and **maxX** control the domain. In the case of polar plots, **minX** is the lowest value for the angle, while **maxX** is the highest value for the angle. For functions of two variables displayed on a three-dimensional Cartesian axis, **minY** and **maxY** are also used; these parameters are ignored for two-dimensional plots.

#### Snippet

```
! Set the domain for the fourth root to positive values only.
DIM p AS Plot
p = Graphics.newPlot
DIM func AS PlotFunction
func = p.newFunction(FUNCTION root4)
func.setDomain(0, 1E30)
end

! The function to plot.
FUNCTION root4(x)
  root4 = SQR(SQR(x))
END FUNCTION
```

#### **SUB setFillColor (red, green, blue, alpha = 1)**

Sets the fill color, used to color the area between the plot and the X axis. The default fill color is transparent.

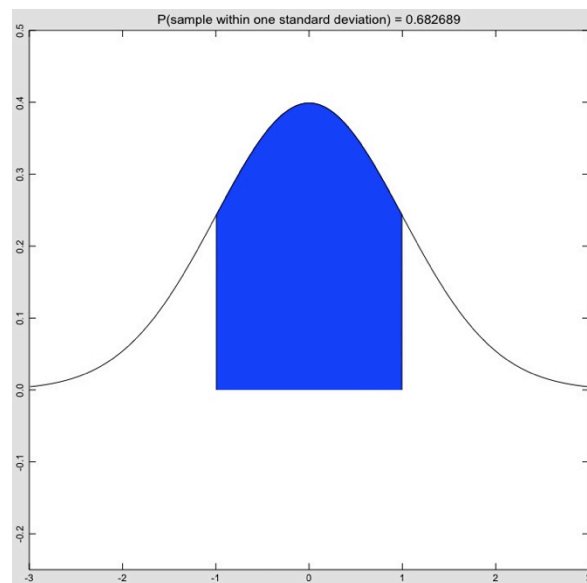
The fill color is typically used to shade an area under a curve. The snippet shades the area evaluated by an integral, so both the geometric and numeric solution are shown.

See *Colors* at the start of the discussion of the `Plot` class for information on RGB colors.

#### Snippet

```
! Find the probability that a
! sample from a normally distributed
! population will fall within one
! standard deviation of the mean.
intfx = Math.romb(function f, -1, 1)

DIM p AS Plot, pf AS PlotFunction,
pfi AS PlotFunction
p = Graphics.newPlot
pf = p.newFunction(FUNCTION f)
pfi = p.newFunction(FUNCTION f)
pfi.setDomain(-1, 1)
pfi.setFillColor(0, 0, 1)
p.setView(-3, -0.25, 3, 0.5, 0)
p.setTitle("P(sample within one
standard deviation) = " & STR(intfx))
System.showGraphics
```



```

FUNCTION f (x AS DOUBLE) AS DOUBLE
f = EXP(-x*x/2)/SQR(2*Math.PI)
END FUNCTION

```

---

**SUB setStyle (style AS INTEGER)**

Sets the style used to draw the line. The available styles are shown in the table, along with blow-ups of the styles in the image. The default line style is 1 (solid line).

Style	Description
0	no line
1	solid line
2	dashed line
3	dotted line
4	dashes and dots




---

## PlotMesh

PlotMesh objects are returned by the newMesh method of the Plot class. The mesh characteristics are controlled by the Plot class. See the newMesh method for samples using this class.

---

**SUB setMesh (x(), y(), z(),)**

Replaces the points in the mesh.

This method is typically used when a plot is updated due to the use of controls to change some of the input parameters to the equations used to generate the mesh. In that case, the points are occasionally updated, replaced by more recent data. Be sure to call repaint in the Plot that is displaying the mesh after updating the points.

See newMesh in the Plot class for a complete discussion of the parameters used to define a mesh.

---

## PlotPoint

PlotPoint objects are returned by the newPlot method of the Plot class. The methods in this class are used to manipulate the plot after it has been created. See the newPlot method for samples using this class.

---

**SUB setColor (red, green, blue)**

Sets the color for any lines used to connect the points in the plot.

See Colors at the start of the discussion of the Plot class for information on RGB colors.

---

**SUB setColors (colors(),)**

Sets the color for the points plotted by this object.

The array should have one row for each point in the plot. Each row has three elements, one each for the red, green and blue values for the color, with each value ranging from 0.0 to 1.0. As the points are plotted, a value is extracted from the array and used as the color for the point.

See Colors at the start of the discussion of the Plot class for information on RGB colors.

See the snippet for newPlot in the Graphics class for an example that uses this method.

---

**SUB setPointColor (red, green, blue)**

Sets the color for any points and error bars shown in the plot.

See Colors at the start of the discussion of the Plot class for information on RGB colors.

---

**SUB setPoints (matrix(,))**

Replaces the points in the plot.

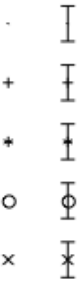
This method is typically used when a plot is showing data as the data is collected. In that case, the points are occasionally updated, replaced by more recent data. See the `accel`, `gyro` and `mag` methods in the `Sensors` class for sample programs that put this ability to good use.

---

**SUB setPointStyle (style AS INTEGER)**

Sets the style used to draw the points in the plot. The point styles are listed in the table and shown in the image. The left column of the image shows the point style without an error bar, while the right column shows the point style with an error bar. The default style is 1 (a dot).

Style	Description
1	dot
2	plus
3	asterisk
4	circle
5	X

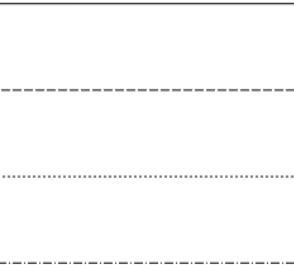


---

**SUB setStyle (style AS INTEGER)**

Sets the style used to draw any line connecting the points. The available styles are shown in the table, along with blow-ups of the styles in the image. The default line style is 1 (solid line).

Style	Description
0	no line
1	solid line
2	dashed line
3	dotted line
4	dashes and dots



---

**SUB setTags (tags() AS STRING)**

Sets the text used for callouts for each point in the plot. **tags** is a one-dimensional array of strings containing the tags for each point, respectively. It is not required to pass a tag for each point, and any extras will be ignored. If this method is not used, tapping on a point shows its coordinates. If this method is used, tapping a point shows a callout with the corresponding message.

See the Stars example in the description of `newPlot` for a sample that uses this feature to display the proper name for a star in a three-dimensional plot of nearby stars.

---

# PlotSurface

`PlotSurface` objects are returned by the `newSurface` method of the `Plot` class. The surface characteristics are controlled by the `Plot` class. See the `newSurface` method for samples using this class.

---

**SUB setSurface (x(,), y(,), z(,))**

Replaces the points in the surface.

This method is typically used when a plot is updated due to the use of controls to change some of the input parameters to the equations used to generate the surface. In that case, the points are occasionally updated, replaced by more recent data. Be sure to call `repaint` in the `Plot` that is displaying the surface after updating the points.

See `newSurface` in the `Plot` class for a complete discussion of the parameters used to define a surface.

---

## PlotVector

`PlotVector` objects are returned by the `newVectorPlot` method of the `Plot` class. The methods in this class are used to manipulate the plot after it has been created. See the `newVectorPlot` method for samples using this class.

---

### **SUB setColor (red, green, blue)**

Sets the color for the vectors plotted by this object.

See *Colors* at the start of the discussion of the `Plot` class for information on RGB colors.

---

### **SUB setColors (colors(),)**

Sets the color for the vectors plotted by this object.

The array should have one row for each point in the plot. Each row has three elements, one each for the red, green and blue values for the color, with each value ranging from 0.0 to 1.0. As the vectors are plotted, a value is extracted from the array and used as the color for the vectors.

See *Colors* at the start of the discussion of the `Plot` class for information on RGB colors.

See the snippet for `newVectorPlot` in the `Graphics` class for an example that uses this method.

---

### **SUB setPoints (matrix(),)**

Replaces the vectors in the plot.

This method is typically used when a plot is showing data as the data is collected. In that case, the vectors are occasionally updated, replaced by more recent data.

---

### **SUB setTags (tags() AS STRING)**

Sets the text used for callouts for each vector in the plot. **tags** is a one-dimensional array of strings containing the tags for each vector, respectively. It is not required to pass a tag for each point, and any extras will be ignored. If this method is not used, tapping on a vector shows its coordinates. If this method is used, tapping a vector shows a callout with the corresponding message.

## Chapter 17 – GUI Classes

Graphical user interface (GUI) classes are used to create and manipulate controls that appear on the graphics view. This allows the creation of programs that allow the user to interact with the program using the same controls found in apps downloaded from the app store.

---

### Activity

Activity objects are used to indicate that the program is busy with a task that may take some time, but the amount of progress cannot be easily determined.

Activity objects are returned by the `newActivity` method of the `Graphics` class. The methods in this class are used to manipulate the indicator after it has been created.

Activity descends from the `Control` class. All methods in the `Control` class also apply to the `Activity` class.

The snippet shows a complete program that creates an activity indicator along with buttons that can start and stop it.

#### Snippet

```
! Create an activity indicator with the
! buttons to control it.
DIM act AS Activity
DIM startButton AS Button, stopButton AS Button

act = Graphics.newActivity(20, 20)
act.setStyle(2)
act.setColor(1, 0, 0)

startButton = Graphics.newButton(70, 10)
startButton.setTitle("Start")

stopButton = Graphics.newButton(160, 10)
stopButton.setTitle("Stop")

! Create the Quit button.
DIM quit AS Button
quit = Graphics.newButton(10, 80)
quit.setTitle("Quit")

System.showGraphics
END

SUB touchUpInside (ctrl AS Button, when AS DOUBLE)
IF ctrl = quit THEN
    STOP
ELSE IF ctrl = startButton THEN
    act.startAnimation
ELSE IF ctrl = stopButton THEN
    act.stopAnimation
END IF
END SUB
```



---

**SUB setColor (red, green, blue, alpha = 1, state AS INTEGER = 1)**

Sets the color for the indicator.

Note that setting the style with `setStyle` also sets the color. To change both the style, which can change the size, and the color, set the style first.

See *Colors* at the start of the discussion of the `Plot` class for information on RGB colors.

---

**SUB setStyle (style AS INTEGER)**

Sets the style of the activity indicator. The default style is a small, white activity indicator.

The style parameter should be a number from 1 to 3, selecting one of the styles shown in the table. If the value is not 1, 2 or 3, the call is ignored.

Style	Description
1	A small white activity indicator.
2	A large white activity indicator.
3	A small gray activity indicator.

---

**SUB startAnimation**

Starts the activity indicator. Since the activity indicator starts by default when it is created, the only reason to use this call is to restart an activity indicator that has been stopped with the `stopAnimation` call.

---

**SUB stopAnimation**

Stops the activity indicator. This hides it, but it can be restarted with a call to `startAnimation`.

---

## Annotation

`Annotation` objects are used to show locations on a `MapView`. They are created by the `newAnnotation` method of the `MapView` class. The methods in this class are used to manipulate the annotation after it has been created.

---

**SUB setLocation (latitude AS DOUBLE, longitude AS DOUBLE)**

Changes the location of the annotation to the latitude and longitude indicated.

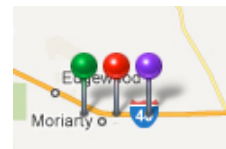
---

**SUB setPinColor (color AS INTEGER)**

Sets the pin color for an annotation.

Annotations are rendered as a map pin with colored head. This method sets the pin color to one of the three predefined colors. The default color is green.

Apple has designated specific uses for each of the predefined pin colors. These are noted in the table.

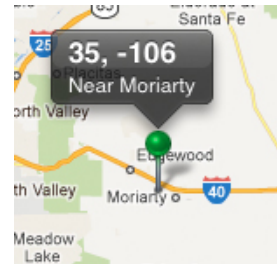


color	Color of the pin head
1	Green; intended for starting locations
2	Red; intended for destinations
3	Purple; intended for user-specified locations

**SUB setSubtitle (title AS STRING)**

Sets the subtitle for an annotation.

The subtitle appears below the title when the annotation is selected. The annotation can be selected either by a tap from the user or using the `selectAnnotation` call in the `MapView` class.

**SUB setTitle (title AS STRING)**

Sets the title for an annotation.

The title appears when the annotation is selected. The annotation can be selected either by a tap from the user or using the `selectAnnotation` call in the `MapView` class.

Snippet

```
DIM map AS MapView, pin AS Annotation
map = Graphics.newMapView(10, 10, 380, 240)
map.setMapRect(35.5, -106.8, 1, 1)

pin = map.newAnnotation(35, -106.1, 1, 2)
pin.setTitle("35, -106")
pin.setSubtitle("Near Moriarty")
map.selectAnnotation(pin)
```

## Button

`Button` objects display on the graphics view as a simple push button. They are used to handle actions that happen immediately after tapping the button, such as quitting the program or performing an operation in a calculator.

`Button` objects are returned by the `newButton` method of the `Graphics` class. The methods in this class are used to manipulate the button after it has been created.

`Button` descends from the `Control` class. All methods in the `Control` class also apply to the `Button` class.

Tapping a button or sliding into a button and releasing the touch inside the button both generate a touch up inside event. If the program has a subroutine named `touchUpInside` with appropriate parameters, it is called to handle the event. From there, the program can detect the button that was pressed by examining the title of the button or comparing the button parameter of the `touchUpInside` subroutine to the various buttons in the program.

See Chapter 13 for a complete discussion of application methods like `touchUpInside`.

**SUB loadImage (path AS STRING, state AS INTEGER = 1, resize AS INTEGER = 1, maintainAspect AS INTEGER = 0)**

Load an image from a file.

The image specified by `path` is loaded and used as the button image.

There are up to four images associated with a button, corresponding to the four states for a button. If no state parameter is passed, the image is used for the normal button state. If the state parameter is passed, it can be a value from 1 to 4; any other value is ignored. The values correspond to the following button states:

State	Description
1	Normal. If no other images are specified, this image is also used for the other button states.
2	Highlighted. A button is highlighted when a tap is in progress. If an image is specified for the normal button state, but not for the highlighted state, the normal button image is overlaid with a bright highlight spot.
3	Disabled. A button can be disabled with a call to <code>setEnabled</code> .
4	Selected. A button can be selected with a call to <code>setSelected</code> .

Buttons are normally resized to fit the image. To prevent the button from being resized, specify 0 for the **resize** parameter. If the button is not resized, the image will be resized to fit the available button area.

Setting the **maintainAspect** parameter to 1 also resizes the image, but the aspect ratio is maintained. If this causes blank areas around the image, the blank areas are filled with the button's background color.

The currently supported image formats for a button are:

Extension	Format
.tiff, .tif	Tagged Image File Format (TIFF)
.jpg, .jpeg	Joint Photographic Experts Group (JPEG)
.gif	Graphic Interchange Format (GIF)
.png	Portable Network Graphic (PNG)
.bmp, .BMPf	Windows Bitmap Format (DIB)
.ico	Windows Icon Format
.cur	Windows Cursor
.xbm	XWindow bitmap

#### Snippet

```
! Create a button with three colors of a logo so
! it appears blue normally, green when pressed, and
! red when disabled.
DIM statelyButton AS Button
statelyButton = Graphics.newButton(10, 10)
statelyButton.loadImage("BlueGears.png")
statelyButton.loadImage("GreenGears.png", 2)
statelyButton.loadImage("RedGears.png", 3, 0)
```



#### SUB setBackgroundColor (red, green, blue, alpha = 1, state AS INTEGER = 1)

Sets the background color for a button.

The background color is used to fill the button shape, or as one of the two colors used for a gradient. Use `setGradientColor` and, optionally, `setGradient` to create a button gradient. The default background color is white. The button border and title colors are set using `setColor`.

The background color can be set explicitly for each of the four button states. If no state is specified, the color applies to the normal button state. See `loadImage` for a description of the four button states.

See *Colors* at the start of the discussion of the `Plot` class for information on RGB colors.

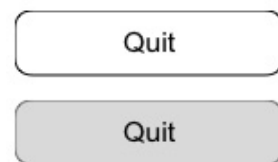
#### Snippet

```
! Create two Quit buttons, one white and one with
! a light gray background.
DIM quit AS Button, grayQuit AS Button
quit = Graphics.newButton(10, 10, 150)
quit.setTitle("Quit")

grayQuit = Graphics.newButton(10, 60, 150)
grayQuit.setTitle("Quit")
grayQuit.setBackgroundColor(0.85, 0.85, 0.85)

System.showGraphics
END

SUB touchUpInside (ctrl AS Button, when AS DOUBLE)
STOP
END SUB
```





---

**SUB setColor (red, green, blue, alpha = 1, state AS INTEGER = 1)**

Sets the title color for a button.

See *Colors* at the start of the discussion of the `Plot` class for information on RGB colors.

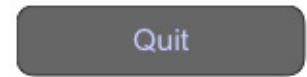
The title color can be set explicitly for each of the four button states. If no state is specified, the color applies to the normal button state. See `loadImage` for a description of the four button states.

**Snippet**

```
! Create a gray quit button with light blue text.
DIM quit AS Button
quit = Graphics.newButton(10, 10, 150)
quit.setTitle("Quit")
quit.setBackgroundColor(0.4, 0.4, 0.4)
quit.setColor(0.8, 0.8, 1)
```

```
System.showGraphics
END
```

```
SUB touchUpInside (ctrl AS Button, when AS DOUBLE)
STOP
END SUB
```



---

**SUB setFont (name AS STRING, size, style AS INTEGER)**

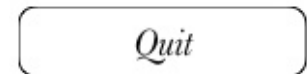
Sets the font for the button title. See `setFont` in the `Graphics` class for a discussion of fonts.

**Snippet**

```
! Create a button with a stylish font.
DIM quit AS Button
quit = Graphics.newButton(10, 10, 150)
quit.setTitle("Quit")
quit.setFont("Baskerville", 20, 2)
```

```
System.showGraphics
END
```

```
SUB touchUpInside (ctrl AS Button, when AS DOUBLE)
STOP
END SUB
```



---

**SUB setGradient (x1, y1, x2, y2)**

Sets the start and end points for a background color gradient. Both the background color and the gradient color must be set, or this call will have no effect.

The first point specifies the relative location where the background color will appear, while the second point specifies the relative location where the gradient color will appear. Intermediate colors are blended using a linear gradient from one point to the other, with solid colors appearing beyond the specified points. The location 0,0 is at the top left of the button, while 1,1 is at the lower right of the button. The default gradient starts at 0.5,0 and ends at 0.5,1.

The gradient can be set explicitly for each of the four button states. If no state is given the gradient applies to the default button state. See `loadImage` for a description of the four button states.

Snippet

```
! Create two buttons, the top one with a default
! gradient location, and the bottom one with a
! slanted gradient.
DIM gradient1 AS Button, gradient2 AS Button
gradient1 = Graphics.newButton(10, 10, 150)
gradient1.setBackgroundColor(1, 1, 1)
gradient1.setGradientColor(1, 1, 0)

gradient2 = Graphics.newButton(10, 60, 150)
gradient2.setBackgroundColor(1, 1, 1)
gradient2.setGradientColor(1, 0, 0)
gradient2.setGradient(0.4, 0, 0.6, 1)

System.showGraphics
END

SUB touchUpInside (ctrl AS Button, when AS DOUBLE)
STOP
END SUB
```



---

**SUB setGradientColor (red, green, blue, alpha = 1, state AS INTEGER = 1)**

Sets the gradient color for a button.

See `setGradient` for a description and example of gradients.

See *Colors* at the start of the discussion of the `Plot` class for information on RGB colors.

The gradient color can be set explicitly for each of the four button states. If no state is specified, the color applies to the normal button state. See `loadImage` for a description of the four button states.

---

**SUB setImage (theImage AS Image, state AS INTEGER = 1, resize AS INTEGER = 1)**

Sets the button image using a preloaded image.

See `loadImage` for a description of how images are used on a button, the four button states controlled by the state parameter, and a complete description of the `resize` and `maintainAspect` parameters.

See the `Image` class for information on creating and manipulating images used by this call.

---

**SUB setStyle (style AS INTEGER = 1)**

Sets the button outline style for a button that has no image. The default style is rounded rectangle.

