So You Want to Learn to Program?

James M. Reneau, M.S.
Assistant Professor
Shawnee State University
Portsmouth Ohio USA

http://www.basicbook.org

James M. Reneau
P.O. Box 278
Russell, Kentucky 41169-2078 USA

Book Version: 20100909a
For BASIC-256 Version 0.9.6.35 or later

So You Want to Learn to Program?

James M. Reneau, M.S. - jim@renejm.com

Table of Contents

Index of Programs

Index of Illustrations

Acknowledgments:

Shout-outs go to all the people who have worked on the BASIC-256 project, at Sourceforge. Most especially, Ian Larsen (aka: DrBlast) for creating the BASIC-256 computer language and his original vision.

Dedications:

To my wife Nancy and my daughter Anna.

Chapter 1: Meeting BASIC-256 – Say Hello.

This chapter will introduce the BASIC-256 environment using the **print** and **say** statements. You will see the difference between commands you send to the computer, strings of text, and numbers that will be used by the program. We will also explore simple mathematics to show off just how talented your computer is. Lastly you will learn what a syntax-error is and how to fix them.

The BASIC-256 Window:

The BASIC-256 window is divided into five sections: the Menu Bar, Tool Bar, Program Area, Text Output Area, and Graphics Output Area (see Illustration 1: The BASIC-256 Screen below).

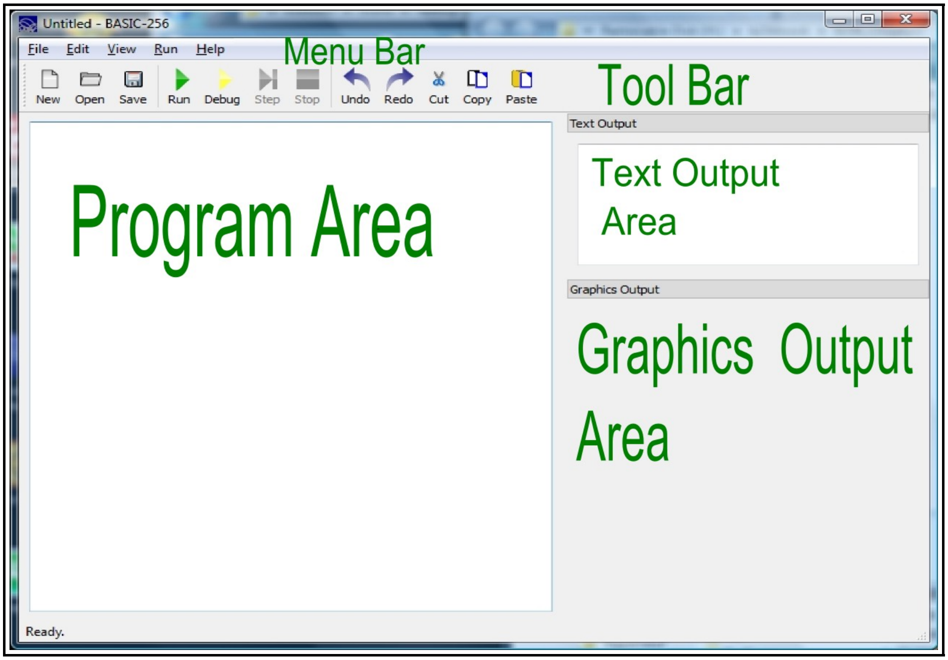


Illustration 1: The BASIC-256 Screen

Menu Bar:

The menu bar contains several different drop down menus. These menus include: “File”, “Edit”, “View”, “Run”, and “About”. The “File” menu allows you to save, reload saved programs, print and exit. The “Edit” menu allows you to cut, copy and paste text and images from the program, text output, and graphics output areas. The “View” menu will allow you to show or hide various parts of the BASIC-256 window. The “Run” menu will allow you to execute and debug your programs. The “About” menu option will display a pop-up dialog with information about BASIC-256 and the version you are using.

Tool Bar:

The menu options that you will use the most are also available on the tool bar.

- New – Start a new program
- Open – Open a saved program
- Save – Save the current program to the computer's hard disk drive or your USB pen drive
- Run – Execute the currently displayed program
- Debug – Start executing program one line at a time
- Step – When debugging – go to next line
- Stop – Quit executing the current program
- Undo – Undo last change to the program.
- Redo – Redo last change that was undone.
- Cut – Move highlighted program text to the clipboard
- Copy – Place a copy of the highlighted program text on the clipboard
- Paste – Insert text from the clipboard into program at current insertion point

Program Area:

Programs are made up of instructions to tell the computer exactly what to do and how to do it. You will type your programs, modify and fix your code, and load saved programs into this area of the screen.

Text Output Area:

This area will display the output of your programs. This may include words and numbers. If the program needs to ask you a question, the question (and what you type) will be displayed here.

Graphics Output Area:

BASIC-256 is a graphical language (as you will see). Pictures, shapes, and graphics you will create will be displayed here.

Your first program – The *say* statement:

Let's actually write a computer program. Let us see if BASIC-256 will say hello to us. In the Program Area type the following one-line program:

```
say “hello”
```

Program 1: Say Hello

Once you have this program typed in, use the mouse, and click on “Run” in the tool bar.

Did BASIC-256 say hello to you through the computer's speakers?

New Concept	`say expression` The **say** statement is used to make BASIC-256 read an expression aloud, to the computer's speakers.

New Concept	“” BASIC-256 treats letters, numbers, and punctuation that are inside a set of double-quotes as a block. This block is called a *string*.

New Concept	“Run” on the tool bar - or - “Run” then “Run” on the menu You must tell BASIC-256 when you want it to start executing a program. It doesn't automatically know when you are done typing your programming code in. You do this by clicking on the “Run” icon on the tool bar or by clicking on “Run” from the menu bar then selecting “Run” from the drop down menu.

To clear out the program you are working on and completely start a new program we use the “New” button on the tool bar. The new button will display the following dialog box:

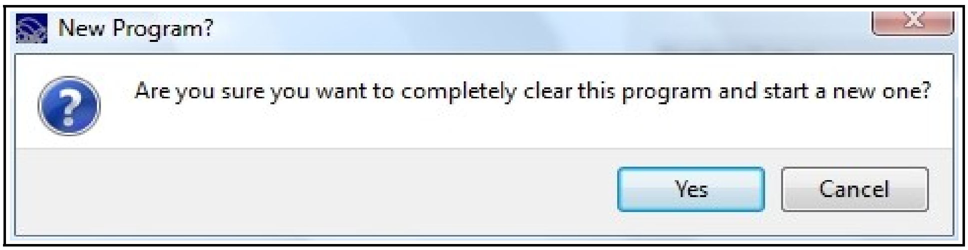


Illustration 2: BASIC-256 - New Dialog

If you are fine with clearing your program from the screen then click on the 

"Yes" button. If you accidentally hit "New" and do not want to start a new program then click on the 

"Cancel" button.

New Concept	"New" on the tool bar - or - "File" then "New" on the menu The "New" command tells BASIC-256 that you want to clear the current statements from the program area and start a totally new program. If you have not saved your program to the computer (Chapter 2) then you will lose all changes you have made to the program.

Explore	Try several different programs using the **say** statement with a string. Say hello to your best friend, have the computer say your favorite color, have fun.

You can also have the **say** statement speak out numbers. Try the following program:

```
say 123456789
```

Program 2: Say a Number

Once you have this program typed in, use the mouse, and click on "Run" in the tool bar.

Did BASIC-256 say what you were expecting?

New Concept	*numbers* BASIC-256 allows you to enter numbers in decimal format. Do not use commas when you are entering large numbers. If you need a number less than zero just place the negative sign before the number. Examples include: 1.56, 23456, -6.45 and .5

BASIC-256 is really good with numbers – Simple Arithmetic:

The brain of the computer (called the Central Processing Unit or CPU for short) works exclusively with numbers. Everything it does from graphics, sound, and all the rest is done by manipulating numbers.

The four basic operations of addition, subtraction, multiplication, and division are carried out using the operators show in Table 1.

Operator	**Operation**
+	Addition expression1 + expression2
–	Subtraction expression1 - expression2
*****	Multiplication expression1 * expression2
/	Division expression1 / expression2

Table 1: Basic Mathematical Operators

Try this program and listen to the talking super calculator.

```
say 12 * (2 + 10)
```

Program 3: Say the Answer

The computer should have said “144” to you.

```
say 5 / 2
```

Program 4: Say another Answer

Did the computer say “2.5”?

New Concept	+ – * / ( ) The four basic mathematical operations: addition (+), subtraction (-), division (/), and multiplication(*) work with numbers to perform calculations. A numeric value is required on both sides of these operators. You may also use parenthesis to group operations together. Examples include: 1 + 1, 5 * 7, 3.14 * 6 + 2, (1 + 2) * 3 and 5 - 5

Explore	Try several different programs using the **say** statement and the four basic mathematical operators. Be sure to try all four of them.

Another use for + (Concatenation):

The + operator also will add strings together. This operation is called concatenation, or "cat" for short. When we concatenate we are joining the strings together, like train cars, to make a longer string.

Let's try it out:

```
say "Hello " + "Bob."
```

Program 5: Say Hello to Bob

The computer should have said hello to Bob.

Try another.

```
say 1 + " more time"
```

Program 6: Say it One More Time

The + in the last example was used as the concatenate operator because the second term was a string and the computer does not know how to perform mathematics with a string (so it 'cats').

New Concept	`+ (concatenate)` Another use for the the plus sign (+) is to tell the computer to concatenate (join) strings together. If one or both operands are a string, concatenation will be performed; if both operands are numeric, then addition is performed.

Explore	Try several different programs using the **say** statement and the + (concatenate) operator. Join strings and numbers together with other strings and numbers.

The text output area - The print statement:

Programs that use the Text to Speech (TTS) **say** statement can be very useful and fun but is is also often necessary to write information (strings and numbers) to the screen so that the output can be read. The **print** statement does just that. In the Program Area type the following two-line program:

```
print “hello”
print “there”
```

Program 7: Print Hello There

Once you have this program typed in, use the mouse, and click on “Run” in the tool bar. The text output area should now show “hello” on the first line and “there” on the second line.

New Concept	**print** *expression* **print** *expression;* The **print** statement is used to display text and numbers on the text output area of the BASIC-256 window. **Print** normally goes down to the next line but you may print several things on the same line by using a ; (semicolon) at the end of the *expression*.

The **print** statement, by default, advances the text area so that the next **print** is on the next line. If you place a ; (semicolon) on the end of the *expression* being printed, it will suppress the line advance so that the next **print** will be on the same line.

```
cls
print “Hello ”;
print “there, ”;
print “my friend.”
```

Program 8: Many Prints One Line

New Concept	**cls** The *cls* statement clears all of the old displayed information from the text output area.

Explore	Try several different programs using the **print** statement. Use strings, numbers, mathematics, and concatenation.

What is a “Syntax error”:

Programmers are human and occasionally make mistakes. “Syntax errors” are one of the types of errors that we may encounter. A “Syntax error” is generated by BASIC-256 when it does not understand the program you have typed in. Usually syntax errors are caused by misspellings, missing commas, incorrect spaces, unclosed quotations, or unbalanced parenthesis. BASIC-256 will tell you what line your error is on and will even attempt to tell you where on the line the error is.

Chapter 2: Drawing Basic Shapes.

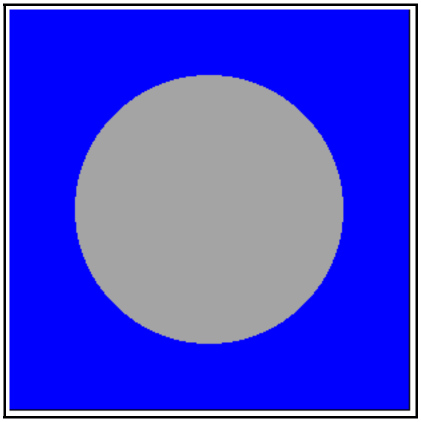
In this chapter we will be getting graphical. You will learn how to draw rectangles, circles, lines and points of various colors. These programs will get more and more complex, so you will also learn how to save your programs to long term storage and how to load them back in so you can run them again or change them.

Drawing Rectangles and Circles:

Let's start the graphics off by writing a graphical program for our favorite sports team, the “Grey Spots”. Their colors are blue and grey.

```
# c2_greyspots.kbs
# a program for our team - the grey spots
clg
color blue
rect 0,0,300,300
color grey
circle 149,149,100
say "Grey Spots, Grey Spots, Grey spots rule!"
```

Program 9: Grey Spots



Sample Output 9: Grey Spots

Warning

Notice: Program listings from here on will have each line numbered. DO NOT type in the line numbers when you are entering the program.

Let's go line by line through the program above. The first line is called a remark or comment statement. A remark is a place for the programmer to place comments in their computer code that are ignored by the system. Remarks are a good place to describe what complex blocks of code is doing, the program's name, why we wrote a program, or who the programmer was.

New Concept	`#`
	`rem`
	
	The **#** and **rem** statements are called remarks. A remark statement allows the programmer to put comments about the code they are working on into the program. The computer sees the **#** or **rem** statement and will ignore all of the rest of the text on the line.

On line two you see the **clg** statement. It is much like the **cls** statement from Chapter 1, except that the **clg** statement will clear the graphic output area of the screen.

New Concept	`clg` The **clg** statement erases the graphics output area so that we have a clean place to do our drawings.

Line three contains the **color** statement. It tells BASIC-256 what color to use for the next drawing action. You may define colors either by using one of the eighteen standard color names or you may define one of over 16 million different colors by mixing the primary colors of light (red, green, and blue) together.

When you are using the numeric method to define your custom color be sure to limit the values from 0 to 255. Zero (0) represents no light of that component color and 255 means to shine the maximum. Bright white is represented by 255, 255, 255 (all colors of light) where black is represented by 0, 0, 0 (no colors at all). This numeric representation is known as the RGB triplet. Illustration 3 shows the named colors and their numeric values.

New Concept	
	`color` *`color_name`* `color` *`red, green, blue`* `color` *`RGB_number`* **color** can also be spelled **colour**. The **color** statement allows you to set the color that will be drawn next. You may follow the **color** statement with a color name (black, white, red, darkred, green, darkgreen, blue, darkblue, cyan, darkcyan, purple, darkpurple, yellow, darkyellow, orange, darkorange, grey/gray, darkgrey/darkgray), with three numbers (0-255) representing how much red, blue, and green should be used to make the color, or with a single value representing red * 256 ^2 + green * 256 + green.

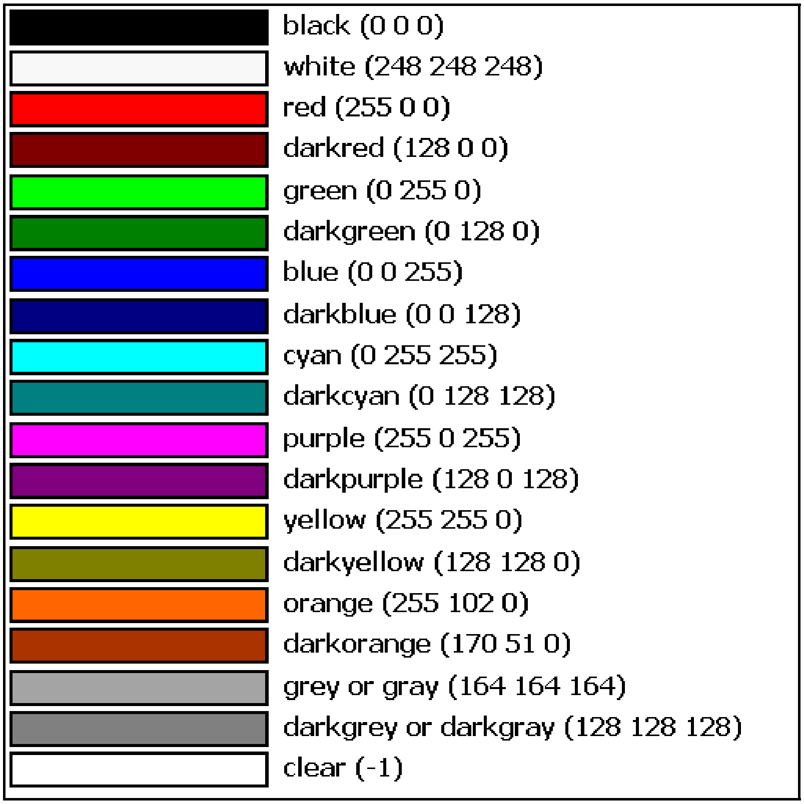


Illustration 3: Color Names

The graphics display area, by default is 300 pixels wide (x) by 300 pixels high (y). A pixel is the smallest dot that can be displayed on your computer monitor. The top left corner is the origin (0,0) and the bottom right is (299,299). Each pixel can be represented by two numbers, the first (x) is how far over it is and the second (y) represents how far down. This way of marking points is known as the Cartesian Coordinate System to mathematicians.

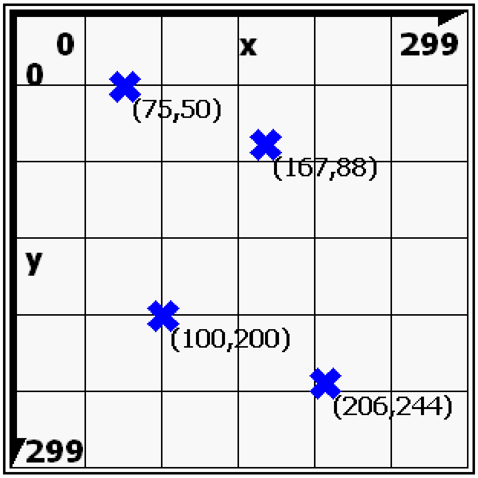


Illustration 4: The Cartesian Coordinate System of the Graphics Output Area

The next statement (line 4) is **rect**. It is used to draw rectangles on the screen. It takes four numbers separated by commas; (1) how far over the left side of the rectangle is from the left edge of the graphics area, (2) how far down the top edge is, (3) how wide and (4) how tall. All four numbers are expressed in pixels (the size of the smallest dot that can be displayed).

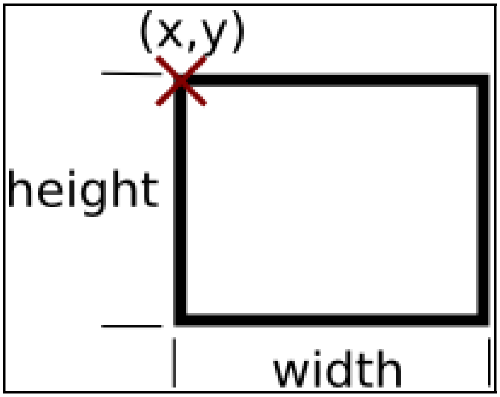


Illustration 5: Rectangle

You can see the the rectangle in the program starts in the top left corner and fills the graphics output area.

New Concept	**`rect`** *`x, y, width, height`* The **rect** statement uses the current drawing color and places a rectangle on the graphics output window. The top left corner of the rectangle is specified by the first two numbers and the width and height is specified by the other two arguments.

Line 6 of Program 9 introduces the **circle** statement to draw a circle. It takes three numeric arguments, the first two represent the Cartesian coordinates for the center of the circle and the third the radius in pixels.

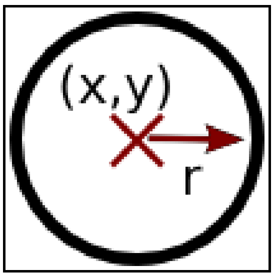


Illustration 6: Circle

New Concept

circle *x, y, radius*

The **circle** statement uses the current drawing color and draws a filled circle with its center at (x, y) with the specified radius.

Explore

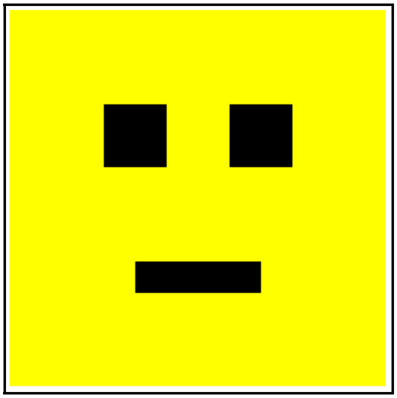
Can you create a graphic screen using colors, rectangles and circles for your school or favorite sports team?

Here are a couple of sample programs that use the new statements **clg**, **color**, **rect** and **circle**. Type the programs in and modify them. Make them a frowning face, alien face, or look like somebody you know.

```
# c2_rectanglesmile.kbs

# clear the screen
clg

# draw the face
color yellow
rect 0,0,299,299

# draw the mouth
color black
rect 100,200,100,25

# put on the eyes
color black
rect 75,75,50,50
```

```
rect 175,75,50,50

say "Hello."
```

Program 10: Face with Rectangles



Sample Output 10: Face with Rectangles

```
# c2_circlesmile.kbs

# clear the screen
clg
color white
rect 0,0,300,300

# draw the face
color yellow
circle 150,150,150

# draw the mouth
color black
circle 150,200,70
color yellow
circle 150,150,70
```

```

# put on the eyes
color black
circle 100,100,30
circle 200,100,30
```

Program 11: Smiling Face with Circles

Sample Output 11: Smiling Face with Circles

Explore	Combine rectangles and circles to create your own face graphic.

Saving Your Program and Loading it Back:

Now that the programs are getting more complex, you may want to save them so that you can load them back in the future.

You may store a program by using the Save button on the tool bar or Save option on the File menu. A dialog will display asking you for a file name, if it is a new program, or will save the changes you have made (replacing the old file).

If you do not want to replace the old version of the program and you want to store it using a new name you may use the Save As option on the File menu to save a copy with a different name.

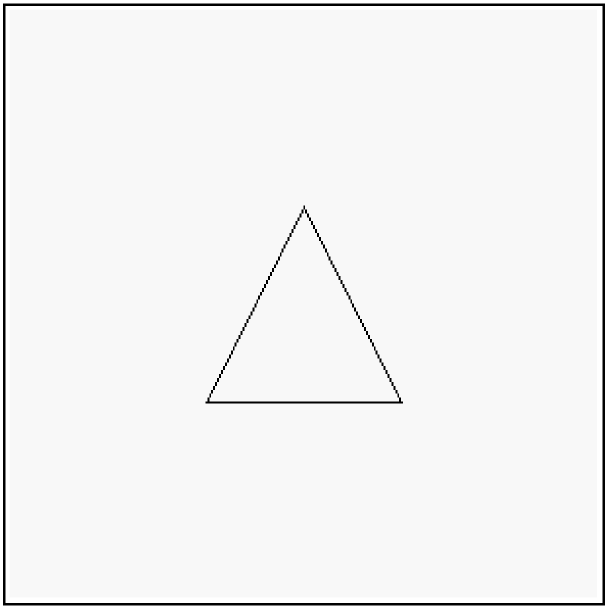
To load a previously saved program you would use the Open button on the tool bar or the Open option on the File menu.

Drawing with Lines:

The next drawing statement is **line**. It will draw a line one pixel wide, of the current color, from one point to another point. Program 12 shows an example of how to use the **line** statement.

```
# c2_triangle.kbs - draw a triangle

clg
color black

line 150, 100, 100, 200
line 100, 200, 200, 200
line 200, 200, 150, 100
```

Program 12: Draw a Triangle



Sample Output 12: Draw a Triangle

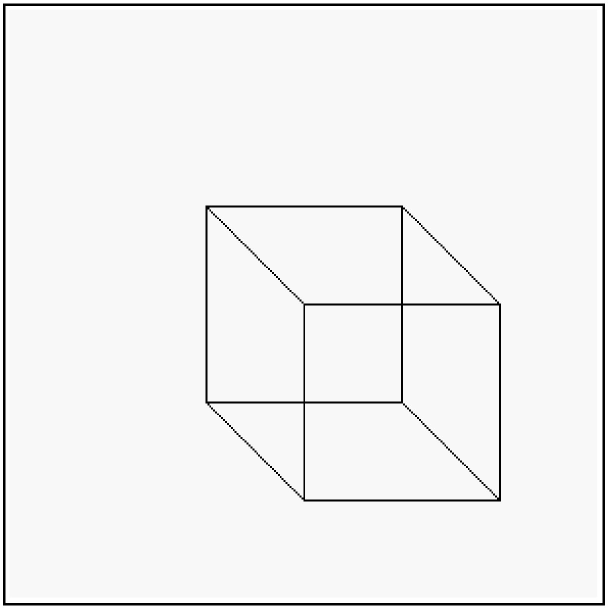
New Concept	`line start_x, start_y, finish_x, finish_y` Draw a line one pixel wide from the starting point to the ending point, using the current color.

Explore	Use a piece of graph-paper to draw other shapes and then write a program to draw them. Try a right triangle, pentagon, star, or other shapes.

The next program is a sample of what you can do with complex lines. It draws a cube on the screen.

```
# c2_cube.kbs - draw a cube

clg
color black

# draw back square
line 150, 150, 150, 250
line 150, 250, 250, 250
line 250, 250, 250, 150
line 250, 150, 150, 150

# draw front square
line 100, 100, 100, 200
line 100, 200, 200, 200
line 200, 200, 200, 100
line 200, 100, 100, 100

# connect the corners
line 100, 100, 150, 150
line 100, 200, 150, 250
line 200, 200, 250, 250
line 200, 100, 250, 150
```

Program 13: Draw a Cube



Sample Output 13: Draw a Cube

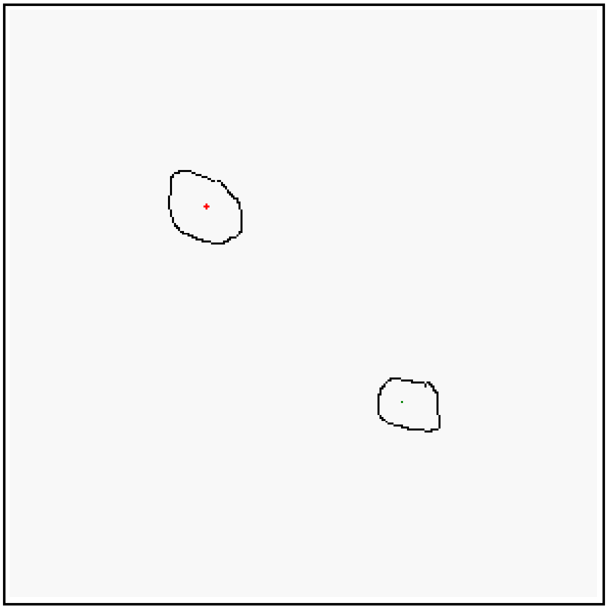
Setting Individual Points on the Screen:

The last graphics statement covered in this chapter is **plot**. The **plot** statement sets a single pixel (dot) on the screen. For most of us these are so small, they are hard to see. Later we will write programs that will draw groups of pixels to make very detailed images.

```
# c2_plot.kbs - use plot to draw points

clg

color red
plot 99,100
plot 100,99
plot 100,100
plot 100,101
```

```
plot 101,100

color darkgreen
plot 200,200
```

Program 14: Use Plot to Draw Points



Sample Output 14: Use Plot to Draw Points (circled for emphasis)

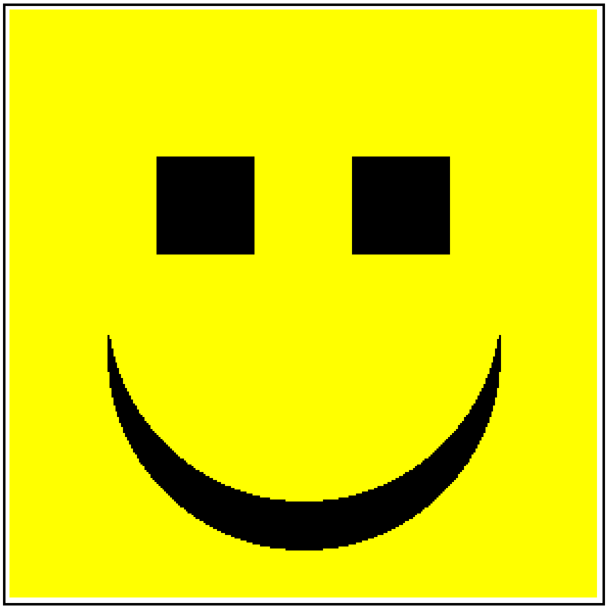
New Concept	**plot** *x, y* Changes a single pixel to the current color.

Big Program	At the end of each chapter there will be one or more big programs for you to look at, type in, and experiment with. These programs will contain only topics that we have covered so far in the book. This "Big Program" takes the idea of a face and makes it talk. Before the program will say each word the lower half of the face is redrawn with a different mouth shape. This creates a rough animation and makes the face more fun.

```
# c2_talkingface.kbs
# draw face background with eyes
color yellow
rect 0,0,300,300
color black
rect 75,75,50,50
rect 175,75,50,50

#erase old mouth
color yellow
rect 0,150,300,150
# draw new mouth
color black
rect 125,175,50,100
# say word
say "i"

color yellow
rect 0,150,300,150
color black
rect 100,200,100,50
say "am"

color yellow
```

```
rect 0,150,300,150
color black
rect 125,175,50,100
say "glad"

color yellow
rect 0,150,300,150
color black
rect 125,200,50,50
say "you"

color yellow
rect 0,150,300,150
color black
rect 100,200,100,50
say "are"

color yellow
rect 0,150,300,150
color black
rect 125,200,50,50
say "my"

# draw whole new face with round smile.
color yellow
rect 0,0,300,300
color black
circle 150,175,100
color yellow
circle 150,150,100
color black
rect 75,75,50,50
rect 175,75,50,50
say "friend"
```

Program 15: Big Program - Talking Face



Sample Output 15: Big Program - Talking Face

Chapter 3: Sound and Music.

Now that we have color and graphics, let's add sound and make some music. Basic concepts of the physics of sound, numeric variables, and musical notation will be introduced. You will be able to translate a tune into frequencies and durations to have the computer synthesize a voice.

Sound Basics – Things you need to know about sound:

Sound is created by vibrating air striking your ear-drum. These vibrations are known as sound waves. When the air is vibrating quickly you will hear a high note and when the air is vibrating slowly you will hear a low note. The rate of the vibration is called frequency.

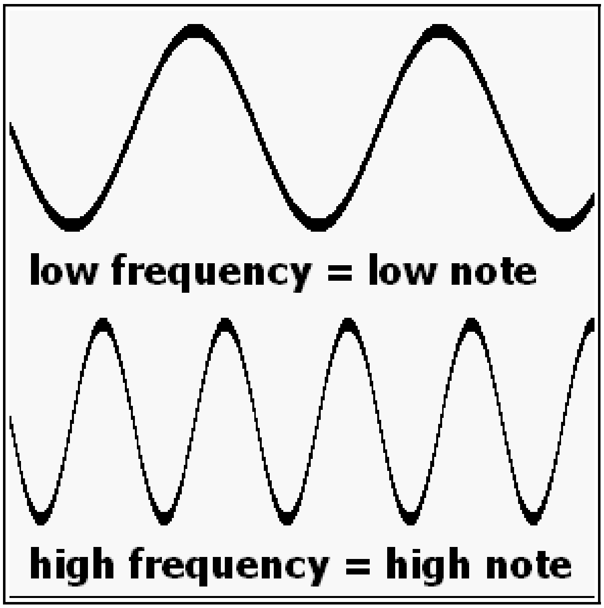


Illustration 7: Sound Waves

Frequency is measured in a unit called hertz (Hz). It represents how many cycles (ups and downs) a wave vibrates through in a second. A normal person can here very low sounds at 20 Hz and very high sounds at 20,000 Hz. BASIC-256 can produce tones in the range of 50Hz to 7000Hz.

Another property of a sound is it's length. Computers are very fast and can measure times accurately to a millisecond (ms). A millisecond (ms) is 1/1000 (one thousandths) of a second.