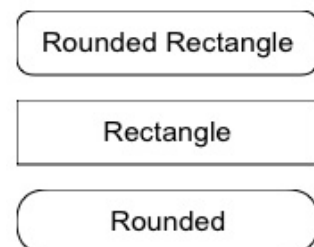
Style	Shape
1	Rounded rectangle
2	Rectangle
3	Rounded

Snippet

```
! Create three buttons, one using each of the
! available button styles.
DIM rrect AS Button, rect AS Button
DIM rounded AS Button

rrect = Graphics.newButton(10, 10, 170)
rrect.setTitle("Rounded Rectangle")

rect = Graphics.newButton(10, 60, 170)
rect.setStyle(2)
rect.setTitle("Rectangle")
```



```

rounded = Graphics.newButton(10, 110, 170)
rounded.setStyle(3)
rounded.setTitle("Rounded")

System.showGraphics
END

SUB touchUpInside (ctrl AS Button, when AS DOUBLE)
STOP
END SUB

```

---

**SUB setTitle (title AS STRING, state AS INTEGER = 1)**

Sets the title of the button. The title is the text that appears on buttons that do not have a button image.

The title can be set explicitly for each of the four button states. If no state parameter is specified, the title applies to the default state. See `loadImage` for a description of the four button states.

**Snippet**

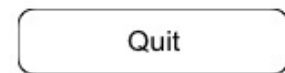
```

! Create a Quit buttons.
DIM quit AS Button
quit = Graphics.newButton(10, 10, 150)
quit.setTitle("Quit")

System.showGraphics
END

SUB touchUpInside (ctrl AS Button, when AS
DOUBLE)
STOP
END SUB

```




---

**FUNCTION title (state AS INTEGER = 1) AS STRING**

Returns the title of the button.

The title can be set explicitly for four button states. This call can return the title for each of the four states.

See `loadImage` for a description of the four button states.

---

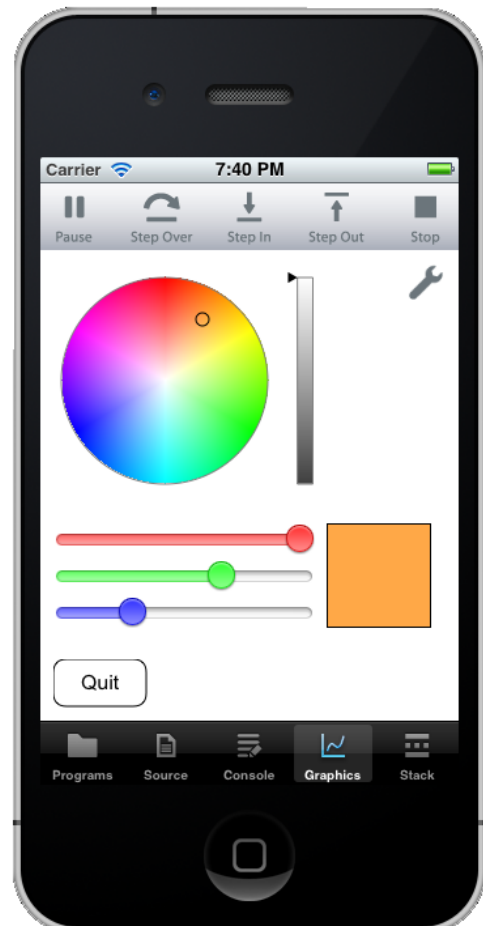
## ColorPicker

`ColorPicker` objects display a color wheel and luminosity slider, allowing colors to be selected using touches and swipes.

`ColorPicker` objects are returned by the `newColorPicker` method of the `Graphics` class. The methods in this class are used to manipulate the color picker after it has been created.

`ColorPicker` descends from the `Control` class. All methods in the `Control` class also apply to the `ColorPicker` class.

Selecting a new color by dragging the circle on the color wheel or the thumb on the luminosity slider generates a value changed event, which is handled by a call to the program's `valueChanged`



subroutine. The red, green and blue methods in this class can then be used to determine the selected color.

See Chapter 13 for a complete discussion of application methods like valueChanged.

The snippet shows a complete program that displays a color picker. Three sliders under the color picker can also be used to select a color; these set the color in the color picker, illustrating how to initialize a color. The color swatch button to the right of the sliders shows the selected color, whether it is selected with the color picker or the sliders.

### Snippet

```
DIM cp AS ColorPicker
cp = Graphics.newColorPicker(10, 10)

DIM swatch AS Label, backGround AS Label
backGround = Graphics.newLabel(219, 209, 80, 80)
backGround.setBackgroundColor(0, 0, 0)
swatch = Graphics.newLabel(220, 210, 78, 78)

DIM rSlider AS Slider, gSlider AS Slider, bSlider AS Slider

rSlider = Graphics.newSlider(10, 210, 200)
rSlider.setMinColor(1, 0.3, 0.3)
rSlider.setThumbColor(1, 0, 0)
rSlider.setValue(1)

gSlider = Graphics.newSlider(10, 238, 200)
gSlider.setMinColor(0.3, 1, 0.3)
gSlider.setThumbColor(0, 1, 0)
gSlider.setValue(1)

bSlider = Graphics.newSlider(10, 266, 200)
bSlider.setMinColor(0.3, 0.3, 1)
bSlider.setThumbColor(0, 0, 1)
bSlider.setValue(1)

DIM quitButton AS Button
quitButton = Graphics.newButton(10, Graphics.height - 47)
quitButton.setTitle("Quit")

System.showGraphics

SUB touchUpInside (ctrl AS Button, time AS DOUBLE)
STOP
END SUB

SUB valueChanged (ctrl AS Control, time AS DOUBLE)
IF ctrl = cp THEN
    rSlider.setValue(cp.red)
    gSlider.setValue(cp.green)
    bSlider.setValue(cp.blue)
ELSE IF ctrl = rSlider OR ctrl = gSlider OR ctrl = bSlider THEN
    cp.setColor(rSlider.value, gSlider.value, bSlider.value)
END IF
swatch.setBackgroundColor(cp.red, cp.green, cp.blue)
END SUB
```

---

### **FUNCTION blue**

Returns the blue component of the currently selected color as a value from 0.0 to 1.0.

**FUNCTION green**

Returns the green component of the currently selected color as a value from 0.0 to 1.0.

**FUNCTION red**

Returns the red component of the currently selected color as a value from 0.0 to 1.0.

**SUB setColor (red, green, blue)**

Sets the selected color, adjusting the luminosity thumb and color wheel selector as appropriate.

---

## Control

The `Control` class is the superclass for all other GUI classes that display an object on the graphics page. It contains a number of methods that can be used in any of these classes. In a few specific cases, the call has no real effect on a class, even though the call itself is legal. These are relatively obvious—a `MapView`, for example, does not change its appearance or behavior when selected. The documentation will point these cases out when they are not obvious.

The classes that descend from `Control` are

Activity	Button	DatePicker	ImageView
Label	MapView	Picker	Progress
Slider	SegmentedControl	Stepper	Switch
Table	TextField	TextView	WebView

**FUNCTION alpha**

Controls can be set to be translucent by calling `setAlpha`. This function returns the alpha value set with `setAlpha`. The default value for all controls is 1.0.

**FUNCTION height**

Returns the height of the control in pixels. Use `setFrame` to change the height of a control.

**FUNCTION isEnabled AS INTEGER**

Most controls perform some action, such as clicking on a button or setting a switch. Disabling the control prevents it from performing that action, and often changes the appearance of the control. This function returns 1 if the control is enabled, and 0 if it is disabled. See `setEnabled`.

**FUNCTION isHidden AS INTEGER**

Returns 1 if the control is hidden and 0 if it is visible. See `setHidden`.

**FUNCTION isHighlighted AS INTEGER**

Returns 1 if the control is highlighted, and 0 if it is not highlighted. See `setHighlighted`.

**FUNCTION isOpaque AS INTEGER**

Returns 1 if the control is opaque, and 0 if it is not opaque. See `setOpaque`.

**FUNCTION isSelected AS INTEGER**

Returns 1 if the control is selected, and 0 if it is not selected. See `setSelected`.

**FUNCTION kind AS INTEGER**

Returns the kind of the control. The value returned and the corresponding control types are shown in the table below.

Kind	Control
1	Button
2	TextField
3	Switch
4	SegmentedControl
5	Label
6	Slider
7	Activity
8	Progress
9	Stepper
10	TextView
11	Table
12	ImageView
13	WebView
14	MapView
15	DatePicker
16	Picker

**SUB setAlpha (alpha)**

The alpha level sets the transparency of the control, with 1, which is the default, being an opaque state that covers anything underneath the control, and 0 being fully transparent, but still functional.

**SUB setBackgroundColor (red, green, blue, alpha = 1)**

Sets the background color for the control. The exact use of the background color varies from control to control, but in general terms, it is the color drawn before the active components of the control are drawn. The default color is white.

See *Colors* at the start of the discussion of the `Plot` class for information on RGB colors.

**SUB setEnabled (flag AS INTEGER)**

Pass a non-zero value to enable the control, or zero to disable it.

Disabled controls are still visible, but cannot be manipulated. Most controls have a different appearance when disabled so the user knows the control is not available. The disabled state is typically used when an operation is temporarily unavailable, such as disabling an undo button if there is nothing to undo.

**SUB setFocus (flag AS INTEGER)**

Pass a non-zero value to cause the control to grab the focus, or zero to release it.

Setting the focus is typically done to tell a `TextView` or `TextField` to begin accepting input. On iPhones, and on iPads with no physical keyboard, this will bring up the software keyboard until the focus is released.

**SUB setFrame (x, y, width, height)**

Sets the location and size of the control in relation to the top left corner of the graphics display.

**SUB setHidden (flag AS INTEGER)**

Pass a non-zero value to hide the control, or zero to show it.

Hidden controls cannot be manipulated or seen.

**SUB setHighlighted (flag AS INTEGER)**

Pass a non-zero value to highlight the control, or zero to set it to an unhighlighted state.

Highlighting is typically done when a control is being tapped. This is handled automatically, so there is rarely any reason to highlight a control from a program. If this is done, though, the control will generally take on a different appearance.

---

**SUB setOpaque (flag AS INTEGER)**

Pass a non-zero value to cause the control to be opaque, or zero to indicate it has some transparency.

This opacity flag is not the same as the alpha value controlled by `setAlpha`. Opacity indicates that, when the control is drawn, it is either not rectangular or there are areas that are always drawn with an alpha value less than 1. This causes any control under this control to always be redrawn when the rectangle occupied by this control needs to be drawn. The alpha value causes everything in the control, even those areas that are normally opaque, to become translucent.

---

**SUB setReadOnly (flag AS INTEGER)**

Pass a non-zero value to cause the control to become read-only, or zero to allow reading and writing.

The read only flag is only used by `TextView` and `TextField`. Setting either control to read only prevents changing the text in the control.

---

**SUB setSelected (flag AS INTEGER)**

Pass a non-zero value to select the control, or zero to unselect it.

Selected buttons are drawn differently than unselected ones. Along with appropriate images or shading, this flag can be used to implement a two-state button that toggles each time it is pressed, like the radio button on a desktop computer.

---

**SUB setTag (tag AS LONG)**

Sets the value of a tag. The default value is 0.

The tag is solely for the benefit of the program. The system calls always ignore the value. The tag can be used to store a small amount of information about a control, or to set up identifiers for each control so they can easily be distinguished from one another.

---

**FUNCTION tag AS LONG**

Returns the tag set by `setTag`.

---

**FUNCTION width**

Returns the width of the control in pixels. Use `setFrame` to change the width of a control.

---

**FUNCTION x**

Returns the horizontal position of the left edge of the control in pixels. Use `setFrame` to change the position of a control.

---

**FUNCTION y**

Returns the vertical position of the top edge of the control in pixels. Use `setFrame` to change the position of a control.

## DatePicker

`DatePicker` objects display on the graphics view as a series of rolling wheels with options displayed on the wheels. They are used to pick dates, times, or dates and times, depending on the way the control is set up.

`DatePicker` objects are returned by the `newDatePicker` method of the `Graphics` class. The methods in this class are used to manipulate the date picker after it has been created.

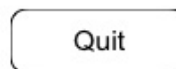
`DatePicker` descends from the `Control` class. All methods in the `Control` class also apply to the `DatePicker` class.

Selecting a date generates a value changed event. If the program has a subroutine named `valueChanged` with appropriate parameters, it is called to handle the event. From there, the program can detect the control that was pressed by comparing the button parameter of the `valueChanged` subroutine to the various buttons in the program. See Chapter 13 for a complete discussion of application methods like `valueChanged`.

The sample shows a short but complete program that displays a data picker that allows selection of dates from 1950 through 2020. Once a date is picked, it is displayed in a text field.



06/27/55



### Snippet

```
! Create a date picker.
DIM dPicker AS DatePicker, selectedDate AS TextField
dPicker = Graphics.newDatePicker(10, 10)
dPicker.setDate(1955, 6, 27)
dPicker.setMinuteInterval(7)
dPicker.setMinimumDate(1950)
dPicker.setMaximumDate(2020)
dPicker.setMode(2)

selectedDate = Graphics.newTextField(10, 250, 380)

! Create the Quit button.
DIM quit AS Button
quit = Graphics.newButton(10, 310, 100)
quit.setTitle("Quit")

System.showGraphics
END

SUB touchUpInside (ctrl AS Button, when AS DOUBLE)
STOP
END SUB

SUB valueChanged (ctrl AS Control, when AS DOUBLE)
DIM dpDate AS Date
dpDate = dPicker.date
selectedDate.setText(dpDate.longDate)
END SUB
```



---

**FUNCTION date AS Date**

Returns the date currently selected on the date picker.

See the Date class for ways to determine specific properties about the date that is returned.

---

**SUB setDate (year AS INTEGER, month AS INTEGER = 1, day AS INTEGER = 1, hour AS INTEGER = 0, minute AS INTEGER = 0, animated AS INTEGER = 1)**

Sets the date displayed on the date picker.

The year should be passed as a four-digit calendar year, such as 2012.

The month defaults to 1, which is January. The months are numbered from 1 to 12, corresponding to the months on the Gregorian calendar.

The day defaults to 1. It is the calendar date for the designated month.

The hour defaults to 0. The hour is specified on a 24 hour clock, with midnight being 0 and 23 being 11 PM.

The minute defaults to 0. Valid values are 0 to 59.

When a date is set, the control will animate the change from the currently displayed date to the new one, spinning the wheels to get to the new date. Pass 0 for this parameter to set the date without taking the time to animate the change.

---

**SUB setMaximumDate (year AS INTEGER, month AS INTEGER = 1, day AS INTEGER = 1, hour AS INTEGER = 0, minute AS INTEGER = 0)**

Sets the maximum date the date picker will display.

The year should be passed as a four-digit calendar year, such as 2012.

The month defaults to 1, which is January. The months are numbered from 1 to 12, corresponding to the months on the Gregorian calendar.

The day defaults to 1. It is the calendar date for the designated month.

The hour defaults to 0. The hour is specified on a 24 hour clock, with midnight being 0 and 23 being 11 PM.

The minute defaults to 0. Valid values are 0 to 59.

---

**SUB setMinimumDate (year AS INTEGER, month AS INTEGER = 1, day AS INTEGER = 1, hour AS INTEGER = 0, minute AS INTEGER = 0)**

Sets the minimum date the date picker will display.

The year should be passed as a four-digit calendar year, such as 2012.

The month defaults to 1, which is January. The months are numbered from 1 to 12, corresponding to the months on the Gregorian calendar.

The day defaults to 1. It is the calendar date for the designated month.

The hour defaults to 0. The hour is specified on a 24 hour clock, with midnight being 0 and 23 being 11 PM.

The minute defaults to 0. Valid values are 0 to 59.

---

**SUB setMinuteInterval (interval AS INTEGER)**

When the date picker is displaying a time, this call sets the number of minutes between each value shown. For example, passing 15 will cause the time to be shown in 15-minute intervals.

If the value passed cannot be evenly divided into 60, the default interval of 1 minute is used.

---

**SUB setMode (mode AS INTEGER)**

The date picker can display the date, the time, or both. By default, it displays the date and time. Use this call to change the picker to display only the date or the time.

Mode	Description
1	Display both the date and time. The specific format varies with the language selected for the iOS device; for US devices, the day of the week, month, day, hour, minute and AM or PM are displayed. The year is not displayed.
2	Display the date, but not the time. In the US, the year, month and date are displayed.
3	Display the time but not the date. In the US, the hour, minute and AM or PM are displayed.

---

## ImageView

ImageView objects are used to place images on the graphics view.

ImageView objects are returned by the `newImageView` method of the `Graphics` class. The methods in this class are used to manipulate the image after it has been created.

ImageView descends from the `Control` class. All methods in the `Control` class also apply to the `ImageView` class.

---

### SUB loadImage (path AS STRING, resize AS INTEGER = 1)

Load an image from a file.

The image specified by **path** is loaded and used as the image.

ImageView controls are normally resized to fit the image. To prevent the button from being resized, specify 0 for the last parameter. If the ImageView is not resized, the image will be resized to fit the available area.

The currently supported image formats are:

Extension	Format
.tiff, .tif	Tagged Image File Format (TIFF)
.jpg, .jpeg	Joint Photographic Experts Group (JPEG)
.gif	Graphic Interchange Format (GIF)
.png	Portable Network Graphic (PNG)
.bmp, .BMPf	Windows Bitmap Format (DIB)
.ico	Windows Icon Format
.cur	Windows Cursor
.xbm	XWindow bitmap

#### Snippet

```
! Create an image.
DIM img AS ImageView
img = Graphics.newImageView(10, 10)
img.loadImage("BlueGears.png")
```




---

### SUB setImage (theImage AS Image, resize AS INTEGER = 1)

Sets the displayed image using a preloaded image.

ImageView controls are normally resized to fit the image. To prevent the button from being resized, specify 0 for the last parameter. If the ImageView is not resized, the image will be resized to fit the available area.

See the `Image` class for information on creating and manipulating images used by this call.

## Label

`Label` objects are used to place text annotations on the screen, often to describe the purpose of other controls, such as a `TextField`.

`Label` objects are returned by the `newLabel` method of the `Graphics` class. The methods in this class are used to manipulate the label after it has been created.

`Label` descends from the `Control` class. All methods in the `Control` class also apply to the `Label` class.

---

### FUNCTION `getText` AS `STRING`

Returns the text from the label. This call returns an empty string if `setText` has never been called, or the text passed by the most recent call to `setText` if `setText` has been called.

---

### SUB `setAlignment` (`alignment` AS `INTEGER`)

Sets the alignment of the text in the label. The default alignment is left aligned text.

Value	Resulting Alignment
1	Left aligned
2	Centered
3	Right Aligned
4	Justified
5	Natural (default)

---

### SUB `setColor` (`red`, `green`, `blue`, `alpha` = 1)

Sets the text color for the label.

See *Colors* at the start of the discussion of the `Plot` class for information on RGB colors.

---

### SUB `setFont` (`name` AS `STRING`, `size`, `style` AS `INTEGER`)

Sets the font for the text. See `setFont` in the `Graphics` class for a discussion of fonts.

---

### SUB `setText` (`text` AS `STRING`)

Sets the text displayed in the label.

---

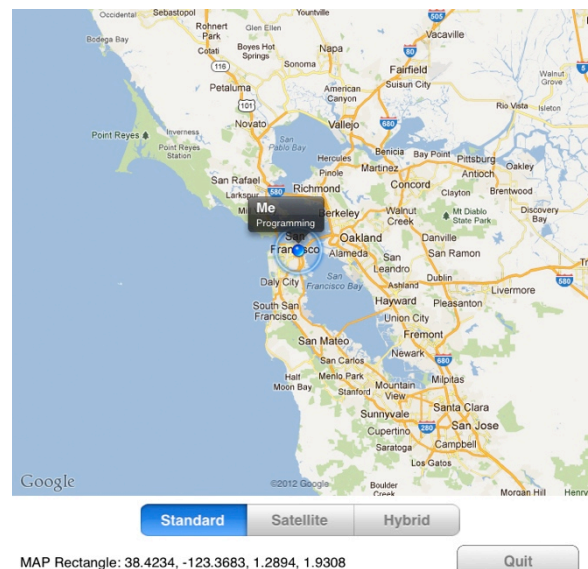
## MapView

`MapView` objects display an interactive map.

`MapView` objects are returned by the `newMapView` method of the `Graphics` class. The methods in this class are used to manipulate the map after it has been created.

`MapView` descends from the `Control` class. All methods in the `Control` class also apply to the `MapView` class.

Locations can be shown on the map two ways. The current location, as determined by the location manager, can be displayed on the map and updated as movement occurs. See `setShowLocation`, `setLocationTitle`, `selectLocation`, `deselectLocation` and `location`. Annotations can be added as pins to label special locations. See `newAnnotation`,



## Chapter 17: GUI Classes

`removeAnnotation`, `selectAnnotation`, `deselectAnnotation`, and the `Annotation` class.

Pinching or swiping the map to change the area viewed generates a value changed event, which is handled by a call to the program's `valueChanged` subroutine. The various functions in this class can then be used to determine the new view area and location. Value changed events are also generated when an annotation is selected or deselected or when the user location changes.

Tapping the map generates a map location event, which passes the location tapped to the program's `mapLocation` subroutine. See Chapter 13 for a complete discussion of application methods like `valueChanged` and `mapLocation`.

This sample shows a simple app that displays a map at the current location, and allows you to tap to see the latitude and longitude, or select map, satellite or hybrid views. It also serves to demonstrate many of the calls in this class.

### Snippet

```
! Create a map showing the current location.
DIM map AS MapView, mapTextField AS TextField
map = Graphics.newMapView(0, 0, Graphics.width, Graphics.height - 110)
map.setMapRect(35.5, -106.8, 1, 1)
map.setShowLocation(1)
map.setLocationTitle("Me", "Programming")
map.setUserTrackingMode(2)

! Set the initial location.
latitude = 36
longitude = -106.3

! Add a text field that will display tapped locations.
mapTextField = Graphics.newTextField(10, Graphics.height - 40,
Graphics.width - 180)

! Add a segmented control to switch between the various map views.
DIM mapType AS SegmentedControl
width = 390
mapType = Graphics.newSegmentedControl((Graphics.width - width)/2,
Graphics.height - 100, width, 40)
mapType.insertSegment("Standard", 1, 0)
mapType.insertSegment("Satellite", 2, 0)
mapType.insertSegment("Hybrid", 3, 0)
mapType.setSelected(1)

! Create the Quit button.
DIM quit AS Button
quit = Graphics.newButton(Graphics.width - 160, Graphics.height - 50, 150,
40)
quit.setTitle("Quit")
quit.setBackgroundColor(1, 1, 1)
quit.setGradientColor(0.8, 0.8, 0.8)
quit.setColor(0.5, 0.5, 0.5)
quit.setFont("Helvetica", 18, 1)
System.showGraphics
END
```

```

FUNCTION format (x AS DOUBLE) AS STRING
s$ = STR(INT(x*10000))
len% = LEN(s$)
s$ = LEFT(s$, len% - 4) & "." & RIGHT(s$, 4)
format = s$
END FUNCTION

SUB touchUpInside (ctrl as Button, when AS DOUBLE)
IF ctrl = quit THEN
    stop
END IF
END SUB

SUB valueChanged (ctrl as Control, when AS DOUBLE)
IF ctrl = map THEN
    rect = map.mapRect
    loc$ = format(rect(1)) & ", " & format(rect(2)) & ", " & format(rect(3))
    & ", " & format(rect(4))
    mapTextField.setText("MAP Rectangle: " & loc$)

    location = map.location
    IF latitude <> location(1) OR longitude <> location(2) THEN
        latitude = location(1)
        longitude = location(2)
        map.setMapRect(latitude + 0.5, longitude - 0.5, 1, 1)
        map.selectLocation
    END IF
ELSE IF ctrl = mapType THEN
    print mapType.selected
    map.setMapType(mapType.selected)
END IF
END SUB

SUB mapLocation (ctrl AS MapView, time AS DOUBLE, latitude AS DOUBLE,
longitude AS DOUBLE)
mapTextField.setText("Tap Location: " & format(latitude) & ", " &
format(longitude))
END SUB

```

---

**FUNCTION annotations AS Annotation()**

Returns an array of all annotations currently on the map.  
Annotations are created with the newAnnotation call.

---

**FUNCTION center AS DOUBLE(2)**

Returns the center of the map. The first element of the array is the latitude, while the second is the longitude.

**Snippet**

```

c = map.center
Print "Latitude: "; c(1); ", Longitude: "; c(2)

```

---

**SUB deselectAnnotation (annotationObject AS Annotation, animated AS INTEGER = 1)**

Deselects an annotation. If the animated flag is set to 0, this is done immediately; if it is a non-zero value, the selection is removed using a visual animation.

A selected annotation displays any title and subtitle associated with the annotation. Deselecting the annotation hides them.

See `newAnnotation` for more about annotations.

---

### **SUB deselectLocation**

Deselects the current location.

If the current location is visible and selected, this call deselects it. Once deselected, any title or subtitle associated with the location is hidden.

---

### **FUNCTION isUpdating AS INTEGER**

Returns 0 if the user location is not currently updating, and 1 if it is updating.

---

### **FUNCTION location AS DOUBLE(8)**

Returns the current user location.

The `MapView` can use the location manager to track and display the current user location. This call returns the current location.

The first two elements of the array returned are the latitude and longitude. Next is the altitude in meters. The fourth and fifth elements of the array are the horizontal position error and vertical position error in meters. The sixth and seventh elements of the array are the speed in meters per second, and the heading as a compass heading. These values will be -1 if the speed and heading cannot be determined. The last element of the array is the time stamp, which is consistent with the time stamps returned by the calls in the `Sensors` class.

The snippet shows a program that prints the current location information. `Map` should already be declared, similar to the snippet at the start of this class description.

#### Snippet

```
location = map.location
PRINT USING "Latitude      : #####.###"; location(1)
PRINT USING "Longitude     : #####.###"; location(2)
PRINT USING "Altitude      : #####.###"; location(3)
PRINT USING "Horiz. Error:  : #####.###"; location(4)
PRINT USING "Vert. Error :  : #####.###"; location(5)
PRINT USING "Speed         : #####.###"; location(6)
PRINT USING "Direction    : #####"; location(7)
PRINT      "Time stamp   : "; location(8)
```

---

### **FUNCTION mapRect AS DOUBLE(4)**

Returns the position and size of the map area shown by the current view.

The first two elements of the returned array are the latitude and longitude of the top left corner of the map view. The next two elements are the width and height of the view in degrees of latitude and longitude, respectively.

Note that the standard of listing latitude first and longitude second is at odds with the general convention of listing horizontal components before vertical components for graphics objects. This call sticks to the convention used for maps, returning latitude first, even though latitude is the vertical measurement of location.

See the snippet at the start of this class for an example of this method in use.

---

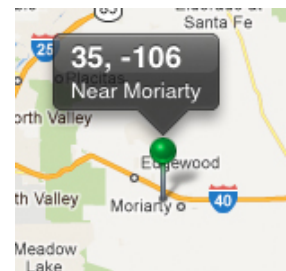
### **FUNCTION newAnnotation (latitude AS DOUBLE, longitude AS DOUBLE, animated AS INTEGER, pinColor AS INTEGER) AS Annotation**

Creates a new annotation and adds it to the map.

Annotations appear on the map as a pin at the specified location. They can be selected, which shows the title and subtitle, either programmatically with `selectAnnotation` or `selectAnnotations`, or by tapping on the head of the pin.

Dropping a pin on the map is normally animated. Override the `animated` parameter, passing zero, to prevent animation of adding the new annotation.

Pins can appear in any of three colors. The default color is green. Apple has designated specific uses for each of the predefined pin colors. These are noted in the table.



Color	Color of the pin head
1	Green; intended for starting locations
2	Red; intended for destinations
3	Purple; intended for user-specified locations

The `Annotation` class contains methods to add, remove, select, deselect or get lists of the current annotations. The `Annotation` class has methods to change the annotation itself.

#### Snippet

```
DIM map AS MapView, pin AS Annotation
map = Graphics.newMapView(10, 10, 380, 240)
map.setMapRect(35.5, -106.8, 1, 1)

pin = map.newAnnotation(35, -106.1, 1, 2)
pin.setTitle("35, -106")
pin.setSubtitle("Near Moriarty")
map.selectAnnotation(pin)
```

---

#### **SUB removeAnnotation (annotationObject AS Annotation)**

Removes an annotation from the map.  
See `newAnnotation` for more about annotations.

#### Snippet

```
DIM map AS MapView, pin AS Annotation
map = Graphics.newMapView(10, 10, 380, 240)
map.setMapRect(35.5, -106.8, 1, 1)

pin = map.newAnnotation(35, -106.1, 1, 2)
map.removeAnnotation(pin)
```

---

#### **SUB selectAnnotation (annotationObject AS Annotation, animated AS INTEGER = 1)**

Selects an annotation. Any previously selected annotations are deselected. If the `animated` flag is set to 0, this is done immediately; if it is a non-zero value, the annotation is removed using a visual animation.

A selected annotation displays any title and subtitle associated with the annotation. Deselecting the annotation hides them.

See `newAnnotation` for more about annotations.

---

#### **FUNCTION selectedAnnotations AS Annotation(n)**

Returns an array containing all of the selected annotations.  
See `newAnnotation` for more about annotations.

---

#### **SUB selectLocation (animated AS INTEGER = 1)**

Selects the current location.

If the current location is visible, this call selects it. Once selected, any title or subtitle associated with the location is shown.

See the snippet at the start of this class for an example of this method in use.

---

#### **SUB setCenter (latitude AS DOUBLE, longitude AS DOUBLE, animated AS INTEGER = 1)**

Sets the location shown by the map so the given latitude and longitude are at the center of the map. The area covered by the map changes as little as possible, but there will be adjustments as the latitude changes to account for the curvature of the Earth.

If the `animated` flag is non-zero, the change in location is animated; if it is zero, the change occurs immediately.

Snippet

```
DIM map AS MapView, pin AS Annotation
map = Graphics.newMapView(10, 10, 380, 240)
map.setCenter(35, -106.1, 0)
```

**SUB setLocationTitle (title AS STRING, subtitle AS STRING = "")**

Sets the title and subtitle that appears near the location when the user location is visible and selected.

See the snippet as the start of this class for an example of this method in use.

**SUB setMapType (mapType AS INTEGER)**

Sets the type of the map.

The map can display as a satellite image, a political map with roads and so forth, or a combination of the two. By default, the map is displayed as a political map. Use this method to change the type of map shown.

mapType	Map Shown
1	Standard political map
2	Satellite image
3	Combination of satellite image and political map

See the snippet as the start of this class for an example of this method in use.

**SUB setMapRect (latitude AS DOUBLE, longitude AS DOUBLE, height AS DOUBLE, width AS DOUBLE, animated AS INTEGER = 1)**

Sets the area displayed by the map.

The latitude and longitude specify the location of the upper left corner of the map. The height and width are the approximate size of the map in degrees of latitude and longitude, respectively. The actual size is usually adjusted slightly to account for the curvature of the Earth. Use `mapRect` to retrieve the actual size of the map.

See the snippet as the start of this class for an example of this method in use.

**SUB setShowLocation (show AS INTEGER)**

Show or hide the location of the user.

See the snippet as the start of this class for an example of this method in use.

**SUB setUserTrackingMode (mode AS INTEGER)**

The user location can be tracked as the location changes. Use this command to set the tracking mode.

mode	Tracking Mode
1	Do not track the user's location
2	Follow the user, but keep the map set so up is north
3	Follow the user, rotating the map so up is the direction of travel

## Picker

`Picker` objects display a custom wheel-driven picker that allows selection of multiple related values.

`Picker` objects are returned by the `newPicker` method of the `Graphics` class. The methods in this class are used to manipulate the picker after it has been created.

`Picker` descends from the `Control` class. All methods in the `Control` class also apply to the `Picker` class.





Changing a value on the wheel generates a value changed event, which is handled by a call to the program's `valueChanged` subroutine. The selected method in this class can then be used to determine the new wheel selections.

See Chapter 13 for a complete discussion of application methods like `valueChanged`.

This sample shows a simple app that displays a color picker. The three wheels set the numeric values for the red, green and blue components of the color; the selections are shown on labels that appear immediately below the picker. It also serves to demonstrate many of the calls in this class.

#### Snippet

```
DIM colors AS Picker, wheels(3) AS INTEGER
colors = Graphics.newPicker(10, 10, 180, 180)

wheels(1) = 1
wheels(2) = 2
wheels(3) = 3
colors.insertWheels(wheels)
FOR t% = 0 TO 10
    colors.insertRow(STR(t%/10.0), t% + 1)
    colors.insertRow(STR(t%/10.0), t% + 1, 2)
    colors.insertRow(STR(t%/10.0), t% + 1, 3)
NEXT

DIM red AS Label, green AS Label, blue AS Label, all AS Label
red = Graphics.newLabel(10, 190, 60, 30)
green = Graphics.newLabel(70, 190, 60, 30)
blue = Graphics.newLabel(130, 190, 60, 30)
all = Graphics.newLabel(10, 220, 180, 30)

DIM quit AS Button
quit = Graphics.newButton(10, 260)
quit.setTitle("Quit")

colors.selectRow(11)
colors.selectRow(7, 2)
valueChanged(colors, 0)

System.showGraphics
END

SUB valueChanged (ctrl AS Control, when AS DOUBLE)
IF ctrl = colors THEN
    r = (colors.selection - 1)/10.0
    g = (colors.selection(2) - 1)/10.0
    b = (colors.selection(3) - 1)/10.0
    red.setBackgroundColor(r, 0, 0)
    green.setBackgroundColor(0, g, 0)
    blue.setBackgroundColor(0, 0, b)
    all.setBackgroundColor(r, g, b)
END IF
END SUB

SUB touchUpInside (ctrl AS Button, when AS DOUBLE)
IF ctrl = quit THEN
    STOP
END IF
END SUB
```

---

**SUB insertRow (cell AS STRING, row AS INTEGER, wheel AS INTEGER = 1)**

Inserts a row in a wheel.

**cell** is the text that will appear on the new row in the wheel. It can be any text string.

The new row will be inserted before the row indicated by the **row** parameter. Rows are indexed from 1, so passing 1 puts the cell at the start of the wheel. Pass one greater than the number of cells in the wheel to put the new cell at the end of the wheel.

If it is used, the **wheel** parameter specifies the wheel the row is inserted into. Wheels are numbered from left to right, starting with 1. If this parameter is not given, the new row is inserted in the first wheel.

See the snippet at the start of this class for an example that uses this method.

---

**SUB insertRows (cells() AS STRING, row AS INTEGER, wheel AS INTEGER = 1)**

Inserts more than one row into a picker wheel.

This method works just like `insertRow`, but passes an array of names for new rows instead of a single name. One new row is created for each entry in the `cells` array.

---

**SUB insertWheels (sections() AS INTEGER)**

Creates one or more new wheels in the picker.

The **sections** array is an array of wheel numbers specifying where to insert the new wheel. Each wheel is inserted before the wheel number passed in the **sections** array. Wheels are numbered from 1.

See the snippet at the start of this class for an example that uses this method.

---

**FUNCTION selection (wheel AS INTEGER = 1) AS INTEGER**

Returns the selected row in a wheel.

Rows and wheels are numbered from 1.

See the snippet at the start of this class for an example that uses this method.

---

**SUB selectRow (row AS INTEGER, wheel AS INTEGER = 1, animated AS INTEGER = 1)**

Selects a row.

Rows are numbered from 1 to the number of rows in the wheel.

The **wheel** parameter specifies the wheel from which to select the row.

Pass a non-zero value for the **animated** parameter to animate the change, or 0 to make the change without an animation.

See the snippet at the start of this class for an example that uses this method.

---

**SUB setShowsSelection (flag AS INTEGER)**

A picker normally shows a selection bar across the wheels. Use this method to turn the selection bar on or off.

---

## Progress

`Progress` objects display a progress bar. Progress bars are used to show the relative completion level of a task that can be predicted ahead of time, like how long it takes to copy a series of files.

`Progress` objects are returned by the `newProgress` method of the `Graphics` class. The methods in this class are used to manipulate the picker after it has been created.

`Progress` descends from the `Control` class. All methods in the `Control` class also apply to the `Progress` class.

The sample shows a progress bar that updates for 10 seconds, after which the program stops.

### Snippet

```
DIM p AS Progress
p = Graphics.newProgress(10, 10)
```



```

DIM lastTime AS DOUBLE

System.showGraphics

SUB nullEvent (time AS DOUBLE) IF lastTime = 0 THEN
    lastTime = time
ELSE
    p.setValue((time - lastTime)/10)
    IF time - lastTime > 10 THEN STOP
END IF
END SUB

```

---

**SUB setProgressColor (red, green, blue, alpha = 1)**

Sets the color of the area used to show progress, which defaults to light blue.

**Snippet**

```

DIM p AS Progress
p = Graphics.newProgress(10, 10)
p.setProgressColor(0, 1, 0)

DIM lastTime AS DOUBLE

System.showGraphics

SUB nullEvent (time AS DOUBLE) IF lastTime = 0 THEN
    lastTime = time
ELSE
    p.setValue((time - lastTime)/10)
    IF time - lastTime > 10 THEN STOP
END IF
END SUB

```




---

**SUB setStyle (style AS INTEGER)**

Sets the style of the progress bar.

The default progress bar has a style of 1; it is a light blue bar with a white track that is shaded to look like a bead of color is filling in a rounded groove. Choosing a style of 2 changes the progress bar to a gray bar with a black track with more complex shading.

**Snippet**

```

DIM p1 AS Progress, p2 AS Progress
p1 = Graphics.newProgress(10, 10)
p2 = Graphics.newProgress(10, 30)
p2.setStyle(2)

DIM lastTime AS DOUBLE

System.showGraphics

SUB nullEvent (time AS DOUBLE)
IF lastTime = 0 THEN
    lastTime = time
ELSE
    p1.setValue((time - lastTime)/10)
    p2.setValue((time - lastTime)/10)
    IF time - lastTime > 10 THEN STOP
END IF

```



END SUB

---

### **SUB setTrackColor (red, green, blue, alpha = 1)**

Sets the color of the track, which defaults to white.

#### Snippet

```
DIM p AS Progress
p = Graphics.newProgress(10, 10)
p.setTrackColor(1, 1, 0)

DIM lastTime AS DOUBLE

System.showGraphics

SUB nullEvent (time AS DOUBLE) IF lastTime = 0 THEN
    lastTime = time
ELSE
    p.setValue((time - lastTime)/10)
    IF time - lastTime > 10 THEN STOP
END IF
END SUB
```




---

### **SUB setValue (value, animated AS INTEGER = 1)**

Sets the value of the progress bar.

The value should be a number from 0, indicating no progress, to 1, indicating the task is complete.

Set the animated flag to 1 to animate the change in value, or 0 to immediately redraw the control.

See the snippets from any other method in this class for an example of setValue.

---

## SegmentedControl

SegmentedControl objects display a row of connected buttons. Tapping one of the buttons selects it and deselects any other button on the segmented control.

SegmentedControl objects are returned by the newSegmentedControl method of the Graphics class. The methods in this class are used to manipulate the picker after it has been created.

SegmentedControl descends from the Control class. All methods in the Control class also apply to the SegmentedControl class.

Tapping on a segment to select it generates a value changed event, which is handled by a call to the program's valueChanged subroutine. The selected method in this class can then be used to determine which segment is selected.

See Chapter 13 for a complete discussion of application methods like valueChanged.

The snippet shows a simple segmented control that echoes the selection in a text field. It illustrates several of the methods in this class.