Let's make some sounds.

```
# c3_sounds.kbs
sound 233, 1000
sound 466, 500
sound 233, 1000
```

Program 16: Play Three Individual Notes

You may have heard a clicking noise in your speakers between the notes played in the last example. This is caused by the computer creating the sound and needing to stop and think a millisecond or so. The *sound* statement also can be written using a list of frequencies and durations to smooth out the transition from one note to another.

```
# c3_soundslist.kbs
sound {233, 1000, 466, 500, 233, 1000}
```

Program 17: List of Sounds

This second sound program plays the same three tones for the same duration but the computer creates and plays all of the sounds at once, making them smoother.

New Concept	**sound** *frequency*, *duration* **sound** {*frequency1*, *duration1*, *frequency2*, *duration2* ...} **sound** *numeric_array* The basic *sound* statement takes two arguments; (1) the frequency of the sound in Hz (cycles per second) and (2) the length of the tone in milliseconds (ms). The second form of the sound statement uses curly braces and can specify several tones and durations in a list. The third form of the sound statement uses an array containing frequencies and durations. Arrays are covered in Chapter 11.

How do we get BASIC-256 to play a tune? The first thing we need to do is to convert the notes on a music staff to frequencies. Illustration 7 shows two octaves of music notes, their names, and the approximate frequency the note makes. In music you will also find a special mark called the rest. The rest means not to play anything for a certain duration. If you are using a list of sounds you can insert a rest by specifying a frequency of zero (0) and the needed duration for the silence.

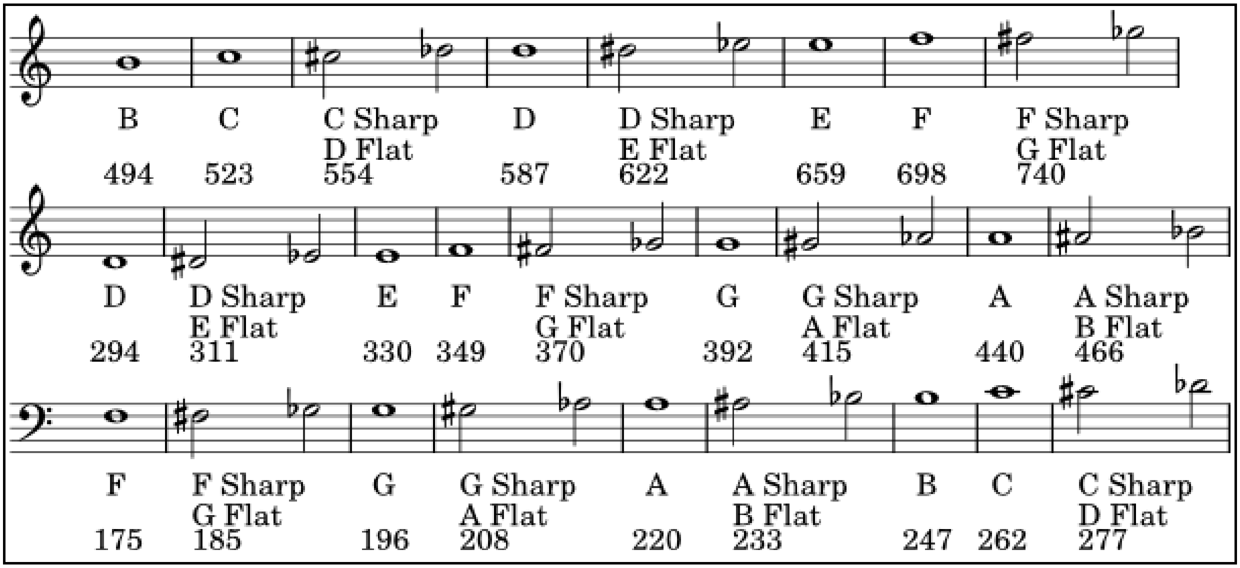


Illustration 8: Musical Notes

Take a little piece of music and then look up the frequency values for each of the notes. Why don't we have the computer play "Charge!". The music is in Illustration 9. You might notice that the high G in the music is not on the musical notes; if a note is not on the chart you can double (to make higher) or half (to make lower) the same note from one octave away.

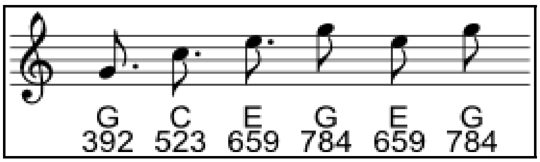


Illustration 9: Charge!

Now that we have the frequencies we need the duration for each of the notes. Table 2 shows most of the common note and rest symbols, how long they are when compared to each other, and a few typical durations.

Duration in milliseconds (ms) can be calculated if you know the speed if the music in beats per minute (BPM) using Formula 1.

$$Note\ Duration = 1000 * 60 / Beats\ Per\ Minute * Relative\ Length$$

Formula 1: Calculating Note Duration

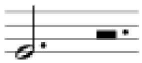
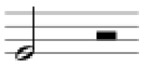
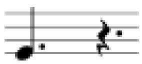
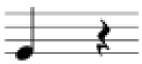
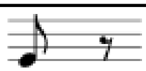
Note Name	Symbols for Note and Rest	Relative Length	At 100 BPM Duration ms	At 120 BPM Duration ms	At 140 BPM Duration ms
Dotted Whole		6.000	3600	3000	2571
Whole		4.000	2400	2000	1714
Dotted Half		3.000	1800	1500	1285
Half		2.000	1200	1000	857
Dotted Quarter		1.500	900	750	642
Quarter		1.000	600	500	428
Dotted Eighth		0.750	450	375	321
Eighth		0.500	300	250	214
Dotted Sixteenth		0.375	225	187	160
Sixteenth		0.250	150	125	107

Table 2: Musical Notes and Typical Durations

Now with the formula and table to calculate note durations, we can write the program to play “Charge!”.

```
# c3_charge.kbs - play charge
sound {392, 375, 523, 375, 659, 375, 784, 250, 659, 250, 784, 250}
say "Charge!"
```

Program 18: Charge!

Explore	Go on-line and find the music for “Row-row-row Your Boat” or another tune and write a program to play it.

Numeric Variables:

Computers are really good at remembering things, where we humans sometimes have trouble. The BASIC language allows us to give names to places in the computer's memory and then store information in them. These places are called variables.

There are four types of variables: numeric variables, string variables, numeric array variables, and string array variables. You will learn how to use numeric variables in this chapter and the others in later chapters.

New Concept	*Numeric variable* A numeric variable allows you to assign a name to a block of storage in the computer's short-term memory. You may store and retrieve numeric (whole or decimal) values from the numeric variable in your program. A numeric variable name must begin with a letter; may contain letters and numbers; and are case sensitive. You may not use words reserved by the BASIC-256 language when naming your variables (see Appendix I). Examples of valid variable names include: a, b6, reader, x, and zoo.

Warning	Variable names are case sensitive. This means that an upper case variable and a lowercase variable with the same letters do not represent the same location in the computer's memory.

Program 19 is an example of a program using numeric variables.

```
1    # c3_numericvariables.kbs
2    numerator = 30
3    denominator = 5
4    result = numerator / denominator
5    print result
```

Program 19: Simple Numeric Variables

The program above uses three variables. On line two it stores the value 30 into the location named “numerator”. Line three stores the value 5 in the variable “denominator”. Line four takes the value from “numerator” divides it by the value in the “denominator” variable and stores the value in the variable named “result”.

Now that we have seen variables in action we could re-write the “Charge!” program using variables and the formula to calculate note durations (Formula 1).

```
# c3_charge2.kbs
# play charge - use variables
beats = 120
dottedeighth = 1000 * 60 / beats * .75
eighth = 1000 * 60 / beats * .5
sound {392, dottedeighth, 523, dottedeighth, 659,
dottedeighth, 784, eighth, 659, eighth, 784,
eighth}
say "Charge!"
```

Program 20: Charge! with Variables

Warning	Change the speed of the music playing by adjusting the value stored in the beats

Big Program	For this chapter's big program let's take a piece of music by J.S. Bach and write a program to play it. The musical score is a part of J.S. Bach's Little Fuge in G.

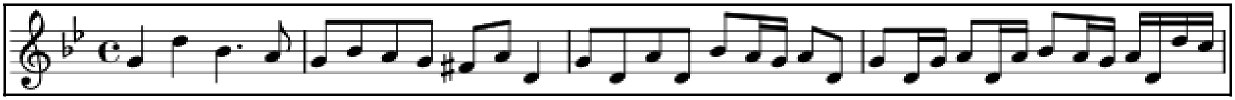


Illustration 10: First Line of J.S. Bach's Little Fuge in G

```
# c3_littlefuge.kbs
# Music by J.S.Bach - XVIII Fuge in G moll.
tempo = 100 # beats per minute
milimin = 1000 * 60 # miliseconds in a minute
q = milimin / tempo # quarter note is a beat
h = q * 2 # half note (2 quarters)
e = q / 2 # eight note (1/2 quarter)
s = q / 4 # sixteenth note (1/4 quarter)
de = e + s # dotted eight - eight + 16th
dq = q + e # doted quarter - quarter + eight

sound{392, q, 587, q, 466, dq, 440, e, 392, e,
466, e, 440, e, 392, e, 370, e, 440, e, 294, q,
392, e, 294, e, 440, e, 294, e, 466, e, 440, s,
392, s, 440, e, 294, e, 392, e, 294, s, 392, s,
440, e, 294, s, 440, s, 466, e, 440, s, 392, s,
440, s, 294, s}
```

Program 21: Big Program - Little Fuge in G

Chapter 4: Thinking Like a Programmer

One of the hardest things to learn is how to think like a programmer. A programmer is not created by simple books or classes but grows from within an individual. To become a “good” programmer takes passion for technology, self learning, basic intelligence, and a drive to create and explore.

You are like the great explorers Christopher Columbus, Neil Armstrong, and Yuri Gagarin (the first human in space). You have an unlimited universe to explore and to create within the computer. The only restrictions on where you can go will be your creativity and willingness to learn.

A program to develop a game or interesting application can often exceed several thousand lines of computer code. This can very quickly become overwhelming, even to the most experienced programmer. Often we programmers will approach a complex problem using a three step process, like:

1. Think about the problem.
2. Break the problem up into pieces and write them down formally.
3. Convert the pieces into the computer language you are using.

Pseudocode:

Pseudocode is a fancy word for writing out, step by step, what your program needs to be doing. The word pseudocode comes from the Greek prefix “pseudo-” meaning fake and “code” for the actual computer programming statements. It is not created for the computer to use directly but it is made to help you understand the complexity of a problem and to break it down into meaningful pieces.

There is no single best way to write pseudocode. Dozens of standards exist and each one of them is very suited for a particular type of problem. In this introduction we will use simple English statements to understand our problems.

How would you go about writing a simple program to draw a school bus (like in Illustration 11)?

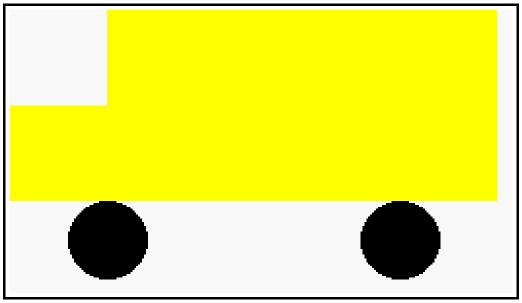


Illustration 11: School Bus

Let's break this problem into two steps:

- draw the wheels
- draw the body

Now let's break the initial steps into smaller pieces and write our pseudocode:

```
Set color to black.
Draw both wheels.
Set color to yellow.
Draw body of bus.
Draw the front of bus.
```

Table 3: School Bus - Pseudocode

Now that we have our program worked out, all we need to do is write it:

Set color to black.	`color black`
Draw both wheels.	`circle 50,120,20`
	`circle 200,120,20`
Set color to yellow.	`color yellow`
Draw body of bus.	`rect 50,0,200,100`
Draw the front of bus.	`rect 0,50,50,50`

Table 4: School Bus - Pseudocode with BASIC-256 Statements

The completed school bus program (Program 22) is listed below. Look at the finished program and you will see comment statements used in the program to help the programmer remember the steps used during the initial problem solving.

```
# schoolbus.kbs
clg
# draw wheels
color black
circle 50,120,20
circle 200,120,20
# draw bus body
color yellow
rect 50,0,200,100
rect 0,50,50,50
```

Program 22: School Bus

In the school bus example we have just seen there were many different ways to break up the problem. You could have drawn the bus first and the wheels last, you could have drawn the front before the back,... We could list dozens of different ways this simple problem could have been tackled.

One very important thing to remember, THERE IS NO WRONG WAY to approach a problem. Some ways are better than others (fewer instructions, easier to read, ...), but the important thing is that you solved the problem.

Explore	Try your hand at writing pseudocode. How would you tell BASIC-256 to draw a stick figure?

Flowcharting:

Another technique that programmers use to understand a problem is called flowcharting. Following the old adage of “a picture is worth a thousand words”, programmers will sometimes draw a diagram representing the logic of a program. Flowcharting is one of the oldest and commonly used methods of drawing this structure.

This brief introduction to flowcharts will only cover a small part of what that can be done with them, but with a few simple symbols and connectors you will be able to model very complex processes. This technique will serve you well not only in programming but in solving many problems you will come across. Here are a few of the basic symbols:

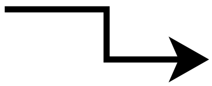
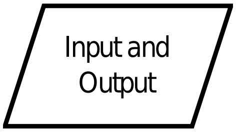
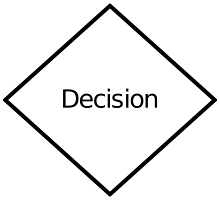
Symbol	Name and Description
	Flow – An arrow represents moving from one symbol or step in the process to another. You must follow the direction of the arrowhead.
	Terminator – This symbol tells us where to start and finish the flowchart. Each flowchart should have two of these: a start and a finish.
	Process – This symbol represents activities or actions that the program will need to take. There should be only one arrow leaving a process.
	Input and Output (I/O) – This symbol represents data or items being read by the system or being written out of the system. An example would be saving or loading files.
	Decision – The decision diamond asks a simple yes/no or true/false question. There should be two arrows that leave a decision. Depending on the result of the question we will follow one path out of the diamond.

Table 5: Essential Flowcharting Symbols

The best way to learn to flowchart is to look at some examples and to try your own hand it it.

Flowcharting Example One:

You just rolled out of bed and your mom has given you two choices for breakfast. You can have your favorite cold cereal or a scrambled egg.

If you do not choose one of those options you can go to school hungry.

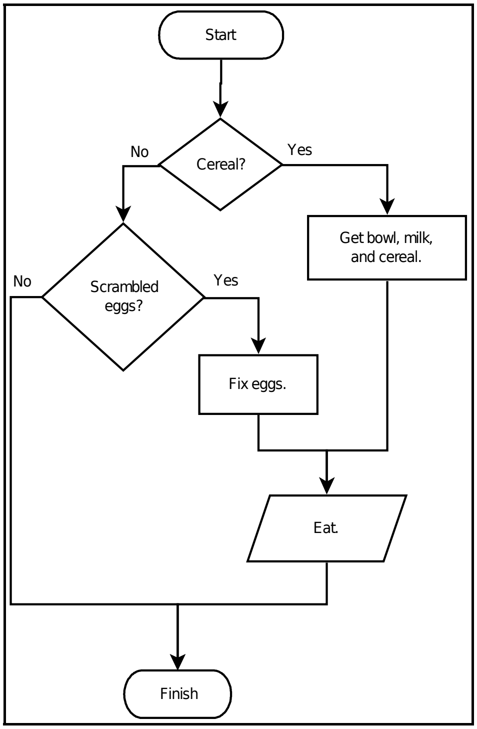


Illustration 12: Breakfast - Flowchart

Take a look at Illustration 12 (above) and follow all of the arrows. Do you see how that picture represents the scenario?

Flowcharting Example Two:

Another food example. You are thirsty and want a soda from the machine. Take a look at Illustration 13 (below).

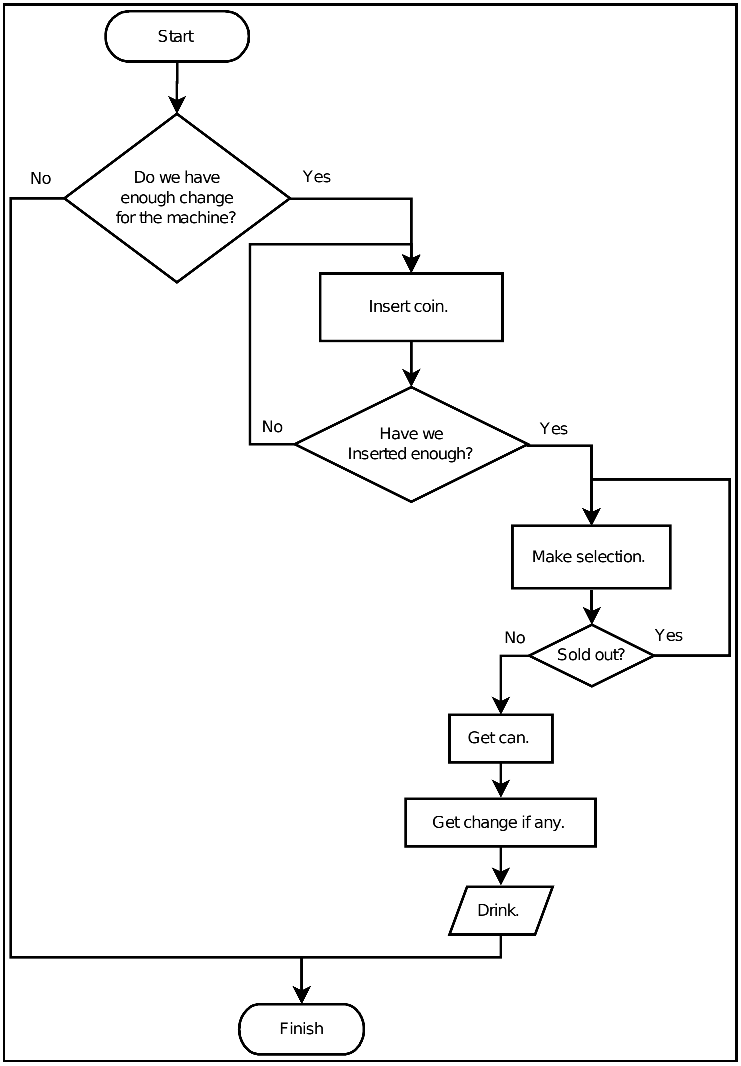


Illustration 13: Soda Machine - Flowchart

Notice in the second flowchart that there are a couple of times that we may need to repeat a process. You have not seen how to do that in BASIC-256, but it will be covered in the next few chapters.

Explore	Try your hand at drawing some simple flow charts. Try a chart for how to brush your teeth or how to cross the street.

Chapter 5: Your Program Asks for Advice.

This chapter introduces a new type of variables (string variables) and how to get text and numeric responses from the user.

Another Type of Variable – The String Variable:

In Chapter 3 you got to see numeric variables, which can only store whole or decimal numbers. Sometimes you will want to store a string, text surrounded by “”, in the computer's memory. To do this we use a new type of variable called the string variable. A string variable is denoted by appending a dollar sign *$* on a variable name.

You may assign and retrieve values from a string variable the same way you use a numeric variable. Remember, the variable name, case sensitivity, and reserved word rules are the same with string and numeric variables.

```
# ilikejim.kbs
name$ = "Jim"
firstmessage$ = name$ + " is my friend."
secondmessage$ = "I like " + name$ + "."
print firstmessage$
say firstmessage$
print secondmessage$
say secondmessage$
```

Program 23: I Like Jim

```
Jim is my friend.
I like Jim.
```

Sample Output 23: I Like Jim

New Concept	*String variable* A string variable allows you to assign a name to a block of storage in the computer's short-term memory. You may store and retrieve text and character values from the string variable in your program. A string variable name must begin with a letter; may contain letters and numbers; are case sensitive; and ends with a dollar sign. Also, you can not use words reserved by the BASIC-256 language when naming your variables (see Appendix I). Examples of valid string variable names include: d$, c7$, book$, X$, and barnYard$.

Warning	You may be tempted to assign a number to a string variable or a string to a numeric variable. If you do you will receive a syntax error.

Input – Getting Text or Numbers From the User:

So far we have told the program everything it needs to know in the programming code. The next statement to introduce is *input*. The *input* statement captures either a string or a number that the user types into the text area and stores that value in a variable.

Let's take Program 23 and modify it so that it will ask you for a name and then say hello to that person.

```
# ilikeinput.kbs
input “enter your name>”, name$
firstmessage$ = name$ + " is my friend."
secondmessage$ = "I like " + name$ + "."
print firstmessage$
say firstmessage$
print secondmessage$
say secondmessage$
```

Program 24: I Like?

```
enter your name>Vance
Vance is my friend.
I like Vance.
```

Sample Output 24: I Like?

New Concept

```
input "prompt", stringvariable$
input "prompt", numericvariable
input stringvariable$
input numericvariable
```

The **input** statement will retrieve a string or a number that the user types into the text output area of the screen. The result will be stored in a variable that may be used later in the program.

A prompt message, if specified, will display on the text output area and the cursor will directly follow the prompt.

If a numeric result is desired (numeric variable specified in the statement) and the user types a string that can not be converted to a number the input statement will set the variable to zero (0).

The “Math-wiz” program shows an example of input with numeric variables.

```
# mathwiz.kbs
input "a? ", a
input "b? ", b
print a + "+" + b + "=" + (a+b)
print a + "-" + b + "=" + (a-b)
print b + "-" + a + "=" + (b-a)
print a + "*" + b + "=" + (a*b)
print a + "/" + b + "=" + (a/b)
print b + "/" + a + "=" + (b/a)
```

Program 25: Math-wiz

```
a? 7
b? 56
7+56=63
7-56=-49
56-7=49
7*56=392
7/56=0.125
56/7=8
```

Sample Output 25: Math-wiz

Big Program

This chapter has two “Big Programs” The first is a fancy program that will say your name and how old you will be in 8 years and the second is a silly story generator.

```
# sayname.kbs
input "What is your name?", name$
input "How old are you?", age
greeting$ =  "It is nice to meet you, " + name$ + "."
print greeting$
say greeting$
greeting$ =  "In 8 years you will be " + (age + 8) + " years old.  Wow, thats old!"
print greeting$
say greeting$
```

Program 26: Fancy – Say Name

```
What is your name?Joe
How old are you?13
It is nice to meet you, Joe.
In 8 years you will be 21 years old.  Wow, thats old!
```

Sample Output 26: Fancy – Say Name

```
# sillystory.kbs

print "A Silly Story."

input "Enter a noun? ", noun1$
input "Enter a verb? ", verb1$
```

```
input "Enter a room in your house? ", room1$
input "Enter a verb? ", verb2$
input "Enter a noun? ", noun2$
input "Enter an adjective? ", adj1$
input "Enter a verb? ", verb3$
input "Enter a noun? ", noun3$
input "Enter Your Name? ", name$


sentence$ = "A silly story, by " + name$ + "."
print sentence$
say sentence$

sentence$ = "One day, not so long ago, I saw a " + noun1$ + " " + verb1$ + " down the stairs."
print sentence$
say sentence$

sentence$ = "It was going to my " + room1$ + " to " + verb2$ + " a " + noun2$
print sentence$
say sentence$

sentence$ = "The " + noun1$ + " became " + adj1$ + " when I " + verb3$ + " with a " + noun3$ + "."
print sentence$
say sentence$

sentence$ = "The End."
print sentence$
say sentence$
```

Program 27: Big Program - Silly Story Generator

```
A Silly Story.
Enter a noun? car
Enter a verb? walk
Enter a room in your house? kitchen
Enter a verb? sing
Enter a noun? television
Enter an adjective? huge
Enter a verb? watch
Enter a noun? computer
Enter Your Name? Jim
A silly story, by Jim.
One day, not so long ago, I saw a car walk down the
stairs.
It was going to my kitchen to sing a television
The car became huge when I watch with a computer.
The End.
```

Sample Output 27: Big Program - Silly Story Generator

Chapter 6: Decisions, Decisions, Decisions.

The computer is a whiz at comparing things. In this chapter we will explore how to compare two expressions, how to work with complex comparisons, and how to optionally execute statements depending on the results of our comparisons. We will also look at how to generate random numbers.

True and False:

The BASIC-256 language has one more special type of data that can be stored in numeric variables. It is the Boolean data type. Boolean values are either true or false and are usually the result of comparisons and logical operations. Also to make them easier to work with there are two Boolean constants that you can use in expressions, they are: *true* and *false*.

New Concept	*true* *false* The two Boolean constants *true* and *false* can be used in any numeric or logical expression but are usually the result of a comparison or logical operator. Actually, the constant *true* is stored as the number one (1) and *false* is stored as the number zero (0).

Comparison Operators:

Previously we have discussed the basic arithmetic operators, it is now time to look at some additional operators. We often need to compare two values in a program to help us decide what to do. A comparison operator works with two values and returns true or false based on the result of the comparison.

Operator	**Operation**
<	Less Than expression1 < expression2 Return true if expression1 is less than expression2, else return false.
<=	Less Than or Equal expression1 <= expression2 Return true if expression1 is less than or equal to expression2, else return false.
>	Greater Than expression1 > expression2 Return true if expression1 is greater than expression2, else return false.
>=	Greater Than or Equal expression1 >= expression2 Return true if expression1 is greater than or equal to expression2, else return false.
=	Equal expression1 = expression2 Return true if expression1 is equal to expression2, else return false.
<>	Not Equal Expression1 <> expression2 Return true if expression1 is not equal to expression2, else return false.

Table 6: Comparison Operators

New Concept	< <= > >= = <> The six comparison operations are: less than (<), less than or equal (<=), greater than (>), greater than or equal (>=), equal (=), and not equal (<>). They are used to compare numbers and strings. Strings are compared alphabetically left to right. You may also use parenthesis to group operations together.

Making Simple Decisions – The If Statement:

The *if* statement can use the result of a comparison to optionally execute a statement or block of statements. This first program (Program 28) uses three *if* statements to display whether your friend is older, the same age, or younger.

```
# compareages.kbs - compare two ages
input "how old are you?", yourage
input "how old is your friend?", friendage

print "You are ";
if yourage < friendage then print "younger than";
if yourage = friendage then print "the same age as";
if yourage > friendage then print "older than";
print " your friend"
```

Program 28: Compare Two Ages

```
how old are you?13
how old is your friend?12
You are older than your friend
```

Sample Output 28: Compare Two Ages

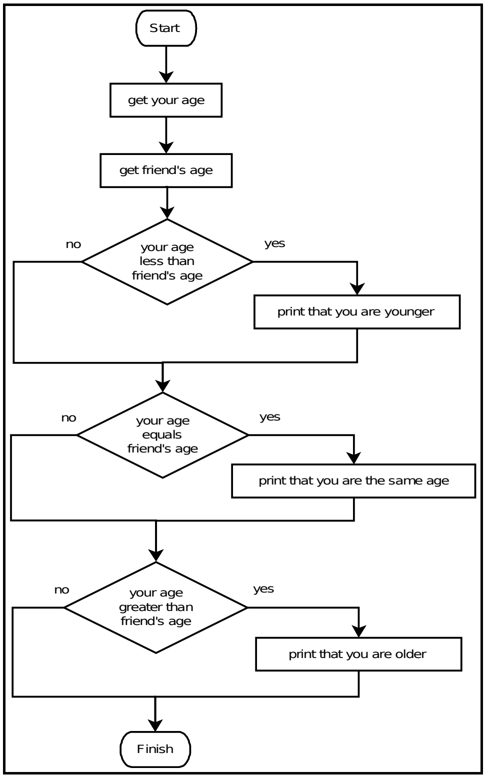


Illustration 14: Compare Two Ages - Flowchart

New Concept

if *condition* **then** *statement*

If the condition evaluates to *true* then execute the statement following the *then* clause.

Random Numbers:

When we are developing games and simulations it may become necessary for us to simulate dice rolls, spinners, and other random happenings. BASIC-256 has a built in random number generator to do these things for us.

New Concept	**rand**
	A random number is returned when rand is used in an expression. The returned number ranges from zero to one, but will never be one ($0 \geq n < 1.0$). Often you will want to generate an integer from 1 to r, the following statement can be used n = int(rand * r) + 1

```
# coinflip.kbs
coin = rand
if coin < .5 then print "Heads."
if coin >= .5 then print "Tails."
```

Program 29: Coin Flip

```
Tails.
```

Sample Output 29: Coin Flip

Warning	In program 5.2 you may have been tempted to use the *rand* expression twice, once in each if statement. This would have created what we call a “Logical Error”. Remember, each time the *rand* expression is executed it returns a different random number.

Logical Operators:

Sometimes it is necessary to join simple comparisons together. This can be done with the four logical operators: *and*, *or*, *xor*, and *not*. The logical operators work very similarly to the way conjunctions work in the English language, except that “or” is used as one or the other or both.

<table>
<tr><th>Operator</th><th>Operation</th></tr>
<tr><td>AND</td><td>Logical And
expression1 AND expression2
If both expression1 and experssion2 are true then return a true value, else return false.
<table>
<tr><th colspan="2" rowspan="2">AND</th><th colspan="2">expression1</th></tr>
<tr><th>TRUE</th><th>FALSE</th></tr>
<tr><th rowspan="2">expression 2</th><th>TRUE</th><td>TRUE</td><td>FALSE</td></tr>
<tr><th>FALSE</th><td>FALSE</td><td>FALSE</td></tr>
</table>
</td></tr>
<tr><td>OR</td><td>Logical Or
expression1 OR expression2
If either expression1 or experssion2 are true then return a true value, else return false.
<table>
<tr><th colspan="2" rowspan="2">OR</th><th colspan="2">expression1</th></tr>
<tr><th>TRUE</th><th>FALSE</th></tr>
<tr><th rowspan="2">expression 2</th><th>TRUE</th><td>TRUE</td><td>TRUE</td></tr>
<tr><th>FALSE</th><td>TRUE</td><td>FALSE</td></tr>
</table>
</td></tr>
</table>

XOR

Logical Exclusive Or
expression1 XOR expression2
If only one of the two expressions is true then return a true value, else return false. The XOR operator works like “or” often does in the English language - “You can have your cake xor you can eat it:.

OR		**expression1**	
		TRUE	**FALSE**
expression 2	**TRUE**	FALSE	TRUE
	FALSE	TRUE	FALSE

NOT

Logical Negation (Not)
NOT expression1
Return the opposite of expression1. If expression 1 was true then return false. If experssion1 was false then return a true.

NOT		
expression 1	**TRUE**	FALSE
	FALSE	TRUE

New Concept

```
and  or  xor  not
```

The four logical operations: logical and, logical or, logical exclusive or, and logical negation (not) join or modify comparisons. You may also use parenthesis to group operations together.

Making Decisions with Complex Results – If/End If:

When we are writing programs it sometimes becomes necessary to do multiple statements when a condition is true. This is done with the alternate format of the *if* statement. With this statement you do not place a statement on the same line as the if, but you place multiple (one or more) statements on lines following the if statement and then close the block of statements with the *end if* statement.

New Concept

```
if condition then
    statement(s) to execute when true
end if
```

The **if/end if** statements allow you to create a block of programming code to execute when a condition is true. It is often customary to indent the statements with in the **if/end if** statements so they are not confusing to read.

```
# dice.kbs
die1 = int(rand * 6) + 1
die2 = int(rand * 6) + 1
total = die1 + die2

print "die 1 = " + die1
print "die 2 = " + die2
print "you rolled " + total
say "you rolled " + total

if total = 2 then
   print "snake eyes!"
   say "snake eyes!"
end if
if total = 12 then
   print "box cars!"
   say "box cars!"
end if
if die1 = die2 then
   print "doubles - roll again!"
   say "doubles - roll again!"
end if
```

Program 30: Rolling Dice

```
die 1 = 6
die 2 = 6
you rolled 12
box cars!
doubles - roll again!
```

Sample Output 30: Rolling Dice

New Concept

"Edit" then "Beautify" on the menu

The "Beautify" option on the "Edit" menu will clean up the format of your program to make it easier to read. It will remove extra spaces from the beginning and ending of lines and will indent blocks of code (like in the *if/end if* statements).

Deciding Both Ways – If/Else/End If:

The third and last form of the *if* statement is the *if/else/end if*. This extends the if/end if statements by allowing you to create a block of code to execute if the condition is true and another block to execute when the condition is false.

New Concept

```
if condition then
    statement(s) to execute when true
else
    statement(s) to execute when false
end if
```

The **if**, **else**, and **end if** statements allow you to define two blocks of programming code. The first block, after the **then** clause, executes if the condition is true and the second block, after the **else** clause, will execute when the condition if false.

Program 31 re-writes Program 29 using the *else* statement.

```
# coinflip2 - coin flip with else
coin = rand
if coin < .5 then
   print "Heads."
   say "Heads."
else
   print "Tails."
   say "Tails."
end if
```

Program 31: Coin Flip – With Else

```
Heads.
```

Sample Output 31: Coin Flip – With Else

Nesting Decisions:

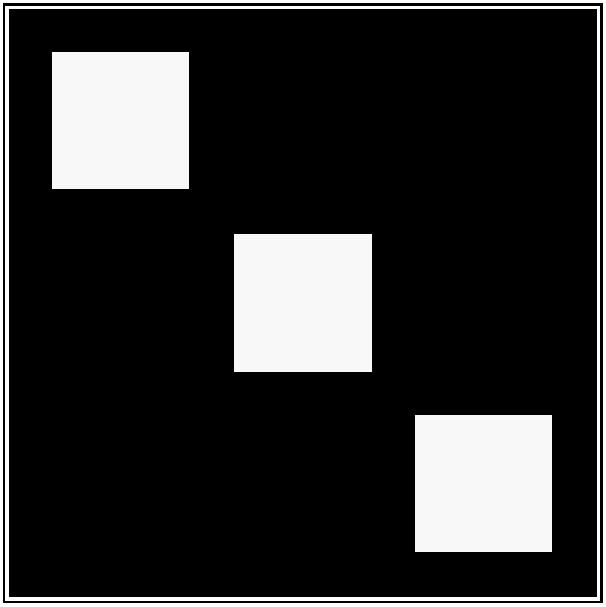
One last thing. With the *if/end if* and the *if/else/end if* statements it is possible to nest an *if* inside the code of another. This can become confusing but you will see this happening in future chapters.

Big Program	This chapter's big program is a program to roll a single 6-sided die and then draw on the graphics display the number of dots.

```
# dieroll.kbs
# hw - height and width of the dots on the dice
hw = 70
# margin - space before each dot
```

```
#   1/4 of the space left over after we draw 3 dots
margin = (300 - (3 * hw)) / 4
# z1 - x and y position of top of top row and column of dots
z1 = margin
# z2 - x and y position of top of middle row and column of dots
z2 = z1 + hw + margin
# z3 - x and y position of top of bottom row and column of dots
z3 = z2 + hw + margin

# get roll
roll = int(rand * 6) + 1
print roll

color black
rect 0,0,300,300

color white
# top row
if roll <> 1 then rect z1,z1,hw,hw
if roll = 6 then rect z2,z1,hw,hw
if roll >= 4 and roll <= 6 then rect z3,z1,hw,hw
# middle
if roll = 1 or roll = 3 or roll = 5 then rect z2,z2,hw,hw
# bottom row
if roll >= 4 and roll <= 6 then rect z1,z3,hw,hw
if roll = 6 then rect z2,z3,hw,hw
if roll <> 1 then rect z3,z3,hw,hw

say "you rolled a " + roll
```

Program 32: Big Program - Roll a Die and Draw It



Sample Output 32: Big Program - Roll a Die and Draw It

Chapter 7: Looping and Counting - Do it Again and Again.

So far our program has started, gone step by step through our instructions, and quit. While this is OK for simple programs, most programs will have tasks that need to be repeated, things counted, or both. This chapter will show you the three looping statements, how to speed up your graphics, and how to slow the program down.

The For Loop:

The most common loop is the *for* loop. The *for* loop repeatedly executes a block of statements a specified number of times, and keeps track of the count. The count can begin at any number, end at any number, and can step by any increment. Program 33 shows a simple for statement used to say the numbers 1 to 10 (inclusively). Program 34 will count by 2 starting at zero and ending at 10.

```
# for.kbs
for t = 1 to 10
   print t
   say t
next t
```

Program 33: For Statement

```
1
2
3
4
5
6
7
8
9
10
```

Sample Output 33: For Statement

```
# forstep2.kbs
for t = 0 to 10 step 2
   print t
   say t
next t
```

Program 34: For Statement – With Step

```
0
2
4
6
8
10
```

Sample Output 34: For Statement – With Step

New Concept

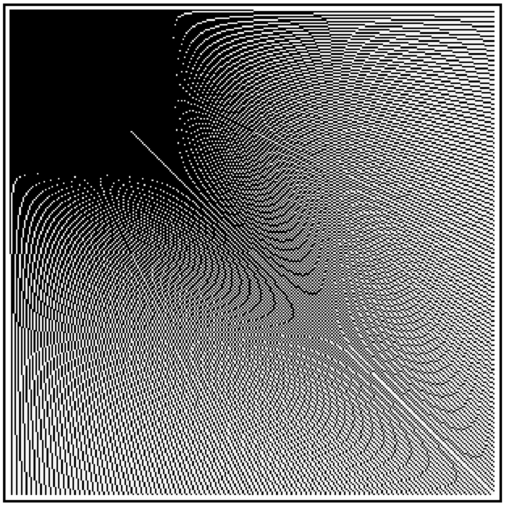
for *variable* **=** *expr1* **to** *expr2* [**step** *expr3*]
    statement(s)
next *variable*

Execute a specified block of code a specified number of times. The *variable* will begin with the value of *expr1*. The *variable* will be incremented by *expr3* (or one if step is not specified) the second and subsequent time through the loop. Loop terminates if *variable* exceeds *expr2*.

Using a loop we can easily draw very interesting graphics. Program 35 will draw a Moiré Pattern. This really interesting graphic is caused by the computer being unable to draw perfectly straight lines. What is actually drawn are pixels in a stair step fashion to approximate a straight line. If you look closely at the lines we have drawn you can see that they actually are jagged.

```
# moire.kbs
clg
color black
for t = 1 to 300 step 3
   line 0,0,300,t
   line 0,0,t,300
next t
```

Program 35: Moiré Pattern



Sample Output 35: Moiré Pattern

Explore	What kind of Moiré Patterns can you draw? Start in the center, use different step values, overlay one on top of another, try different colors, go crazy.

For statements can even be used to count backwards. To do this set the step to a negative number.

```
# forstepneg1.kbs
for t = 10 to 0 step -1
   print t
   pause 1.0
next t
```

Program 36: For Statement – Countdown

```
10
9
8
7
6
5
4
3
2
1
0
```

Sample Output 36: For Statement – Countdown

New Concept	**pause** *seconds* The pause statement tells BASIC-256 to stop executing the current program for a specified number of seconds. The number of seconds may be a decimal number if a fractional second pause is required.

Do Something Until I Tell You To Stop:

The next type of loop is the *do/until*. The *do/until* repeats a block of code one or more times. At the end of each iteration a logical condition is tested. The loop repeats as long as the condition is *false*. Program 37 uses the do/until loop to repeat until the user enters a number from 1 to 10.

```
# dountil.kbs
do
   input "enter a number from 1 to 10?",n
until n>=1 and n<=10
print "you entered " + n
```

Program 37: Get a Number from 1 to 10

```
enter a number from 1 to 10?66
enter a number from 1 to 10?-56
enter a number from 1 to 10?3
you entered 3
```

Sample Output 37: Get a Number from 1 to 10

New Concept	**do** *statement(s)* **until** *condition* Do the statements in the block over and over again while the condition is false. The statements will be executed one or more times.

Program 38 uses a *do/until* loop to count from 1 to 10 like Program 33 did with a *for* statement.

```
# dountilfor.kbs
t = 1
do
  print t
  t = t + 1
until t >= 11
```

Program 38: Do/Until Count to 10

```
1
2
3
4
5
6
7
8
9
10
```

Sample Output 38: Do/Until Count to 10

Do Something While I Tell You To Do It:

The third type of loop is the *while/end while*. It tests a condition before executing each iteration and if it evaluates to true then executes the code in the loop. The *while/end while* loop may execute the code inside the loop zero or more times.

Sometimes we will want a program to loop forever, until the user stops the program. This can easily be accomplished using the Boolean *true* constant (see Program 39).

```
# whiletrue.kbs
while true
   print "nevermore ";
end while
```

Program 39: Loop Forever

```
nevermore.
nevermore.
nevermore.
nevermore.
nevermore.
… runs until you stop it
```

Sample Output 39: Loop Forever

New Concept

```
while condition
    statement(s)
end while
```

Do the statements in the block over and over again while the condition is true. The statements will be executed zero or more times.

Program 40 uses a while loop to count from 1 to 10 like Program 33 did with a *for* statement.

```
# whilefor.kbs
t = 1
while t <= 10
   print t
   t = t + 1
end while
```

Program 40: While Count to 10

```
1
2
3
4
5
6
7
8
9
10
```

Sample Output 40: While Count to 10

Fast Graphics:

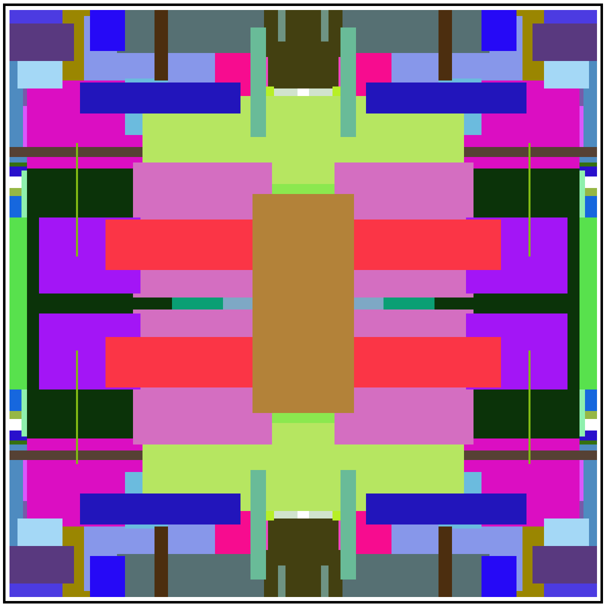
When we need to execute many graphics quickly, like with animations or games, BASIC-256 offers us a fast graphics system. To turn on this mode you execute the *fastgraphics* statement. Once *fastgraphics* mode is started the graphics output will only be updated once you execute the *refresh* statement.

New Concept	**fastgraphics** **refresh** Start the **fastgraphics** mode. In fast graphics the screen will only be updated when the **refresh** statement is executed. Once a program executes the **fastgraphics** statement it can not return to the standard graphics (slow) mode.

```
# kalidescope.kbs
clg
fastgraphics
for t = 1 to 100
   r = int(rand * 256)
   g = int(rand * 256)
   b = int(rand * 256)
   x = int(rand * 300)
   y = int(rand * 300)
   h = int(rand * 100)
   w = int(rand * 100)
   color rgb(r,g,b)
   rect x,y,w,h
   rect 300-x-w,y,w,h
   rect x,300-y-h,w,h
   rect 300-x-w,300-y-h,w,h
```

```
    next t
    refresh
```

Program 41: Kalidescope



Sample Output 41: Kalidescope

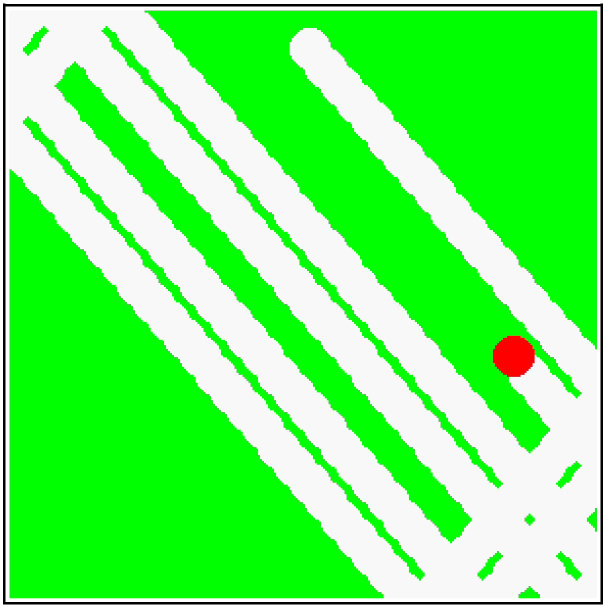
Explore	In Program 41, try running it with the *fastgraphics* statement removed or commented out. Do you see the difference?

Big Program	In this chapter's “Big Program” let's use a while loop to animate a ball bouncing around on the graphics display area.

```
# bouncingball.kbs
fastgraphics
clg

# starting position of ball
x = rand * 300
y = rand * 300
# size of ball
r = 10
# speed in x and y directions
dx = rand * r + 2
dy = rand * r + 2

color green
rect 0,0,300,300

while true
   # erase old ball
   color white
   circle x,y,r
   # calculate new position
   x = x + dx
   y = y + dy
   # if off the edges turn the ball around
   if x < 0 or x > 300 then
      dx = dx * -1
      sound 1000,50
   end if
   # if off the top or bottom turn the ball around
   if y < 0 or y > 300 then
      dy = dy * -1
      sound 1500,50
   end if
```

```
      # draw new ball
      color red
      circle x,y,r
      # update the display
      refresh
   end while
```

Program 42: Big Program - Bouncing Ball



Sample Output 42: Big Program - Bouncing Ball

Chapter 8: Custom Graphics – Creating Your Own Shapes.

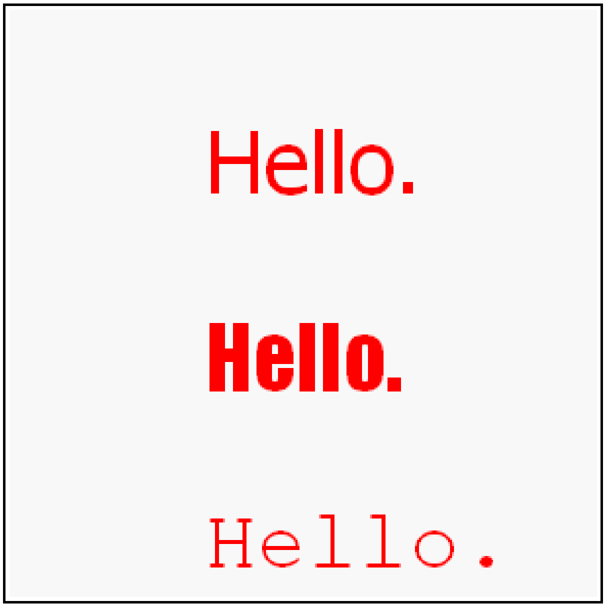
This chapter we will show you how to draw colorful words and special shapes on your graphics window. Several topics will be covered, including: fancy text; drawing polygons on the graphics output area; and stamps, where we can position, re-size, and rotate polygons. You also will be introduced to angles and how to measure them in radians.

Fancy Text for Graphics Output:

You have been introduced to the *print* statement (Chapter 1) and can output strings and numbers to the text output area. The *text* and *font* commands allow you to place numbers and text on the graphics output area.

```
# graphichello.kbs
clg
color red
font "Tahoma",33,100
text 100,100,"Hello."
font "Impact",33,50
text 100,150,"Hello."
font "Courier New",33,50
text 100,250,"Hello."
```

Program 43: Hello on the Graphics Output Area



Sample Output 43: Hello on the Graphics Output Area

New Concept

font *font_name, size_in_point, weight*

Set the font, size, and weight for the next *text* statement to use to render text on the graphics output area.

Argument	Description
font_name	String containing the system font name to use. A font must be previously loaded in the system before it may be used. Common font names under Windows include: "Verdana", "Courier New", "Tahoma", "Arial", and "Times New Roman".
size_in_point	Height of text to be rendered in a measurement known as point. There are 72 points in an inch.
weight	Number from 1 to 100 representing how dark letter should be. Use 25 for light, 50 for normal, and 75 for bold.

New Concept

text *x, y, expression*

Draw the contents of the *expression* on the graphics output area with it's top left corner specified by *x* and *y*. Use the font, size, and weight specified in the last **font** statement.

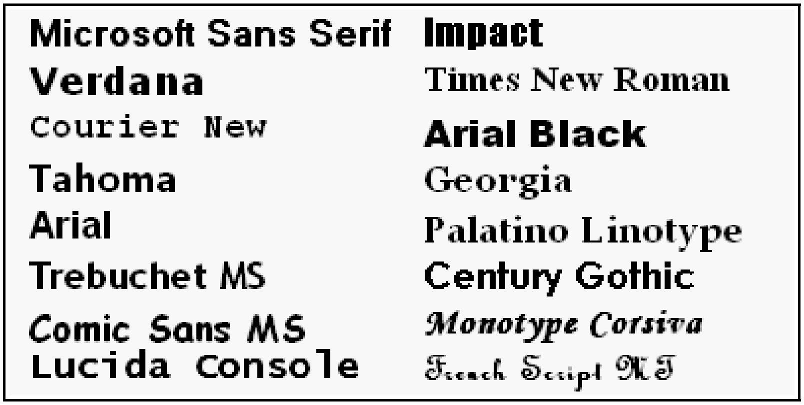


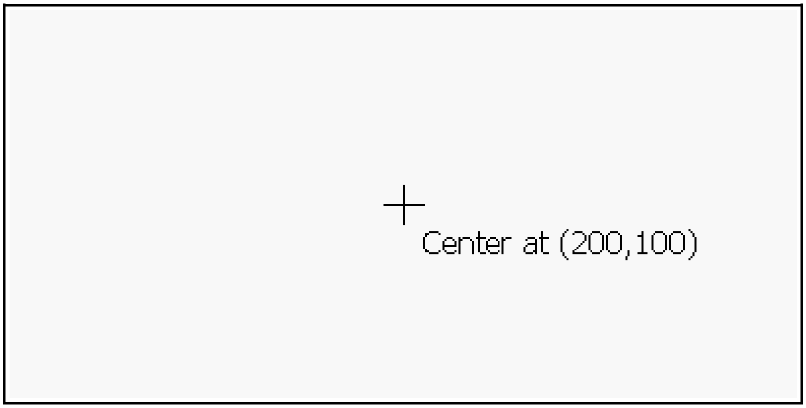
Illustration 15: Common Windows Fonts

Resizing the Graphics Output Area:

By default the graphics output area is 300x300 pixels. While this is sufficient for many programs, it may be too large or too small for others. The graphsize statement will re-size the graphics output area to what ever custom size you require. Your program may also use the graphwidth and graphheight functions to see what the current graphics size is set to.

```
# resizegraphics.kbs
graphsize 500,500
xcenter = graphwidth/2
ycenter = graphheight/2

color black
line xcenter, ycenter - 10, xcenter, ycenter + 10
line xcenter - 10, ycenter, xcenter + 10, ycenter

font "Tahoma",12,50
text xcenter + 10, ycenter + 10, "Center at (" +
xcenter + "," + ycenter + ")"
```

Program 44: Re-size Graphics



Sample Output 44: Re-size Graphics

New Concept	**graphsize** *width, height* Set the graphics output area to the specified *height* and *width*.

New Concept	**graphwidth** or **graphwidth**() **graphheight** or **graphheight**() Functions that return the current graphics height and width for you to use in your program.

Creating a Custom Polygon:

In previous chapters we learned how to draw rectangles and circles. Often we want to draw other shapes. The *poly* statement will allow us to draw a custom polygon anywhere on the screen.

Let's draw a big red arrow in the middle of the graphics output area. First, draw it on a piece of paper so we can visualize the coordinates of the vertices of the arrow shape.

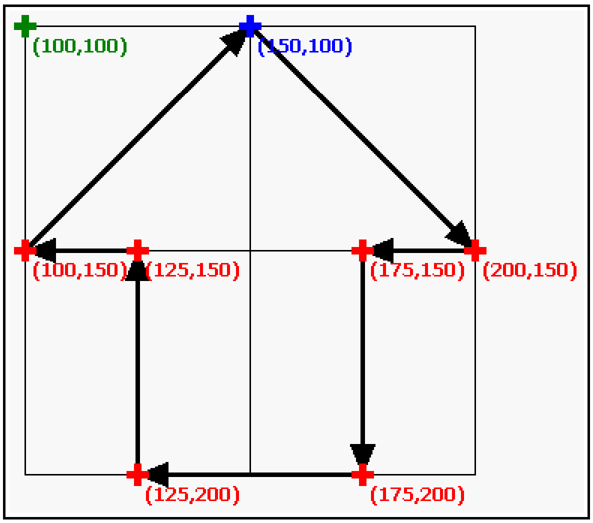
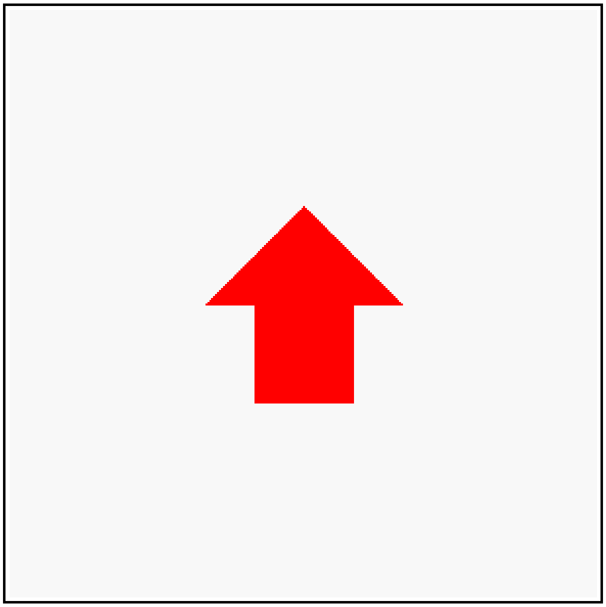


Illustration 16: Big Red Arrow

Now start at the top of the arrow going clockwise and write down the x and y values.

```
# bigredarrow.kbs
clg
color red
poly {150, 100, 200, 150, 175, 150, 175, 200,
125, 200, 125, 150, 100, 150}
```

Program 45: Big Red Arrow



Sample Output 45: Big Red Arrow

New Concept	**poly** {*x1, y1, x2, y2 ...*} **poly** *numeric_array* Draw a polygon

Stamping a Polygon:

The *poly* statement allowed ue to place a polygon at a specific location on the screen but it would be difficult to move it around or adjust it. These problems are solved with the *stamp* statement. The stamp statement takes a location on the screen, optional scaling (re-sizing), optional rotation, and a polygon definition to allow us to place a polygon anywhere we want it in the screen.

Let's draw an equilateral triangle (all sides are the same length) on a piece of paper. Put the point (0,0) at the top and make each leg 10 long (see Illustration 17).

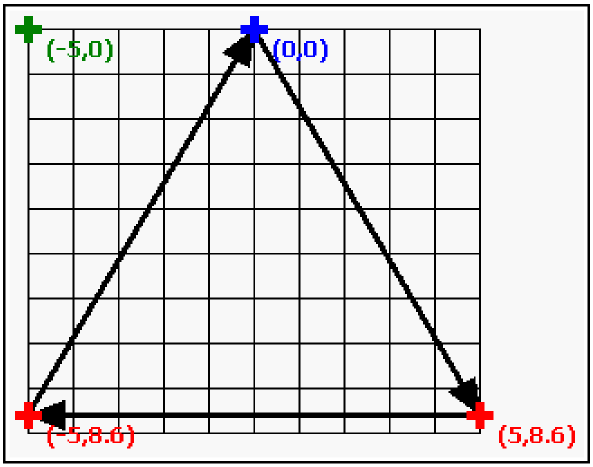
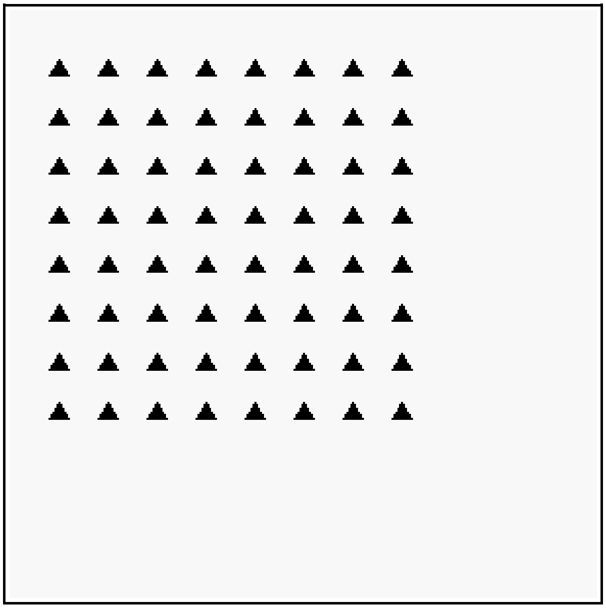


Illustration 17: Equilateral Triangle

Now we will create a program, using the simplest form of the *stamp* statement, to fill the screen with triangles. Program 46 Will do just that. It uses the triangle stamp inside two nested loops to fill the screen.

```
# stamptri.kbs
clg
color black
for x = 25 to 200 step 25
   for y = 25 to 200 step 25
      stamp x, y, {0, 0, 5, 8.6, -5, 8.6}
   next y
next x
```

Program 46: Fill Screen with Triangles



Sample Output 46: Fill Screen with Triangles

New Concept	
	stamp *x, y, {x1, y1, x2, y2 ...}* **stamp** *x, y, numeric_array* **stamp** *x, y, scale, {x1, y1, x2, y2 ...}* **stamp** *x, y, scale, numeric_array* **stamp** *x, y, scale, rotate, {x1, y1, x2, y2 ...}* **stamp** *x, y, scale, rotate, numeric_array* Draw a polygon with it's origin (0,0) at the screen position (x,y). Optionally scale (re-size) it by the decimal scale where 1 is full size. Also you may also rotate the stamp clockwise around it's origin by specifying how far to rotate as an angle expressed in radians (0 to 2π).

New Concept	
	Radians 0 to 2π Angles in BASIC-256 are expressed in a unit of measure known as a radian. Radians range from 0 to 2π. A right angle is π/2 radians and an about face is π radians. You can convert degrees to radians with the formula $r=d/180*\pi$.

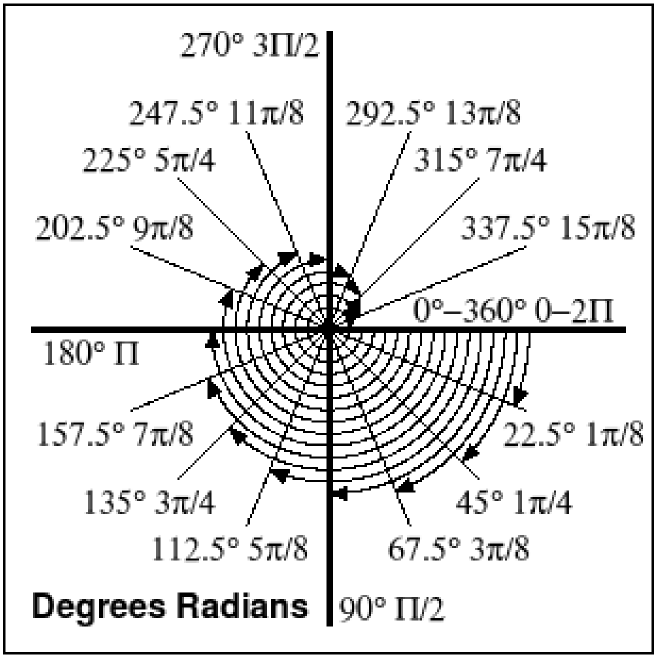
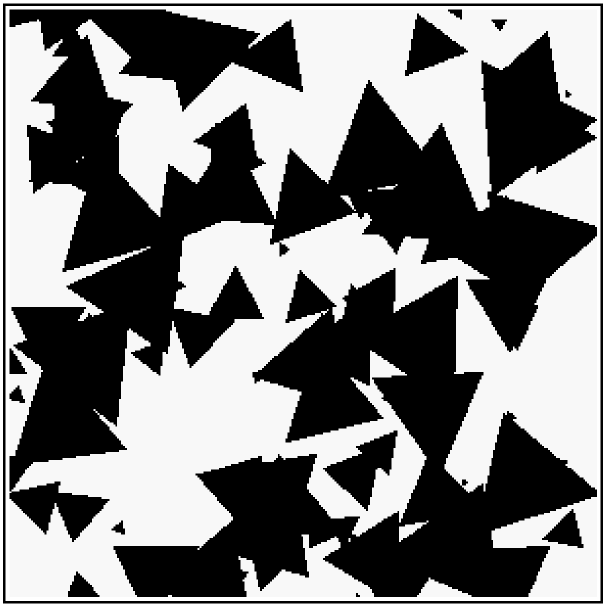


Illustration 18: Degrees and Radians

Let's look at another example of the stamp program. Program 47 used the same isosceles triangle as the last program but places 100 of them at random locations, randomly scaled, and randomly rotated on the screen.

```
# stamptri2.kbs
clg
color black
for t = 1 to 100
  x = rand * graphwidth
  y = rand * graphheight
  s = rand * 7
  r = rand * 2 * pi
  stamp x, y, s, r, {0, 0, 5, 8.6, -5, 8.6}
next t
```

Program 47: One Hundred Random Triangles



Sample Output 47: One Hundred Random Triangles

New Concept	`pi` The constant *pi* can be used in expressions so that you do not have to remember the value of π. Π is approximately 3.1415.

Explore	In Program 47, add statements to make the color random. Also create your own polygon to stamp.

Big Program	Let's send flowers to somebody special. The following program draws a flower using rotation and a stamp.

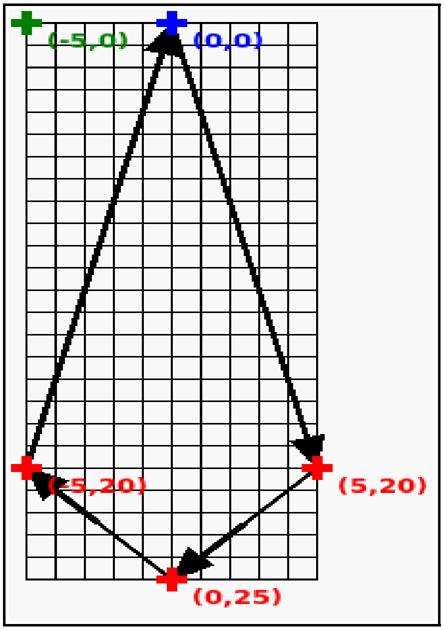
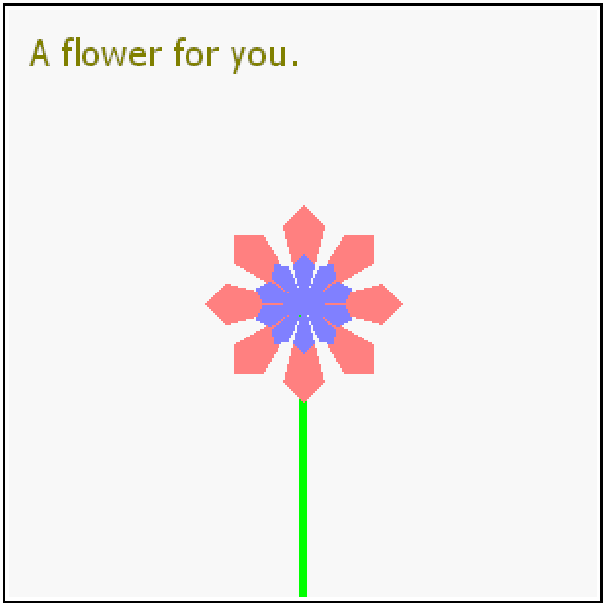


Illustration 19: Big Program - A Flower For You - Flower Petal Stamp

```
# aflowerforyou.kbs
clg

color green
rect 148,150,4,150

color 255,128,128
for r = 0 to 2*pi step pi/4
   stamp graphwidth/2, graphheight/2, 2, r, {0, 0, 5, 20, 0, 25, -5, 20}
next r

color 128,128,255
for r = 0 to 2*pi step pi/5
   stamp graphwidth/2, graphheight/2, 1, r, {0,
```

```
    0, 5, 20, 0, 25, -5, 20}
next r

message$ = "A flower for you."

color darkyellow
font "Tahoma", 14, 50
text 10, 10, message$
say message$
```

Program 48: Big Program - A Flower For You



Sample Output 48: Big Program - A Flower For You

Chapter 9: Subroutines – Reusing Code.