#### Snippet

```
DIM sc as SegmentedControl, scTextField AS TextField
sc = Graphics.newSegmentedControl(10, 10, 200, 30)
sc.insertSegment("Yes", 1, 0)
sc.insertSegment("No", 2, 0)
sc.insertSegment("Maybe", 2, 0)
sc.setSelected(1)
sc.setApportionByContent(1)
```

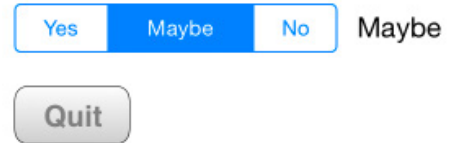
```
scTextField = Graphics.newTextField(220, 10, 60, 30)
scTextField.setText("Yes")
```

```
DIM quit AS Button
quit = Graphics.newButton(10, 60)
quit.setTitle("Quit")
quit.setBackgroundColor(1, 1, 1)
quit.setGradientColor(0.8, 0.8, 0.8)
quit.setColor(0.5, 0.5, 0.5)
quit.setFont("Helvetica", 18, 1)
```

```
System.showGraphics
```

```
SUB touchUpInside (ctrl AS Button, when AS DOUBLE)
IF ctrl = quit THEN
    STOP
END IF
END SUB
```

```
SUB valueChanged (ctrl AS Control, when AS DOUBLE)
IF ctrl = sc THEN
    scTextField.setText(sc.title(sc.selected))
END IF
END SUB
```




---

```
SUB insertSegment (title AS STRING, index AS INTEGER, animated AS INTEGER = 1)
```

Inserts a new segment in the control.

**title** is the text to place in the segment.

**index** is the number of the existing segment the new segment will appear before. The first segment is segment 1, so passing 1 for this value puts the new segment to the left of all existing segments. Use one more than the number of existing segments to place the new segment to the right of all existing segments.

If the **animated** parameter is 0, the change is made immediately; if it is non-zero, the change to the control is animated.

See the snippet at the start of the class for an example of `insertSegment`.

---

```
SUB removeAllSegments
```

Removes all of the existing segments from the control.

---

```
SUB removeSegment (index AS INTEGER, animated AS INTEGER = 1)
```

Removes a segment from the control.

**index** is the number of the segment to remove. The leftmost segment is numbered 1, and so forth, proceeding to the right.

If the **animated** parameter is 0, the change is made immediately; if it is non-zero, the change to the control is animated.

Using

```
sc.removeSegment(2)
```

in the snippet from the start of this class would remove the segment titled `Maybe`.

---

```
FUNCTION isSegmentEnabled (index AS INTEGER) AS INTEGER
```

Returns 1 if the indicated segment is enabled, or 0 if it is not.

Use `setSegmentEnabled` to enable or disable a segment.

---

**FUNCTION selected AS INTEGER**

Returns the number of the selected segment, counting from 1.

See the snippet at the start of the class for an example of `selected`.

---

**SUB setApportionByContent (flag AS INTEGER)**

Pass 0 to force all segments in the control to have the same width, or a non-zero value to automatically resize the segments based on the width of the title in each segment. The default behavior is to create segments with the same width.

See the snippet at the start of the class for an example of `setApportionByContent`.

---

**SUB setSegmentEnabled (index AS INTEGER, flag AS INTEGER)**

Enables or disables a segment.

A disabled segment cannot be selected by tapping, and is also drawn with a slightly less pronounced button. Disable a segment by passing 0 for **flag**, or enable it by passing a non-zero value. Segments are typically disabled when the function they represent is temporarily unavailable.

Using

```
sc.setSegmentEnabled(2, 0)
```



in the snippet from the start of this class would disable the segment titled Maybe. The new control is shown here; note that there is a slight difference in the way the Maybe button is drawn.

---

**SUB setSelected (index AS INTEGER)**

Selects the indicated segment.

Selecting a segment changes its color to the tint color, which defaults to blue, and deselects any other segment.

See the snippet at the start of the class for an example of `setSelected`.

---

**SUB setStyle (style AS INTEGER)**

Deprecated.

Styles for segmented controls are no longer supported in iOS. While this call remains to prevent legacy programs from crashing, it no longer does anything.

---

**SUB setTintColor (red, green, blue, alpha = 1)**

Sets the tint color for a segmented control whose style is set to 3 (Bar) with the `setStyle` method. The tint color is ignored for all other styles.

Snippet

```
DIM sc as SegmentedControl, scTextField AS TextField
sc = Graphics.newSegmentedControl(10, 10, 200, 30)
sc.insertSegment("Yes", 1, 0)
sc.insertSegment("No", 2, 0)
sc.insertSegment("Maybe", 2, 0)
sc.setSelected(1)
sc.setApportionByContent(1)
sc.setStyle(3)
sc.setTintColor(0.7, 0.7, 0)
```



```

DIM quit AS Button
quit = Graphics.newButton(10, 60)
quit.setTitle("Quit")
quit.setBackgroundColor(0.8, 0.8, 0)
quit.setGradientColor(0.6, 0.6, 0)
quit.setColor(1, 1, 1)
quit.setFont("Helvetica", 18, 1)

System.showGraphics

SUB touchUpInside (ctrl AS Button, when AS DOUBLE)
IF ctrl = quit THEN
    STOP
END IF
END SUB

```

---

**SUB setTitle (title AS STRING, index AS INTEGER)**

Sets the title for the indicated segment.

---

**FUNCTION title (index AS INTEGER) AS STRING**

Returns the title for the indicated segment.

See the snippet at the start of the class for an example of `title`.

---

## Slider

Slider objects display a slot and a button called the thumb. They are used to allow the selection of a more or less continuous value, like an exposure setting for a camera.

Slider objects are returned by the `newSlider` method of the `Graphics` class. The methods in this class are used to manipulate the slider after it has been created.

Slider descends from the `Control` class. All methods in the `Control` class also apply to the `Slider` class.

Dragging the thumb generates a value changed event, which is handled by a call to the program's `valueChanged` subroutine. The `value` method in this class can then be used to determine the new slider value.

See Chapter 13 for a complete discussion of application methods like `valueChanged`.

The snippet shows a simple slider whose value is displayed in a text field. It illustrates several of the methods in this class.

### Snippet

```

DIM sl AS Slider, slTextField as TextField
sl = Graphics.newSlider(10, 10)
sl.setMinValue(-50)
sl.setMaxValue(50)
sl.setValue(10)

slTextField = Graphics.newTextField(150, 10)

```



```

DIM quit AS Button
quit = Graphics.newButton(10, 40)
quit.setTitle("Quit")
quit.setBackgroundColor(1, 1, 1)
quit.setGradientColor(0.8, 0.8, 0.8)
quit.setColor(0.5, 0.5, 0.5)
quit.setFont("Helvetica", 18, 1)

System.showGraphics

SUB touchUpInside (ctrl AS Button, when AS DOUBLE)
IF ctrl = quit THEN
    STOP
END IF
END SUB

SUB valueChanged (ctrl AS Control, when AS DOUBLE)
IF ctrl = sl THEN
    slTextField.setText(STR(sl.value))
END IF
END SUB

```

---

**SUB setContinuous (flag AS INTEGER)**

By default, the slider sends a value changed event each time the value changes even slightly during a swipe gesture. If the program only needs to know the final value of the control, and does not need to track these intermediate values, call `setContinuous` with a parameter of 0. This will cause the control to generate a value changed event only after the touch event that is changing the control value has completed. Pass a non-zero value to restore the original behavior.

---

**SUB setMaxColor (red, green, blue, alpha = 1)**

Sets the color for the bar that appears to the right of the thumb.

**Snippet**

```

DIM sl AS Slider, slTextField as TextField
sl = Graphics.newSlider(10, 10)
sl.setMaxColor(1, 1, 0)
sl.setValue(0.5)

```



```

DIM quit AS Button
quit = Graphics.newButton(10, 40)
quit.setTitle("Quit")

```

```

System.showGraphics

```

```

SUB touchUpInside (ctrl AS Button, when AS DOUBLE)
IF ctrl = quit THEN
    STOP
END IF
END SUB

```

---

**SUB setMaxValue (value)**

Sets the value that is returned by the `value` method when the slider is all the way to the right. The default value is 1.0.

See the snippet at the start of the class for an example of `setMaxValue`.



---

**SUB setMinColor (red, green, blue, alpha = 1)**

Sets the color for the bar that appears to the left of the thumb.

**Snippet**

```

DIM sl AS Slider, slTextField as TextField
sl = Graphics.newSlider(10, 10)
sl.setMinColor(1, 1, 0)
sl.setValue(0.5)

DIM quit AS Button
quit = Graphics.newButton(10, 40)
quit.setTitle("Quit")

System.showGraphics

SUB touchUpInside (ctrl AS Button, when AS DOUBLE)
IF ctrl = quit THEN
    STOP
END IF
END SUB

```



---

**SUB setMinValue (value)**

Sets the value that is returned by the value method when the slider is all the way to the left. The default value is 0.0.

See the snippet at the start of the class for an example of setMinValue.

---

**SUB setThumbColor (red, green, blue, alpha = 1)**

Sets the color for the slider thumb.

**Snippet**

```

DIM sl AS Slider, slTextField as TextField
sl = Graphics.newSlider(10, 10)
sl.setThumbColor(1, 1, 0)
sl.setValue(0.5)

DIM quit AS Button
quit = Graphics.newButton(10, 40)
quit.setTitle("Quit")

System.showGraphics

SUB touchUpInside (ctrl AS Button, when AS DOUBLE)
IF ctrl = quit THEN
    STOP
END IF
END SUB

```



---

**SUB setValue (value)**

Sets the slider value.

See the snippet at the start of the class for an example of setValue.

---

**FUNCTION value**

Returns the current value for the slider.

See the snippet at the start of the class for an example of value.

## Stepper

Stepper objects display a pair of buttons that are designed to raise or lower a value.

Stepper objects are returned by the `newStepper` method of the `Graphics` class. The methods in this class are used to manipulate the stepper after it has been created.

Stepper descends from the `Control` class. All methods in the `Control` class also apply to the `Stepper` class.

Tapping either button generates a value changed event, which is handled by a call to the program's `valueChanged` subroutine. Touching and holding the control generates a series of value changed events; again, the `valueChanged` subroutine is called as the value changes. The `value` method in this class can then be used to determine the new slider value.

See Chapter 13 for a complete discussion of application methods like `valueChanged`.

The snippet shows a stepper whose value is displayed in a text field. It illustrates several of the methods in this class.

### Snippet

```
DIM st AS Stepper, stTextField as TextField
st = Graphics.newStepper(10, 10)
st.setStepValue(5)
st.setMinValue(-50)
st.setMaxValue(50)
st.setValue(-40)

stTextField = Graphics.newTextField(120, 10)
stTextField.setText(str(st.value))

DIM quit AS Button
quit = Graphics.newButton(10, 50)
quit.setTitle("Quit")
quit.setBackgroundColor(1, 1, 1)
quit.setGradientColor(0.8, 0.8, 0.8)
quit.setColor(0.5, 0.5, 0.5)
quit.setFont("Helvetica", 18, 1)

System.showGraphics

SUB touchUpInside (ctrl AS Button, when AS DOUBLE)
IF ctrl = quit THEN
    STOP
END IF
END SUB

SUB valueChanged (ctrl AS Control, when AS DOUBLE)
IF ctrl = st THEN
    stTextField.setText(STR(st.value))
END IF
END SUB
```




---

### SUB setAutorepeat (flag AS INTEGER)

By default, touching and holding the stepper will pause for a moment, then begin to update the control repeatedly. Pass 0 to this method to prevent this, instead updating the value once per touch. Pass a non-zero value to restore the original behavior.

**SUB setMaxValue (value)**

Sets the maximum value that is returned by the `value` method. The default value is 100.  
See the snippet at the start of the class for an example of `setMaxValue`.

**SUB setMinValue (value)**

Sets the minimum value that is returned by the `value` method. The default value is 0.  
See the snippet at the start of the class for an example of `setMinValue`.

**SUB setStepValue (value)**

Sets the step size for the stepper. The step size is the amount the value will change each time the control is touched, unless, of course, the control is already at the maximum or minimum value, and cannot be changed further. The default step size is 1.

See the snippet at the start of the class for an example of `setStepValue`.

**SUB setValue (value)**

Sets the value for the stepper; this is the value returned by the `value` method.  
See the snippet at the start of the class for an example of `setValue`.

**FUNCTION value**

Returns the current value for the stepper.  
See the snippet at the start of the class for an example of `value`.

**SUB setWraps (flag AS INTEGER)**

By default, if the control reaches the maximum value, the value cannot be increased, and if it reaches the minimum value, the value cannot be decreased. Pass a non-zero value to this method to change that behavior so increasing the value past the maximum wraps to the minimum value, and decreasing the value while it is at the minimum wraps to the maximum value. Pass 0 to the method to restore the default behavior.

---

## Switch

Switch objects display an on off switch. These are often paired with a Label that explains what is being turned on or off.

Switch objects are returned by the `newSwitch` method of the Graphics class. The methods in this class are used to manipulate the switch after it has been created.

Switch descends from the Control class. All methods in the Control class also apply to the Switch class.

Tapping the switch flips the On/Off state and generates a value changed event, which is handled by a call to the program's `valueChanged` subroutine. The `isOn` method in this class can then be used to determine the new switch value.

The appearance of switches has changed over the years. In iOS 6 and older versions, the switch appears in blue rather than green, and has a text label indicating whether it is on or off. These appearance differences are controlled by the version of iOS; there is no way to programmatically select the appearance.

See Chapter 13 for a complete discussion of application methods like `valueChanged`.

The snippet shows a switch whose value is displayed in a text field. It illustrates several of the methods in this class.

Snippet

```
DIM sw as Switch, swTextField as TextField
sw = Graphics.newSwitch(10, 10)
sw.setOn(1)
```



```

swTextField = Graphics.newTextField(100, 10)

DIM quit AS Button
quit = Graphics.newButton(10, 50)
quit.setTitle("Quit")
quit.setBackgroundColor(1, 1, 1)
quit.setGradientColor(0.8, 0.8, 0.8)
quit.setColor(0.5, 0.5, 0.5)
quit.setFont("Helvetica", 18, 1)

System.showGraphics

SUB touchUpInside (ctrl AS Button, when AS DOUBLE)
IF ctrl = quit THEN
    STOP
END IF
END SUB

SUB valueChanged (ctrl AS Control, when AS DOUBLE)
IF ctrl = sw THEN
    IF sw.isOn THEN
        swTextField.setText("On")
    ELSE
        swTextField.setText("Off")
    END IF
END IF
END SUB

```

---

**FUNCTION isOn AS INTEGER**

Returns 1 if the switch is on, and 0 if it is off.

See the snippet at the start of the class for an example of isOn.

---

**SUB setColor (red, green, blue, alpha = 1)**

Sets the color that appears on the switch when it is on.

---

Snippet

```

DIM sw as Switch, swTextField as TextField
sw = Graphics.newSwitch(10, 10)
sw.setColor(0, 1, 0)

```

```

DIM quit AS Button
quit = Graphics.newButton(10, 50)
quit.setTitle("Quit")

```

System.showGraphics

```

SUB touchUpInside (ctrl AS Button, when AS DOUBLE)
IF ctrl = quit THEN
    STOP
END IF
END SUB

```



---

**SUB setOn (flag AS INTEGER)**

Pass a non-zero value to this method to turn the switch on, and 0 to turn it off.

See the snippet at the start of the class for an example of setOn.

---

## Table

`Table` objects display a scrollable, selectable list of strings called cells. These cells can be placed in one or more groups, called sections, which can optionally have a section title.

`Table` objects are returned by the `newTable` method of the `Graphics` class. The methods in this class are used to manipulate the table after it has been created.

`Table` descends from the `Control` class. All methods in the `Control` class also apply to the `Table` class.

Tapping a cell generates a cell selected event, which is handled by a call to the program's `cellSelected` subroutine. The `selection` method in this class can then be used to determine which cell is selected. See Chapter 13 for a complete discussion of application methods like `cellSelected`.

The snippet shows a table whose value is displayed in a text field. It illustrates several of the methods in this class. This table has three sections labeled Spanish, English and Roman. The section labeled Spanish has two cells labeled Uno and Dos.

### Snippet

```
DIM tb AS Table, tbTextField AS TextField

tbTextField = Graphics.newTextField(10, 10)

DIM tableValues(7) AS STRING
tb = Graphics.newTable(10, 50)
tableValues(1) = "Four"
tableValues(2) = "Five"
tableValues(3) = "Six"
tableValues(4) = "Seven"
tableValues(5) = "Eight"
tableValues(6) = "Nine"
tableValues(7) = "Ten"
tb.insertRow("One", 1)
tb.insertRow("Three", 3, 1)
tb.insertRow("Two", 2)
tb.insertRows(tableValues, 4)

DIM sections(1) AS INTEGER
sections = [3, 1]
tb.insertSections(sections)

tb.insertRow("Uno", 1, 1)
tb.insertRow("Dos", 2, 1)
tb.insertRow("I", 1, 3)
tb.insertRow("II", 2, 3)
tb.insertRow("III", 3, 3)
tb.insertRow("IV", 4, 3)

tb.setSectionText("Spanish", 1)
tb.setSectionText("Roman", 3)
tb.setSectionText("English", 2)

tb.setSelection(1)
tb.setFont("Arial", 16, 1)

DIM quit AS Button
quit = Graphics.newButton(10, 300)
quit.setTitle("Quit")
```

Four

Spanish
Uno
Dos
English
One
Two
Three
<b>Four</b>
Five
Six
Seven
Eight
Quit

```

System.showGraphics

SUB touchUpInside (ctrl AS Button, when AS DOUBLE)
IF ctrl = quit THEN
    STOP
END IF
END SUB

SUB cellSelected (ctrl AS Control, time AS DOUBLE, row AS INTEGER, section
AS INTEGER)
tbTextField.setText(tb.getText(row, section))
END SUB

```

---

**FUNCTION allowsMultipleSelection AS INTEGER**

Returns the 1 if the table allows multiple cells to be selected at the same time, and 0 if it does not.  
See setAllowsMultipleSelection for a way to change this flag.

---

**FUNCTION allowsSelection AS INTEGER**

Returns the 1 if the table allows cells to be selected by the user, and 0 if it does not.  
See setAllowsSelection for a way to change this flag.

---

**SUB deleteRows (cells(,) AS INTEGER)**

Deletes one or more rows from a table.

**cells** is an array with any number of rows and two columns. In each row, the first element is the cell number of the cell to delete, and the second is the section number of the cell to delete. Both cells and section numbers are counted from 1. Cells may be listed in any order, but are always deleted starting at the end of the table and working toward the beginning. This prevents confusion with the number of a cell; the number passed in the array refers to the cell number before any deletions take place, regardless of the order in the table.

If the cell or section refers to a cell that does not exist, the entry is ignored.

Snippet

```

! Delete the number 2 from each of the three sections in the
! table shown in the example at the start of the class.
DIM rows(3, 2) AS INTEGER
rows(1, 1) = 2 : ! Row 2
rows(1, 2) = 1 : ! Section 1
rows(2, 1) = 2 : ! Row 2
rows(2, 2) = 2 : ! Section 2
rows(3, 1) = 2 : ! Row 2
rows(3, 2) = 3 : ! Section 3
tb.deleteRows(rows)

```

---

**SUB deleteSections (sections() AS INTEGER)**

Deletes one or more sections from a table. Any cells in the section are also deleted.

**sections** is a singly-subscripted array with any number of entries. Each entry refers to the number of a section to delete, counting from 1. Sections may be listed in any order, but are always deleted starting at the end of the table and working toward the beginning. This prevents confusion with the number of a section; the number passed in the array refers to the section number before any deletions take place, regardless of the order in the table.

If an entry in the array refers to a section that does not exist, the entry is ignored.

Snippet

```
! Delete the second section in the table shown in the example at
! the start of the class.
DIM section(1) AS INTEGER
section(1) = 2
tb.deleteSections(section)
```

---

**SUB deselect (row AS INTEGER, section AS INTEGER = 1, animated AS INTEGER = 1)**

Deselects the indicated cell in the specified row and section, or in the first section if no section is specified. If the **animated** flag is set, the change is animated.

Snippet

```
! Deselect the initial selection from the table shown in the the
! example at the start of the class.
deselect(1)
```

---

**FUNCTION getText (row AS INTEGER, section AS INTEGER = 1) AS STRING**

Returns the string in the specified cell.

See the snippet at the start of the class for an example of `getText`.

---

**SUB insertRow (title AS STRING, row AS INTEGER, section AS INTEGER = 1)**

Inserts a new row in the table.

**title** is the string that will appear in the new cell.

**row** is the row number the new cell will appear before, starting at 1. Pass 1 to place the cell at the start of the section, or the number of rows in the section plus one to place the cell at the end of the section.

**section** is the number of the section the row will be inserted into, counting from 1.

See the snippet at the start of the class for an example of `insertRow`.

---

**SUB insertRows (titles() AS STRING, row AS INTEGER, section AS INTEGER = 1)**

Inserts one or more news row in the table.