This chapter introduces the concept of setting labels within your code and then jumping to those labels. This will allow a program to execute the code in a more complex order. You will also see the subroutine. A *gosub* acts like a jump with the ability to jump back.

Labels and Goto:

In Chapter 7 we saw how to use language structures to perform looping. In Program 49 we can see an example of looping forever using a label and a *goto* statement.

```
# gotodemo.kbs
top:
print "hi"
goto top
```

Program 49: Goto With a Label

```
hi
hi
hi
hi
... repeats forever
```

Sample Output 49: Goto With a Label

New Concept

`label:`

A label allows you to name a place in your program so you may jump to that location later in the program. You may have multiple labels in a single program.

A label name is followed with a colon (:); must be on a line with no other statements; must begin with a letter; may contain letters and numbers; and are case sensitive. Also, you can not use words reserved by the BASIC-256 language when naming your variables (see Appendix I).

Examples of valid labels include: top:, far999:, and About:.

New Concept

`goto label`

The **goto** statement causes the execution to jump to the statement directly following the label.

Some programmers use labels with *goto* statements throughout their programs. While it is sometimes easier to program with *goto* statements they can add complexity to large programs, making the program more difficult to debug and maintain. It is recommended that you keep the use of *goto* statements to an absolute minimum.

Let's take a look at another example of a label and *goto* statement. In Program 50 we create a colorful clock.

```
# textclock.kbs
fastgraphics
font "Tahoma", 20, 100
color blue
rect 0, 0, 300, 300
color yellow
text 0, 0, "My Clock."
showtime:
color blue
rect 100, 100, 200, 100
color yellow
text 100, 100, hour + ":" + minute + ":" + second
refresh
pause 1.0
goto showtime
```

Program 50: Text Clock

Sample Output 50: Text Clock

New Concept

```
hour  or hour()
minute or minute()
second  or second()
month or month()
day  or day()
year or year()
```

The functions **year**, **month**, **day**, **hour**, **minute**, and **second** return the components of the system clock. They allow your program to tell what time it is.

year	Returns the system 4 digit year.
month	Returns month number 0 to 11. 0 – January, 1-February...
day	Returns the day of the month 1 to 28,29,30, or 31.
hour	Returns the hour 0 to 23 in 24 hour format. 0 – 12 AM, 1- 1 AM, … 13 – 12 PM, 14 – 1 PM, ...
minute	Returns the minute 0 to 59 in the current hour.
second	Returns the second 0 to 59 in the current minute.

Reusing Blocks of Code – The Gosub Statement:

Throughout many programs we will find lines or even whole sections of code being needed over and over again. To help with this problem BASIC-256 includes the concept of a subroutine. A subroutine is a block of code that can be called by other parts of the program to do a task or part of a task. When a subroutine is finished it returns control back to where it was called.

Program 51 shows an example of a subroutine that is called three times.

```
# gosubdemo.kbs
gosub showline
print "hi"
gosub showline
print "there"
gosub showline
end

showline:
print "------------------"
return
```

Program 51: Gosub

```
------------------
hi
------------------
there
------------------
```

Sample Output 51: Gosub

New Concept	**gosub** *label* The **gosub** statement causes the execution to jump to the subroutine defined by the *label*.

New Concept	**return** Execute the **return** statement within a subroutine to send control back to where it was called from.

New Concept	**end** Terminates the program (stop).

Now that we have seen the subroutine in action let's write a new digital clock program using a subroutine to format the time and date better (Program 52).

```
# textclockimproved.kbs

fastgraphics

while true
   color blue
   rect 0, 0, graphwidth, graphheight
   color white
   font "Times New Roman", 40, 100

   line$ = ""
   n = month + 1
   gosub addtoline
   line$ = line$ + "/"
   n = day
   gosub addtoline
   line$ = line$ + "/"
   line$ = line$ + year
```

```
   text 50,100, line$

   line$ = ""
   n = hour
   gosub addtoline
   line$ = line$ + ":"
   n = minute
   gosub addtoline
   line$ = line$ + ":"
   n = second
   gosub addtoline
   text 50,150, line$
   refresh
end while

addtoline:
  ## append a two digit number in n to the string line$
  if n < 10 then line$ = line$ + "0"
  line$ = line$ + n
 return
```

Program 52: Text Clock - Improved

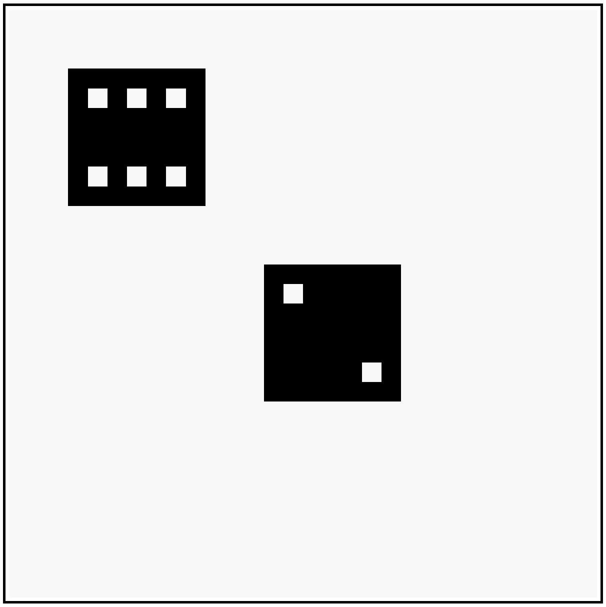
Sample Output 52: Text Clock - Improved

Big Program	In our “Big Program” this chapter, let's make a program to roll two dice, draw them on the screen, and give the total. Let's use a gosub to draw the image so that we only have to write it once.

```
# roll2dice.kbs
clg
total = 0

x = 30
y = 30
roll = int(rand * 6) + 1
total = total + roll
gosub drawdie

x = 130
y = 130
roll = int(rand * 6) + 1
total = total + roll
gosub drawdie

print "you rolled " + total + "."
end

drawdie:
# set x,y for top left and roll for number of dots
# draw 70x70 with dots 10x10 pixels
color black
rect x,y,70,70
color white
# top row
if roll <> 1 then rect x + 10, y + 10, 10, 10
if roll = 6 then rect x + 30, y + 10, 10, 10
if roll >= 4 and roll <= 6 then rect x + 50, y + 10, 10, 10
# middle
if roll = 1 or roll = 3 or roll = 5 then rect x + 30, y + 30, 10, 10
# bottom row
```

```
   if roll >= 4 and roll <= 6 then rect x + 10, y + 50, 10, 10
   if roll = 6 then rect x + 30, y + 50, 10, 10
   if roll <> 1 then rect x + 50, y + 50, 10, 10
   return
```

Program 53: Big Program - Roll Two Dice Graphically



Sample Output 53: Big Program - Roll Two Dice Graphically

Chapter 10: Mouse Control – Moving Things Around.

This chapter will show you how to make your program respond to a mouse. There are two different ways to use the mouse: tracking mode and clicking mode. Both are discussed with sample programs.

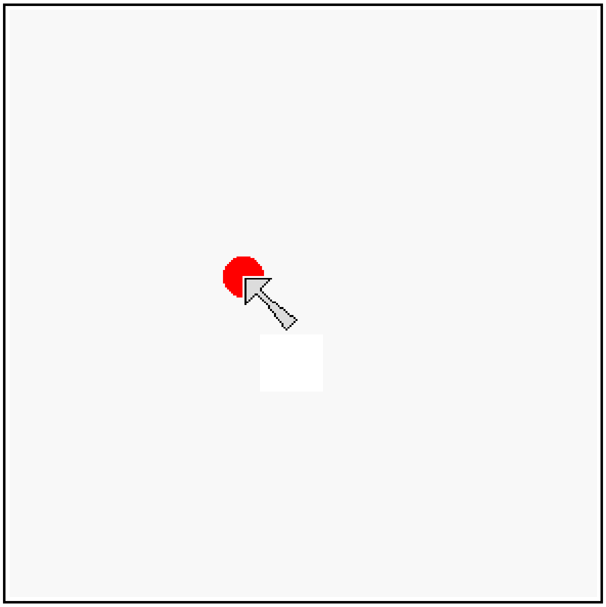
Tracking Mode:

In mouse tracking mode, there are three numeric functions (**mousex**, **mousey**, and **mouseb**) that will return the coordinates of the mouse pointer over the graphics output area. If the mouse is not over the graphics display area then the mouse movements will not be recorded (the last location will be returned).

```
# mousetrack.kbs
print "Move the mouse around the graphics window."
print "Click left mouse button to quit."

fastgraphics

# do it over and over until the user clicks left
while mouseb <> 1
  # erase screen
  color white
  rect 0, 0, graphwidth, graphheight
  # draw new ball
  color red
  circle mousex, mousey, 10
  refresh
end while

print "all done."
```

```
end
```

Program 54: Mouse Tracking



Sample Output 54: Mouse Tracking

New Concept

```
mousex or mousex()
mousey or mousey()
mouseb or mouseb()
```

The three mouse functions will return the current location of the mouse as it is moved over the graphics display area. Any mouse motions outside the graphics display area are not recorded, but the last known coordinates will be returned.

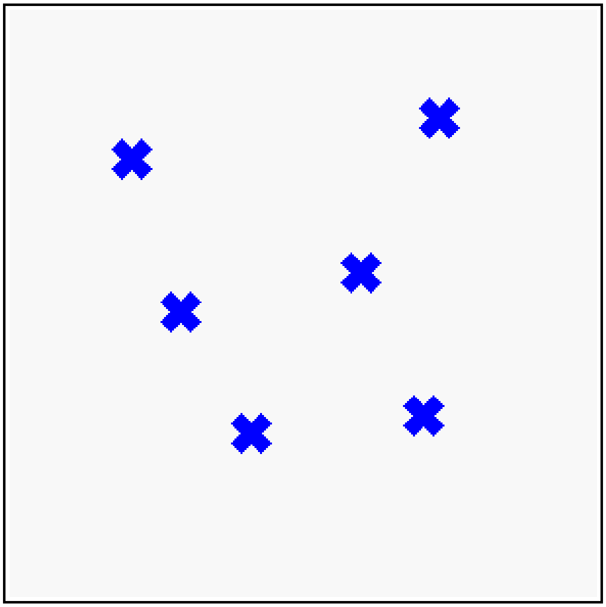
<table>
<tr><td>mousex</td><td>Returns the x coordinate of the mouse pointer position. Ranges from 0 to graphwidth -1.</td></tr>
<tr><td>mousey</td><td>Returns the y coordinate of the mouse pointer position. Ranges from 0 to graphheight -1.</td></tr>
<tr><td>mouseb</td><td>
<table>
<tr><td>0</td><td>Returns this value when no mouse button is being pressed.</td></tr>
<tr><td>1</td><td>Returns this value when the “left” mouse button is being pressed.</td></tr>
<tr><td>2</td><td>Returns this value when the “right” mouse button is being pressed.</td></tr>
<tr><td>4</td><td>Returns this value when the “center” mouse button is being pressed.</td></tr>
</table>
If multiple mouse buttons are being pressed at the same time then the value returned will be the button values added together.</td></tr>
</table>

Clicking Mode:

The second mode for mouse control is called “Clicking Mode”. In clicking mode, the mouse location and the button (or combination of buttons) are stored when the click happens. Once a click is processed by the program a *clickclear* command can be executed to reset the click, so the next one can be recorded.

```
# mouseclick.kbs
# X marks the spot where you click
print "Move the mouse around the graphics window"
print "click left mouse button to mark your spot"
print "click right mouse button to stop."
clg
clickclear
while clickb <> 2
   # clear out last click and
   # wait for the user to click a button
   clickclear
   while clickb = 0
      pause .01
   end while
   #
   color blue
   stamp clickx, clicky, 5, {-1, -2, 0, -1, 1, -2, 2, -1, 1, 0, 2, 1, 1, 2, 0, 1, -1, 2, -2, 1, -1, 0, -2, -1}
end while
print "all done."
end
```

Program 55: Mouse Clicking



Sample Output 55: Mouse Clicking

New Concept

```
clickx  or clickx()
clicky  or clicky()
clickb or clickb()
```

The values of the three click functions are updated each time a mouse button is clicked when the pointer is on the graphics output area. The last location of the mouse when the last click was received are available from these three functions.

New Concept

```
clickclear
```

The **clickclear** statement resets the **clickx**, **clicky**, and **clickb** functions to zero so that a new click will register when **clickb** <> 0.

Big Program

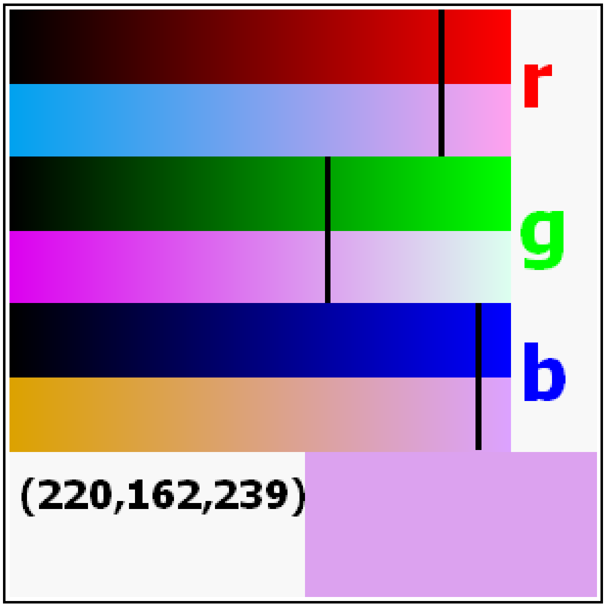
The big program this chapter uses the mouse to move color sliders so that we can see all 16,777,216 different colors on the screen.

```
# colorchooser.kbs
fastgraphics

print "colorchooser - find a color"
print "click and drag red, green and blue sliders"

# variables to store the color parts
r = 128
g = 128
b = 128

gosub display

while true
  # wait for click
  while mouseb = 0
    pause .01
  end while
  # change color sliders
  if mousey < 75 then
    r = mousex
    if r > 255 then r = 255
  end if
  if mousey >= 75 and mousey < 150 then
    g = mousex
    if g > 255 then g = 255
  end if
```

```
    if mousey >= 150 and mousey < 225 then
      b = mousex
      if b > 255 then b = 255
    end if
    gosub display
  end while
  end

  display:
  clg
  # draw red
  color 255, 0, 0
  font "Tahoma", 30, 100
  text 260, 10, "r"
  for t = 0 to 255
      color t, 0, 0
      line t,0,t,37
      color t, g, b
      line t, 38, t, 75
  next t
  color black
  rect r-1, 0, 3, 75
  # draw green
  color 0, 255, 0
  font "Tahoma", 30, 100
  text 260, 85, "g"
  for t = 0 to 255
      color 0, t, 0
      line t,75,t, 75 + 37
      color r, t, b
      line t, 75 + 38, t, 75 + 75
  next t
  color black
  rect g-1, 75, 3, 75
  # draw blue
  color 0, 0, 255
```

```
font "Tahoma", 30, 100
text 260, 160, "b"
for t = 0 to 255
    color 0, 0, t
    line t, 150, t, 150 + 37
    color r, g, t
    line t, 150 + 38, t, 150 + 75
next t
color black
rect b-1, 150, 3, 75
# draw swatch
color black
font "Tahoma", 15, 100
text 5, 235, "(" + r + "," + g + "," + b + ")"
color r,g,b
rect 151,226,150,75
refresh
return
```

Program 56: Big Program - Color Chooser



Sample Output 56: Big Program - Color Chooser

Chapter 11: Keyboard Control – Using the Keyboard to Do Things.

This chapter will show you how to make your program respond to the user when a key is pressed (arrows, letters, and special keys) on the keyboard.

Getting the Last Key Press:

The *key* function returns the last raw keyboard code generated by the system when a key was pressed. Certain keys (like control-c and function-1) are captured by the BASIC256 window and will not be returned by key. After the last key press value has been returned the function value will be set to zero (0) until another keyboard key has been pressed.

The key values for printable characters (0-9, symbols, letters) are the same as their upper case Unicode values regardless of the status of the caps-lock or shift keys.

```
# readkey.kbs
print "press a key - Q to quit"
do
   k = key
   if k <> 0 then
      if k >=32 and k <= 127 then
         print chr(k) + "=";
      end if
      print k
   end if
until k = asc("Q")
end
```

Program 57: Read Keyboard

```
press a key - Q to quit
A=65
Z=90
M=77
16777248
&=38
7=55
```

Sample Output 57: Read Keyboard

New Concept	**key** **key**()
	The **key** function returns the value of the last keyboard key the user has pressed. Once the key value is read by the function, it is set to zero to denote that no key has been pressed.

New Concept	*Unicode* The Unicode standard was created to assign numeric values to letters or characters for the world's writing systems. There are more than 107,000 different characters defined in the Unicode 5.0 standard. See: http://www.unicode.org

New Concept	**`asc`**`(`*`expression`*`)` The **asc** function returns an integer representing the Unicode value of the first character of the string *expression*.

New Concept	**`chr`**`(`*`expression`*`)` The **chr** function returns a string, containing a single character with the Unicode value of the integer *expression*.

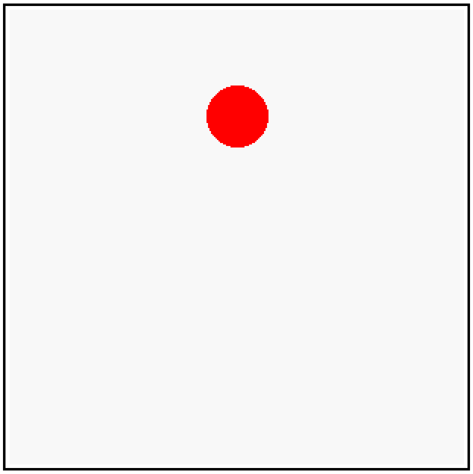
How about we look at a more complex example? Program 58 Draws a red ball on the screen and the user can move it around using the keyboard.

```
# moveball.kbs
print "use i for up, j for left, k for right, m for down, q to quit"

fastgraphics
clg
ballradius = 20
```

```

# position of the ball
# start in the center of the screen
x = graphwidth /2
y = graphheight / 2

# draw the ball initially on the screen
gosub drawball

# loop and wait for the user to press a key
while true
   k = key
   if k = asc("I") then
      y = y - ballradius
      if y < ballradius then y = graphheight - ballradius
      gosub drawball
   end if
   if k = asc("J") then
      x = x - ballradius
      if x < ballradius then x = graphwidth - ballradius
      gosub drawball
   end if
   if k = asc("K") then
      x = x + ballradius
      if x > graphwidth - ballradius then x = ballradius
      gosub drawball
   end if
   if k = asc("M") then
      y = y + ballradius
      if y > graphheight - ballradius then y = ballradius
      gosub drawball
   end if
   if k = asc("Q") then end
```

```
end while

drawball:
color white
rect 0, 0, graphwidth, graphheight
color red
circle x, y, ballradius
refresh
return
```

Program 58: Move Ball



Sample Output 58: Move Ball

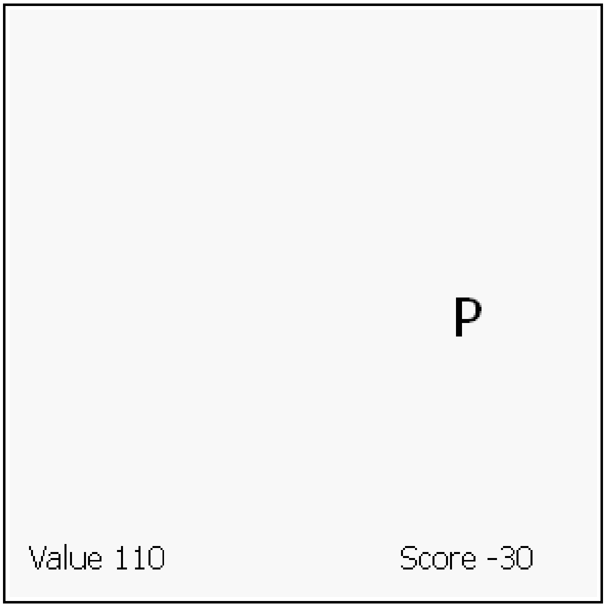
Big Program	The big program this chapter is a game using the keyboard. Random letters are going to fall down the screen and you score points by pressing the key as fast as you can.

```
# fallinglettergame.kbs
```

```

speed = .15 # drop speed - lower to make faster
nletters = 10 # letters to play

score = 0
misses = 0
color black

fastgraphics

clg
font "Tahoma", 20, 50
text 20, 80, "Falling Letter Game"
text 20, 140, "Press Any Key to Start"
refresh
# clear keyboard and wait for any key to be pressed
k = key
while key = 0
   pause speed
end while

for n = 1 to nletters
   letter = int((rand * 26)) + asc("A")
   x = 10 + rand * 225
   for y = 0 to 250 step 20
      clg
      # show letter
      font "Tahoma", 20, 50
      text x, y, chr(letter)
      # show score and points
      font "Tahoma", 12, 50
      value = (250 - y)
      text 10, 270, "Value "+ value
      text 200, 270, "Score "+ score
      refresh
```

```
         k = key
         if k <> 0 then
            if k = letter then
               score = score + value
            else
               score = score - value
            end if
            goto nextletter
         end if
         pause speed
      next y
      misses = misses + 1
   nextletter:
   next n

   clg
   font "Tahoma", 20, 50
   text 20, 40, "Falling Letter Game"
   text 20, 80, "Game Over"
   text 20, 120, "Score: " + score
   text 20, 160, "Misses: " + misses
   refresh
   end
```

Program 59: Big Program - Falling Letter Game



Sample Output 59: Big Program - Falling Letter Game

Chapter 12: Images, WAVs, and Sprites

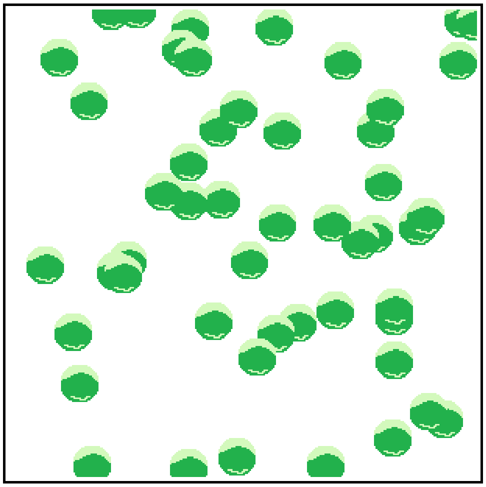
This chapter will introduce the really advanced multimedia and graphical statements. Loading images from files, playing sounds asynchronously from WAV files, and really cool animation using sprites.

Images From a File:

So far we have seen how to create shapes and graphics using the built in drawing statements. The **imgload** statement allows you to load a picture from a file and display it in your BASIC-256 programs.

```
# imgload_ball.kbs - Show Imgload
clg
for i = 1 to 50
   imgload rand * graphwidth, rand * graphheight, "greenball.png"
next i
```

Program 60: Imgload a Graphic



Sample Output 60: Imgload a Graphic

Program 60 Shows an example of this statement in action. The last argument is the name of a file on your computer. It needs to be in the same folder as the program, unless you specify a full path to it. Also notice that the coordinates (x,y) represent the CENTER of the loaded image and not the top left corner.

	Most of the time you will want to save the program into the same folder that the image or sound file is in BEFORE you run the program. This will set your current working directory so that BASIC-256 can find the file to load.

New Concept

imgload *x, y, filename*
imgload *x, y, scale, filename*
imgload *x, y, scale, rotation, filename*

Read in the picture found in the file and display it on the graphics output area. The values of *x* and *y* represent the location to place the CENTER of the image.

Images may be loaded from many different file formats, including: BMP, PNG, GIF, JPG, and JPEG.

Optionally scale (re-size) it by the decimal scale where 1 is full size. Also you may also rotate the image clockwise around it's center by specifying how far to rotate as an angle expressed in radians (0 to 2π).

The **imgload** statement also allows optional scaling and rotation like the **stamp** statement does. Look at Program 61 for an example.

```
# imgload_picasso.kbs - Show Imgload with rotation and scaling
graphsize 500,500
clg
for i = 1 to 50
   imgload graphwidth/2, graphheight/2, i/50, 2*pi*i/50, "picasso_selfport1907.jpg"
next i
say "hello Picasso."
```

Program 61: Imgload a Graphic with Scaling and Rotation

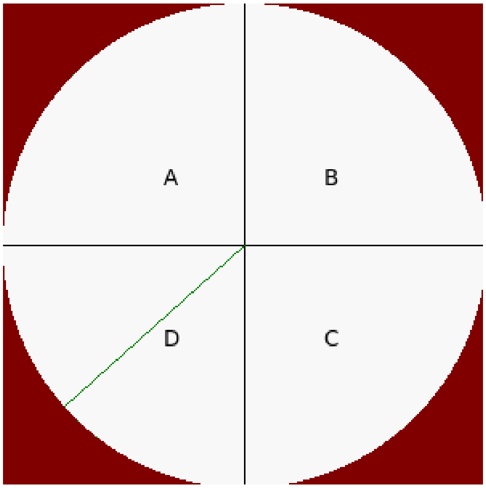
Sample Output 61: Imgload a Graphic with Scaling and Rotation

Playing Sounds From a WAV file:

So far we have explored making sounds and music using the **sound** command and text to speech with the **say** statement. BASIC-256 will also play sounds stored in WAV files. The playback of a sound from a WAV file will happen in the background. Once the sound starts the program will continue to the next statement and the sound will continue to play.

```
# spinner.kbs
fastgraphics
wavplay "roll.wav"

# setup spinner
angle = rand * 2 * pi
speed = rand * 2
color darkred
rect 0,0,300,300

for t = 1 to 100
# draw spinner
   color white
   circle 150,150,150
   color black
   line 150,300,150,0
   line 300,150,0,150
   text 100,100,"A"
   text 200,100,"B"
   text 200,200,"C"
   text 100,200,"D"
   color darkgreen
   line 150,150,150 + cos(angle)*150, 150 +
sin(angle)*150
   refresh
# update angle for next redraw
   angle = angle + speed
   speed = speed * .9
   pause .05
next t

# wait for sound to complete
wavwait
```

Program 62: Spinner with Sound Effect



Sample Output 62: Spinner with Sound Effect

New Concept	**wavplay** *filename* **wavwait** **wavstop** The **wavplay** statement loads a wave audio file (.wav) from the current working folder and plays it. The playback will be synchronous meaning that the next statement in the program will begin immediately as soon as the audio begins playing. **Wavstop** will cause the currently playing wave audio file to stop the synchronous playback and **wavwait** will cause the program to stop and wait for the currently playing sound to complete.

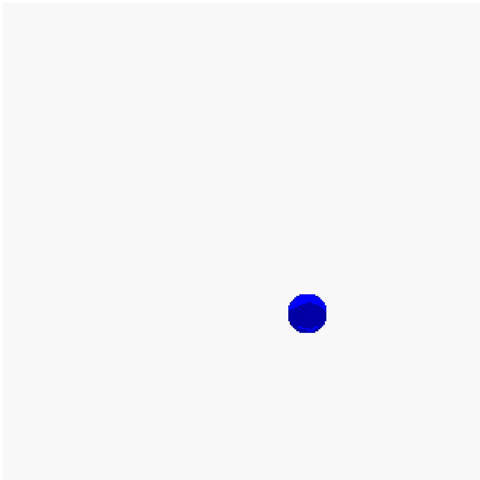
Moving Images - Sprites:

Sprites are special graphical objects that can be moved around the screen without having to redraw the entire screen. In addition to being mobile you can detect when one sprite overlaps (collides) with another. Sprites make programming complex games and animations much easier.

```
# sprite_1ball.kbs

color white
rect 0, 0, graphwidth, graphheight

spritedim 1

spriteload 0, "blueball.png"
spriteplace 0, 100,100
spriteshow 0

dx = rand * 10
dy = rand * 10

while true
   if spritex(0) <=0 or spritex(0) >= graphwidth -1 then
      dx = dx * -1
      wavplay "4359__NoiseCollector__PongBlipF4.wav"
   end if
   if spritey(0) <= 0 or spritey(0) >= graphheight -1 then
      dy = dy * -1
      wavplay "4361__NoiseCollector__pongblipA_3.wav"
```

```
      endif
      spritemove 0, dx, dy
      pause .05
   end while
```

Program 63: Bounce a Ball with Sprite and Sound Effects



Sample Output 63: Bounce a Ball with Sprite and Sound Effects

As you can see in Program 63 the code to make a ball bounce around the screen, with sound effects, is much easier than earlier programs to do this type of animation. When using sprites we must tell BASIC-256 how many there will be (**spritedim**), we need to set them up (**spriteload** or **spriteplace**), make them visible (**spriteshow**), and then move them around (**spritemove**). In addition to these statements there are functions that will tell us where the sprite is on the screen (**spritex** and **spritey**), how big the sprite is (**spritew** and spriteh) and if the sprite is visible (**spritev**).

New Concept

spritedim *numberofsprites*

The **spritedim** statement initializes, or allocates in memory, places to store the specified number of sprites. You may allocate as many sprites as your program may require but your program may slow down if you create too many sprites.

New Concept

spriteload *spritenumber, filename*

This statement reads an image file (GIF, BMP, PNG, JPG, or JPEG) from the specified path and creates a sprite.

By default the sprite will be placed with its center at 0,0 and it will be hidden. You should move the sprite to the desired position on the screen (**spritemove** or **spriteplace**) and then show it (**spriteshow**).

New Concept

spritehide *spritenumber*
spriteshow *spritenumber*

The **spriteshow** statement causes a loaded, created, or hidden sprite to be displayed on the graphics output area.

Spritehide will cause the specified sprite to not be drawn on the screen. It will still exist and may be shown again later.

New Concept

spriteplace *spritenumber, x, y*

The **spriteplace** statement allows you to place a sprite's center at a specific location on the graphics output area.

New Concept	**spritemove** *spritenumber, dx, dy* Move the specified sprite *x* pixels to the right and *y* pixels down. Negative numbers can also be specified to move the sprite left and up. A sprite's center will not move beyond the edge of the current graphics output window (0,0) to (**graphwidth**-1, **graphheight**-1). You may move a hidden sprite but it will not be displayed until you show the sprite using the **showsprite** statement.

New Concept	**spritev**(*spritenumber*) This function returns a true value if a loaded sprite is currently displayed on the graphics output area. False will be returned if it is not visible.

New Concept

spriteh (*spritenumber*)
spritew (*spritenumber*)
spritex (*spritenumber*)
spritey (*spritenumber*)

These functions return various pieces of information about a loaded sprite.

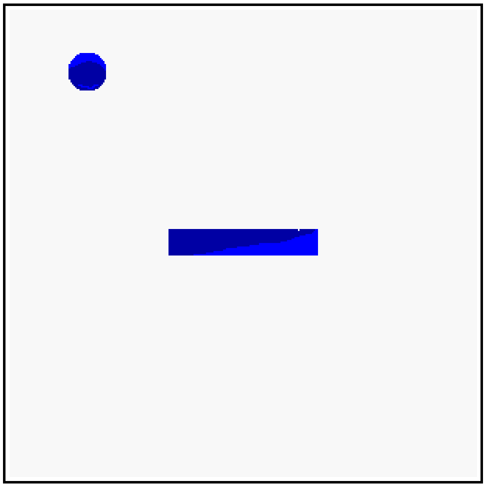
spriteh	Returns the height of a sprite in pixels.
spritew	Returns the width of a sprite in pixels.
spritex	Returns the position on the x axis of the center of the sprite.
spritey	Returns the position on the y axis of the center of the sprite.

The second sprite example (Program 64) we now have two sprites. The first one (number zero) is stationary and the second one (number one) will bounce off of the walls and the stationary sprite.

```
# sprite_bumper.kbs

color white
rect 0, 0, graphwidth, graphheight

spritedim 2

# stationary bumber
spriteload 0, "paddle.png"
spriteplace 0,graphwidth/2,graphheight/2
spriteshow 0

# moving ball
spriteload 1, "blueball.png"
```

```
spriteplace 1, 50, 50
spriteshow 1
dx = rand * 5 + 5
dy = rand * 5 + 5

while true
   if spritex(1) <=0 or spritex(1) >= graphwidth -1 then
      dx = dx * -1
   end if
   if spritey(1) <= 0 or spritey(1) >= graphheight -1 then
      dy = dy * -1
   end if
   if spritecollide(0,1) then
      dy = dy * -1
      print "bump"
   end if
   spritemove 1, dx, dy
   pause .05
end while
```

Program 64: Sprite Collision



Sample Output 64: Sprite Collision

New Concept	**spritecollide** (*spritenumber1, spritenumber2*) This function returns true of the two sprites collide with or overlap each other.

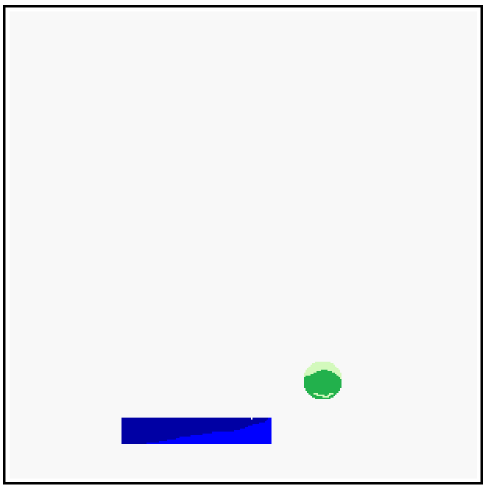
Big Program	The "Big Program" for this chapter uses sprites and sounds to create a paddle ball game.

```
# sprite_paddleball.kbs

color white
rect 0, 0, graphwidth, graphheight

```

```
spritedim 2

spriteload 1, "greenball.png"
spriteplace 1, 100,100
spriteshow 1
spriteload 0, "paddle.png"
spriteplace 0, 100,270
spriteshow 0

dx = rand * .5 + .25
dy = rand * .5 + .25

bounces = 0

while spritey(1) < graphheight -1
   k = key
   if chr(k) = "K" then
      spritemove 0, 20, 0
   end if
   if chr(k) = "J" then
      spritemove 0, -20, 0
   end if
   if spritecollide(0,1) then
      # bounce back ans speed up
      dy = dy * -1
      dx = dx * 1.1
      bounces = bounces + 1
      wavstop
      wavplay
"96633__CGEffex__Ricochet_metal5.wav"
      # move sprite away from paddle
      while spritecollide(0,1)
         spritemove 1, dx, dy
      end while
   end if
   if spritex(1) <=0 or spritex(1) >= graphwidth
```

```
-1 then
      dx = dx * -1
      wavstop
      wavplay "4359__NoiseCollector__PongBlipF4.wav"
   end if
   if spritey(1) <= 0 then
      dy = dy * -1
      wavstop
      wavplay "4361__NoiseCollector__pongblipA_3.wav"
   end if
   spritemove 1, dx, dy
end while

print "You bounced the ball " + bounces + " times."
```

Program 65: Paddleball with Sprites



Sample Output 65: Paddleball with Sprites

Chapter 13: Arrays – Collections of Information.

We have used simple string and numeric variables in many programs, but they can only contain one value at a time. Often we need to work with collections or lists of values. We can do this with either one-dimensioned or two-dimensioned arrays. This chapter will show you how to create, initialize, use, and re-size arrays.

One-Dimensional Arrays of Numbers:

A one-dimensional array allows us to create a list in memory and to access the items in that list by a numeric address (called an index). Arrays can be either numeric or string depending on the type of variable used in the *dim* statement.

```
# numeric1d.kbs

dim a(10)

a[0] = 100
a[1] = 200
a[3] = a[1] + a[2]

input "Enter a number", a[9]
a[8] = a[9] - a[3]

for t = 0 to 9
    print "a[" + t + "] = " + a[t]
next t
```

Program 66: One-dimensional Numeric Array

```
Enter a number63
a[0] = 100
a[1] = 200
a[2] = 0
a[3] = 200
a[4] = 0
a[5] = 0
a[6] = 0
a[7] = 0
a[8] = -137
a[9] = 63
```

Sample Output 66: One-dimensional Numeric Array

New Concept

```
dim variable(items)
dim variable$(items)
dim variable(rows, columns)
dim variable$(rows, columns)
```

The **dim** statement creates an array in the computer's memory the size that was specified in the parenthesis. Sizes (items, rows, and columns) must be integer values greater than one (1).

The **dim** statement will initialize the elements in the new array with either zero (0) if numeric or the empty string (“”), depending on the type of variable.

New Concept

```
variable[index]
variable[rowindex, columnindex]
variable$[index]
variable$[rowindex, columnindex]
```

You can use an array reference (variable with index(s) in square brackets) in your program almost anywhere you can use a simple variable. The index or indexes must be integer values between zero (0) and one less than the size used in the *dim* statement.

It may be confusing, but BASIC-256 uses zero (0) for the first element in an array and the last element has an index one less than the size. Computer people call this a zero-indexed array.

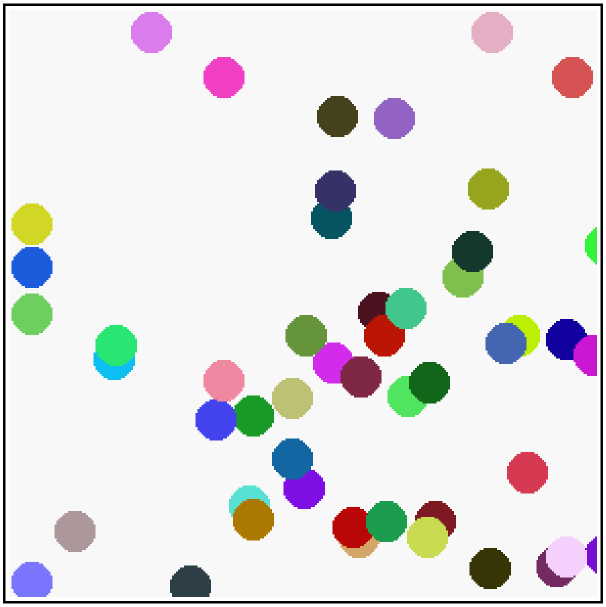
We can use arrays of numbers to draw many balls bouncing on the screen at once. Program 66 uses 5 arrays to store the location of each of the balls, it's direction, and color. Loops are then used to initialize the arrays and to animate the balls. This program also uses the *rgb()* function to calculate and save the color values for each of the balls.

```
# manyballbounce.kbs
fastgraphics

r = 10  # size of ball
balls = 50  # number of balls

dim x(balls)
dim y(balls)
dim dx(balls)
dim dy(balls)
dim colors(balls)

```

```
for b = 0 to balls-1
   # starting position of balls
   x[b] = 0
   y[b] = 0
   # speed in x and y directions
   dx[b] = rand * r + 2
   dy[b] = rand * r + 2
   # each ball has it's own color
   colors[b] = rgb(rand*256, rand*256, rand*256)
next b

color green
rect 0,0,300,300

while true
   # erase screen
   clg
   # now position and draw the balls
   for b = 0 to balls -1
      x[b] = x[b] + dx[b]
      y[b] = y[b] + dy[b]
      # if off the edges turn the ball around
      if x[b] < 0 or x[b] > 300 then
         dx[b] = dx[b] * -1
      end if
      # if off the top of bottom turn the ball around
      if y[b] < 0 or y[b] > 300 then
         dy[b] = dy[b] * -1
      end if
      # draw new ball
      color colors[b]
      circle x[b],y[b],r
   next b
   # update the display
   refresh
```

```
      pause .05
   end while
```

Program 67: Bounce Many Balls



Sample Output 67: Bounce Many Balls

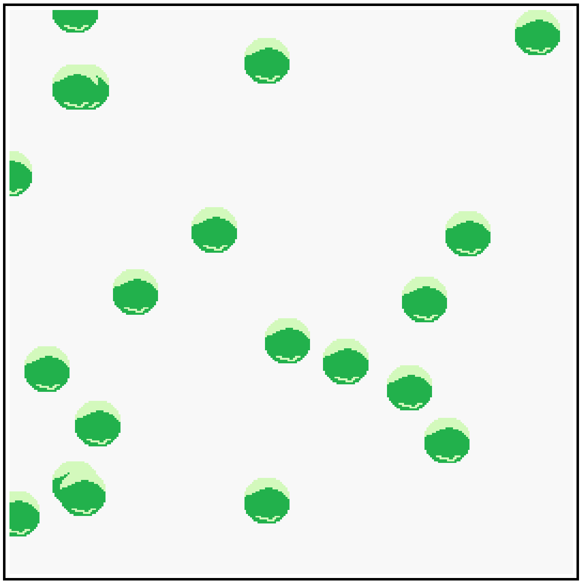
New Concept	**rgb**(*redexp, greenexp, blueexp*) The **rgb** function returns a single number that represents a color expressed by the three values. Remember that color component values have the range from 0 to 255.

Another example of a ball bouncing can be seen in Program 68. This second example uses sprites and two arrays to keep track of the direction each sprite is moving.

```
#manyballsprite.kbs

# another way to bounce many balls using sprites

fastgraphics
color white
rect 0, 0, graphwidth, graphheight

n = 20
spritedim n

dim dx(n)
dim dy(n)

for b = 0 to n-1
   spriteload b, "greenball.png"
   spriteplace b,graphwidth/2,graphheight/2
   spriteshow b
   dx[b] = rand * 5 + 2
   dy[b] = rand * 5 + 2
next b

while true
   for b = 0 to n-1
      if spritex(b) <=0 or spritex(b) >= graphwidth -1 then
         dx[b] = dx[b] * -1
      end if
      if spritey(b) <= 0 or spritey(b) >= graphheight -1 then
         dy[b] = dy[b] * -1
      end if
      spritemove b, dx[b], dy[b]
   next b
   refresh
```

```
34     end while
```

Program 68: Bounce Many Balls Using Sprites



Sample Output 68: Bounce Many Balls Using Sprites

Arrays of Strings:

Arrays can also be used to store string values. To create a string array use a string variable in the *dim* statement. All of the rules about numeric arrays apply to a string array except the data type is different. You can see the use of a string array in Program 69.

```
# listoffriends.kbs
print "make a list of my friends"
input "how many friends do you have?", n

dim names$(n)

for i = 0 to n-1
    input "enter friend name ?", names$[i]
next i

cls
print "my friends"
for i = 0 to n-1
    print "friend number ";
    print i + 1;
    print " is " + names$[i]
next i
```

Program 69: List of My Friends

```
make a list of my friends
how many friends do you have?3
enter friend name ?Bill
enter friend name ?Ken
enter friend name ?Sam
 - screen clears -
my friends
friend number 1 is Bill
friend number 2 is Ken
friend number 3 is Sam
```

Sample Output 69: List of My Friends

Assigning Arrays:

We have seen the use of the curly brackets ({}) to play music, draw polygons, and define stamps. The curly brackets can also be used to assign an entire array with custom values.

```
# arrayassign.kbs
dim number(3)
dim name$(3)

number = {1, 2, 3}
name$ = {"Bob", "Jim", "Susan"}

for i = 0 to 2
   print number[i] + " " + name$[i]
next i
```

Program 70: Assigning an Array With a List

```
1 Bob
2 Jim
3 Susan
```

Sample Output 70: Assigning an Array With a List

New Concept	*array* = {*value0*, *value1*, … } *array$* = {*value0*, *value1*, … } An array may be assigned values (starting with index 0) from a list of values enclosed in curly braces. This works for numeric and string arrays.

Sound and Arrays:

In Chapter 3 we saw how to use a list of frequencies and durations (enclosed in curly braces) to play multiple sounds at once. The sound statement will also accept a list of frequencies and durations from an array. The array should have an even number of elements; the frequencies should be stored in element 0, 2, 4, ...; and the durations should be in elements 1, 3, 5, ...

The sample (Program 71) below uses a simple linear formula to make a fun sonic chirp.

```
# spacechirp.kbs

# even values 0,2,4... - frequency
# odd values 1,3,5... - duration

# chirp starts at 100hz and increases by 40 for each of the 50 total sounds in list, duration is always 10

dim a(100)
for i = 0 to 98 step 2
    a[i] = i * 40 + 100
    a[i+1] = 10
next i
sound a
```

Program 71: Space Chirp Sound

Explore	What kind of crazy sounds can you program. Experiment with the formulas you use to change the frequencies and durations.

Graphics and Arrays:

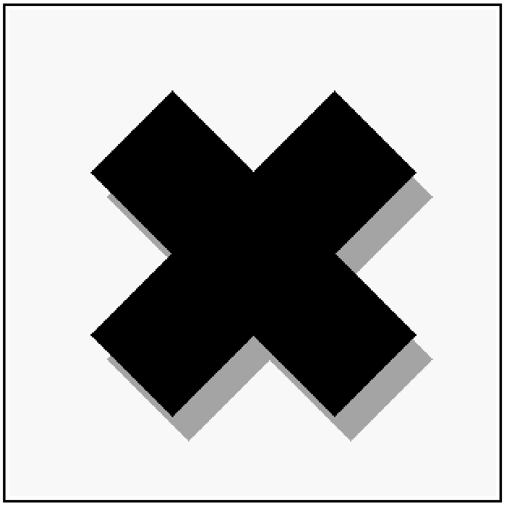
In Chapter 8 we also saw the use of lists for creating polygons and stamps. Arrays may also be used to draw stamps and polygons. This may help simplify your code by allowing the same stamp or polygon to be defined once, stored in an array, and used in various places in your program.

In an array used for stamps and polygons, the even elements (0, 2, 4, ...) contain the x value for each of the points and the odd element (1, 3, 5, ...) contain the y value for the points. The array will have two values for each point in the shape.

In Program 72 we will use the stamp from the mouse chapter to draw a big X with a shadow. This is accomplished by stamping a gray shape shifted in the direction of the desired shadow and then stamping the object that is projecting the shadow.

```
# shadowstamp.kbs

dim xmark(24)
xmark = {-1, -2, 0, -1, 1, -2, 2, -1, 1, 0, 2, 1,
1, 2, 0, 1, -1, 2, -2, 1, -1, 0, -2, -1}

clg
color grey
stamp 160,165,50,xmark
color black
stamp 150,150,50,xmark
```

Program 72: Shadow Stamp

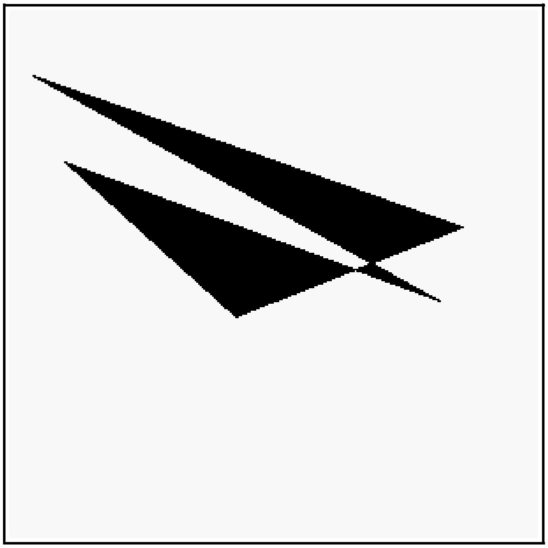


Sample Output 72: Shadow Stamp

Arrays can also be used to create stamps or polygons mathematically. In Program 73 we create an array with 10 elements (5 points) and assign random locations to each of the points to draw random polygons. BASIC-256 will fill the shape the best it can but when lines cross, as you will see, the fill sometimes leaves gaps and holes.

```
# mathpoly.kbs

dim shape(10)

for t = 0 to 8 step 2
    x = 300 * rand
    y = 300 * rand
    shape[t] = x
    shape[t+1] = y
next t

clg
color black
poly shape
```

Program 73: Randomly Create a Polygon



Sample Output 73: Randomly Create a Polygon

Advanced - Two Dimensional Arrays:

So far in this chapter we have explored arrays as lists of numbers or strings. We call these simple arrays one-dimensional arrays because they resemble a line of values. Arrays may also be created with two-dimensions representing rows and columns of data. Program 74 uses both one and two-dimensional arrays to calculate student's average grade.

```
# grades.kbs
# calculate average grades for each student
# and whole class

nstudents = 3 # number of students
nscores = 4 # number of scores per student

dim students$(nstudents)

dim grades(nstudents, nscores)
# store the scores as columns and the students as rows
# first student
students$[0] = "Jim"
grades[0,0] = 90
grades[0,1] = 92
grades[0,2] = 81
grades[0,3] = 55
# second student
students$[1] = "Sue"
grades[1,0] = 66
grades[1,1] = 99
grades[1,2] = 98
grades[1,3] = 88
# third student
```

```
students$[2] = "Tony"
grades[2,0] = 79
grades[2,1] = 81
grades[2,2] = 87
grades[2,3] = 73

total = 0
for row = 0 to nstudents-1
   studenttotal = 0
   for column = 0 to nscores-1
      studenttotal = studenttotal + grades[row, column]
      total = total + grades[row, column]
   next column
   print students$[row] + "'s average is ";
   print studenttotal / nscores
next row
print "class average is ";
print total / (nscores * nstudents)

end
```

Program 74: Grade Calculator

```
Jim's average is 79.5
Sue's average is 87.75
Tony's average is 80
class average is 82.416667
```

Sample Output 74: Grade Calculator

Really Advanced - Array Sizes:

Sometimes we need to create programming code that would work with an array of any size. If you specify a question mark as a index, row, or column number in the square bracket reference of an array BASIC-256 will return the dimensioned size. In Program 70 we modified Program 67 to display the array regardless of it's length. You will see the special [?] used on line 16 to return the current size of the array.

```
1     # size.kbs
2     dim number(3)
3     number = {77, 55, 33}
4     print "before"
5     gosub shownumberarray
6
7     # create a new element on the end
8     redim number(4)
9     number[3] = 22
10    print "after"
11    gosub shownumberarray
12    #
13    end
14    #
15    shownumberarray:
16    for i = 0 to number[?] - 1
17        print i + " " + number[i]
18    next i
19    return
```

Program 75: Get Array Size

```
before
0 77
1 55
2 33
after
0 77
1 55
2 33
3 22
```

Sample Output 75: Get Array Size

New Concept

```
array[?]
array$[?]
array[?,]
array$[?,]
array[,?]
array$[,?]
```

The [?] reference returns the length of a one-dimensional array or the total number of elements (rows * column) in a two-dimensional array. The [?,] reference returns the number of rows and the [,?] reference returns the number of columns of a two dimensional array.

Really Really Advanced - Resizing Arrays:

BASIC-256 will also allow you to re-dimension an existing array. The *redim* statement will allow you to re-size an array and will preserve the existing data. If the new array is larger, the new elements will be filled with zero (0) or the empty string (“”). If the new array is smaller, the values beyond the new size will be truncated (cut off).

```
# redim.kbs
dim number(3)
number = {77, 55, 33}
# create a new element on the end
redim number(4)
number[3] = 22
#
for i = 0 to 3
    print i + " " + number[i]
next i
```

Program 76: Re-Dimension an Array

```
0 77
1 55
2 33
3 22
```

Sample Output 76: Re-Dimension an Array

New Concept

redim *variable*(*items*)
redim *variable$*(*items*)
redim *variable*(*rows*, *columns*)
redim *variable$*(*rows*, *columns*)

The **redim** statement re-sizes an array in the computer's memory. Data previously stored in the array will be kept, if it fits.

When resizing two-dimensional arrays the values are copied in a linear manner. Data may be shifted in an unwanted manner if you are changing the number of columns.

Big Program

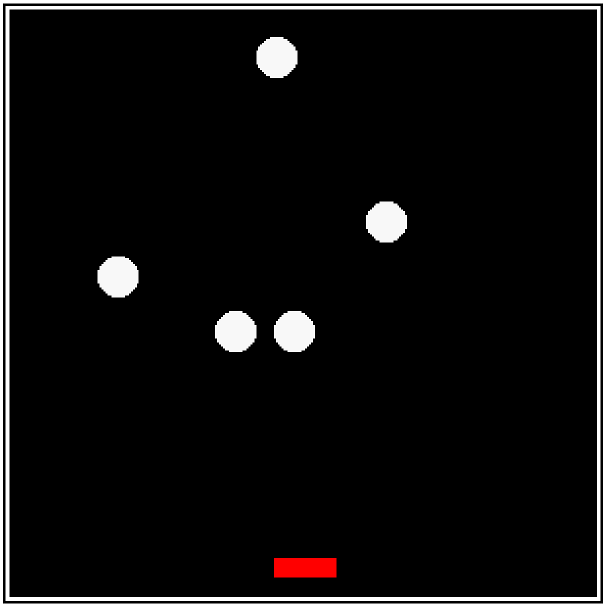
The “Big Program” for this chapter uses three numeric arrays to store the positions and speed of falling space debris. You are not playing pong but you are trying to avoid all of them to score points.

```
# spacewarp.kbs
# The falling space debris game

balln = 5 # number of balls
dim ballx(balln)    # arrays to hold ball position and speed
dim bally(balln)
dim ballspeed(balln)
ballr = 10      # radius of balls

minx = ballr   # minimum x value for balls
maxx = graphwidth - ballr     # maximum x value for balls
miny = ballr   # minimum y value for balls
maxy = graphheight - ballr    # maximum y value for balls
score = 0       # initial score
playerw = 30   # width of player
playerm = 10   # size of player move
playerh = 10   # height of player
playerx = (graphwidth - playerw)/2 # initial position of player
keyj = asc("J")     # value for the 'j' key
keyk = asc("K")     # value for the 'k' key
keyq = asc("Q")     # value for the 'q' key
growpercent = .20   # random growth - bigger is faster
speed = .15    # the lower the faster
```

```
print "spacewarp - use j and k keys to avoid the falling space debris"
print "q to quit"

fastgraphics

# setup initial ball positions and speed
for n = 0 to balln-1
   gosub setupball
next n

more = true
while more
   pause speed
   score = score + 1

   # clear screen
   color black
   rect 0, 0, graphwidth, graphheight

   # draw balls and check for collission
   color white
   for n = 0 to balln-1
      bally[n] = bally[n] + ballspeed[n]
      if bally[n] > maxy then gosub setupball
      circle ballx[n], bally[n], ballr
      if ((bally[n]) >= (maxy-playerh-ballr)) and ((ballx[n]+ballr) >= playerx) and ((ballx[n]-ballr) <= (playerx+playerw)) then more = false
   next n

   # draw player
   color red
   rect playerx, maxy - playerh, playerw, playerh
   refresh

```

```
   # make player bigger
   if (rand<growpercent) then playerw = playerw + 1

   # get player key and move if key pressed
   k = key
   if k = keyj then playerx = playerx - playerm
   if k = keyk then playerx = playerx + playerm
   if k = keyq then more = false

   # keep player on screen
   if playerx < 0 then playerx = 0
   if playerx > graphwidth - playerw then playerx = graphwidth - playerw

end while

print "score " + string(score)
print "you died."
end

setupball:
bally[n] = miny
ballx[n] = int(rand * (maxx-minx)) + minx
ballspeed[n] = int(rand * (2*ballr)) + 1
return
```

Program 77: Big Program - Space Warp Game



Sample Output 77: Big Program - Space Warp Game

Chapter 14: Mathematics – More Fun With Numbers.

In this chapter we will look at some additional mathematical operators and functions that work with numbers. Topics will be broken down into four sections: 1) new operators; 2) new integer functions, 3) new floating point functions, and 4) trigonometric functions.

New Operators:

In addition to the basic mathematical operations we have been using since the first chapter, there are three more operators in BASIC-256. Operations similar to these three operations exist in most computer languages. They are the operations of modulo, integer division, and power.

Operation	Operator	Description
Modulo	%	Return the remainder of an integer division.
Integer Division	\	Return the whole number of times one integer can be divided into another.
Power	^	Raise a number to the power of another number.

Modulo Operator:

The modulo operation returns the remainder part of integer division. When you do long division with whole numbers, you get a remainder –

that is the same as the modulo.

```
# mod.kbs
input "enter a number ", n
if n % 2 = 0 then print "divisible by 2"
if n % 3 = 0 then print "divisible by 3"
if n % 5 = 0 then print "divisible by 5"
if n % 7 = 0 then print "divisible by 7"
end
```

Program 78: The Modulo Operator

```
enter a number 10
divisible by 2
divisible by 5
```

Sample Output 78: The Modulo Operator

New Concept

`expression1 % expression2`

The Modulo (%) operator performs integer division of *expression1* divided by *expression2* and returns the remainder of that process.

If one or both of the expressions are not integer values (whole numbers) they will be converted to an integer value by truncating the decimal (like in the *int()* function) portion before the operation is performed.

You might not think it, but the modulo operator (%) is used quite often by programmers. Two common uses are; 1) to test if one number divides into another (Program 78) and 2) to limit a number to a specific range (Program 79).

```
# moveballmod.kbs
# rewrite of moveball.kbs using the modulo operator to wrap the ball around the screen

print "use i for up, j for left, k for right, m for down, q to quit"

fastgraphics
clg
ballradius = 20

# position of the ball
# start in the center of the screen
x = graphwidth /2
y = graphheight / 2

# draw the ball initially on the screen
gosub drawball

# loop and wait for the user to press a key
while true
   k = key
   if k = asc("I") then
  # y can go negative, + graphheight keeps it positive
      y = (y - ballradius + graphheight) % graphheight
      gosub drawball
   end if
   if k = asc("J") then
      x = (x - ballradius + graphwidth) % graphwidth
      gosub drawball
   end if
   if k = asc("K") then
```

```
      x = (x + ballradius) % graphwidth
      gosub drawball
   end if
   if k = asc("M") then
      y = (y + ballradius) % graphheight
      gosub drawball
   end if
   if k = asc("Q") then end
end while

drawball:
color white
rect 0, 0, graphwidth, graphheight
color red
circle x, y, ballradius
refresh
return
```

Program 79: Move Ball - Use Modulo to Keep on Screen

Integer Division Operator:

The Integer Division (\) operator does normal division but it works only with integers (whole numbers) and returns an integer value. As an example, 13 divided by 4 is 3 remainder 1 – so the result of the integer division is 3.

```
# integerdivision.kbs
input "dividend ", dividend
input "divisor ", divisor
print dividend + " / " + divisor + " is ";
print dividend \ divisor;
print "r";
print dividend % divisor;
```

Program 80: Check Your Long Division

```
dividend 43
divisor 6
43 / 6 is 7r1
```

Sample Output 80: Check Your Long Division

New Concept

expression1 **** *expression2*

The Integer Division (\\) operator performs division of *expression1 / expression2* and returns the whole number of times *expression1* goes into *expression2*.

If one or both of the expressions are not integer values (whole numbers), they will be converted to an integer value by truncating the decimal (like in the *int()* function) portion before the operation is performed.

Power Operator:

The power operator will raise one number to the power of another number.

```
# power.kbs
for t = 0 to 16
    print "2 ^ " + t + " = ";
    print 2 ^ t
next t
```

Program 81: The Powers of Two

```
2 ^ 0 = 1
2 ^ 1 = 2
2 ^ 2 = 4
2 ^ 3 = 8
2 ^ 4 = 16
2 ^ 5 = 32
2 ^ 6 = 64
2 ^ 7 = 128
2 ^ 8 = 256
2 ^ 9 = 512
2 ^ 10 = 1024
2 ^ 11 = 2048
2 ^ 12 = 4096
2 ^ 13 = 8192
2 ^ 14 = 16384
2 ^ 15 = 32768
2 ^ 16 = 65536
```

Sample Output 81: The Powers of Two

New Concept

expression1 **^** *expression2*

The Power (^) operator raises *expression1* to the *expression2* power.

The mathematical expression $a=b^c$ would be written in BASIC-256 as a = b ^ c.

New Integer Functions:

The three new integer functions in this chapter all deal with how to convert strings and floating point numbers to integer values. All three functions handle the decimal part of the conversion differently.

In the *int()* function the decimal part is just thrown away, this has the same effect of subtracting the decimal part from positive numbers and adding it to negative numbers. This can cause troubles if we are trying to round and there are numbers less than zero (0).

The *ceil()* and *floor()* functions sort of fix the problem with *int()*. Ceil() always adds enough to every floating point number to bring it up to the next whole number while floor(0) always subtracts enough to bring the floating point number down to the closest integer.

We have been taught to round a number by simply adding 0.5 and drop the decimal part. If we use the int() function, it will work for positive numbers but not for negative numbers. In BASIC-256 to round we should always use a formula like $a = floor(b+0.5)$.

	Function	Description
New Concept	**int**(*expression*)	Convert an expression (string, integer, or decimal value) to an integer (whole number). When converting a floating point value the decimal part is truncated (ignored). If a string does not contain a number a zero is returned.
	ceil(*expression*)	Converts a floating point value to the next highest integer value.

	Function	Description
New Concept	**floor** (*expression*)	Converts a floating point expression to the next lowers integer value. You should use this function for rounding $a=floor(b+0.5)$.

```
# intceilfloor.kbs
for t = 1 to 10
   n = rand * 100 - 50
   print n;
   print "  int=" + int(n);
   print "  ceil=" + ceil(n);
   print "  floor=" + floor(n)
next t
```

Program 82: Difference Between Int, Ceiling, and Floor

```
-46.850173  int=-46  ceil=-46  floor=-47
-43.071987  int=-43  ceil=-43  floor=-44
23.380133  int=23  ceil=24  floor=23
4.620722  int=4  ceil=5  floor=4
3.413543  int=3  ceil=4  floor=3
-26.608505  int=-26  ceil=-26  floor=-27
-18.813465  int=-18  ceil=-18  floor=-19
7.096065  int=7  ceil=8  floor=7
23.482759  int=23  ceil=24  floor=23
-45.463169  int=-45  ceil=-45  floor=-46
```

Sample Output 82: Difference Between Int, Ceiling, and Floor

New Floating Point Functions:

The mathematical functions that wrap up this chapter are ones you may need to use to write some programs. In the vast majority of programs these functions will not be needed.

	Function	**Description**
New Concept	**`float`**`(`*`expression`*`)`	Convert expression (string, integer, or decimal value) to a decimal value. Useful in changing strings to numbers. If a string does not contain a number a zero is returned.
	`abs``(`*`expression`*`)`	Converts a floating point or integer expression to an absolute value.
	`log``(`*`expression`*`)`	Returns the natural logarithm (base *e*) of a number.
	`log10``(`*`expression`*`)`	Returns the base 10 logarithm of a number.