**titles** is an array of any length containing the strings that will appear in the new cells. The length of the array determines the number of cells that will be inserted.

**row** is the row number the new cells will appear before, starting at 1. Pass 1 to place the cells at the start of the section, or the number of rows in the section plus one to place the cells at the end of the section.

**section** is the number of the section the rows will be inserted into, counting from 1.

See the snippet at the start of the class for an example of `insertRows`.

---

**SUB insertSections (sections() AS INTEGER)**

Inserts one or more news sections in the table.

**sections** is an array containing the number of the existing section the new section will be inserted before, counting from 1. Sections are inserted from the end of the table working toward the beginning, regardless of the order they appear in the array. This insures the section numbers in the original array refer to the sections numbers before the insertions begin. The length of the array determines the number of sections that will be inserted.

See the snippet at the start of the class for an example of `insertSections`.

---

**FUNCTION rows (section AS INTEGER = 1) AS INTEGER**

Returns the number of rows in the specified section.

---

**SUB scroll (row AS INTEGER, section AS INTEGER = 1, animated AS INTEGER = 1, position AS INTEGER = 0)**

Scrolls the table so the cell specified by **row** and **section** is visible.

If the **animated** parameter is non-zero, the scroll action is animated; if it is 0, the table is updated without animation.

The **position** parameter specifies the preferred position for the cell once the scroll action is complete. It can be any value from this table.

position	Cell Position
1	No preference
2	Top
3	Middle
4	Bottom

Snippet

```
! Scroll to the Roman numeral I in the table in the example
! at the start of the class. Scroll with no animation, and
! place the cell at the bottom of the table.
tb.scroll(1, 3, 0, 4)
```

---

**FUNCTION sections AS INTEGER**

Returns the number of sections in the table.

---

**FUNCTION selection () (,) AS INTEGER**

Returns the selected cells in the table.

The returned array can have zero or more rows, each of which will have two columns. The first column in each row is the row number of a selected cell, while the second is the section number, both counting from 1. Use UBOUND to determine the number of cells selected.

Snippet

```
! Print the selected cells in the table tb.
s = tb.selection
PRINT "Selected Cells:"
IF UBOUND(s, 1) = 0 THEN
    PRINT "  No cells selected."
ELSE
    FOR i = 1 TO UBOUND(s, 1)
        PRINT "  Row = "; s(i, 1); "; section = "; s(i, 2)
    NEXT
END IF
```

---

**SUB setAllowsMultipleSelection (flag AS INTEGER)**

Pass a non-zero value to allow multiple cells to be selected at the same time, or 0 to only allow selection of one cell at a time.

The default is to not allow multiple selection.

---

**SUB setAllowsSelection (flag AS INTEGER)**

Pass a non-zero value to allow cells to be selected, or 0 to prevent cells from being selected.

The default is to allow cells to be selected.

---

**SUB setFont (name AS STRING, size, style AS INTEGER)**

Sets the font for the cells. See `setFont` in the `Graphics` class for a discussion of fonts.

See the snippet at the start of the class for an example of `setFont`.

---

**SUB setSectionTitle (title AS STRING, section AS INTEGER)**

Sets the title of the specified section.



Sections with no titles can be used to organize the table in the program, but there is no visual indication in the table that sections without titles exist. If a title is provided, it appears above all of the cells in the section, and stays at the top of the table as the user scrolls through the section.

See the snippet at the start of the class for an example of `setSectionTitle`.

---

**SUB setSelection (row AS INTEGER, section AS INTEGER = 1, animated AS INTEGER = 1, position AS INTEGER = 1)**

Sets the selected cell to the cell specified by the **row** and **section** parameters.

If a non-zero value is passed for **animated**, the selection is animated. If zero is passed, the change is made without animation.

The table will scroll to show the selected cell. The **position** parameter specifies where the cell will be in the table after scrolling occurs.

---

position	Cell Position
1	No preference
2	Top
3	Middle
4	Bottom

---

**SUB setText (cell AS STRING, row AS INTEGER, section AS INTEGER = 1)**

Sets the text displayed in a cell.

**row** and **section** are the row and section number for the cell to change, counting from 1.

---

**FUNCTION visibleRows (,) AS INTEGER**

Returns the rows that are visible in the table.

The returned array can have zero or more rows, each of which will have two columns. The first column in each row is the row number of a selected cell, while the second is the section number, both counting from 1. Use `UBOUND` to determine the number of cells selected.

#### Snippet

```
! Print the visible cells in the table tb.
s = tb.visibleRows
PRINT "Visible Cells:"
IF UBOUND(s, 1) = 0 THEN
    PRINT "    No cells selected."
ELSE
    FOR i = 1 TO UBOUND(s, 1)
        PRINT "    Row = "; s(i, 1); "; section = "; s(i, 2)
    NEXT
END IF
```

---

## TextField

`TextField` objects display a one-line field for entering text.

`TextField` objects are returned by the `newTextField` method of the `Graphics` class. The methods in this class are used to manipulate the text field after it has been created.

`TextField` descends from the `Control` class. All methods in the `Control` class also apply to the `TextField` class.

Tapping a text field starts the editing process, either displaying the software keyboard or allowing entry with a physical keyboard. As text is changed, text changed events are posted. These are handled by the program's `textChanged` subroutine. When the return key (or its equivalent) is pressed, a value changed event is posted. The

value changed event is handled by the program's valueChanged subroutine. See Chapter 13 for a complete discussion of application methods like textChanged and valueChanged.

The snippet shows a text field used to enter titles for a button. It illustrates several of the methods in this class. As text is typed, it is shown in a prototype of the button. When the return key is pressed, the text is placed in the final button.

#### Snippet

```
DIM label1 AS Label, label2 AS Label, label3 AS Label

    label1 = Graphics.newLabel(10, 10, 110)
    label1.setText("Button name:")
    label1.setBackgroundColor(0.9, 0.9, 0.9)
    label1.setAlignment(3)

    label2 = Graphics.newLabel(10, 49, 110)
    label2.setText("Prototype:")
    label2.setBackgroundColor(0.9, 0.9, 0.9)
    label2.setAlignment(3)

    label3 = Graphics.newLabel(10, 99, 110)
    label3.setText("Final:")
    label3.setBackgroundColor(0.9, 0.9, 0.9)
    label3.setAlignment(3)

DIM tf AS TextField, button1 AS Button, button2 AS Button
tf = Graphics.newTextField(130, 10, 160, 22)
tf.setBackgroundColor(1, 1, 1)

button1 = Graphics.newButton(130, 42, 160)

button2 = Graphics.newButton(130, 92, 160)

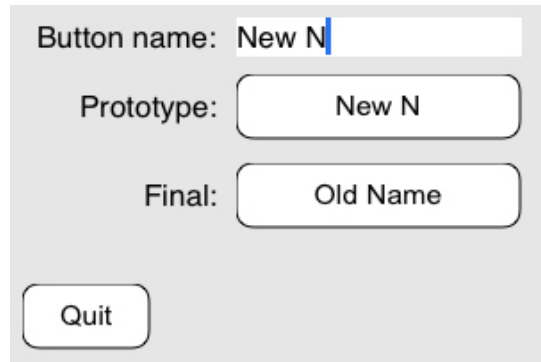
DIM quit AS Button
quit = Graphics.newButton(10, 160)
quit.setTitle("Quit")

Graphics.setColor(0.9, 0.9, 0.9)
Graphics.fillRect(0, 0, 1024, 1024)
System.showGraphics

SUB touchUpInside (ctrl AS Button, when AS DOUBLE)
    IF ctrl = quit THEN
        STOP
    END IF
END SUB

SUB textChanged (ctrl AS Control, time AS DOUBLE)
    button1.setTitle(tf.getText)
END SUB

SUB valueChanged (ctrl AS Control, time AS DOUBLE)
    button2.setTitle(tf.getText)
END SUB
```




---

#### FUNCTION getText AS STRING

Returns the text in the text field.

See the snippet at the start of the class for an example of `getText`.

---

**SUB setAlignment (alignment AS INTEGER)**

Sets the alignment of the text in the text field. The default alignment is left aligned text.

Value	Resulting Alignment
1	Left aligned
2	Centered
3	Right Aligned
4	Justified
5	Natural (default)

See the snippet at the start of the class for an example of `setAlignment`.

---

**SUB setAutoCapitalization (value AS INTEGER)**

Pass a non-zero value to automatically capitalize the first word in each sentence (the default) or zero to leave letter case as it is typed.

---

**SUB setAutoCorrection (value AS INTEGER)**

Pass a non-zero value to automatically present corrections to mistyped words (the default) or zero to turn off auto correction.

---

**SUB setColor (red, green, blue, alpha = 1)**

Sets the text color.

See *Colors* at the start of the discussion of the `Plot` class for information on RGB colors.

---

**SUB setFont (name AS STRING, size, style AS INTEGER)**

Sets the font for the text. See `setFont` in the `Graphics` class for a discussion of fonts.

---

**SUB setKeyboardAppearance (value AS INTEGER)**

Pass 2 to the method to set the keyboard to an alert-style keyboard, or 1 to use the default keyboard.

---

**SUB setKeyboardType (value AS INTEGER)**

Sets the type of software keyboard.

value	Keyboard Style
1	Default
2	ASCII Capable
3	Numbers and Punctuation
4	URL
5	Number Pad
6	Phone Pad
7	Name and Phone Pad
8	Email Address
9	Decimal Pad
10	Twitter
11	Alphabet

---

**SUB setSecureTextEntry (value AS INTEGER)**

Pass a non-zero value to use secure text entry, or zero for normal text entry.

With secure text entry, only the last letter typed is visible in the text field. All other letters are displayed as a spot.

---

**SUB setSpellChecking (value AS INTEGER)**

Pass a non-zero value to use automatic spell checking, or zero to disable it.

---

**SUB setText (text AS STRING)**

Sets the text displayed in the text field.

---

## TextView

TextView objects display and edit multiple lines of text.

TextView objects are returned by the `newTextView` method of the `Graphics` class. The methods in this class are used to manipulate the text view after it has been created.

TextView descends from the `Control` class. All methods in the `Control` class also apply to the `TextView` class.

Tapping a text field starts the editing process, either displaying the software keyboard or allowing entry with a physical keyboard. As text is changed, text changed events are posted. These are handled by the program's `textChanged` subroutine.

See Chapter 13 for a complete discussion of application methods like `textChanged`.

The snippet shows two text views. It illustrates several of the methods in this class. The top text field is editable; as text is typed in this text field, it is copied into the text field below. The bottom text field is not editable, and several other options have been set for it.

### Snippet

```
DIM editable AS TextView, echo AS TextView
editable = Graphics.newTextView(10, 10, 160, 120)

echo = Graphics.newTextView(10, 140, 160, 120)
echo.setEditable(0)
echo.setAlignment(2)
echo.setColor(1, 0, 0)
echo.setFont("Ariel", 16, 2)

DIM quit AS Button
quit = Graphics.newButton(10, 270)
quit.setTitle("Quit")

Graphics.setColor(0.9, 0.9, 0.9)
Graphics.fillRect(0, 0, 1024, 1024)
System.showGraphics

SUB touchUpInside (ctrl AS Button, when AS DOUBLE)
IF ctrl = quit THEN
    STOP
END IF
END SUB

SUB textChanged (ctrl AS Control, time AS DOUBLE)
echo.setText(editable.getText)
END SUB
```

bands which have  
connected then with  
another and to assume  
among the powers of the  
earth, the separate and  
equal station to which the  
Laws of Nature and of  
Nature's God entitle them.

*When in the course  
of human events it  
becomes necessary  
for one people to  
dissolve the political  
hands which have*

Quit

---

**FUNCTION `getText` AS STRING**

Returns the text in the text view.

See the snippet at the start of the class for an example of `getText`.

---

**SUB `replaceText` (`start` AS INTEGER, `length` AS INTEGER, `text` AS STRING)**

Replaces the text with new text.

The text to replace starts at the first location, and ends **`length`** characters past the start location. Pass a **`length`** of 0 to insert text.

The text that replaces the old text is passed as the **`text`** parameter.

---

**SUB `scroll` (`start` AS INTEGER, `length` AS INTEGER)**

Scrolls the text view so the text at the indicated location is visible.

**`start`** is the offset from the start of the text of the first character that should be visible, while **`length`** is the number of characters to make visible.

---

**FUNCTION `selectionLength` AS INTEGER**

Returns the number of characters in the current selection.

---

**FUNCTION `selectionStart` AS INTEGER**

Returns the offset from the start of the text of the current selection. This is the location where the next character typed will be entered. If `selectionLength` is non-zero, the selected text is replaced when the character is typed.

---

**SUB `setAlignment` (`alignment` AS INTEGER)**

Sets the alignment of the text in the text view. The default alignment is left aligned text.

Value	Resulting Alignment
1	Left aligned
2	Centered
3	Right Aligned
4	Justified
5	Natural (default)

See the snippet at the start of the class for an example of `setAlignment`.

---

**SUB `setAutoCapitalization` (`value` AS INTEGER)**

Pass a non-zero value to automatically capitalize the first word in each sentence (the default) or zero to leave letter case as it is typed.

---

**SUB `setAutoCorrection` (`value` AS INTEGER)**

Pass a non-zero value to automatically present corrections to mistyped words (the default) or zero to turn off auto correction.

---

**SUB `setColor` (`red`, `green`, `blue`, `alpha` = 1)**

Sets the text color.

See *Colors* at the start of the discussion of the `Plot` class for information on RGB colors.

---

**SUB `setEditable` (`value` AS INTEGER)**

Pass a non-zero value to allow the text to be edited, or a zero value to make the text view read-only.

See the snippet at the start of the class for an example of `setAlignment`.

---

**SUB setFont (name AS STRING, size, style AS INTEGER)**

Sets the font for the text. See `setFont` in the `Graphics` class for a discussion of fonts.

See the snippet at the start of the class for an example of `setAlignment`.

---

**SUB setKeyboardAppearance (value AS INTEGER)**

Pass 2 to the method to set the keyboard to an alert-style keyboard, or 1 to use the default keyboard.

---

**SUB setKeyboardType (value AS INTEGER)**

Sets the type of software keyboard.

value	Keyboard Style
1	Default
2	ASCII Capable
3	Numbers and Punctuation
4	URL
5	Number Pad
6	Name and Phone Pad
7	Email Address
8	Decimal Pad
9	Twitter
10	Alphabet

---

**SUB setSecureTextEntry (value AS INTEGER)**

Pass a non-zero value to use secure text entry, or zero for normal text entry.

With secure text entry, only the last letter typed is visible in the text view. All other letters are displayed as a spot.

---

**SUB setSelection (start AS INTEGER, length AS INTEGER)**

Selects the indicated range of text.

**start** is the offset from the start of the text of the first character to select. **length** is the number of characters to select; pass zero to set the insertion point (the location where new text will be placed) without selecting any text.

---

**SUB setSpellChecking (value AS INTEGER)**

Pass a non-zero value to use automatic spell checking, or zero to disable it.

---

**SUB setText (text AS STRING)**

Sets the text displayed in the text view.

---

## WebView

`WebView` objects display web pages and many kinds of documents. The objects also provide basic web navigation functions, such as moving forward and backward through the browsing history.

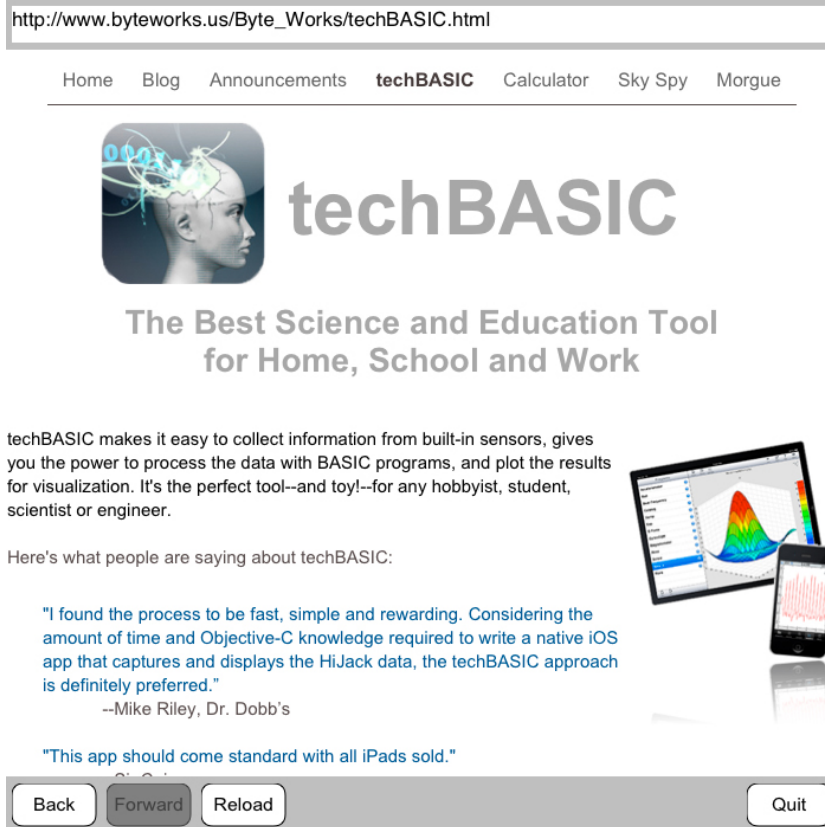
`WebView` objects are returned by the `newWebView` method of the `Graphics` class. The methods in this class are used to manipulate the web view after it has been created.

`WebView` descends from the `Control` class. All methods in the `Control` class also apply to the `WebView` class.

A web start load event is generated when a web page or document starts to load. This is handled by the program's `startedLoad` subroutine. An end load event is generated with the web page finishes loading; this is handled by the `finishedLoad` subroutine. If an error occurs during loading, a load error event is triggered and

passed to the `loadError` subroutine. See Chapter 13 for a complete discussion of application methods like `startedLoad`, `finishedLoad` and `loadError`.

The snippet shows a simple but functioning web browser. It illustrates several of the methods in this class.



#### Snippet

```
DIM back AS Button, forward AS Button, reload AS Button, quit AS Button
DIM url AS TextField
DIM webPage AS WebView
DIM busy AS Activity

pageHeight = Graphics.height
pageWidth = Graphics.width
Graphics.setColor(0.75, 0.75, 0.75)
Graphics.fillRect(0, 0, pageWidth, pageHeight)

back = Graphics.newButton(5, pageHeight - 42)
back.setTitle("Back")
back.setEnabled(0)

forward = Graphics.newButton(85, pageHeight - 42)
forward.setTitle("Forward")
forward.setEnabled(0)

reload = Graphics.newButton(165, pageHeight - 42)
reload.setTitle("Reload")

quit = Graphics.newButton(pageWidth - 77, pageHeight - 42)
quit.setTitle("Quit")
```

## Chapter 17: GUI Classes

```
webPage = Graphics.newWebView(0, 42, pageWidth, pageHeight - 90)
webPage.setScalesPage(1)

url = Graphics.newTextField(5, 5, pageWidth - 10)
url.setBackgroundColor(1, 1, 1)
url.setAutoCapitalization(1)

busy = Graphics.newActivity(pageWidth/2, pageHeight/2)
busy.setColor(0, 0, 0)
busy.stopAnimation

System.showGraphics

home$ = "http://www.byteworks.us"
webPage.loadURL(home$)

SUB touchUpInside (ctrl AS Button, when AS DOUBLE)
IF ctrl = back THEN
    webPage.goBack
ELSE IF ctrl = forward THEN
    webPage.goForward
ELSE IF ctrl = reload THEN
    webPage.reload
ELSE IF ctrl = quit THEN
    STOP
end IF
END SUB

SUB startedLoad (ctrl AS WebView, when AS DOUBLE, message AS
STRING)busy.startAnimation
url.setText(message)
END SUB

SUB finishedLoad (ctrl AS WebView, when AS DOUBLE, url AS STRING)
busy.stopAnimation
back.setEnabled(webPage.canGoBack)
forward.setEnabled(webPage.canGoForward)
END SUB

SUB loadError (ctrl AS WebView, when AS DOUBLE, message AS STRING, err AS
LONG)
print "Web page failed to load. Error = "; err
END SUB

SUB valueChanged (ctrl AS Control, when AS DOUBLE)
IF ctrl = url THEN
    webPage.loadURL(url.getText)
END IF
END SUB
```

---

### **FUNCTION canGoBack AS INTEGER**

Returns one if the browser can go back to a previous page, and zero if it cannot.  
See the snippet at the start of the class for an example of `canGoBack`.