Advanced - Trigonometric Functions:

Trigonometry is the study of angles and measurement. BASIC-256 includes support for the common trigonometric functions. Angular measure is done in radians (0-2p). If you are using degrees (0-360) in your programs you must convert to use the “trig” functions.

New Concept	Function	Description
	cos(*expression*)	Return the cosine of an angle.
	sin(*expression*)	Return the sine of an angle.
	tan(*expression*)	Return the tangent of an angle.
	degrees(*expression*)	Convert Radians (0 – 2π) to Degrees (0-360).
	radians(*expression*)	Convert Degrees (0-360) to Radians (0 – 2π).
	acos(*expression*)	Return the inverse cosine.
	asin(*expression*)	Return the inverse sine.
	atan(*expression*)	Return the inverse tangent.

The discussion of the first three functions will refer to the sides of a right triangle. Illustration 20 shows one of these with it's sides and angles labeled.

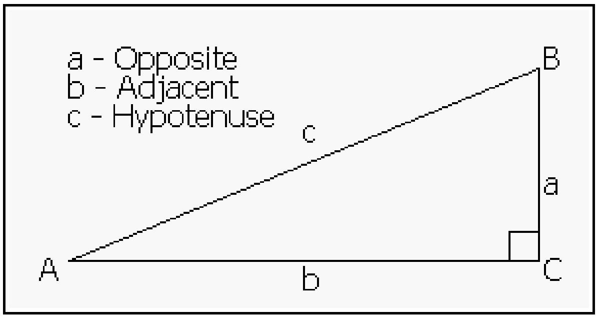


Illustration 20: Right Triangle

Cosine:

A cosine is the ratio of the length of the adjacent leg over the length of the hypotenuse $\cos A=\frac{b}{c}$. The cosine repeats itself every 2π radians and has a range from -1 to 1. Illustration 20 graphs a cosine wave from 0 to 2π radians.

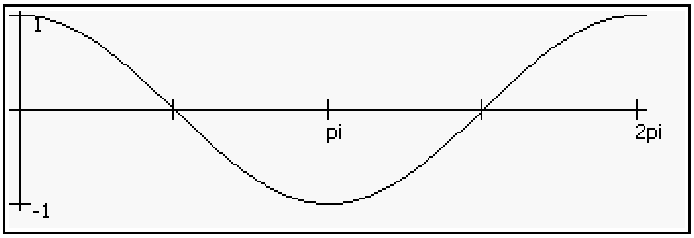


Illustration 21: Cos() Function

Sine:

The sine is the ratio of the adjacent side over the hypotenuse $\sin A=\frac{a}{c}$. The sine repeats itself every 2π radians and has a range from -1 to 1. You have seen diagrams of sine waves in Chapter 3 as music was discussed.

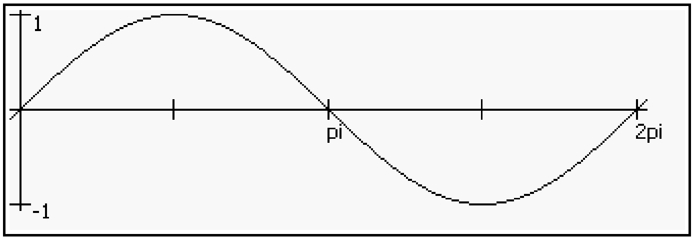


Illustration 22: Sin() Function

Tangent:

The tangent is the ratio of the adjacent side over the opposite side $\tan A = \frac{a}{b}$. The sine repeats itself every π radians and has a range from -∞ to ∞. The tangent has this range because when the angle gets very small the length of the opposite side becomes very small.

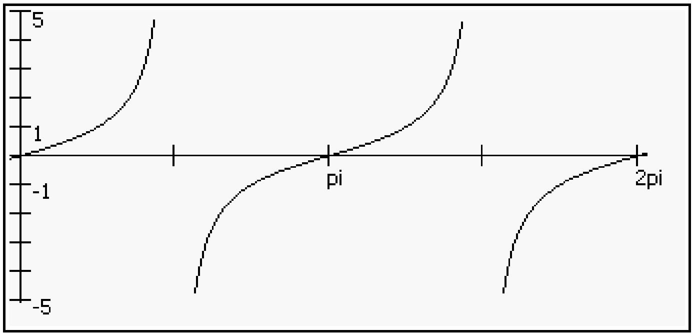


Illustration 23: Tan() Function

Degrees Function:

The **degrees**() function does the quick mathematical calculation to convert an angle in radians to an angle in degrees. The formula used is $degrees = radians / 2\pi * 360$.

Radians Function:

The **radians**() function will convert degrees to radians using the formula $radians = degrees / 360 * 2\pi$. Remember all of the trigonometric functions in BASIC-256 use radians and not degrees to measure angles.

Inverse Cosine:

The inverse cosine function **acos**() will return an angle measurement in radians for the specified cosine value. This function performs the opposite of the *cos()* function.

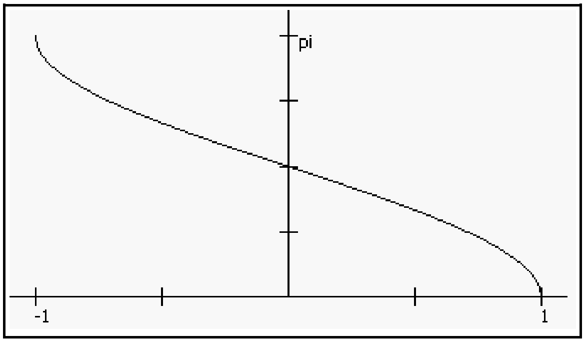


Illustration 24: Acos() Function

Inverse Sine:

The inverse sine function **asin**() will return an angle measurement in radians for the specified sine value. This function performs the opposite of the sin*()* function.

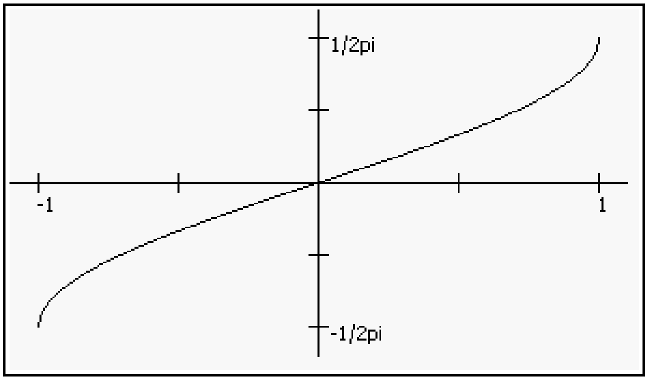


Illustration 25: Asin() Function

Inverse Tangent:

The inverse tangent function **atan**() will return an angle measurement in radians for the specified tangent value. This function performs the opposite of the **tan**() function.

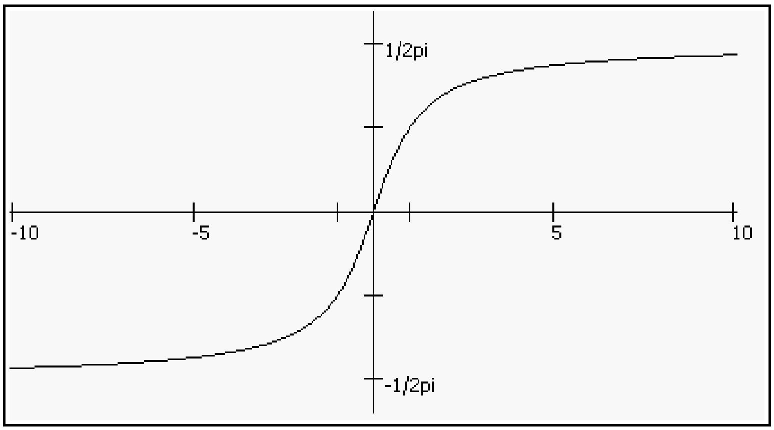


Illustration 26: Atan() Function

Big Program	The big program this chapter allows the user to enter two positive whole numbers and then performs long division. This program used logarithms to calculate how long the numbers are, modulo and integer division to get the individual digits, and is generally a very complex program. Don't be scared or put off if you don't understand exactly how it works, yet.

```
# longdivision.kbs
# show graphically the long division of two positive integers

input "dividend? ", b
input "divisor? ", a

originx = 100
originy = 20
height = 12
width = 9
margin = 2

b = int(abs(b))
a = int(abs(a))

clg

# display original problem
row = 0
col = -1
number = a
underline = false
gosub drawrightnumber
row = 0
col = 0
number = b
```

```
gosub drawleftnumber
line originx - margin, originy, originx + (width * 11), originy
line originx - margin, originy, originx - margin, originy + height

# calculate how many digits are in the dividend
lb = ceil(log10(abs(b)))

r = 0
bottomrow = 0    ## row for bottom calculation display

# loop through all of the digits from the left to the right
for tb = lb-1 to 0 step -1
   # drop down the next digit to running remainder and remove from dividend
   r = r * 10
   r = r + (b \ (10 ^ tb))
   b = b % (10 ^ tb)
   # display running remainder
   row = bottomrow
   bottomrow = bottomrow + 1
   col = lb - tb - 1
   number = r
   underline = false
   gosub drawrightnumber
   # calculate new digit in answer and display
   digit = r \ a
   row = -1
   col = lb - tb - 1
   gosub drawdigit
   # calculate quantity to remove from running and display
   number = digit * a
   r = r - number
```

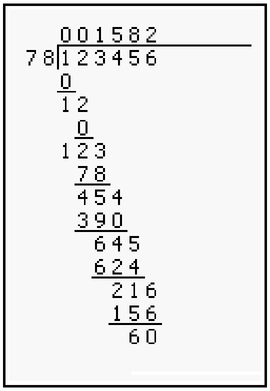
```
   col = lb - tb - 1
   row = bottomrow
   bottomrow = bottomrow + 1
   underline = true
   gosub drawrightnumber
next tb
#
# print remainder at bottom
row = bottomrow
col = lb - 1
number = r
underline = false
gosub drawrightnumber
end

drawdigit:
# pass row and col convert to x y
text col * width + originx, row * height + originy, digit
if underline then
   line col * width + originx - margin, (row + 1) * height + originy, (col + 1) * width + originx - margin, (row + 1) * height + originy
end if
return

drawleftnumber:
# pass start row, col, and number - from left column
if number < 10 then
   digit = number
   gosub drawdigit
else
   lnumber = ceil(log10(abs(number)))
   for tnumber = lnumber-1 to 0 step -1
      digit = (number \ (10 ^ tnumber)) % 10
      gosub drawdigit
```

```
            col = col + 1
         next tnumber
      endif
      return

      drawrightnumber:
      # pass start row, col, and number - from right
      column
      if number < 10 then
         digit = number
         gosub drawdigit
      else
         lnumber = ceil(log10(abs(number)))
         for tnumber = 0 to lnumber - 1
            digit = (number \ (10 ^ tnumber)) % 10
            gosub drawdigit
            col = col - 1
         next tnumber
      endif
      return
```

Program 83: Big Program - Long Division

```
dividend? 123456
divisor? 78
```

Sample Output 83: Big Program - Long Division (one)



Sample Output 83: Big Program - Long Division

Chapter 15: Working with Strings.

We have used strings to store non-numeric information, build output, and capture input. We have also seen, in Chapter 11, using the Unicode values of single characters to build strings.

This chapter shows several new functions that will allow you to manipulate string values.

The String Functions:

BASIC-256 includes eight common functions for the manipulation of strings. Table 7 includes a summary of them.

Function	Description
string(*expression*)	Convert expression (string, integer, or decimal value) to a string value.
length(*string*)	Returns the length of a string.
left(*string, length*)	Returns a string of length characters starting from the left.
right(*string, length*)	Returns a string of length characters starting from the right.
mid(*string, start, length*)	Returns a string of length characters starting from the middle of a string.
upper(*expression*)	Returns an upper case string.
lower(*expression*)	Returns a lower case string.
instr(*haystack, needle*)	Searches the string “haystack” for the “needle” and returns it's location.

Table 7: Summary of String Functions

String() Function:

The **string**() function will take an expression of any format and will return a string. This function is a convenient way to convert an integer or floating point number into characters so that it may be manipulated as a string.

```
# string.kbs
a$ = string(10 + 13)
print a$
b$ = string(2 * pi)
print b$
```

Program 84: The String Function

```
23
6.283185
```

Sample Output 84: The String Function

New Concept	**string**(*expression*) Convert expression (string, integer, or decimal value) to a string value.

Length() Function:

The *length()* function will take a string expression and return it's length in characters (or letters).

```
# length.kbs
# prints 6, 0, and 17
print length("Hello.")
print length("")
print length("Programming Rulz!")
```

Program 85: The Length Function

```
6
0
17
```

Sample Output 85: The Length Function

New Concept	**length**(*expression*) Returns the length of the string expression. Will return zero (0) for the empty string “”.

Left(), Right() and Mid() Functions:

The **left**(), **right**(), and **mid**() functions will extract sub-strings (or parts of a string) from a larger string.

```
# leftrightmid.kbs
a$ = "abcdefghijklm"
# prints "abcd"
print left(a$,4)
# prints "lm"
print right(a$,2)
# prints "def" and "jklm"
print mid(a$,4,3)
print mid(a$,10,9)
```

Program 86: The Left, Right, and Mid Functions

```
abcd
kl
def
jklm
```

Sample Output 86: The Left, Right, and Mid Functions

New Concept	**left**(*string, length*) Return a sub-string from the left end of a string. If length is equal or greater then the actual length of the string the entire string will be returned.

New Concept	**right**(*string, length*) Return a sub-string from the right end of a string. If length is equal or greater then the actual length of the string the entire string will be returned.

New Concept

mid(*string, start, length*)

Return a sub-string of specified length from somewhere on the middle of a string. The start parameter specifies where the sub-string begins (1 = beginning of string).

Upper() and Lower() Functions:

The **upper**() and **lower**() functions simply will return a string of upper case or lower case letters. These functions are especially helpful when you are trying to perform a comparison of two strings and you do not care what case they actually are.

```
# upperlower.kbs
a$ = "Hello."
# prints "hello."
print lower(a$)
# prints "HELLO."
print upper(a$)
```

Program 87: The Upper and Lower Functions

```
hello.
HELLO.
```

Sample Output 87: The Upper and Lower Functions

New Concept

lower(*string*)
upper(*string*)

Returns an all upper case or lower case copy of the string expression. Non-alphabetic characters will not be modified.

Instr() Function:

The **instr**() function searches a string for the first occurrence of another string. The return value is the location in the big string of the smaller string. If the substring is not found then the function will return a zero (0).

```
# instr.kbs
a$ = "abcdefghijklm"
# find location of "hi"
print instr(a$,"hi")
# find location of "bye"
print instr(a$,"bye")
```

Program 88: The Instr Function

```
8
0
```

Sample Output 88: The Instr Function

New Concept	**instr**(*haystack, needle*) Find the sub-string (*needle)* in another string expression (*haystack*). Return the character position of the start. If sub-string is not found return a zero (0).

Big Program	The decimal (base 10) numbering system that is most commonly used uses 10 different digits (0-9) to represent numbers. Imagine if you will what would have happened if there were only 5 digits (0-4) – the number 23 ($2*10^1+3*10^0$) would become 43 ($4*5^1+3*5^0$) to represent the same number of items. This type of transformation is called radix (or base) conversion. The computer internally does not understand base 10 numbers but converts everything to base 2 (binary) numbers to be stored in memory. The “Big Program” this chapter will convert a positive integer from any base 2 to 36 (where letters are used for the 11th - 26th digits) to any other base.

```
# radix.kbs
# convert a number from one base (2-36) to another

digits$ = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ"

message$ = "from base"
gosub getbase
frombase = base
```

```

input "number in base " + frombase + " >", number$
number$ = upper(number$)

# convert number to base 10 and store in n
n = 0
for i = 1 to length(number$)
   n = n * frombase
   n = n + instr(digits$, mid(number$, i, 1)) - 1
next i

message$ = "to base"
gosub getbase
tobase = base

# now build string in tobase
result$ = ""
while n <> 0
   result$ = mid(digits$, n % tobase + 1, 1) + result$
   n = n \ tobase
end while

print "in base " + tobase + " that number is " + result$
end

getbase: # get a base from 2 to 36
do
   input message$+"> ", base
until base >= 2 and base <= 36
return
```

Program 89: Big Program - Radix Conversion

```
from base> 10
number in base 10 >999
to base> 16
in base 16 that number is 3E7
```

Sample Output 89: Big Program - Radix Conversion

Chapter 16: Files – Storing Information For Later.

We have explored the computer's short term memory with variables and arrays but how do we store those values for later? There are many different techniques for long term data storage.

BASIC-256 supports writing and reading information from files on your hard disk. That process of input and output is often written as I/O.

This chapter will show you how to read values from a file and then write them for long term storage.

Reading Lines From a File:

Our first program using files is going to show you many of the statements and constants you will need to use to manipulate file data. There are several new statements and functions in this program.

```
#readlfile.kbs
input "file name>", fn$
if not exists(fn$) then
   print fn$ + " does not exist."
   end
end if
#
n = 0
open fn$
while not eof
    l$ = readline
    n = n + 1
```

```
        print n + " " + l$
    end while
    #
    print "the file " + fn$ + " is " + size + " bytes long."
    close
```

Program 90: Read Lines From a File

```
file name>test.txt
1 These are the times that
2 try men's souls.
3 - Thomas Paine
the file test.txt is 58 bytes long.
```

Sample Output 90: Read Lines From a File

New Concept

exist(*expression*)

Look on the computer for a file name specified by the string *expression*. Drive and path may be specified as part of the file name, but if they are omitted then the current working directory will be the search location.

Returns *true* if the file exists; else returns *false*.

New Concept

```
open expression
open (expression)
open filenumber, expression
open (filenumber, expression)
```

Open the file specified by the *expression* for reading and writing to the specified file number. If the file does not exist it will be created so that information may be added (see *write* and *writeline*). Be sure to execute the *close* statement when the program is finished with the file.

BASIC-256 may have a total of eight (8) files open 0 to 7. If no file number is specified then the file will be opened as file number zero (0).

New Concept

```
eof
eof()
eof(filenumber)
```

The **eof** function returns a value of *true* if we are at the end of the file for reading or *false* if there is still more data to be read.

If filenumber is not specified then file number zero (0) will be used.

New Concept	**readline** **readline**() **readline**(*filenumber*) Return a string containing the contents of an open file up to the end of the current line. If we are at the end of the file [*eof(filenumber) = true*] then this function will return the empty string (“”). If filenumber is not specified then file number zero (0) will be used.

New Concept	**size** **size**() **size**(*filenumber*) This function returns the length of an open file in bytes. If filenumber is not specified then file number zero (0) will be used.

New Concept	**close** **close**() **close** *filenumber* **close**(*filenumber*) The **close** statement will complete any pending I/O to the file and allow for another file to be opened with the same number. If filenumber is not specified then file number zero (0) will be used.

Writing Lines to a File:

In Program 90 we saw how to read lines from a file. The next two programs show different variations of how to write information to a file. In Program 91 we open and clear any data that may have been in the file to add our new lines and in Program 92 we append our new lines to the end (saving the previous data).

```
# resetwrite.kbs
open "resetwrite.dat"

print "enter a blank line to close file"

# clear file (reset) and start over
reset
repeat:
input ">", l$
if l$ <> "" then
   writeline l$
```

```
      goto repeat
   end if

   # go the the start and display contents
   seek 0
   k = 0
   while not eof()
      k = k + 1
      print k + " " + readline()
   end while

   close
   end
```

Program 91: Clear File and Write Lines

```
enter a blank line to close file
>this is some
>data, I am typing
>into the program.
>
1 this is some
2 data, I am typing
3 into the program.
```

Sample Output 91: Clear File and Write Lines

New Concept	**reset** *or* **reset**() *or* **reset** *filenumber* **reset**(*filenumber*) Clear any data in an open file and move the file pointer to the beginning. If filenumber is not specified then file number zero (0) will be used.

New Concept	**seek** *expression* **seek**(*expression*) **seek** *filenumber*,*expression* **seek** (*filenumber*,*expression*) Move the file pointer for the next read or write operation to a specific location in the file. To move the current pointer to the beginning of the file use the value zero (0). To seek to the end of a file use the **size**() function as the argument to the see statement. If filenumber is not specified then file number zero (0) will be used.

New Concept

writeline *expression*
writeline(*expression*)
writeline *filenumber*,*expression*
writeline (*filenumber*,*expression*)

Output the contents of the expression to an open file and then append an end of line mark to the data. The file pointer will be positioned at the end of the write so that the next write statement will directly follow.

If filenumber is not specified then file number zero (0) will be used.

```
# appendwrite.kbs
open "appendwrite.dat"

print "enter a blank line to close file"

# move file pointer to end of file and append
seek size()
repeat:
input ">", l$
if l$ <> "" then
   writeline l$
   goto repeat
end if

# move file pointer to beginning and show contents
seek 0
k = 0
while not eof()
   k = k + 1
   print k + " " + readline()
end while
```

```

close
end
```

Program 92: Append Lines to a File

```
enter a blank line to close file
>sed sed sed
>vim vim vim
>
1 bar bar bar
2 foo foo foo
3 grap grap grap
4 sed sed sed
5 vim vim vim
```

Sample Output 92: Append Lines to a File

Read() Function and Write Statement:

In the first three programs of this chapter we have discussed the **readline**() function and **writeline** statement. There are two other statements that will read and write a file. They are the **read**() function and **write** statement.

New Concept

```
read
read()
read(filenumber)
```

Read the next word or number (token) from a file. Tokens are delimited by spaces, tab characters, or end of lines. Multiple delimiters between tokens will be treated as one.

If filenumber is not specified then file number zero (0) will be used.

New Concept

```
write expression
write (expression)
write filenumber,expression
write (filenumber,expression)
```

Write the string expression to a file file. Do not add an end of line or a delimiter.

If filenumber is not specified then file number zero (0) will be used.

Big Program

This program uses a single text file to help us maintain a list of our friend's telephone numbers.

```
# phonelist.kbs
# add a phone number to the list and show
filename$ = "phonelist.txt"

```

```
print "phonelist.kbs - Manage your phone list."
do
   input "Add, List, Quit (a/l/q)?",action$
   if left(lower(action$),1) = "a" then gosub addrecord
   if left(lower(action$),1) = "l" then gosub listfile
until left(lower(action$),1) = "q"
end

listfile:
if exists(filename$) then
   # list the names and phone numbers in the file
   open filename$
   print "the file is " + size + " bytes long"
   while not eof
      # read next line from file and print it
      print readline
   end while
   close
else
   print "No phones on file.  Add first."
end if
return

addrecord:
input "Name to add?", name$
input "Phone to add", phone$
open filename$
# seek to the end of the file
seek size()
# we are at end of file - add new line
writeline name$ + ", " + phone$
close
return
```

Program 93: Big Program - Phone List

```
phonelist.kbs - Manage your phone list.
Add, List, Quit (a/l/q)?l
the file is 46 bytes long
jim, 555-5555
sam, 555-7777
doug, 555-3333
Add, List, Quit (a/l/q)?a
Name to add?ang
Phone to add555-0987
Add, List, Quit (a/l/q)?l
the file is 61 bytes long
jim, 555-5555
sam, 555-7777
doug, 555-3333
ang, 555-0987
Add, List, Quit (a/l/q)?q
```

Sample Output 93: Big Program - Phone List

Chapter 17: Stacks, Queues, Lists, and Sorting

This chapter introduces a few advanced topics that are commonly covered in the first Computer Science class at the University level. The first three topics (Stack, Queue, and Linked List) are very common ways that information is stored in a computer system. The last two are algorithms for sorting information.

Stack:

A stack is one of the common data structures used by programmers to do many tasks. A stack works like the “discard pile” when you play the card game “crazy-eights”. When you add a piece of data to a stack it is done on the top (called a “push”) and these items stack upon each other. When you want a piece of information you take the top one off the stack and reveal the next one down (called a “pop”). Illustration 27 shows a graphical example.

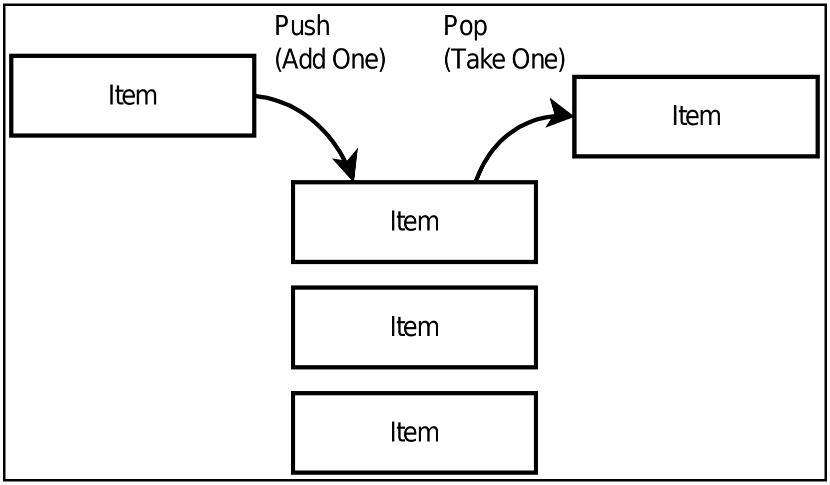


Illustration 27: What is a Stack

The operation of a stack can also be described as "last-in, first-out" or LIFO for short. The most recent item added will be the next item removed. Program 94 implements a stack using an array and a pointer to the most recently added item. In the "`pushstack`" subroutine you will see array logic that will re-dimension the array to make sure there is enough room available in the stack for virtually any number of items to be added.

```
# stack.kbs
# implementing a stack using an array

dim stack(1) # array to hold stack with initial size
nstack = 0  # number of elements on stack

value = 1
gosub pushstack
value = 2
gosub pushstack
value = 3
```

```
gosub pushstack
value = 4
gosub pushstack
value = 5
gosub pushstack

while nstack > 0
   gosub popstack
   print value
end while

end

popstack: #
# get the top number from stack and set it in value
if nstack = 0 then
   print "stack empty"
else
   nstack = nstack - 1
   value = stack[nstack]
end if
return

pushstack: #
# push the number in the variable value onto the stack
# nake the stack larger if it is full
if nstack = stack[?] then redim stack(stack[?] + 5)
stack[nstack] = value
nstack = nstack + 1
return
```

Program 94: Stack

Queue:

The queue (pronounced like the letter Q) is another very common data structure. The queue, in its simplest form, is like the lunch line at school. The first one in the line is the first one to get to eat. Illustration 28 shows a block diagram of a queue.

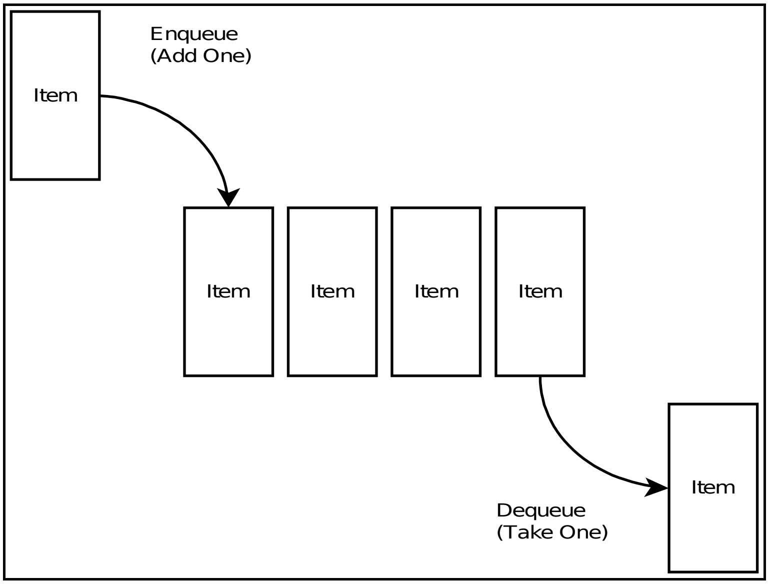


Illustration 28: What is a Queue

The terms enqueue (pronounced in-q) and dequeue (pronounced dee-q) are the names we use to describe adding a new item to the end of the line (tail) or removing an item from the front of the line (head). Sometimes this is described as a “first-in, first-out” or FIFO. The example in Program 95 uses an array and two pointers that keep track of the head of the line and the tail of the line.

```
# queue.kbs
# implementing a queue using an array
```

```

queuesize = 4  # maximum number of entries in the queue at any one time
dim queue(queuesize) # array to hold queue with initial size
tail = 0  # location in queue of next new entry
head = 0  # location in queue of next entry to be returnrd (served)
inqueue = 0  # number of entries in queue

value = 1
gosub enqueue
value = 2
gosub enqueue

gosub dequeue
print value

value = 3
gosub enqueue
value = 4
gosub enqueue

gosub dequeue
print value
gosub dequeue
print value

value = 5
gosub enqueue
value = 6
gosub enqueue
value = 7
gosub enqueue

# empty everybody from the queue
while inqueue > 0
```

```
   gosub dequeue
   print value
end while

end

dequeue: #
if inqueue = 0 then
   print "queue is empty"
else
   inqueue = inqueue - 1
   value = queue[head]
   print "dequeue value=" + value + " from=" + head + " inqueue=" + inqueue
   # move head pointer - if we are at end of array go back to the begining
   head = head + 1
   if head = queuesize then head = 0
end if
return

enqueue: #
if inqueue = queuesize then
   print "queue is full"
else
   inqueue = inqueue + 1
   queue[tail] = value
   print "enqueue value=" + value + " to=" + tail + " inqueue=" + inqueue
   # move tail pointer - if we are at end of array go back to the begining
   tail = tail + 1
   if tail = queuesize then tail = 0
end if
return
```

Program 95: Queue

Linked List:

In most books the discussion of this material starts with the linked list. Because BASIC-256 handles memory differently than many other languages this discussion was saved after introducing stacks and queues.

A linked list is a sequence of nodes that contains data and a pointer or index to the next node in the list. In addition to the nodes with their information we also need a pointer to the first node. We call the first node the "Head". Take a look at Illustration 29 and you will see how each node points to another.

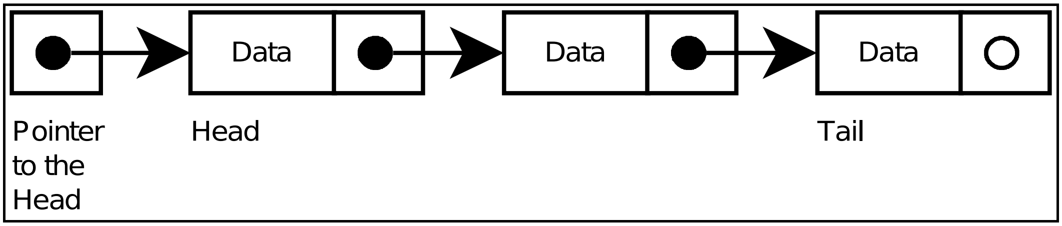


Illustration 29: Linked List

An advantage to the linked list, over an array, is the ease of inserting or deleting a node. To delete a node all you need to do is change the pointer on the previous node (Illustration 30) and release the discarded node so that it may be reused.

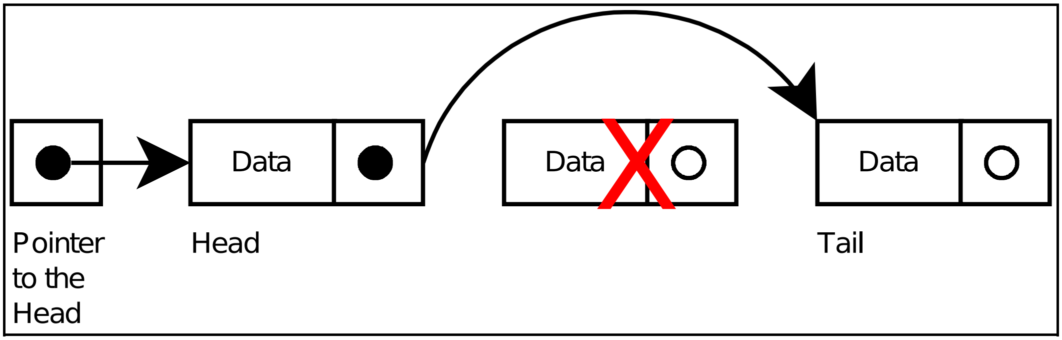


Illustration 30: Deleting an Item from a Linked List

Inserting a new node is also as simple as creating the new node, linking the new node to the next node, and linking the previous node to the first node. Illustration 31 Shows inserting a new node into the second position.

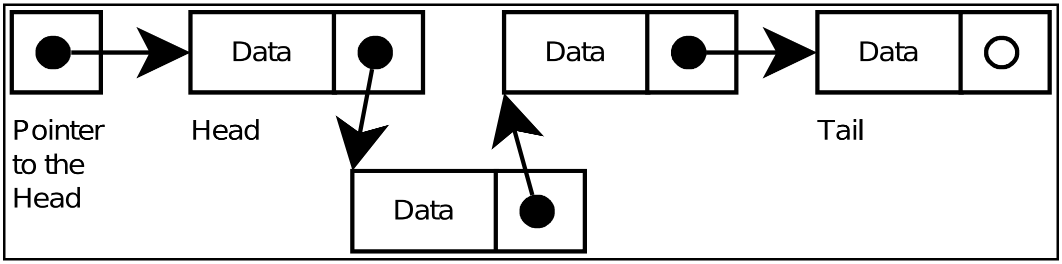


Illustration 31: Inserting an Item into a Linked List

Linked lists are commonly thought of as the simplest data structures. In the BASIC language we can't allocate memory like in most languages so we will simulate this behavior using arrays. In Program 96 we use the data$ array to store the text in the list, the nextitem array to contain the index to the next node, and the freeitem array to contain a stack of free (unused) array indexes.

```
# linkedlist.kbs
```

```

n = 8 # maximum size of list
dim data$(n)  # data for item in list
dim nextitem(n)  # pointer to next item in list
dim freeitem(n)  # list of free items

# initialize freeitem stack
for t = 0 to n-1
   freeitem[t] = t
next t
lastfree = n-1

head = -1    # start of list -  -1 = pointer to nowhere

# list of 3 items
text$ = "Head"
gosub append
text$ = "more"
gosub append
text$ = "stuff"
gosub append
gosub displaylist
gosub displayarrays
gosub wait

print "delete item 2"
r = 2
gosub delete
gosub displaylist
gosub displayarrays
gosub wait
```

```
print "insert item 1"
r = 1
text$ = "bar"
gosub insert
gosub displaylist
gosub displayarrays
gosub wait

print "insert item 2"
r = 2
text$ = "foo"
gosub insert
gosub displaylist
gosub displayarrays
gosub wait

print "delete item 1"
r = 1
gosub delete
gosub displaylist
gosub displayarrays
gosub wait

end

wait: ## wait for enter
input "press enter? ", garbage$
print
return

```

```
displaylist: # showlist by following the linked list
print "list..."
k = 0
i = head
do
   k = k + 1
   print k + " ";
   print data$[i]
   i = nextitem[i]
until i = -1
return

displayarrays: # show data actually stored and how
print "arrays..."
for i = 0 to n-1
   print i + " " + data$[i] + " >" + nextitem[i];
   for k = 0 to lastfree
      if freeitem[k] = i then print " <<free";
   next k
   if head = i then print " <<head";
   print
next i
return

insert: # insert text$ at position r
if r = 1 then
   gosub createitem
   nextitem[index] = head
   head = index
```

```
else
   k = 2
   i = head
   while i <> -1 and k <> r
      k = k + 1
      i = nextitem[i]
   end while
   if i <> -1 then
      gosub createitem
      nextitem[index] = nextitem[i]
      nextitem[i] = index
   else
      print "can't insert beyond end of list"
   end if
end if
return

delete: # delete element r from linked list
if r = 1 then
   index = head
   head = nextitem[index]
   gosub freeitem
else
   k = 2
   i = head
   while i <> -1 and k <> r
      k = k + 1
      i = nextitem[i]
   end while
   if i <> -1 then
      index = nextitem[i]
```

```
      nextitem[i] = nextitem[nextitem[i]]
      gosub freeitem
   else
      print "can't delete beyond end of list"
   end if
end if
return

append: # append  text$ to end of linked list
if head = -1 then
   gosub createitem
   head = index
else
   i = head
   while nextitem[i] <> -1
      i = nextitem[i]
   end while
   gosub createitem
   nextitem[i] = index
endif
return

freeitem: # free element in index and add back to the free stack
lastfree = lastfree + 1
freeitem[lastfree] = index
return

createitem: # save text$ in data and return index to new location
if lastfree < 0 then
   print "no free cell to allocate"
```

```
    end
  end if
  index = freeitem[lastfree]
  data$[index] = text$
  nextitem[index] = -1
  lastfree = lastfree - 1
  return
```

Program 96: Linked List

Explore	Re-write Program 96 to implement a stack and a queue using a linked list.

Slow and Inefficient Sort - Bubble Sort:

The “Bubble Sort” is probably the worst algorithm ever devised to sort a list of values. It is very slow and inefficient except for small sets of items. This is a classic example of a bad algorithm.

The only real positive thing that can be said about this algorithm is that it is simple to explain and to implement. Illustration 32 shows a flow-chart of the algorithm. The bubble sort goes through the array over and over again swapping the order of adjacent items until the sort is complete,

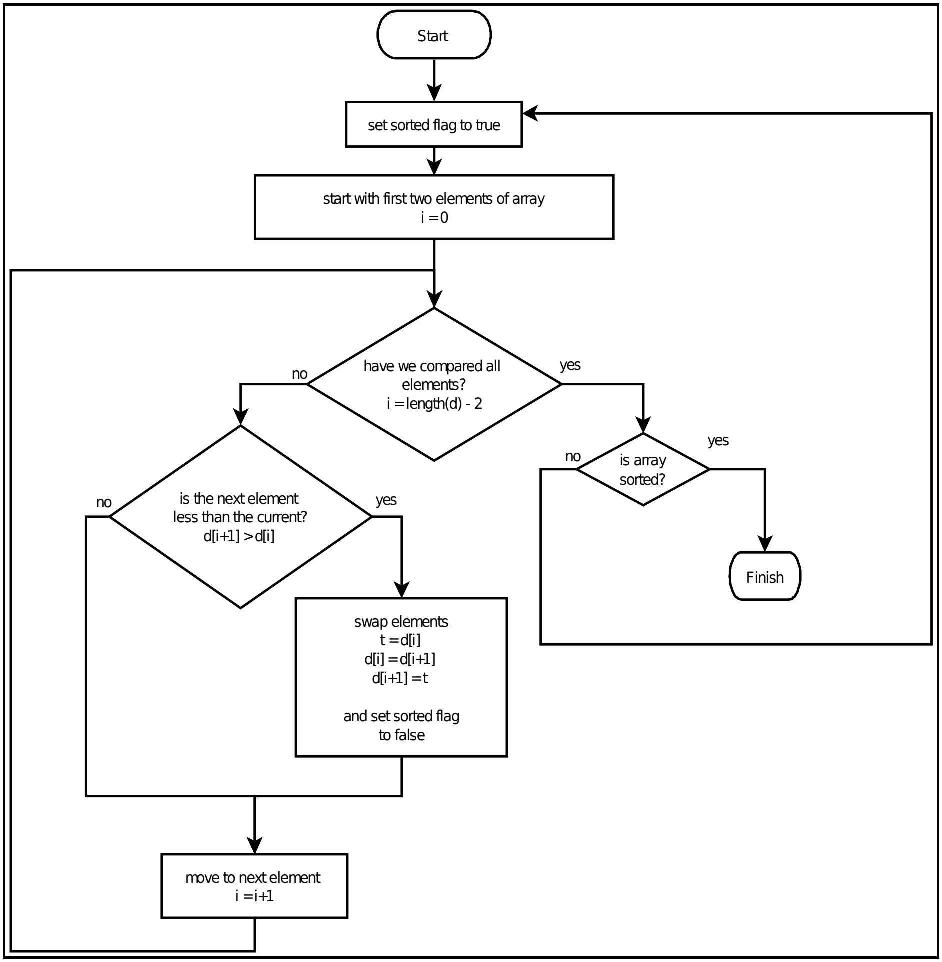


Illustration 32: Bubble Sort - Flowchart

```
# bubblesort.kbs
# implementing a simple sort

# a bubble sort is one of the SLOWEST algorithms
# for sorting but it is the easiest to implement
# and understand.
#
# The algorithm for a bubble sort is
```

```
# 1. Go through the array swaping adjacent values
#    so that lower value comes first.
# 2. Do step 1 over and over until there have
#    been no swaps (the array is sorted)
#

dim d(20)

# fill array with unsorted numbers
for i = 0 to d[?]-1
   d[i] = rand * 1000
next i

print "*** Un-Sorted ***"
gosub displayarray

gosub bubblesort

print "*** Sorted ***"
gosub displayarray
end

displayarray:
# print out the array's values
for i = 0 to d[?]-1
   print d[i] + " ";
next i
print
return

bubblesort:
do
   sorted = true
   for i = 0 to d[?] - 2
      if d[i] > d[i+1] then
```

```
            sorted = false
            temp = d[i+1]
            d[i+1] = d[i]
            d[i] = temp
         end if
      next i
   until sorted
   return
```

Program 97: Bubble Sort

Better Sort – Insertion Sort:

The insertion sort is another algorithm for sorting a list of items. It is usually faster than the bubble sort, but in the worst case case could take as long.

The insertion sort gets it's name from how it works. The sort goes through the elements of the array (index = 1 to length -1) and inserts the value in the correct location in the previous array elements. Illustration 33 shows a step-by-step example.

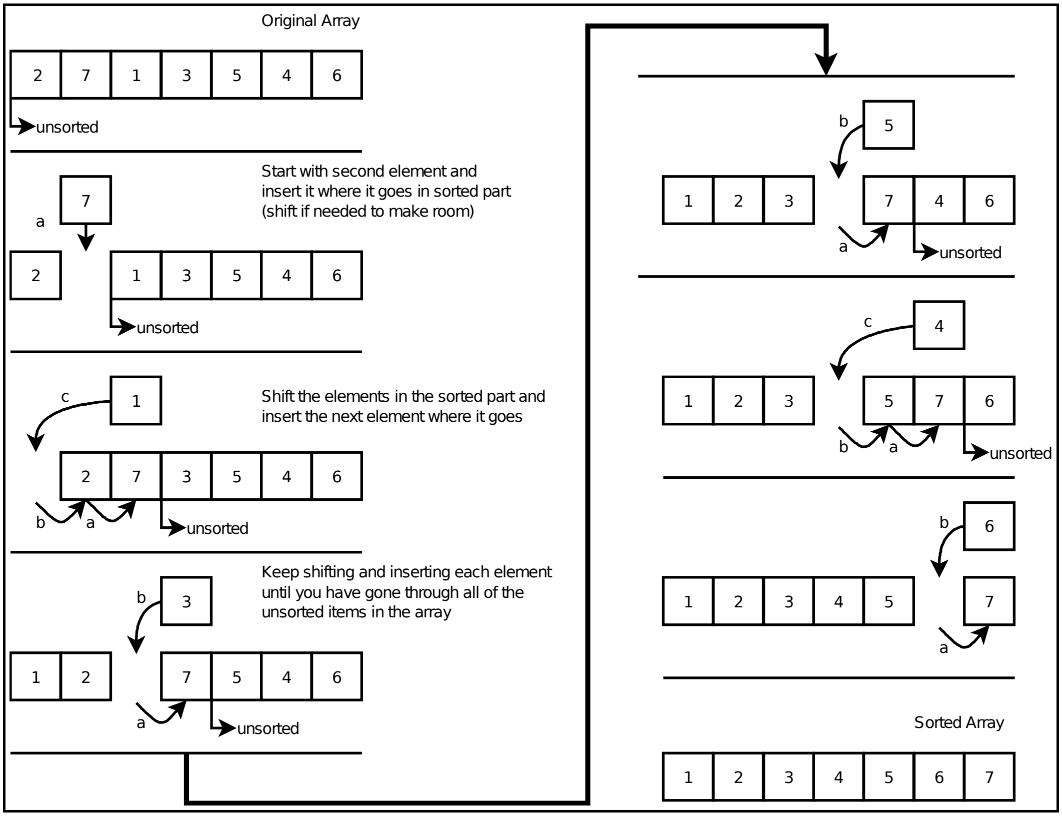


Illustration 33: Insertion Sort - Step-by-step

```
# insertionsort.kbs
# implementing an efficient sort

dim d(20)

# fill array with unsorted numbers
for i = 0 to d[?]-1
   d[i] = rand * 1000
next i

print "*** Un-Sorted ***"
gosub displayarray
```

```

gosub insertionsort

print "*** Sorted ***"
gosub displayarray
end

displayarray:
# print out the array's values
for i = 0 to d[?]-1
   print d[i] + " ";
next i
print
return

insertionsort:
# loops thru the list starting at the second element.
# takes current element and inserts it
# in the the correct sorted place in the previously
# sorted elements

# moving from backward from the current location
# and sliding elements with a larger value foward
# to make room for the current value in the correct
# place (in the partially sorted array)

for i = 1 to d[?] - 1
   currentvalue = d[i]
   j = i - 1
   done = false
   do
      if d[j] > currentvalue then
         # shift value and stop looping if we
```

```
are at begining
            d[j+1] = d[j]
            j = j - 1
            if j < 0 then done = true
         else
            # j is the element before where we want
to insert
            done = true
         endif
      until done
      d[j+1] = currentvalue
   next i
   return
```

Program 98: Insertion Sort

Explore	Re-write Program 98 using a linked list like in Program 96.

Explore	Research other sorting algorithms and write them in BASIC-256.

Chapter 18 – Runtime Error Trapping

As you have worked through the examples and created your own programs you have seen errors that happen while the program is running. These errors are called "runtime errors". BASIC-256 includes a group of special commands that allow your program to recover from or handle these errors.

Trapping errors, when you do not mean too, can cause problems. Error trapping should only be used when needed and disabled when not.

Error Trap:

When error trapping is turned on, with the **onerror** statement, the program will jump to a specified subroutine when an error occurs. If we look at Program 99 we will see that the program calls the subroutine when it tries to read the value of z (an undefined variable). If we try to run the same program with line one commented out or removed the program will terminate when the error happens.

```
1     onerror errortrap
2
3     print "z = " + z
4     print "Still running after error"
5     end
6
7     errortrap:
8     print "I trapped an error."
9     return
```

Program 99: Simple Runtime Error Trap

```
I trapped an error.
z = 0
Still running after error
```

Sample Output 99: Simple Runtime Error Trap

New Concept	**onerror** label Create an error trap that will automatically jump to the subroutine at the specified label when an error occurs.

Finding Out Which Error:

Sometimes just knowing that an error happened is not enough. There are functions that will return the error number (**lasterror**), the line where the error happened in the program (**lasterrorline**), a text message describing the error (**lasterrormessage**), and extra command specific error messages (**lasterrorextra**).

Program 100 modifies the previous program to print details of what error actually happened. More complex logic could be added to your error trap, specifically to change the behavior with different errors happen.

```
onerror errortrap

print "z = " + z
print "Still running after error"
end

errortrap:
print "Error Trap - Activated"
```

```
   print "   Error = " + lasterror
   print "   On Line = " + lasterrorline
   print "   Message = " + lasterrormessage
   return
```

Program 100: Runtime Error Trap - With Messages

```
Error Trap - Activated
   Error = 12
   On Line = 3
   Message = Unknown variable
z = 0
Still running after error
```

Sample Output 100: Runtime Error Trap - With Messages

New Concept

```
lasterror or lasterror()
lasterrorline or lasterrorline()
lasterrormessage or lasterrormessage()
lasterrorextra or lasterrorextra()
```

The four “last error” functions will return information about the last trapped error. These values will remain unchanged until another error is encountered.

lasterror	Returns the number of the last trapped error. If no errors have been trapped this function will return a zero. See Appendix J: Error Numbers for a complete list of trappable errors.
lasterrorline	Returns the line number, of the program, where the last error was trapped.
lasterrormessage	Returns a string describing the last error.
lasterrorextra	Returns a string with additional error information. For most errors this function will not return any information.

Turning Off Error Trapping:

Sometimes in a program we will want to trap errors during part of the program and not trap other errors. You will see examples of this type of error trapping logic in subsequent chapters.

The **offerror** statement turns error trapping off. This causes all errors encountered to stop the program.

```
onerror errortrap
print "z = " + z
print "Still running after first error"

offerror
print "z = " + z
print "Still running after second error"

end

errortrap:
print "Error Trap - Activated"
return
```

Program 101: Turning Off the Trap

```
Error Trap - Activated
z = 0
Still running after first error
ERROR on line 6: Unknown variable
```

Sample Output 101: Turning Off the Trap

Chapter 19: Database Programming

This chapter will show how BASIC-256 can connect to a simple relational database and use it to store and retrieve useful information.

What is a Database:

A database is simply an organized collection of numbers, string, and other types of information. The most common type of database is the "Relational Database". Relational Databases are made up of four major parts: tables, rows, columns, and relationships (see Table 8).

Table	A table consists of a predefined number or columns any any number of rows with information about a specific object or subject. Also known as a relation.
Row	Also called a tuple.
Column	This can also be referred to as an attribute.
Relationship	A reference of the key of one table as a column of another table. This creates a connection between tables.

Table 8: Major Components of a Relational Database

The SQL Language:

Most relational databases, today, use a language called SQL to actually extract and manipulate data. SQL is actually an acronym for Structured Query Language. The original SQL language was developed by IBM in the 1970s and has become the primary language used by relational databases.

SQL is a very powerful language and has been implemented by dozens of software companies, over the years. Because of this complexity there are many different dialects of SQL in use. BASIC-256 uses the SQLite database engine. Please see the SQLite web-page at http://www.sqlite.org for more information about the dialect of SQL shown in these examples.

Creating and Adding Data to a Database:

The SQLite library does not require the installation of a database sever or the setting up of a complex system. The database and all of its parts are stored in a simple file on your computer. This file can even be copied to another computer and used, without problem.

The first program (Program 102: Create a Database) creates a new sample database file and tables. The tables are represented by the Entity Relationship Diagram (ERD) as shown in Illustration 34.

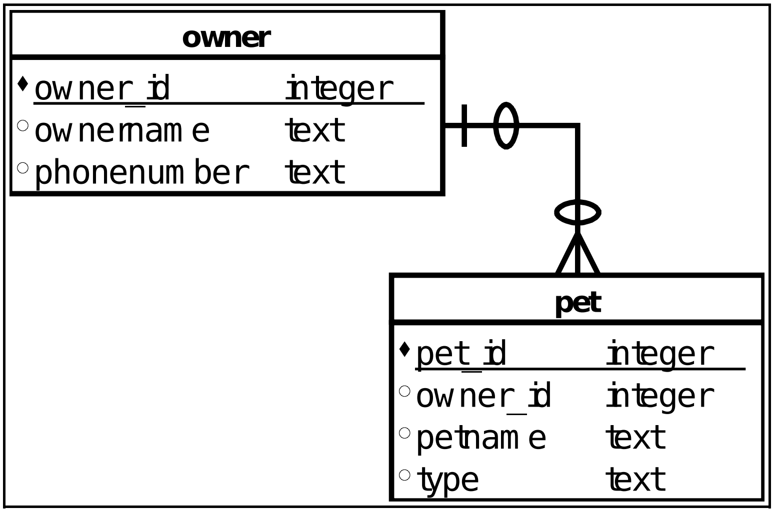


Illustration 34: Entity Relationship Diagram of Chapter Database

```
# delete old database and create a database with two tables
errors = 0
file$ = "pets.sqlite3"
if exists(file$) then kill(file$)
dbopen file$

stmt$ =  "CREATE TABLE owner (owner_id INTEGER, ownername TEXT, phonenumber TEXT, PRIMARY KEY (owner_id));"
gosub execute

stmt$ =  "CREATE TABLE pet (pet_id INTEGER, owner_id INTEGER, petname TEXT, type TEXT, PRIMARY KEY (pet_id), FOREIGN KEY (owner_id) REFERENCES owner (owner_id));"
gosub execute

# wrap everything up
dbclose
print file$ + " created. " + errors + " errors."
end

execute:
print stmt$
onerror executeerror
dbexecute stmt$
offerror
return

executeerror:
errors = errors + 1
print "ERROR: " + lasterror + " " + lasterrormessage + " " + lasterrorextra
return
```

Program 102: Create a Database

```
CREATE TABLE owner (owner_id INTEGER, ownername TEXT,
phonenumber TEXT, PRIMARY KEY (owner_id));
CREATE TABLE pet (pet_id INTEGER, owner_id INTEGER,
petname TEXT, type TEXT, PRIMARY KEY (pet_id),
FOREIGN KEY (owner_id) REFERENCES owner (owner_id));
pets.sqlite3 created. 0 errors.
```

Sample Output 102: Create a Database

So far you have seen three new database statements: **dbopen** – will open a database file and create it if it does not exist, **dbexecute** – will execute an SQL statement on the open database, and **dbclose** – closes the open database file.

New Concept	**dbopen** filename Open an SQLite database file. If the database does not exist then create a new empty database file.

New Concept	**dbexecute** sqlstatement Perform the SQL statement on the currently open SQLite database file. No value will be returned but a trappable runtime error will occur if there were any problems executing the statement on the database.

New Concept	**dbclose** Close the currently open SQLite database file. This statement insures that all data is written out to the database file.

These same three statements can also be used to execute other SQL statements. The INSERT INTO statement (Program 103) adds new rows of data to the tables and the UPDATE statement (Program 104) will change an existing row's information.

```
# add rows to the database

file$ = "pets.sqlite3"
dbopen file$

owner_id = 0
pet_id = 0

ownername$ = "Jim": phonenumber$ = "555-3434"
gosub addowner
petname$ = "Spot": type$ = "Cat"
gosub addpet
petname$ = "Fred": type$ = "Cat"
gosub addpet
petname$ = "Elvis": type$ = "Cat"
gosub addpet

ownername$ = "Sue": phonenumber$ = "555-8764"
gosub addowner
petname$ = "Alfred": type$ = "Cat"
gosub addpet
petname$ = "Fido": type$ = "Dog"
gosub addpet

```

```
ownername$ = "Amy": phonenumber$ = "555-9932"
gosub addowner
petname$ = "Bones": type$ = "Dog"
gosub addpet

ownername$ = "Dee": phonenumber$ = "555-4433"
gosub addowner
petname$ = "Sam": type$ = "Goat"
gosub addpet

# wrap everything up
dbclose
end

addowner:
owner_id = owner_id + 1
stmt$ = "INSERT INTO owner (owner_id, ownername, phonenumber) VALUES (" + owner_id + "," + chr(34) + ownername$ + chr(34) + "," + chr(34) + phonenumber$ + chr(34) + ");"
print stmt$
onerror adderror
dbexecute stmt$
offerror
return

addpet:
pet_id = pet_id + 1
stmt$ = "INSERT INTO pet (pet_id, owner_id, petname, type) VALUES (" + pet_id + "," + owner_id + "," + chr(34) + petname$ + chr(34) + "," + chr(34) + type$ + chr(34) + ");"
print stmt$
onerror adderror
dbexecute stmt$
offerror
return
```

```

adderror:
print "ERROR: " + lasterror + " " +
lasterrormessage + " " + lasterrorextra
return
```

Program 103: Insert Rows into Database

```
INSERT INTO owner (owner_id, ownername, phonenumber)
VALUES (1,"Jim","555-3434");
INSERT INTO pet (pet_id, owner_id, petname, type)
VALUES (1,1,"Spot","Cat");
INSERT INTO pet (pet_id, owner_id, petname, type)
VALUES (2,1,"Fred","Cat");
INSERT INTO pet (pet_id, owner_id, petname, type)
VALUES (3,1,"Elvis","Cat");
INSERT INTO owner (owner_id, ownername, phonenumber)
VALUES (2,"Sue","555-8764");
INSERT INTO pet (pet_id, owner_id, petname, type)
VALUES (4,2,"Alfred","Cat");
INSERT INTO pet (pet_id, owner_id, petname, type)
VALUES (5,2,"Fido","Dog");
INSERT INTO owner (owner_id, ownername, phonenumber)
VALUES (3,"Amy","555-9932");
INSERT INTO pet (pet_id, owner_id, petname, type)
VALUES (6,3,"Bones","Dog");
INSERT INTO owner (owner_id, ownername, phonenumber)
VALUES (4,"Dee","555-4433");
INSERT INTO pet (pet_id, owner_id, petname, type)
VALUES (7,4,"Sam","Goat");
```

Sample Output 103: Insert Rows into Database

```
# update a database row

dbopen "pets.sqlite3"

# create and populate
s$ =  "UPDATE owner SET phonenumber = " +
chr(34) + "555-5555" + chr(34) + " where
owner_id = 1;"
print s$
dbexecute s$
dbclose
```

Program 104: Update Row in a Database

```
UPDATE owner SET phonenumber = "555-5555" where
owner_id = 1;
```

Sample Output 104: Update Row in a Database

Retrieving Information from a Database:

So far we have seen how to open, close, and execute a SQL statement that does not return any values. A database would be pretty useless if we could not get information out of it.

The SELECT statement, in the SQL language, allows us to retrieve the desired data. After a SELECT is executed a “record set” is created that contains the rows and columns of data that was extracted from the database. Program 105 shows three different SELECT statements and how the data is read into your BASIC-256 program.

```
# Get data from the pets database

```

```
dbopen "pets.sqlite3"

# show owners and their phone numbers
print "Owners and Phone Numbers"
dbopenset "SELECT ownername, phonenumber FROM owner ORDER BY ownername;"
while dbrow()
   print dbstring(0) + " " + dbstring(1)
end while
dbcloseset

print

# show owners and their pets
print "Owners with Pets"
dbopenset "SELECT owner.ownername, pet.pet_id, pet.petname, pet.type FROM owner JOIN pet ON pet.owner_id = owner.owner_id ORDER BY ownername, petname;"
while dbrow()
   print dbstring(0) + " " + dbint(1) + " " + dbstring(2) + " " + dbstring(3)
end while
dbcloseset

print

# show average number of pets
print "Average Number of Pets"
dbopenset "SELECT AVG(c) FROM (SELECT COUNT(*) AS c FROM owner JOIN pet ON pet.owner_id = owner.owner_id GROUP BY owner.owner_id) AS numpets;"
while dbrow()
   print dbfloat(0)
end while
dbcloseset
```

```

# wrap everything up
dbclose
```

Program 105: Selecting Sets of Data from a Database

```
Owners and Phone Numbers
Amy 555-9932
Dee 555-4433
Jim 555-5555
Sue 555-8764

Owners with Pets
Amy 6 Bones Dog
Dee 7 Sam Goat
Jim 3 Elvis Cat
Jim 2 Fred Cat
Jim 1 Spot Cat
Sue 4 Alfred Cat
Sue 5 Fido Dog

Average Number of Pets
1.75
```

Sample Output 105: Selecting Sets of Data from a Database

New Concept	**dbopenset** sqlstatement Execute a SELECT statement on the database and create a “record set” to allow the program to read in the result. The “record set” may contain 0 or more rows as extracted by the SELECT.

New Concept	**`dbrow` or `dbrow ()`** Function to advance the result of the last **dbopenset** to the next row. Returns false if we are at the end of the selected data. You need to advance to the first row, using **dbrow**, after a **dbopenset** statement before you can read any data.

New Concept	**`dbint ( column )`** **`dbfloat ( column )`** **`dbstring ( column )`** These functions will return data from the current row of the record set. You must know the zero based numeric column number of the desired data.

dbint	Return the cell data as an integer.
dbfloat	Return the cell data as a floating point number.
dbstring	Return the cell data as a string.

New Concept	**`dbcloseset`** Close and discard the results of the last **dbopenset** statement.

Chapter 20: Connecting with a Network

This chapter discusses how to use the BASIC-256 networking statements. Networking in BASIC-256 will allow for a simple “socket” connection using TCP (Transmission Control Protocol). This chapter is not meant to be a full introduction to TCP/IP socket programming.

Socket Connection:

TCP stream sockets create a connection between two computers or programs. Packets of information may be sent and received in a bi-directional (or two way) manner over the connection.

To start a connection we need one computer or program to act as a server (to wait for the incoming telephone call) and the other to be a client (to make the telephone call). Illustration 35 shows graphically how a stream connection is made.

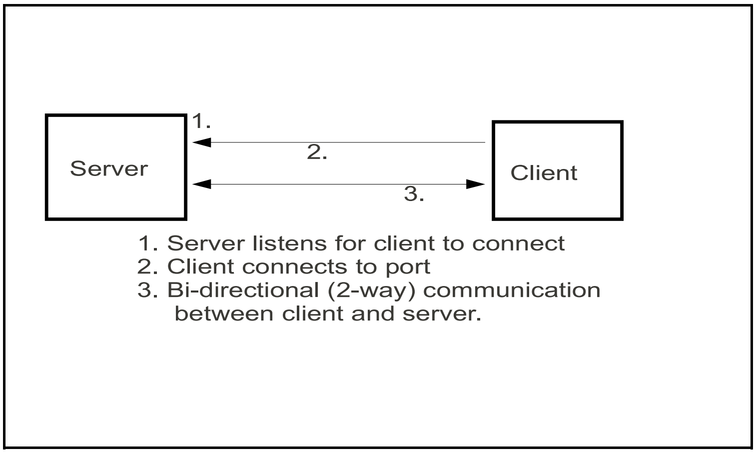


Illustration 35: Socket Communication

Just like with a telephone call, the person making the call (client) needs to know the phone number of the person they are calling (server). We call that number an IP address. BASIC-256 uses IP version 4 addresses that are usually expressed as four numbers separated by periods (999.999.999.999).

In addition to having the IP address for the server, the client and server must also talk to each-other over a port. You can think of the port as a telephone extension in a large company. A person is assigned an extension (port) to answer (server) and if you want to talk to that person you (client) call that extension.

The port number may be between 0 and 65535 but various Internet and other applications have been reserved ports in the range of 0-1023. It is recommended that you avoid using these ports.

A Simple Server and Client:

```
# simple_server.kbs
print "listening to port 9999 on " + netaddress()
NetListen 9999
NetWrite "The simple server sent this message."
NetClose
```

Program 106: Simple Network Server

```
# simple _client.kbs
input "What is the address of the simple_server?", addr$
if addr$ = "" then addr$ = "127.0.0.1"
#
NetConnect addr$, 9999
print NetRead
NetClose
```

Program 107: Simple Network Client

```
listening to port 9999 on xx.xx.xx.xx
```

Sample Output 106: Simple Network Server

```
What is the address of the simple_server?
The simple server sent this message.
```

Sample Output 107: Simple Network Client

New Concept	**netaddress** **netaddress** ( ) Function that returns a string containing the numeric IPv4 network address for this machine.

New Concept

```
netlisten portnumber
netlisten ( portnumbrer )
netlisten socketnumber, portnumber
netlisten ( socketnumber, portnumber )
```

Open up a network connection (server) on a specific port address and wait for another program to connect. If *socketnumber* is not specified socket number zero (0) will be used.

New Concept

```
netclose
netclose ( )
netclose socketnumber
netclose ( socketnumber )
```

Close the specified network connection (socket). If *socketnumber* is not specified socket number zero (0) will be closed.

New Concept

```
netwrite string
netwrite ( string )
netwrite socketnumber, string
netwrite ( socketnumber, string )
```

Send a string to the specified open network connection. If *socketnumber* is not specified socket number zero (0) will be written to.