---

### **FUNCTION canGoForward AS INTEGER**

Returns one if the browser can go forward to a subsequent page, and zero if it cannot.



See the snippet at the start of the class for an example of `canGoForward`.

---

**SUB goBack**

Go back to the previous page.

See the snippet at the start of the class for an example of `canGoBack`.

---

**SUB goForward**

Go back to the subsequent page.

See the snippet at the start of the class for an example of `canGoForward`.

---

**FUNCTION isLoading AS INTEGER**

Returns one if the browser is loading a page, and zero if it is not.

---

**SUB loadDocument (file AS STRING)**

While the obvious use for a web view is to display a web page, a web view can also load and display many types of documents. The following table shows the currently supported documents, along with the file extension that must be used to indicate the document type. For Microsoft Office documents, the document must be saved with Microsoft Office 97 or newer formats.

Extension	Kind of Document
.xls	Excel
.key.zip	Keynote
.key	Keynote '09
.numbers.zip	Numbers
.numbers	Numbers '09
.pages.zip	Pages
.pages	Pages '09
.pdf	PDF
.ppt	Powerpoint
.rtf	Rich Text Format
.rtfd.txt	Rich Text Format Directory
.doc	Word

---

**SUB loadURL (url AS STRING)**

Load a web page.

See the snippet at the start of the class for an example of `loadURL`.

---

**SUB reload**

Reloads the current web page.

---

**SUB setScalesPage (flag AS INTEGER)**

Pass 0 to force the web page to load at its natural size, or 1 to scale the web page to fit the current view and allow the user to scale the page.

---

**SUB stopLoading**

Stop loading the current web page.



## Appendix A – Error Messages

This appendix shows all of the errors that techBASIC can generate. It starts with a table that shows the error number and error message.

There are two fundamentally different kinds of errors. Compiler errors are generated when the program is compiled, before it starts to run. Run-time errors occur while your program is running. Runtime errors can be intercepted with an `ON ERROR` handler and dealt with by your program; compiler errors have to be corrected before the program will run at all.

---

### Compiler Errors

---

#### **'#' expected**

On a statement like `OPEN`, which requires a file number, there was no `#` character where the file number was expected.

#### **'(' expected**

A left parenthesis was expected, generally right after the name of a built-in function like `SIN`, but was not found.

#### **')' expected**

A right parenthesis was expected, but not found. This can occur in several situations, such as after the parameters for a subroutine or function, or when dimensioning an array.

#### **',' expected**

A comma was expected, but not found. This can occur many places, such as in statements like `PUT` or function calls with multiple parameters, such as `RIGHT`.

#### **'.' expected**

Something appeared after a function that returns an object, but it was not a `.` operator.

#### **';' expected**

The `PRINT USING` statement requires a `;` after the format string, but it was not found.

#### **= expected**

The `=` token was missing from a `FOR`, `MAT` or `LET` statement.

#### **']' expected**

The `]` character to close an array constant was not found.

#### **An array cannot be used here**

An array variable was used in an expression, but only scalar or string values are allowed where the array was found. For example, an array cannot be divided by another array.

#### **Array expected**

An array was expected in an expression or as the argument for `SIZE`, `LBOUND` or `UBOUND`, but a scalar was found.

---

**Arrays, variables and subroutines must have distinct names**

An array, variable or subroutine was found with the same name as an existing Array, variable or subroutine. Arrays cannot have the same name as a non-array variable, and variables cannot have the same name as subroutines. This restriction in standard BASIC is what allows techBASIC to make LET, MAT and CALL optional when entering programs.

---

**AS expected**

The NAME and OPEN statement both expect AS to appear as part of the statement, but it was not found.

---

**BASE may only appear once, before any use of an array**

A BASE statement was found, but it appeared after another BASE statement or after an array. The BASE statement can only be used once, and must appear before the first use of an array.

---

**BASE must be set or 0 or 1**

An attempt was made to set BASE to something other than 0 or 1.

---

**Cannot mix arrays and scalars here**

Addition and subtraction are legal for arrays and for variables, but you cannot add or subtract an array and variables. This error is flagged when an attempt is made to do so.

---

**Cannot redimension a variable**

A second DIM statement was encountered for the same non-array variable. Only one DIM statement may be used for each non-array variable.

---

**CASE expected**

Either CASE was omitted after the SELECT token, or a block inside a SELECT CASE block was found that did not start with CASE.

---

**Class name expected**

An identifier was found where a type was expected, but it was not the name of a predefined class. For example,

```
DIM p AS Plots
```

would cause this error, since the built in class is named Plot, not Plots.

---

**Duplicate subroutine or function name**

A subroutine or function was found with the same name as an existing subroutine or function. Each subroutine and function must have a unique name.

---

**END FUNCTION expected**

The end of the program was reached while compiling a FUNCTION. The END FUNCTION statement is required.

---

**END IF expected**

An IF-THEN statement was started, but no END IF was found.

Remember to include an END IF for all block IF statements. For example,

```
IF found THEN CALL Process
```

is a perfectly legal BASIC statement, contained entirely on one line, but

```
IF found THEN
  CALL Process
```

needs an `END IF`.

---

#### **End of line expected**

This error occurs when the compiler expects a statement to be complete, but other tokens were found on the line. Keep in mind that it can easily occur because something was left out—like a + sign in an equation on a `LET` statement.

---

#### **END SELECT expected**

No `END SELECT` was found at the end of a `SELECT CASE` statement.

---

#### **END SUB expected**

The end of the program was reached while compiling a subroutine. The `END SUB` statement is required.

---

#### **Expression expected**

An expression was expected, but one was not found. This can occur anywhere an expression is required, such as after the = sign of a `LET` statement. It occurs both when the statement is missing entirely and when a token that cannot start an expression is found.

---

#### **FOR expected**

The `FOR` token in an `OPEN` statement was missing or misplaced.

---

#### **Further errors suppressed**

If more than 25 errors are found in a program, the compiler stops and reports this error.

---

#### **Function name expected**

A method, such as `createPlot` in a `Plot` object, expected to be passed the name of a user-defined function, but found something else.

---

#### **GOTO expected**

The tokens `ON ERROR` were not followed by the required `GOTO` token.

---

#### **GOTO or GOSUB expected**

An `ON` statement was followed by an expression, indicating that the next token should be either `GOTO` or `GOSUB`, but neither token was found. This could be caused by an error in the expression.

---

#### **Identifier expected**

The compiler expected to find the name of a variable, array, subroutine or function, but found some other token. In most cases, the reason is obvious, but be sure to check to see if a reserved word was used as a variable name.

---

#### **Illegal character**

A character that is not allowed outside a comment or string constant was found in the program.

---

#### **INPUT expected**

The `INPUT` token in a `LINE INPUT` statement was missing.

---

#### **Integer expected**

Something other than an integer was used on the `BASE` statement. Expressions are not allowed on the `BASE` statement.

---

**LEN required for RANDOM files**

An attempt was made to open a random access file, but no length was specified. A length is required for random access files.

---

**Line number or label expected**

A line number was expected for a GOTO or GOSUB, but the token found was not a line number or line label.

---

**LOOP expected**

A DO statement was started, but the program or procedure finished without finding a matching LOOP.

Make sure there is exactly one LOOP for each DO. Keep in mind that other mismatched statements might cause this error even if there is a matching LOOP for the DO. For example,

```
DO
  GET #1, CH
  IF CH = 13 THEN
    PRINT
  LOOP WHILE NOT EOF(1)
```

would cause this error if the first value read is not 13, even though the real error is that the IF statement has no matching END IF.

---

**NEXT expected**

No NEXT was found at the end of a FOR statement.

---

**NEXT variable does not match the FOR control variable**

A NEXT statement specified a loop variable, but the variable did not match the loop variable used in the FOR statement the NEXT statement completes.

---

**Object name expected**

A . operation appeared after an identifier that was not an object.

---

**Out of memory**

techBASIC ran out of memory compiling the program.

---

**OUTPUT, INPUT, APPEND, RANDOM or BINARY expected**

An OPEN statement found something other than one of these valid methods for opening a file.

---

**Statement expected**

A line started with something other than one of the tokens that can start a BASIC statement.

---

**String constant does not end with "**

A string constant was started, but no closing " character was found.

---

**SUB or FUNCTION cannot be nested**

A SUB or FUNCTION appeared inside another SUB or FUNCTION. This can easily happen because the END SUB or END FUNCTION statement was omitted.

---

**Subroutine not found**

The name of a subroutine was expected after a CALL statement, but the token found was not the name of a subroutine.

**THEN expected**

The THEN token was not found for an IF statement.

**TO expected**

The TO token was not found in a FOR statement.

**Type expected**

In a location where a type was expected, such as after AS in a DIM statement or parameter list, a token was found which was not the name of a type or an identifier that could have been an object.

**WEND expected**

No WEND was found at the end of a WHILE statement.

---

## Runtime Errors

Runtime errors occur when the program encounters an error while it is running. These are easy to distinguish from compiler errors because they always start with “Runtime error:” The error number shown is the error number returned by the ERR function.

The table is followed by an expanded description of the short text error message printed by techBASIC, along with common causes for the error.

Error	Message
1	Type mismatch
2	Integer overflow
3	Label not found
4	RETURN or POP with no corresponding GOSUB
5	READ with no corresponding DATA
6	Mismatched quotes
7	Bad input on INPUT or DATA
8	Subroutine, function or field not found: <name>
9	Not enough parameters
10	Too many parameters
11	Arrays of pointers are not supported
12	Values passed BYREF must be assignable
13	The array index is out of bounds
14	File number must be in the range 1..32767
15	The length of a file buffer must be greater than 0
16	The file <name> is not open
17	The file <name> is already open
18	Could not open <name>
19	The file <number> is not open for input
20	Error while reading <number>
21	Error closing <number>
22	INPUT format error while reading <number>
23	The file <file> is not open for output
24	Error while writing to <number>
25	Could not create the directory <name>
26	The directory <name> does not exist
27	<name> could not be renamed
28	An attempt was made to access the file <name> which is outside of the sandbox
29	Null object: An attempt was made to access an object that does not exist
30	This value cannot be printed

## Appendix A – Error Messages

31	Subroutine calls are nested too deep
32	Redimensioned array
33	Out of memory
34	The file <name> cannot be killed because it is open
35	The total number of entries in an array is too large
36	Stopped by user
37	The index is too large or too small for the given array
38	There were no formats in the format specifier
39	Unfinished format specifier
40	The format specifier is too long
41	The number of array subscripts or indices does not match
42	The matrix is singular
43	Attempting to GET a value as an array that was not written as an array
44	Could not open or read the image file <name>
45	No image exists
46	Desired accuracy was not achieved in the allowed number of steps
47	Failed due to numeric instability
48	There were not enough points for the specified regression
50	The second argument of MOD cannot be zero
51	The specified value is not allowed
52	The network domain could not be opened (<number>)
53	Error while accessing file <number>
54	Error accessing the domain: <name>
55	The network operation did not complete in the allowed time
56	The UUID does not conform to the allowed format
57	Audio input could not be started or stopped

---

### **<name> could not be renamed**

An attempt was made to rename a file, but the file could not be renamed. This can happen if the file does not exist, is locked, or if the program has insufficient privileged to rename the file.

---

### **An attempt was made to access the file <name> which is outside of the sandbox**

Some files are protected from access by techBASIC. If you access one of these files, and it is detected by techBASIC rather than the operating system, this error will be flagged.

---

### **Arrays of pointers are not supported**

This error is not used in the iOS implementation of techBASIC.

---

### **Attempting to GET a value as an array that was not written as an array**

An attempt was made to use the GET statement to read the value of an array from a file, but the value in the file was not written using PUT to write an array.

---

### **Audio input could not be started or stopped**

An audio command was used that needed to start or stop the audio input system, but the command failed.

---

### **Bad input on INPUT or DATA**

An attempt was made to read a number using the READ statement, but a non-numeric value was found.

---

### **Could not create the directory <name>**

An attempt was made to create a new directory, but the attempt failed.  
Check the name to make sure it does not contain a / character.

---

### **Could not open <name>**

An attempt was made to open a file, but it could not be opened.



This error occurs when the file is opened for input. The most likely reason the file cannot be opened is that it does not exist. Files cannot be opened for APPEND, OUTPUT or INPUT unless they already exist.

---

**Could not open or read the image file <name>**

An attempt was made to open an image file, but it could not be opened.

The most likely reason the file cannot be opened is that it does not exist.

---

**Desired accuracy was not achieved in the allowed number of steps**

Some numeric integration methods put an upper limit of iterative steps that will be made in an attempt to reach a desired degree of accuracy. This error occurs when the number of steps is exceeded.

Check the integral visually to see if there are issues like discontinuities. Try increasing the number of steps or decreasing the desired accuracy. Try a different integration method.

---

**Error accessing the domain: <name>**

An error was encountered while accessing a network connection. The name is the URL used to open the connection.

---

**Error while accessing file <number>**

An error was encountered while accessing a network connection. The number indicates the channel number used to open the network connection.

---

**Error while reading <number>**

An error occurred while reading from a file, probably because an attempt was made to read past the end of the file.

---

**Error while writing to <number>**

The operating system reported an error while writing to the file. This could be caused by a disk full error.

---

**Failed due to numeric instability**

Linear regression failed.

This occurs when the regression matrix is singular or nearly singular. This typically means there is no good fit for the data.

---

**File number must be in the range 1..32767**

The programs used a file number outside the allowed range.

Change the file number to a number in the allowed range.

---

**Subroutine calls are nested too deep**

More than 4096 recursive GOSUB calls were attempted, or SUB or FUNCTION calls were made to a depth of 2048. techBASIC blocks excessive GOSUB calls to prevent crashes due to out of memory errors.

---

**INPUT format error while reading <number>**

An attempt was made to read a number from a file using the INPUT statement, but a non-numeric value was found. The number is the file number read from.

---

**Integer overflow**

An integer value was outside the range -32768..32767, or a long value was outside the range -2147483648..2147483647.

Converting intermediate values to SINGLE or DOUBLE is one effective way to avoid an integer overflow.

---

**Label not found**

A GOTO, GOSUB, ON ... GOTO or ON ... GOSUB was made to a label that was not found in the program.

---

### Mismatched quotes

A READ statement was reading a string, and found an initial " character, but not a final one.

---

### No image exists

An Image object was used before a valid image was loaded into the object.

Load an image with the `getCameraImage`, `getPhotoImage` or `load` methods before trying to use the image. After loading the image, use the `hasImage` method to verify the image was successfully loaded.

---

### Not enough parameters

A method was called with too few parameters.

---

### Null object: An attempt was made to access an object that does not exist

An object was dereferenced, but nothing had been assigned to the object.

This generally happens because a statement assigning something to the object was omitted. For example,

```
DIM p AS Plot
p = graphics.newPlot
p.setTitle("Plot Title")
```

will work just fine, but

```
DIM p AS Plot
p.setTitle("Plot Title")
```

will generate this error.

---

### Out of memory

techBASIC ran out of memory while running the program.

---

### READ with no corresponding DATA

A READ statement was encountered, but there was no DATA statement to read the data from.

---

### Redimensioned array

An array was dimensioned with the DIM statement after it was already dimensioned with another DIM statement, or after it was dimensioned by being used in an expression. The number of subscripts in the new DIM statement was different than the number of subscripts when the array was previously dimensioned.

It is allowed to use a DIM statement to change the number of elements in a subscript, but not the number of subscripts. For example,

```
DIM a(4, 5)
DIM a(10, 12)
```

is allowed, but

```
DIM a(4, 5)
DIM a(10) : ! Not valid.
```

is not allowed.

---

**RETURN or POP with no corresponding GOSUB**

A RETURN or POP was encountered, but there was no GOSUB to return from.

Each GOSUB or ON ... GOSUB call records a location to return to when the subroutine completes. POP can be used to remove this location, or RETURN can return from the subroutine. There must be exactly one POP or RETURN for each GOSUB. This error occurs when there are more POP or RETURN statements than GOSUB calls.

---

**Stopped by user.**

This error occurs when the user presses the Stop key to stop the running program.

---

**Subroutine, function or field not found: <name>**

An object was accessed, but the named entity was not found. For example, this error is flagged when the name of a method in a built-in object is misspelled.

```
DIM p AS Plot
p = graphics.newPlots: ! Error: should be createPlot
```

---

**The array index is out of bounds**

An attempt was made to access an element of an array that did not exist. For example,

```
DIM a(4)
FOR i = 0 TO 4
  a(i) = i : ! Error; arrays index from 1 by default (See BASE)
NEXT
```

is an error, since the initial index of an array is 1.

---

**The directory <name> does not exist**

The CHDIR command was used to change the default directory, but the directory does not exist.

Check the spelling of the directory name carefully for errors.

---

**The index is too large or too small for the given array**

The LBOUND, UBOUND or SIZE function was used with a value that is not a valid subscript for the array. For example, the array

```
DIM a(3, 4, 4)
```

has 3 subscripts, so 1, 2, or 3 is a valid subscript index, but 0 or 4 is not. The statement

```
upper = UBOUND(a, 3)
```

is valid, and will set upper to 4, but the statement

```
upper = UBOUND(a, 4)
```

is not valid, since the array a only has 3 subscripts.

---

**The length of a file buffer must be greater than 0**

An attempt was made to open a file using a LEN value that was negative or 0. File buffer lengths specified by LEN must be 1 or larger.

---

**The total number of entries in an array is too large**

An attempt was made to create an array with more than 32767 elements in the array. Arrays are limited to 32767 total elements.

---

**The file <name> cannot be killed because it is open**

The KILL command was used to delete a file, but the file was open at the time. Close the file first, then delete it.

---

**The file <name> is already open**

An attempt was made to open a file that was already open. Close the file, then open it.

---

**The file <name> is not open**

An attempt was made to perform an operation on a file that was not open.

This is not limited to trying to read or write a file that is not open. This error can also occur from other operations, like CLOSE or EOF.

---

**The file <number> is not open for input**

An attempt was made to read from a file that was not open for input. This can mean that the file was not open at all, or that the file was open for output only.

---

**The file <file> is not open for output**

An attempt was made to write to a file that was not open for output. This can mean that the file was not open at all, or that the file was open for input only.

---

**The format specifier is too long**

Individual format specifiers for numbers are limited to 24 characters. For example,

```
PRINT USING "%.#####"; 3.1415926535897932
```

is fine, but

```
PRINT USING "%.#####"; 3.1415926535897932
```

will generate an error. Keep in mind that even double-precision values are only accurate to about 16 decimal digits.

---

**The matrix is singular**

An attempt was made to invert or find the determinant of a matrix, but the matrix either has no determinant or inverse, or it is so close to being singular that numeric precision prevented its calculation.

---

**The network domain could not be opened (<number>)**

An attempt was made to open a network connection, but the domain was invalid. The numeric error code gives more information about the reason for the failure.

Error	Description
1	The address family for hostname is not supported.
2	Temporary failure in name resolution.
3	Invalid value for ai_flags.
4	Non-recoverable failure in name resolution.
5	ai_family not supported.
6	Memory allocation failure.
7	No address associated with hostname.
8	Hostname nor servname provided, or not known.
9	Servname not supported for ai_socktype.
10	ai_socktype not supported.
11	System error.
12	Invalid value for hints.
13	Resolved protocol is unknown.
14	Argument buffer overflow.

---

**The network operation did not complete in the allowed time**

Some network operations are limited to five seconds for completion; these are operations that are expected to complete fairly quickly. This error is flagged if the network operation does not complete in the allowed time.

---

**The number of array subscripts or indices does not match**

An array operation was performed that either attempted to change the number of subscripts in an array, or attempted to perform an operation on two arrays with inappropriate sizes.

This example is not valid because the number of subscripts is changed.

```
DIM a(4, 4)
a = [1, 2, 3, 4] : ! Not valid: Changes the number of subscripts
```

The following operation is not valid because multiplication of matrices only works when the number of elements in the second subscript of the first array matches the number of elements in the first subscript of the second array.

```
DIM a(3, 4), b(5, 6)
! Not valid: The size of the second subscript of A must match the
! first subscript of b
a = a*b
PRINT a
```

This example corrects the error, resulting in a valid, if uninteresting, multiplication.

```
DIM a(3, 4), b(4, 6)
a = a*b
PRINT A
```

---

**The second argument of MOD cannot be zero**

The MOD operator was used with a second operand of zero. Since the MOD operator returns the remainder from an integer division, the second argument cannot be zero.

---

**The specified value is not allowed**

A value was passed to a library method, but the value is not allowed.

This generally means the value passed as a parameter was NaN (not a number) or infinity.

---

**The UUID does not conform to the allowed format**

A UUID was supplied while creating a new BLE object, and it did not conform to one of the allowed formats.

The correct format for a 32 bit UUID is a series of four hexadecimal characters, such as "12eF". The format for a 128 bit UUID is "hhhhhhhh-hhhh-hhhh-hhhh-hhhhhhhhhhhh", where each letter h is replaced by a hexadecimal digit.

---

**There were no formats in the format specifier**

A PRINT USING format specifier had no imbedded format specifiers, but needed one.

```
PRINT USING "PI = "; 3.1415926535897932
```

is not valid, since there is no specifier for the number. Use

```
PRINT USING "PI = #.#####"; 3.1415926535897932
```

instead.

---

**There were not enough points for the specified regression**

Regression was attempted without enough points for the specified order.

Linear regression requires at least two points, and polynomial regression requires one more point than the order specified. For example, fitting a second order polynomial like

$$y = A + Bx + Cx^2$$

requires at least three points.

This error only occurs if there are too few points. If enough points are coincident that there are not enough distinct points, the error *Failed due to numeric instability* will occur.

---

**This value cannot be printed**

An error occurred while evaluating an expression that resulted in an unprintable value. This error should always be accompanied by some other error that describes the reason this one occurred.

---

**Too many parameters**

A method was called with too many parameters.

---

**Type mismatch**

An error in an expression resulted in an operation with values that are not compatible with one another, or with an operation.

There are many examples, such as attempting to add strings (use the & operator), assigning an array to a non-array variable, or passing a value of one type to a BYREF parameter of another type. Typographical errors in expressions or misspelling array names frequently cause this error.

---

**Unfinished \ format specifier**

A \ character was found that started a string format specifier, but no ending \ was found in the format specifier.

If you need a backslash, and are not trying to create a string format specifier, escape the backslash with an underscore character:

```
PRINT USING "Print \    \ like this, and backslash characters like this:
_\"; "strings"
```

---

**Unknown error**

An unknown error occurred, presumably because the ERROR statement was used with an error number that does not appear in this appendix.

---

**Values passed BYREF must be assignable**

An attempt was made to pass an expression to a method that expected a variable it could change.

```
x = 4
Square(x)
PRINT x
END

SUB Square(BYREF y)
Y = Y*Y
END SUB
```

is allowed, and prints 16. The parameter, passed by reference, allows the subroutine to change the value in the main program. This, however, does not work, since the value passed is now an expression:

```
x = 4
Square(x + 1) : ! Cannot pass an expression by reference.
PRINT x
END

SUB Square(BYREF y)
Y = Y*Y
END SUB
```





## Appendix B – Character Sets

---

### The ASCII Character Set

The ASCII character set establishes numeric equivalents for 95 printing characters. The tie between the ASCII character set and computers is so pervasive that virtually all keyboards built for use in the United States allow input of all of the ASCII characters—and only the ASCII characters. (Some facility for entering international characters is generally available, but these aren’t actually on the keyboard.)

The ASCII character set also defines 33 nonprinting characters, numbered 0 to 31 and 127. All of these have a suggested meaning, and many are now nearly universal. This wasn’t always true. To get an idea of how long the ASCII character set has been around, consider that character 127, rub, is used as a delete character. The reason it’s character 127 is that this character is made up of seven 1 bits. When a mistake was made punching code into a paper tape—yes, a long yellow strip of paper used to store programs and data—deleting a character meant backing up and punching all seven holes out, or “rubbing out” the letter. And for the first 10 years or so of the microcomputer revolution, it was rare to find a keyboard with the entire ASCII character set available.

The complete ASCII character set is shown below. To find the number for a particular character, add the values to the top and left of the given character. For example, the ordinal value for the character A is 64 + 1, or 65.

	0	16	32	48	64	80	96	112
0	nul	dle		0	@	P	`	p
1	soh	dc1	!	1	A	Q	a	q
2	stx	dc2	"	2	B	R	b	r
3	etx	dc3	#	3	C	S	c	s
4	eot	dc4	\$	4	D	T	d	t
5	enq	nak	%	5	E	U	e	u
6	ack	syn	&	6	F	V	f	v
7	bel	etb	'	7	G	W	g	w
8	bs	can	(	8	H	X	h	x
9	ht	em	)	9	I	Y	i	y
10	lf	sub	*	:	J	Z	j	z
11	vt	esc	+	;	K	[	k	{
12	ff	fs	,	<	L	\	l	
13	cr	gs	-	=	M	]	m	}
14	co	rs	.	>	N	^	n	~
15	si	us	/	?	O	_	o	rub



# Index

---

## *Special Characters*

- · 36, 61, 65, 72  
 \_ · 46  
 , · 36  
 : · 36, 46  
 ! · 35, 36, **39**, 41, 54, 111  
 ? · 114  
 . · 61  
 ( · 36, 61  
 ) · 36, 61  
 @ · 36  
 \* · 36, 61, 66, 111, 133  
 / · 36, 61, 67  
 & · 35, 41, 54, 60, 65, 111  
 # · 35, 36, 41, 54, 108, 109, 113, 114, 121  
 % · 35, 41, 54  
 ^ · 67, 111  
 + · 36, 61, 64, 72, 110, 111  
 < · 36, 61, 71  
 << · 61, 68  
 <= · 61, 71  
 <> · 61, 71  
 = · 36, 61, 71, 141  
 > · 36, 61, 71  
 >= · 61, 71  
 >> · 61, 68  
 ~ · 35, 41, 54  
 \$ · 35, 37, 42, 54, 110  
 π · **80**

---

## A

about · 18, 32  
 ABS function · 42, **75**, 226  
 absolute value · 75  
 accel function · **218**  
 accelAvailable function · **220**  
 ACOS function · **76**  
 Activity class · 237, **287**  
 A-D Converter · 216  
 addAttachment subroutine · **163**  
 addBCC subroutine · **163**  
 addCC subroutine · **163**  
 addition · 64  
 addPresentationFormat · 195  
 addService subroutine · **204**  
 addTo subroutine · **163**  
 addUserDescription · 195  
 aliases · 8, 24  
 allowsMultipleSelection function · **320**  
 allowsSelection function · **320**

alpha function · **295**  
 altimeter · 227  
 altimeterAvailable function · **220**  
 AND · 61, **70**, 86, 87, 124, 133  
 Annotation class · **288**  
 annotations function · **303**  
 arc cosine · 76  
 arc sine · 76  
 arc tangent · 76  
 arrays · **49–50**, 51, 52, 54, 55–56, 60, 61, 66  
     as parameters · 139, 141  
     constants · 38  
     expressions · 60  
     subscripts · 73  
 AS · 55, 56, 130, 135, 138, 139, 143  
 ASC function · **85**, 124, 133  
 ascender function · **231**, **247**  
 ASCII character set · **35**, 38, 42, 71, 85, **347**  
 ASCII files · 121  
 ASIN function · **76**  
 aspectY function · **260**  
 assembly language · 41  
 assignment compatibility · 51  
 assignment statement · 74, 75  
 ATAN function · **76**  
 Audio class · **187**

---

## B

barometer · 227  
 BASE statement · 49, **55**, 56, 82, 84, 85  
 binary conversions · 61–62  
 binary files · 122, 126, 128, 130  
 binary operators · 60  
 bit shift left · **68**  
 bit shift right · **68**  
 BITAND · **69**  
 BITAND · 61  
 BITNOT · **69**  
 BITNOT · 61  
 BITOR · **69**  
 BITOR · 61  
 bitwise and · **69**  
 bitwise exclusive or · **69**  
 bitwise not · **69**  
 bitwise or · **69**  
 BITXOR · **69**  
 BITXOR · 61  
 BLE central · 189  
 BLE class · **189**  
 BLE slave · 189, 190  
**BLEATTRequest** class · **192**  
 BLEATTRequest class · **205**  
 BLEATTRequest object · 151

## Index

BLEAvailable function · **190**  
BLECharacteristic class · **193**  
BLECharacteristicInfo subroutine · **147**  
BLECharacteristicsDiscovery subroutine · 199, 202  
BLEDescriptor class · **194**  
BLEDescriptorInfo subroutine · **147**  
BLEDescriptorInfo subroutine · 202  
BLEDiscoveredPeripheral subroutine · **148**, 191, 192  
BLEMutableCharacteristic class · **195**  
BLEMutableCharacteristicInfo subroutine · **149**  
BLEMutableService class · **197**  
BLEMutableServiceInfo subroutine · **149**  
bleName function · **198**  
BLEPeripheral class · **198**  
BLEPeripheralInfo subroutine · **150**, 190, 191  
BLEPeripheralManager class · **204**  
BLEPeripheralManagerInfo event · **206**  
BLEPeripheralManagerInfo subroutine · **151**  
BLERetrievedPeripherals subroutine · **153**, 191, 192  
BLEService class · **207**  
BLEServiceInfo subroutine · **153**, 199, 200, 201  
BLEState subroutine · **153**  
blue function · **294**  
Bluetooth · 189  
Bluetooth 4.0  
    see Bluetooth LE · 187  
Bluetooth Low Energy  
    see Bluetooth LE · 187  
breakpoints · 14, 19, 28, 33  
built in functions · 73  
Button class · 237  
Button class · **289**  
BYTE · 35, **41**, 49, 54, 57, 61, 63, 64, 104, 123

---

## C

CALL statement · **143**, 144  
Callout class · **229**, 261  
callouts · 13, 27  
canGoBack function · **330**  
canGoForward function · **330**  
canSendMail function · **164**  
case sensitivity · 10, 24, 35  
CASE statement · **97**, 98, 108  
CDBL function · 62, **77**  
cellSelected subroutine · **154**, 319, 320  
center function · **303**  
characteristic function · **192**  
characteristics function · **197**, **207**  
CHDIR statement · **132**  
CHR function · 86, 87, 108, 124, 133  
CINT function · 62, **77**, 104, 124  
classes  
    Activity · 237  
    Button · 237  
    Callout · 261  
    ColorPicker · 237  
    DatePicker · 237  
    ImageView · 238

Label · 238  
MapView · 238  
Picker · 238  
Plot · 238  
PlotFunction · 261, 262, 269, 270  
PlotMesh · 265  
PlotPoint · 266  
PlotSurface · 271  
PlotVector · 272  
Progress · 238  
SegmentedControl · 238  
Slider · 239  
Stepper · 239  
Switch · 239  
Table · 239  
TextField · 239  
TextView · 239  
WebView · 240  
classes · 53  
    Activity · 287  
    Annotation · 288  
    Audio · 187  
    BLE · 189  
    **BLEATTRequest** · 192  
    BLECharacteristic · 193  
    BLEDescriptor · 194  
    BLEMutableCharacteristic · 195  
    BLEMutableService · 197  
    BLEPeripheral · 198  
    BLEPeripheralManager · 204  
    BLEService · 207  
    Button · 289  
    Callout · 229  
    ColorPicker · 293  
    Comm · 208  
    Control · 295  
    Date · 161  
    DatePicker · 298  
    Email · 162  
    Err · 164  
    Event · 165  
    Graphics · 230  
    Hijack · 216  
    Image · 246  
    ImageView · 300  
    Label · 301  
    MapView · 301  
    Math · 167  
    Picker · 306  
    Plot · 258  
    PlotFunction · 282  
    PlotMesh · 284  
    PlotPoint · 284  
    PlotSurface · 285  
    PlotVector · 286  
    Progress · 308  
    SegmentedControl · 310  
    Sensors · 218  
    Slider · 313  
    Stepper · 316  
    Switch · 317

- System · 181
- Table · 319
- TextField · 323
- TextView · 326
- WebView · 328
- CLEAR statement · **100**
- clearAttachments subroutine · **164**
- clearBCC subroutine · **164**
- clearCC subroutine · **164**
- clearConsole subroutine · **181, 186**
- clearTo subroutine · **164**
- CLNG function · **77**
- CLOSE statement · 95, 121, 122, 123, 127, 128, 129, 130, 268
- color · **260**
- ColorPicker class · 156, 159, 237, **293**
- Comm class · **208**
- comments · 36
- comparison operators · 60, 70
- CON function · **82, 84**
- connect subroutine · 150, **190**
- console · 113
- Console view · 12, 26
- constants · 35, 71
  - arrays · 38
  - floating-point · 37
  - hexadecimal · 37
  - integer · 36
  - long integer · 37
  - matrices · 38
  - string · 36, 38
  - vectors · 38
- consumed function · **165**
- continuation lines · 46
- Control class · **295**
- conversions
  - binary · 61–62
  - unary · 62–64
- COS function · 57, **78, 93, 107, 270**
- cosecant · 78
- COSH function · **78**
- cosine · 78
- COT function · **78**
- cotangent · 78
- crop subroutine · **247**
- CSC function · **78**
- CSNG function · 62, **79**
- CURDIR function · 132, **133**

---

## D

- data formats · 41
- DATA statement · **116, 117**
- Date class · **161**
- date function · **181, 299**
- DatePicker class · 156, 159, 237, **298**
- day function · **161**
- dayOfWeek function · **161**
- dayOfYear function · **162**

- debugger · 14, 28
- default prefix
  - see directories ·
- DEG function · **79**
- degrees · 79, 80
- deleteRows subroutine · **320**
- deleteSections subroutine · **320**
- descender function · **231, 247**
- description function · **165**
- descriptors function · **193**
- descriptors subroutine · 199
- deselect subroutine · **321**
- deselectAnnotation subroutine · **303**
- deselectLocation subroutine · **304**
- DET function · **83, 140**
- detectPulses subroutine · **187**
- device orientation · 154, 182
- didRotate subroutine · **154**
- DIM statement · 35, 49, 50, 53, 54, 55, **57, 100**
  - SHARED · 57
- DIR function · 125, **133, 134**
- directories · 133
  - changing · 132
  - creating · 135
  - default · 120, 132, 133
  - deleting · 135
  - files in · 133
- directory names · 119
- dirFTP function · **208**
- disconnect subroutine · 191
- disconnect subroutine · **191**
- discoverCharacteristics subroutine · **199**
- discoverDescriptors subroutine · 147, 153
- discoverDescriptors subroutine · **199**
- discoverIncludedServices subroutine · 153, **200**
- discoverServices subroutine · **201**
- division · 67
- DO statement · **91, 95, 128**
- DOT function · **83, 84**
- DOUBLE · 35, **37, 41, 49, 54, 57, 59, 61, 62, 63, 64, 65, 66, 77**
- drawArc subroutine · **231, 247**
- drawImage subroutine · **231**
- drawLine subroutine · **232, 247**
- drawOval subroutine · **232, 248**
- drawPoly subroutine · **232, 248**
- drawRect subroutine · **233, 249**
- drawRoundRect subroutine · **233, 249**
- drawText subroutine · **233, 249**

---

## E

- Email class · 182
- Email class · **162**
- END statement · 42, **100**
  - END FUNCTION statement · **143**
  - END IF statement · **95**
  - END SELECT statement · **97**
  - END SUB statement · **144**

## Index

EOF function · 95, 121, 122, 123, 126, 128, **131**  
equal · 71  
erf function · **167**  
erfc function · **167**  
erl function · **165**  
Err class · **164**  
ERROR statement · 42, **99**, 100  
errors · 16, 29  
Event class · **165**  
events  
    null event · **157**  
    number of touches · 166, 167  
    tap · 146, **157**, 165, 166  
    touches began · 145, **158**, 165, 166  
    touches canceled · 146, **158**, 165, 166  
    touches ended · 146, **158**, 165, 166  
    touches moved · 146, 165, 166  
    touches mved · **158**  
EXISTS function · **133**  
existsFTP function · **209**  
EXP function · **79**, 112, 139  
exponent · 37, 41, 49, 79, 89, 104, 111  
exponentiation · 67  
exporting console text · 16, 30  
exporting graphics · 16, 30  
exporting programs · 16, 30  
expression  
    array · 60  
    conversions in · 61  
    logical · 59  
    mathematical · 59  
    operator precedence · 60  
    string · 60

---

## F

false · 59  
file names · 119, 133  
    character set · 119  
file numbers · 121  
file types  
    bin · 119, 122, 126, 128  
    txt · 119  
files  
    deleting · 135  
    letter case · 133  
    names · 119  
    renaming · 135  
    size · 131  
fillArc subroutine · **234, 249**  
fillOval subroutine · **235, 250**  
fillPoly subroutine · **235, 251**  
fillRect subroutine · **235, 251**  
fillRoundRect subroutine · **236, 251**  
find · 10, 24  
    options · 10, 24  
finishedLoad subroutine · **156**, 328  
finishedLoad subroutine · 330  
floating-point · 41, 51, 56, 59, 64, 74, 77, 104, 109, 122

    constants · 37  
    see also DOUBLE ·  
    see also SINGLE ·  
folders  
    renaming · 7, 23  
font function · **236, 252**  
fontCount function · **236, 252**  
fontName function · **236, 252**  
fonts  
    changing size · 19, 33  
fontSize function · **236, 252**  
FOR statement · 13, 26, **92**  
FTP · 208, 209, 210, 211, 213, 215  
full path names · 119  
full screen mode · **184**  
FUNCTION statement · 45, 73, 116, **143**  
    parameters · 138  
    recursion · 142  
    variable scope · 142  
functions · 11  
    built in · 73  
    see also FUNCTION statement ·

---

## G

gamma function · **167**  
GET statement · 122, 123, 128, 129, 130, **131**  
getCameraImage subroutine · **252**  
getPhotoImage subroutine · **253**  
getText function · **301, 321, 324, 327**  
goBack subroutine · **331**  
goForward subroutine · **331**  
GOSUB statement · **137**, 138  
GOTO statement · 94, 95, **98**  
Graphics class · **230**  
graphics output · 13, 26  
graphics view · **184**  
greater than · 71  
greater than or equal · 71  
green function · **295**  
gyro function · **220**  
gyroAvailable function · **222**

---

## H

hasCamera function · **253**  
hasImage function · **253**  
hasPhotoLibrary function · **253**  
heading function · **222**  
headingAvailable function · **222**  
height function · **295**  
height function · **253**  
height function · **237**  
Help · 12, 26  
HEX function · **86**  
hexadecimal · 37  
hide program list · 19, 33  
Hijack class · **216**

hour function · **162**  
 HTTP · 208, 214, 215  
 hyperbolic cosine · 78  
 hyperbolic sine · 81  
 hyperbolic tangent · 82

---

## I

identifiers · **35**, 41, 42, 53, 54  
     case sensitivity · 35  
     length · 35  
 IDN function · 49, 66, **83**  
 IF statement · 59, **95**  
 Image class · 182  
 Image class · **246**  
 ImageView class · 238  
 ImageView class · **300**  
 includedServices function · **197, 208**  
 infinity · **41**, 64, 65, 66, 67, 70, 75, 82  
 Innovative Systems · 41  
 INPUT statement · 13, 26, 95, **113**, 119, 123, 128  
     see also LINE INPUT ·  
 insertRow subroutine · **308, 321**  
 insertRows subroutine · **308, 321**  
 insertSections subroutine · **321**  
 insertSegment subroutine · **311**  
 insertWheels subroutine · **308**  
 INSTR function · 88  
 INT function · **79**  
 INTEGER · 35, **36**, 41, 49, 54, 56, 57, 59, 61, 62, 63, 64,  
     66, 67, 77, 104  
 integers · 41  
     constants · 36, 37  
     storage · 41  
 INV function · **83**  
 iPad · 5  
 iPad 3<sup>rd</sup> generation · 189  
 iPhone · 21  
 iPhone 4s · 189  
 iPod · 21  
 isAntialiased function · **237, 253**  
 isBroadcasted function · **193, 195**  
 isConnected function · **201**  
 ISDIR function · **133**  
 ISDIR function · 126  
 isDirFTP function · **209, 210**  
 isEnabled function · **295**  
 isHidden function · **295**  
 isHighlighted function · **295**  
 isInf function · **168**  
 isLoading function · **331**  
 isNaN function · **168**  
 isNotifying function · **193, 195**  
 isOn function · **318**  
 isOpaque function · **295**  
 isPixelGraphics function · **237**  
 isPrimary function · **197, 208**  
 isSegmentEnabled function · **311**  
 isSelected function · **295**

isUpdating function · **304**

---

## K

keyboard · 9, 24, 103, 131, 347  
 KILL statement · 127, 128, **135**  
 killFTP subroutine · **210**  
 kind function · **295**

---

## L

Label class · 238  
 Label class · **301**  
 label function · **165**  
 labels · **45**  
 LBOUND function · **84**  
 LCASE function · **86**  
 LEFT function · 86  
 LEN function · 86, **87**  
 less than · 71  
 less than or equal · 71  
 LET statement · **74**  
 LINE INPUT function · 122  
 LINE INPUT statement · 12, 26, **115**, 119  
 line numbers · **45**  
     see also labels · 45  
 lineHeight function · **237, 253**  
 lines · 35  
 load subroutine · **255**  
 loadDocument subroutine · **331**  
 loadError subroutine · **156**, 329  
 loadError subroutine · 330  
 loadImage subroutine · **289, 300**  
 loadURL subroutine · **331**  
 LOC function · **131**  
 location function · **223**  
 location function · **304**  
 locationAvailable function · **223**  
 LOF function · 128, 129, 130, **131**  
 LOG function · **79**  
 LOG10 function · **79**  
 LOG2 function · **80**  
 logarithm · 80  
 logarithm · 79  
     base 10 · 79  
     base 2 · 80  
 logAxis subroutine · **260**  
 logGamma function · **168**  
 logical AND · 70  
 logical expression · 59  
 logical NOT · 72  
 logical OR · 70  
 logical value · 59  
 LONG · 35, **37**, 41, 49, 54, 57, 59, 61, 62, 63, 64, 66, 67,  
     77, 104  
 long integers  
     constants · 37  
     storage · 41

## Index

longDate function · 162  
longTime function · 162  
LOOP statement · 91  
LTRIM function · 87  
LUBackSub function · 168  
LUDComp function · 169  
l-value · 73–74

---

## M

mag function · 223  
magAvailable function · 225  
mantissa · 63, 104  
mapLocation subroutine · 156, 302, 303  
MapView class · 156, 159, 238  
MapView class · 301  
MAT statement · 75  
MATCH function · 88  
Math class · 167  
mathematical expression · 59  
matrices  
    addition · 64  
    assignment · 75  
    constants · 38, 82  
    determinant · 83  
    identity · 83  
    inverse · 83  
    multiplication · 66  
    subtraction · 65  
    transpose · 84  
    zero · 85  
maximumX function · 260  
maximumY function · 260  
maximumZ function · 260  
mean function · 170  
MID function · 87  
minimumX function · 261  
minimumY function · 261  
minimumZ function · 261  
minute function · 162  
mkDirFTP subroutine · 211  
MOD · 67  
month function · 162  
multiplication · 66

---

## N

NAME statement · 135  
NaN · 42, 64, 65, 66, 67, 70, 75, 82  
natural logarithm · 79  
network communications · 208  
newActivity function · 237  
newAnnotation function · 304  
newBLEPeripheralManager function · 191  
newButton function · 237  
newCallout function · 261  
**newCharacteristic** function · 197  
newColorPicker function · 237

**newConstantCharacteristic** function · 198  
newCylindrical function · 261  
newDatePicker function · 237  
newEmail function · 182  
newFunction function · 262  
newImage function · 182  
newImage subroutine · 253  
newImageView function · 238  
newLabel function · 238  
newMapView function · 238  
newMesh function · 265  
newPicker function · 238  
newPlot function · 266  
newPlot function · 238  
newPolar function · 269  
newProgress function · 238  
newSegmentedControl function · 238  
newService function · 205  
newSlider function · 239  
newSpherical function · 270  
newStepper function · 239  
newSurface function · 271  
newSwitch function · 239  
newTable function · 239  
newTextField function · 239  
newTextView function · 239  
newVectorPlot function · 272  
newWebView function · 240  
NEXT statement · 92  
normal function · 170  
NOT · 61, 72  
not equal · 71  
nullEvent subroutine · 157, 184, 309, 310  
number function · 165

---

## O

objects · 42, 50, 53  
offset function · 192  
ON ERROR GOTO statement · 99, 100, 101, 164  
ON-GOSUB statement · 138  
ON-GOTO statement · 98  
OPEN statement · 95, 121, 122, 123, 127, 128, 130  
openTCPIP subroutine · 211  
openUDP subroutine · 212  
operator precedence · 60  
optional parameters · 141  
OR · 61, 70  
orientation · 154, 182  
orientation function · 182  
osVersion function · 182  
overflow · 65, 66, 67

---

## P

packDbl function · 171  
packInt function · 171  
packLng function · 172



packSng function · **172**  
 parameters · 138, 143, 144  
     arrays as · 139, 141  
     built-in functions · 73  
     optional · 141  
     pass by reference · 51, 141, 142  
     pass by value · 140  
     type compatibility · 51  
     unary conversions · 62  
 parentheses · 61, 71, 73, 138, 139, 143  
 partial path names · 119, 120  
 pausing a program · 15, 28  
 permissions function · **195**  
 PI function · **172**  
 PI function · **80**  
 Picker class · 159, 238  
 Picker class · **306**  
 pinching · 13, 14, 27  
 pixel function · **255**  
 pixel graphics · 230, 243  
 pixels function · **256**  
 pixilation · 230, 243  
 Plot class · 238  
 Plot class · **258**  
 PlotFunction class · 261, 262, 269, 270  
 PlotFunction class · **282**  
 PlotMesh class · 265  
 PlotMesh class · **284**  
 PlotPoint class · 266  
 PlotPoint class · **284**  
 PlotSurface class · 271  
 PlotSurface class · **285**  
 PlotVector class · 272  
 PlotVector class · **286**  
 poly function · **172**  
 polyFit function · **173**  
 polygons · 232, 235, 248, 251  
 POP statement · **138**  
 POS function · **88**  
 power  
     see exponent ·  
 power use · 19, 33  
 preferences · 18, 33  
 PRINT statement · 12, 13, 26, **103**, 119  
     printing to files · 108  
     printing to strings · 108  
 PRINT USING statement · **109**, 119  
     printing to files · 113  
     printing to strings · 113  
 programs  
     aliases · 8, 24  
     copying · 8, 24  
     moving · 8, 23  
 programs  
     creating new programs · 6, 22  
     deleting · 7, 23  
     editing · 9, 24  
     renaming · 7, 23  
     running · 6, 22  
     searching · 10, 24  
     viewing · 6, 22

Progress class · 238  
 Progress class · **308**  
 properties function · **193**, **196**  
 pulseHistogram subroutine · **188**  
 PUT statement · 122, 127, 128, 130, **132**

---

## R

RAD function · **80**  
 radians · 79, 80  
 rand function · **175**  
 random access files · 129, 130, 131, 132  
 random numbers · 80–81  
 READ statement · **116**  
 readCharacteristic subroutine · 147  
 readCharacteristic subroutine · **202**  
 readDescriptor subroutine · 147  
 readDescriptor subroutine · **202**  
 readFTP subroutine · **213**  
 readHTTP subroutine · **214**  
 readyToUpdateSubscribers subroutine · **157**  
 receive function · **216**  
 recursion · 137, 142  
 red function · **295**  
 redo · 11, 25  
 reload subroutine · **331**  
 REM statement · **39**  
 removeAllSegments subroutine · **311**  
 removeAnnotation subroutine · **305**  
 removeCallout subroutine · **273**  
 removeSegment subroutine · **311**  
 removeService subroutine · **206**  
 repaint subroutine · **240**, **273**  
 replace · 10, 24  
 replaceText subroutine · **327**  
 repondToRequest subroutine · 151  
 reserved symbols · **36**  
 reserved words · **36**  
 respondToRequest subroutine · **205**  
 RESTORE statement · **117**  
 RESUME statement · **100**  
 retrieveConnectedPeripherals subroutine · **191**  
 retrievePeripherals subroutine · **192**  
 RETURN statement · **138**  
 RGBA color · **260**  
 RIGHT function · 86, **88**  
 RMDIR statement · **135**  
 RND function · **80**, 232, 248  
 romb function · **175**  
 rotate subroutine · **274**  
 rotation · 14, 27  
 rows function · **321**  
 rSquare function · **176**  
 rssi function · **202**  
 RTRIM function · **89**  
 running a program · 15, 28

---

**S**

- sample function · **225**
- sample programs · 19, 33
- sampleRate function · **188**
- samples
  - Array Parameters · 140
  - Binary files · 122
  - Catalog · 125
  - File Dump Example · 123
  - File random access · 129
  - List Files Example · 125
  - Move Up One Directory · 132
  - Print Directory · 133
  - Print Text File · 94
  - Reading an entire file · 129
  - Test files · 121
  - toUpper · 86
  - Writing and reading variable length files · 127
- sandbox · 121
- save subroutine · **256**
- scale function · **241**
- screen resolution · 230
- scroll subroutine · **321, 327**
- SEC function · **81**
- secant · 81
- second function · **162**
- sections function · **322**
- SEEK statement · 128, 129, **132**
- SegmentedControl class · 159, 238
- SegmentedControl class · **310**
- SELECT statement · **97**
- selectAnnotation subroutine · **305**
- selectedAnnotations function · **305**
- selection function · **308, 312, 322**
- selectionLength function · **327**
- selectionStart function · **327**
- selectLocation subroutine · **305**
- selectRow subroutine · **308**
- send subroutine · **164**
- send subroutine · **218**
- Sensors class · **218**
- services function · **202**
- setAccelRate subroutine · **227**
- setAlignment subroutine · **301, 325, 327**
- setAllowedGestures subroutine · **274**
- setAllowedOrientations subroutine · **182**
- setAllowsMultipleSelection subroutine · **322**
- setAllowsSelection subroutine · **322**
- setAlpha subroutine · **296**
- setAntialiased subroutine · **241, 256**
- setApportionByContent subroutine · **312**
- setAspectY subroutine · **275**
- setAutoCapitalization subroutine · **325, 327**
- setAutoCorrection subroutine · **325, 327**
- setAutorepeat subroutine · **316**
- setAxisFont subroutine · **275**
- setAxisKind subroutine · **275**
- setAxisStyle subroutine · **275**
- setBackgroundColor subroutine · **229, 277, 290, 296**
- setBorderColor subroutine · **277**
- setCenter subroutine · **305**
- setColor subroutine · **282, 284, 286, 288, 291, 295, 301, 318, 325, 327**
- setColor subroutine · **242, 256**
- setColorMap subroutine · **277**
- setColors subroutine · **284, 286**
- setConsumed subroutine · **165**
- setContinuous subroutine · **314**
- setDate subroutine · **299**
- setDomain subroutine · **283**
- setEditable subroutine · **327**
- setEnabled subroutine · **296**
- setFillColor subroutine · **283**
- setFocus subroutine · **296**
- setFont subroutine · **230, 291, 301, 322, 325, 328**
- setFont subroutine · **242, 257**
- setFontColor subroutine · **230**
- setFrame subroutine · **296**
- setGradient subroutine · **291**
- setGradientColor subroutine · **292**
- setGridColor subroutine · **278**
- setGyroRate subroutine · **227**
- setHidden subroutine · **296**
- setHighlighted subroutine · **296**
- setImage subroutine · **292, 300**
- setKeyboardAppearance subroutine · **183, 325, 328**
- setKeyboardType subroutine · **183, 325, 328**
- setLabelColor subroutine · **279**
- setLabelFont subroutine · **279**
- setLocation subroutine · **288**
- setLocationTitle subroutine · **306**
- setMagRate subroutine · **227**
- setMapRect subroutine · **306**
- setMapType subroutine · **306**
- setMaxColor subroutine · **314**
- setMaximumDate subroutine · **299**
- setMaxValue subroutine · **314, 317**
- setMesh subroutine · **284**
- setMeshColor subroutine · **279**
- setMeshSize subroutine · **279**
- setMeshStyle subroutine · **279**
- setMessage subroutine · **164**
- setMinColor subroutine · **315**
- setMinimumDate subroutine · **299**
- setMinuteInterval subroutine · **299**
- setMinValue subroutine · **315, 317**
- setMode subroutine · **299**
- setNotify subroutine · **202**
- setNullEventTime subroutine · **184**
- setOn subroutine · **318**
- setOpaque subroutine · **297**
- setPermissions · 196
- setPinColor subroutine · **288**
- setPixelGraphics subroutine · **243**
- setPointColor subroutine · **284**
- setPoints subroutine · **285, 286**
- setPointStyle subroutine · **285**
- setProgressColor subroutine · **309**
- setProperties · 196
- setRate subroutine · **218**

- setReadOnly subroutine · **297**
- setRect subroutine · **279**
- setRotation subroutine · **280**
- setScale subroutine · **280**
- setScale subroutine · **243**
- setScale3D subroutine · **280**
- setScalePage subroutine · **331**
- setSectionTitle subroutine · **322**
- setSecureTextEntry subroutine · **325, 328**
- setSeed subroutine · **177**
- setSegmentEnabled subroutine · **312**
- setSelected subroutine · **297, 312**
- setSelection subroutine · **323, 328**
- setShowLocation subroutine · **306**
- setShowsSelection subroutine · **308**
- setSpellChecking subroutine · **326, 328**
- setStepValue subroutine · **317**
- setStyle subroutine · **284, 285, 288, 292, 309, 312**
- setSubject subroutine · **164**
- setSubtitle subroutine · **289**
- setSurface subroutine · **285**
- setSurfaceColor subroutine · **280**
- setSurfaceStyle subroutine · **280**
- setTag subroutine · **297**
- setTags subroutine · **285, 286**
- setText subroutine · **301, 323, 326, 328**
- setThumbColor subroutine · **315**
- setTintColor subroutine · **312**
- setTitle subroutine · **281, 289, 293, 313**
- setTitleFont subroutine · **281**
- setToolsHidden subroutine · **243**
- setToolsLocation subroutine · **244**
- setTrackColor subroutine · **310**
- setTranslation subroutine · **281**
- setTranslation subroutine · **244**
- setTranslation3D subroutine · **281**
- setUpdate subroutine · **244**
- setUserTrackingMode subroutine · **306**
- setValue · 193, 197
- setValue subroutine · **310, 315, 317**
- setView subroutine · **281**
- setView3D subroutine · **281**
- setWraps subroutine · **317**
- setXAxisLabel subroutine · **281**
- setYAxisLabel subroutine · **281**
- setZAxisLabel subroutine · **282**
- SGN function · **81**
- SHARED · 57
- sharing console text · 16, 30
- sharing graphics · 16, 19, 30, 33
- sharing programs · 16, 30
- shortDate function · **162**
- shortTime function · **162**
- showAlert subroutine · **244**
- showConsole subroutine · **184**
- showGraphics subroutine · **184**
- showGrid subroutine · **282**
- showSource subroutine · **185**
- SIN function · 57, **81**, 93
- sine · 81
- SINGLE · 35, **37**, 41, 49, 54, 57, 59, 61, 62, 63, 64, 65, 66, 79
- SINH function · **81**
- SIZE function · **84**
- sleep · 19, 33
- Slider class · 159, 239
- Slider class · **313**
- SPC function · **107**
- SQR function · **82**
- square root · 82
- Stack view · 16, 29
- startAdvertising subroutine · **206**
- startAltimeter subroutine · **227**
- startAnimation subroutine · **288**
- startBLE subroutine · **192**
- startedLoad subroutine · **157, 328, 330**
- startScan subroutine · 148, **192**
- startSoundInput subroutine · **188**
- state function · **206**
- statusHTTP function · **215**
- stDev function · **177**
- step in · 15, 28
- step out · 15, 28
- step over · 14, 28
- Stepper class · 159, 239
- Stepper class · **316**
- STOP statement · **100**
- stop subroutine · **218**
- stopAccel subroutine · **228**
- stopAltimeter subroutine · **228**
- stopAnimation subroutine · **288**
- stopBLE subroutine · **192**
- stopGyro subroutine · **228**
- stopHeading subroutine · **228**
- stopLoading subroutine · **331**
- stopLocation subroutine · **228**
- stopMag subroutine · **228**
- stopping a program · 15, 29
- stopScan subroutine · **192**
- stopSoundInput subroutine · **189**
- STR function · **89**, 107
- STRING · 35, **38**, 42, 49, 54, 57
- strings · 53, 71
  - concatenation · 65
  - constants · 36, **38**
  - converting numbers to · 89
  - converting to numbers · 89
  - expressions · 60
- stringWidth function · **245, 257**
- SUB statement · 45, 116, **144**
  - parameters · 138, 142
  - variable scope · 142
- subroutines · 11
- subs · 11
- subtraction · 65
- succeeded function · **164, 257**
- swiping · 13, 14, 27
- Switch class · 159, 239
- Switch class · **317**
- System class · **181**

---

**T**

TAB function · **107**  
 Table class · 239  
 Table class · 154, **319**  
 tabs · 106  
 TAN function · **82**  
 tangent · 82  
 TANH function · **82**  
 tap subroutine · 146, **157**  
 TCP/IP · 208, 211  
 techBASIC Sampler · 1  
 term · 60  
 text files · 121, 130  
 text input · 12, 26  
 text output · 12, 26  
 textChanged subroutine · **156, 158, 323, 324, 326**  
 TextField class · 159, 239  
 TextField class · 158, **323**  
 TextView class · 239  
 TextView class · 158, **326**  
 ticks function · **185**  
 title function · **313**  
 title function · **293, 297**  
 tokens · 35–39  
 touchesBegan subroutine · 145, **158**  
 touchesCanceled subroutine · 146, **158**  
 touchesEnded subroutine · 146, **158**  
 touchesMoved subroutine · 146, **158**  
 touchUpInside subroutine · **159, 246, 287, 289, 330**  
 translation · 13, 14, 27  
 translationX function · **282**  
 translationX function · **245**  
 translationY function · **282**  
 translationY function · **245**  
 trap function · **177**  
 TRN function · **84**  
 true · 59  
 type characters · 54  
 types · 41

---

**U**

UBOUND function · 84, **85, 234, 250**  
 UCASE function · **89**  
 UDP · 216  
 UDP/IP · 208, 212  
 unary addition · 72  
 unary conversions · 62–64  
 unary negation · 72  
 unary subtraction · 72  
 undo · 11, 25  
 Unicode · **35**  
 Unicode character set · **35, 38, 42**  
 unpackDbl function · **178**  
 unpackInt function · **179**  
 unpackLng function · **179**  
 unpackSng function · **180**  
 updateValue function · **206**

UTF-8 · **35**  
 uuid function · **194, 197, 198, 203, 208**

---

**V**

VAL function · **89**  
 value · 194, 197  
 value function · **315, 317**  
 value function · **193**  
 valueChanged subroutine · **159, 294, 298, 302, 303, 307, 310, 311, 313, 314, 316, 317, 318, 324, 330**  
 var function · **180**  
 variable scope · 142  
 Variable view · 15, 29  
 vector graphics · 230, 243  
 vectors  
   addition · 64  
   assignment · 75  
   constants · 38, 82  
   dot product · 83  
   multiplication · 66  
   subtraction · 65  
   zero · 85  
 version function · **186**  
 vibrate subroutine · **186**  
 visibleRows function · **323**

---

**W**

WebView class · 240  
 WebView class · 156, 157, **328**  
 WEND statement · **94**  
 what function · **166**  
 when function · **166**  
 where function · **166**  
 WHILE statement · **94**  
 white space · 39  
 width function · **258, 297**  
 width function · **245**  
 workspace · 100  
 writeCharacteristic subroutine · 147  
 writeCharacteristic subroutine · **203**  
 writeDescriptor subroutine · 147  
 writeDescriptor subroutine · **204**  
 writeFTP subroutine · **215**  
 writeUDP subroutine · **216**

---

**X**

x function · **297**

---

**Y**

y function · **297**  
 year function · **162**

---

*Z*

ZER function · **85**  
zero function · **181**  
zooming · 13, 14, 27