New Concept

```
netconnect servername, portnumber
netconnect ( servername, portnumber )
netconnect socketnumber, servername,
    portnumber
netconnect ( socketnumber, servername,
    portnumber )
```

Open a network connection (client) to a server. The IP address or host name of a server are specified in the *servername* argument, and the specific network port number. If *socketnumber* is not specified socket number zero (0) will be used for the connection.

New Concept

```
netread
netread ( )
netread ( socketnumber )
```

Read data from the specified network connection and return it as a string. This function is blocking (it will wait until data is received). If *socketnumber* is not specified socket number zero (0) will be read from.

Network Chat:

This example adds one new function (**netdata**) to the networking statements we have already introduced. Use of this new function will allow our network clients to process other events, like keystrokes, and then read network data only when there is data to be read.

The network chat program (Program 108) combines the client and server program into one. If you start the application and it is unable to connect to a server the error is trapped and the program then becomes a server. This is one of many possible methods to allow a single program to fill both roles.

```
# chat.kbs
# uses port 9999 for server

input "Chat to address (return for server or local host)?", addr$
if addr$ = "" then addr$ = "127.0.0.1"
#
# try to connect to server - if there is not one become one
OnError startserver
NetConnect addr$, 9999
OffError
print "connected to server"

chatloop:
while true
   # get key pressed and send it
   k = key
   if k <> 0 then
      gosub show
      netwrite string(k)
   end if
   # get key from network and show it
   if NetData() then
      k = int(NetRead())
      gosub show
   end if
   pause .01
end while
end

show:
if k=16777220 then
```

```
      print
   else
      print chr(k);
   end if
   return

   startserver:
   OffError
   print "starting server - waiting for chat
   client"
   NetListen 9999
   print "client connected"
   goto chatloop
   return
```

Program 108: Network Chat

The following is observed when the user on the client types the message “HI SERVER” and then the user on the server types “HI CLIENT”.

```
Chat to address (return for server or local host)?
starting server - waiting for chat client
client connected
HI SERVER
HI CLIENT
```

Sample Output 108.1: Network Chat (Server)

```
Chat to address (return for server or local host)?
connected to server
HI SERVER
HI CLIENT
```

Sample Output 108.2: Network Chat (Client)

New Concept

netdata or **netdata**()

Returns true if there is network data waiting to be read. This allows for the program to continue operations without waiting for a network packet to arrive.

Big Program

The big program this chapter creates a two player networked tank battle game. Each player is the white tank on their screen and the other player is the black tank. Use the arrow keys to rotate and move. Shoot with the space bar.

```
# battle.kbs
# uses port 9998 for server

kspace = 32
kleft = 16777234
kright = 16777236
kup = 16777235
kdown = 16777237
dr = pi / 16   # direction change
dxy = 2.5 # move speed
scale = 20     # tank size
shotscale = 4  # shot size
shotdxy = 5    # shot move speed
port = 9998    # port to communicate on
```

```

dim tank(30)
tank = {-1,-.66, -.66,-.66, -.66,-.33, -.33, -.33, 0,-1, .33,-.33, .66,-.33, .66,-.66, 1,-.66, 1,1, .66,1, .66,.66, -.66,.66, -.66,1, -1,1}
dim shot(14)
shot = {0,-1, .5,-.5, .25,0, .5,.75, -.25,.75, -.25,0, -.5,-.5}

print "Tank Battle - You are the white tank."
print "Your mission is to shoot and kill the"
print "black one. Use arrows to move and"
print "space to shoot."
print
input "Address (return for server or local host)?", addr$
if addr$ = "" then addr$ = "127.0.0.1"

# try to connect to server - if there is not one become one
OnError startserver
NetConnect addr$, port
OffError
print "connected to server"

playgame:

myx = 100
myy = 100
myr = 0
mypx = 0  # projectile position direction and remaining length (no shot when mypl=0)
mypy = 0
mypr = 0
mypl = 0
yourx = 200
```

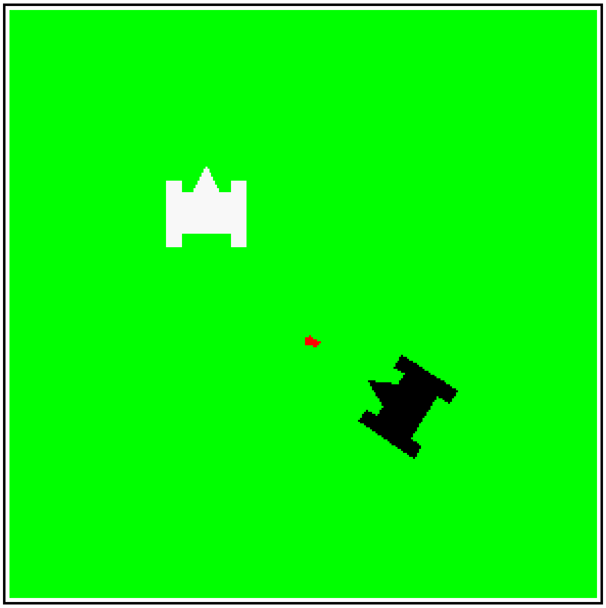
```
youry = 200
yourr = pi
yourpx = 0     # projectile position direction and remaining length
yourpy = 0
yourpr = 0
yourpl = 0
gosub writeposition

fastgraphics
while true
   # get key pressed and move tank on the screen
   k = key
   if k <> 0 then
      if k = kup then
         myx = myx + sin(myr) * dxy
         myy = myy - cos(myr) * dxy
      end if
      if k = kdown then
         myx = myx - sin(myr) * dxy
         myy = myy + cos(myr) * dxy
      end if
      if k = kspace then
         mypr = myr
         mypx = myx + sin(mypr) * scale
         mypy = myy - cos(mypr) * scale
         mypl = 100
      end if
      if myx < scale then myx = graphwidth - scale
      if myx > graphwidth-scale then myx = scale
      if myy < scale then myy = graphheight - scale
      if myy > graphheight-scale then myy = scale
      if k = kleft then myr = myr - dr
      if k = kright then myr = myr + dr
```

```
         gosub writeposition
      end if
      # move my projectile (if there is one)
      if mypl > 0 then
         mypx = mypx + sin(mypr) * shotdxy
         mypy = mypy - cos(mypr) * shotdxy
         if mypx < shotscale then mypx = graphwidth - shotscale
         if mypx > graphwidth-shotscale then mypx = shotscale
         if mypy < shotscale then mypy = graphheight - shotscale
         if mypy > graphheight-shotscale then mypy = shotscale
         if (mypx-yourx)^2 + (mypy-youry)^2 < scale^2 then
            NetWrite "!"
            print "You killed your opponent. Game over."
            end
         end if
         mypl = mypl - 1
         gosub writeposition
      end if
      # get position from network
      gosub getposition
      #
      gosub draw
      #
      pause .1
end while

writeposition: ###
# 10 char for x, 10 char for y, 10 char for r (rotation)
position$ =  left(myx + "          ",10)+left(myy + "          ",10)+left(myr + "
```

```
",10)+left(mypx + "          ",10)+left(mypy + "
",10)+left(mypr + "          ",10)+left(mypl + "
",10)
NetWrite position$
return

getposition: ###
# get position from network and set variables
for the opponent
while NetData()
   position$ = NetRead()
   if position$ = "!" then
      print "You Died. - Game Over"
      end
   end if
   yourx = 300 - float(mid(position$,1,10))
   youry = 300 - float(mid(position$,11,10))
   yourr = pi + float(mid(position$,21,10))
   yourpx = 300 - float(mid(position$,31,10))
   yourpy = 300 - float(mid(position$,41,10))
   yourpr = pi + float(mid(position$,51,10))
   yourpl = pi + float(mid(position$,61,10))
end while
return

draw: ###
clg
color green
rect 0,0,graphwidth,graphheight
color white
stamp myx, myy, scale, myr, tank
if mypl > 0 then
   stamp mypx, mypy, shotscale, mypr, shot
end if
color black
stamp yourx, youry, scale, yourr, tank
if yourpl > 0 then
```

```
      color red
      stamp yourpx, yourpy, shotscale, yourpr, shot
   end if
   refresh
   return

   startserver:
   OffError
   print "starting server - waiting for chat
   client"
   NetListen port
   print "client connected"
   goto playgame
   return
```

Program 109: Network Tank Battle



Sample Output 109: Network Tank Battle

Appendix A: Loading BASIC-256 on your PC or USB Pen Drive

This chapter will walk you step by step through downloading and installing BASIC-256 on your Microsoft Windows PC. The instructions are written for Windows XP with Firefox 3.x as your Web browser. Your specific configuration and installation may be different but the general steps should be similar.

1 – Download:

Connect to the Internet and navigate to the Web site http://www.basic256.org and follow the download link. Once you are at the Sourceforge project page click on the green "Download Now!"button (Illustration 36) to start the download process.

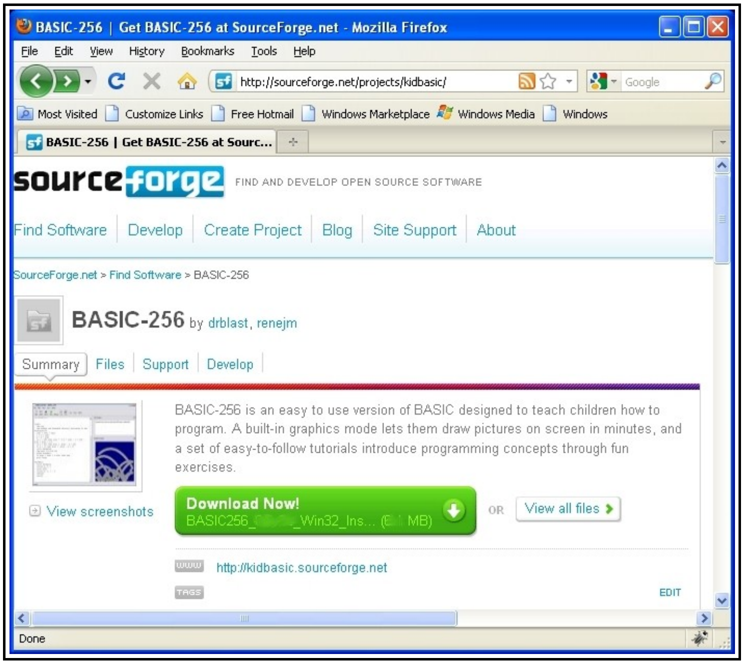


Illustration 36: BASIC-256 on Sourceforge

The download process may ask you what you want to do with the file. Click the “Save File” button (Illustration 37).

Illustration 37: Saving Install File

Firefox should display the “Downloads” window and actually download the BASIC-256 installer. When it is finished it should look like Illustration 38. Do not close this window quite yet, you will need it to start the Installation.

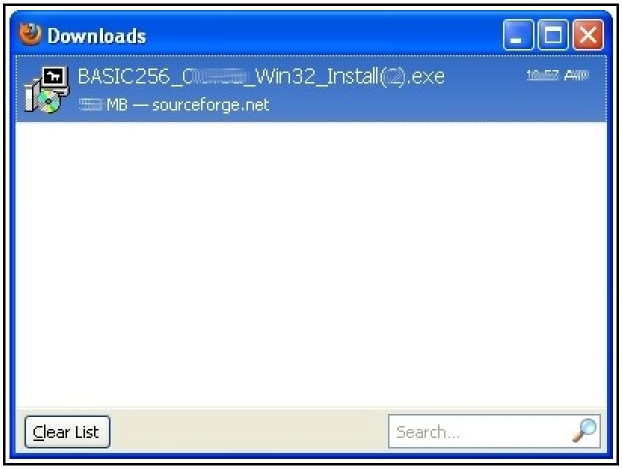


Illustration 38: File Downloaded

2 – Installing:

Once the file has finished downloading (Illustration 38) use your mouse and click on the file from the download list. You will then see one or two dialogs asking if you really want to execute this file (Illustration 39) (Illustration 40). You need to click the “OK” or “Run” buttons on these dialogs.

Illustration 39: Open File Warning

Illustration 40: Open File Security Warning

After the security warnings are cleared you will see the actual BASIC-256 Installer application. Click the “Next>” button on the first screen (Illustration 41).

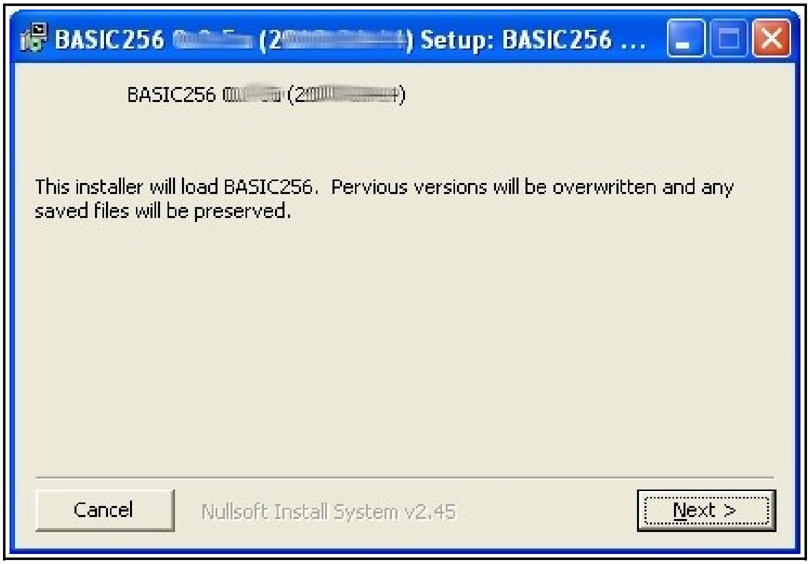


Illustration 41: Installer - Welcome Screen

Read and agree to the GNU GPL software license and click on “I Agree” (Illustration 42). The GNU GPL license is one of the most commonly used “Open Source” and”Free” license to software. You have the right to use, give away, and modify the programs released under the GPL. This license only relates to the BASIC-256 software and not the contents of this book.

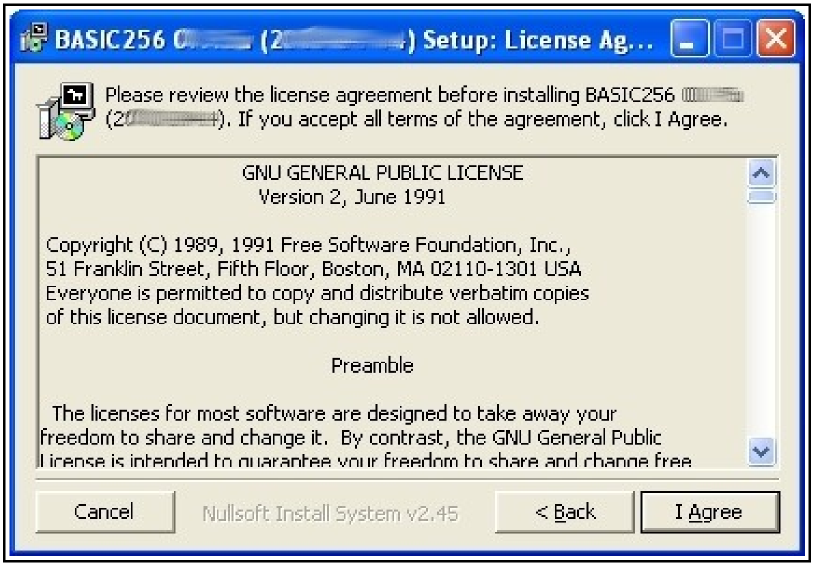


Illustration 42: Installer - GPL License Screen

The next Installer screen asks you what you want to install (Illustration 43). If you are installing BASIC-256 to a USB or other type of removable drive then it is suggested that you un-check the “Start Menu Shortcuts”. For most users who are installing to a hard drive, should do a complete install. Click “Next>”.

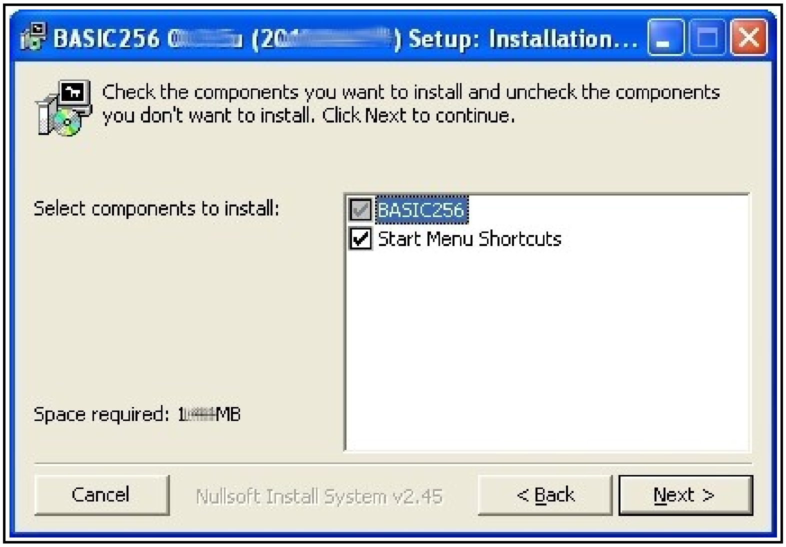


Illustration 43: Installer - What to Install

Illustration 44 shows the last screen before the install begins. This screen asks you what folder to install the BASIC-256 executable files into. If you are installing to your hard drive then you should accept the default path.

Illustration 44: Installer - Where to Install

The installation is complete when you see this screen (Illustration 45). Click “Close”.

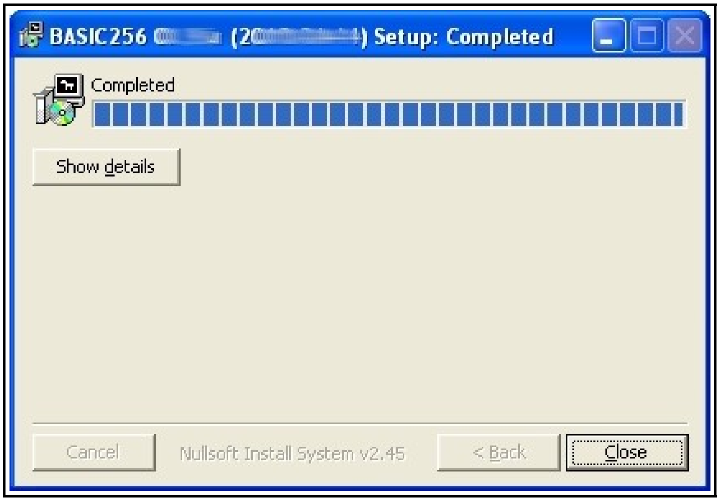


Illustration 45: Installer - Complete

3 – Starting BASIC-256

The installation is complete. You may now click on the Windows “Start” button and then “All Programs >” (Illustration 46).

Illustration 46: XP Start Button

You will then see a menu for BASIC-256. You may open the program by clicking on it, uninstall it, or view the documentation from this menu (Illustration 47).

Illustration 47: BASIC-256 Menu from All Programs

Appendix B: Language Reference - Statements

Chapter number where this statement is introduced is shown in parentheses.

circle – Draw a Circle on the Graphics Output Area (2)

circle *x*, *y*, *radius*

The circle command draws a filled circle on the graphics output area. The center of the circle is defined by the x and y parameters and the size is defined as radius.

Example:

```
clg
color 255,128,128
circle 150,150,150
color red
circle 150,150,100
```

changedir – Change Your Current Working Directory (16)

changedir *path*

The **changedir** command allows you to change the current working directory for you application. When you specify a file without a full path (in **imgload**, **open**, **spriteload**, or other statement that requests a file name) the application uses this directory. You can check your currently set path using the **currentdir** function.

clg – Clear Graphics Output Area (2)

```
clg
```

This command clears the graphics output area. The graphics output area is not cleared automatically when an program is run. This will sometimes leave undesired graphics visible. If you are using graphics it is advised that you always clear the output window, first.

clickclear – Clear the Last Mouse Click (10)

```
clickclear
```

When the mouse is being read in click mode the x position, y position, and button click information are stored when the mouse button is clicked. These values can be retrieved with the clickx(), clicky(), and clickb() functions. The stored values can be reset to zero (0) using clickclear.

close – Close the Currently Open File (16)

```
close
close()
close filenumber
close (filenumber)
```

Closes open file. This will flush any pending disk output. If file number parameter is not specified then file number zero (0) will be used.

cls – Clear Text Output Window (1)

```
cls
```

This command clears the Text Output window. The Text Output window is automatically cleared when a program is run.

color or **colour**– Set Color for Drawing (2)

```
color colorname
color rgbvalue
color red, green, blue
```

Sets the foreground color for all graphical commands. The color may be specified by the color name (see Appendix E), an integer representing the RGB value, or by three numbers representing the RGB value as separate component colors.

A special color named CLEAR or represented by -1 tells the drawing commands to erase the pixels from the drawing and make them transparent.

Example:

```
clg
color black
rect 100,100,100,100
color 255,128,128
circle 150,150,75
```

dbclose (19)

```
dbclose
```

Close the currently open SQLite database file.

dbcloseset (19)

```
dbcloseset
```

Close the currently open record set opened by **DBOpenSet**.

dbexecute (19)

```
dbexecute statement
dbexecute ( statement )
```

Execute an SQL statement on the open SQLite database file. This statement does not create a record set but will return an error if the statement did not execute.

dbopen (19)

```
dbopen filename
dbopen ( filename )
```

Open an SQLite database file. If the file does not exist then create it.

dbopenset (19)

```
dbopenset statement
dbopenset ( statement )
```

Perform an SQL statement and create a record set so that the program may loop through and use the results.

decimal ()

```
decimal n
decimal ( n )
```

Description...

dim – Dimension a New Array (13)

```
dim variable(items)
dim variable$(items)
dim variable(rows, columns)
dim variable$(rows, columns)
```

The **dim** statement creates an array in the computer's memory the size that was specified in the parenthesis. Sizes (*items*, *rows*, and *columns*) must be integer values greater than or equal to one (1). The **dim** statement will initialize the elements in the new array with either zero (0) if numeric or the empty string (“”), depending on the type of variable.

do / until – Do / Until Loop (7)

```
do
    statement(s)
until condition
```

Repeat the statements in the block over and over again. Stop repeating when the condition is true. The statements will be executed one or more times.

end – Stop Running the Program (9)

```
end
```

Terminates the program (stop).

fastgraphics – Turn Fast Graphics Mode On (8)

```
fastgraphics
```

The **fastgraphics** statement will switch BASIC-256 into fast graphics mode. In this mode the graphics output area is only refreshed (drawn), when the program requests. This speeds up graphically intense programs. The **refresh** statement signals that draw process. Once fast graphics mode is entered in a program you may not return to the default slow graphics.

font – Set Font, Size, and Weight (8)

```
font fontname, point, weight
```

The **font** command sets the font that will be used by the next **text** command. You must specify the name of the font or font family, the point size, and the weight.

Each computer may have several different fonts available but "Helvetica", "Times", "Courier", "System", "Symbol" should be available on most computers. The point size represents how tall the letters will be drawn. Weight is used to specify how dark the letters will be drawn (25-light, 50-normal, 63-demi bold, 75-bold, 100-black).

Example:

```
clg
color black
n = 5
dim fonts$(n)
fonts$ = {"Helvetica", "Times", "Courier",
"System", "Symbol"}
for t = 0 to n-1
    font fonts$[t], 32, 50
    text 10, t*50, fonts$[t]
next t
```

for/next – Loop and Count (7)

```
for variable = expr1 to expr2 [step expr3]
    statement(s)
next variable
```

Execute a block of code a specified number of times. The variable will begin with the value of *expr1* and be incremented and the looping will continue until the variable is greater than *expr2*. If the **step** clause is included in the statement the increment will be *expr3* and not the default value of one (1).

goto – Jump to a Label (9)

```
goto label
```

The **goto** statement causes the execution to jump to the statement directly following the label.

gosub/return – Jump to a Subroutine and Return (9)

```
gosub label
```

```
return
```

The **gosub** statement causes the execution to jump to the subroutine defined by the label. Execute the **return** statement within a subroutine to send control back to where it was called from.

graphsize – Set Graphic Display Size (8)

```
graphsize width, height
```

Set the graphics output area to the specified *height* and *width*.

if then – Test if Something is True - Single Line(6)

```
if condition then statement
```

If the condition evaluates to true then execute the statement following the **then** clause.

if then / end if – Test if Something is True – Multiple Line (6)

```
if condition then
    statement(s) to execute when true
end if
```

The **if** and **end if** statements allow you to create a block of programming code to execute when a condition is true. It is often customary to indent the statements within the **if**/**end if** statements so they are not confusing to read.

if then / else / end if – Test if Something is True – Multiple Line with Else (6)

```
if condition then
    statement(s) to execute when true
else
    statement(s) to execute when false
end if
```

The **if**, **else**, and **end if** statements allow you to define two blocks of programming code. The first block, after the **then** clause, executes if the condition is true and the second block, after the **else** clause, will execute when the condition is false.

imgload – Load an image from a file and display (12)

```
imgload x, y, filename
imgload x, y, scale, filename
imgload x, y, scale, rotation, filename
```

Read in the picture found in the file and display it on the graphics output area. The values of x and y represent the location to place the CENTER of the image.

Images may be loaded from many different file formats, including: BMP, PNG, GIF, JPG, and JPEG.

Optionally scale (re-size) it by the decimal scale where 1 is full size. Also you may also rotate the image clockwise around it's center by specifying how far to rotate as an angle expressed in radians (0 to 2π).

input – Get a String Value from the User (7)

```
input "prompt", stringvariable$
input "prompt", numericvariable
input stringvariable$
input numericvariable
```

The **input** statement will retrieve a string or a number that the user types into the text output area of the screen. The result will be stored in a variable that may be used later in the program.

A prompt message, if specified, will display on the text output area and the cursor will directly follow the prompt.

If a numeric result is desired (numeric variable specified in the statement) and the user types a string that can not be converted to a number the input statement will set the variable to zero (0).

kill – Delete a File ()

```
kill filename
kill ( filename )
```

Delete a file from the file system

line – Draw a Line on the Graphics Output Area (2)

```
line start_x, start_y, finish_x, finish_y
```

Draw a line one pixel wide from the starting point to the ending point, using the current color.

netclose (20)

```
netclose
netclose ( )
netclose socket
netclose ( socket )
```

Close the specified network connection (socket). If socket number is not number zero (0) will be used.

netconnect (20)

```
netconnect server, port
netconnect ( server, port )
netconnect socket, server, port
netconnect ( socket, server, port )
```

Open a network connection (client) to a server. The IP address or host name of a server are specified in the *server_name* argument, and the specific network port number in the *port_number* argument. If socket number is not specified zero (0) will be used.

netlisten (20)

```
netlisten port
netlisten ( port )
netlisten socket, port
netlisten ( socket, port )
```

Open up a network connection (server) on a specific port address and wait for another program to connect. If socket number is not specified zero (0) will be used.

netwrite (20)

```
netwrite string
netwrite ( string )
netwrite socket, string
netwrite ( socket, string )
```

Send a string to the specified open network connection. If socket number is not specified zero (0) will be used.

offerror (18)

```
offerror
```

Turns off error trapping and restores the default error behavior.

onerror (18)

```
onerror label
```

Causes the subroutine at *label* to be executed when an runtime error occurs. Program control may be resumed at the next statement with a **return** statement in the subroutine.

open – Open a file for Reading and Writing (16)

```
open filename
open filenumber, filename
```

Open the file specified for reading and writing. If the file does not exist it will be created so that information may be added (see **write** and **writeline**). Be sure to execute the **close** statement when the program is finished with the file.

BASIC-256 may have up to eight (8) files opened at any one time. The files will be numbered from zero(0) to seven(7). If a file number is not specified then file number zero (0) will be used.

pause – Pause the Program (7)

```
pause seconds
```

The **pause** statement tells BASIC-256 to stop executing the current program for a specified number of seconds. The number of seconds may be a decimal number if a fractional second pause is required.

plot – Put a Point on the Graphics Output Area (2)

```
plot x, y
```

Changes a single pixel to the current color.

poly – Draw a Polygon on the Graphics Output Area (8)

```
poly {x1, y1, x2, y2 ...}
poly numeric_array
```

Draw a polygon. The array or list should contain an even number of elements so that the each vertex of the polygon is represented by first two values.

print – Display a String on the Text Output Window (1)

```
print expression
print expression;
```

The **print** statement is used to display text and numbers on the text output area of the BASIC-256 window. Print normally goes down to the next line but you may output several things on the same line by using a ; (semicolon) at the end of the expression.

putslice – Display a Captured Part of the Graphics Output

```
putslice x, y, slice
putslice x, y, slice, rgbcolor
```

This statement will draw the captured slice (see the **getslice** function) back onto the graphics output area. If an RGB color is specified then the slice will be drawn with pixels of that color being omitted (transparent).

rect – Draw a Rectangle on the Graphics Output Area (2)

```
rect x, y, width, height
```

The rect command draws a filled rectangle on the graphics output area. The top left corner will be placed at the point (x, y).

Example:

```
clg
color darkblue
rect 75,75,100,100
color blue
rect 100,100,100,100
```

redim – Re-Dimension an Array (12)

redim *variable*(*items*)
redim *variable$*(*items*)
redim *variable*(*rows*, *columns*)
redim *variable$*(*rows*, *columns*)

The **redim** statement re-sizes an array in the computer's memory. Data previously stored in the array will be kept, if it fits.

When resizing two-dimensional arrays the values are copied in a linear manner. Data may be shifted in an unwanted manner if you are changing the number of columns.

refresh – Update Graphics Output Area (8)

refresh

In fast graphics mode (see **fastgraphics**) the graphics output area is only refreshed, drawn, when the program requests. This speeds up graphically intense programs. The refresh statement signals that draw process.

rem – Remark or Comment (2)

rem *comment text*

```
# comment text
```

Insert remark, also called a comment, into a program. Any text, on a line, following the **rem** or **#** will be ignored by BASIC-256. Remarks are used by programmers to place information about what the program does, who wrote or changed it, and how it works.

reset – Clear an Open File (16)

```
reset
reset()
reset filenumber
```

Clear any data from an open file and move the file pointer to the beginning.
If file number is not specified then file number zero (0) will be used.

say – Use Text-To-Speech to Speak (1)

```
say expression
```

The **say** statement is used to make BASIC-256 read an expression aloud,
to the computer's speakers.

seek – Move the File I/O Pointer (16)

```
seek expression
seek (expression)
seek filenumber, expression
seek (filenumber, expression)
```

Move the file pointer for the next read or write operation to a specific location in the file. To move the current pointer to the beginning of the file use the value zero (0). To seek to the end of a file use the size() function as the argument to the seek statement.

If file number parameter is not specified then file number zero (0) will be used.

spritedim – Initialize Sprites for Drawing (12)

```
spritedim numberofsprites
```

The **spritedim** statement initializes, or allocates in memory, places to store the specified number of sprites. Each sprite will need to be loaded (**spriteload**) or created (**spriteslice**) before it may be displayed. You may allocate as many sprites as your program may require but your program may be slow if you create many sprites.

Sprites are drawn on the graphics output area in order by their assigned sprite number. A sprite will be drawn under any sprite with a higher number and over all sprites with a lower number.

Sprites are numbered from zero (0) to one less than the number specified in this command (numberofsprites -1).

spritehide – Hide a Sprite (12)

```
spritehide spritenumber
```

This statement will cause the specified sprite to not be drawn on the screen. It will still exist and may be shown using the **spriteshow** statement.

spriteload – Load an Image File Into a Sprite (12)

```
spriteload spritenumber, filename
```

This statement reads an image file (GIF, BMP, PNG, JPG, or JPEG) from the specified path and creates a sprite. The sprite muse be allocated using the **spritedim** statement before you may load it.

By default the sprite will be placed with its center at 0,0 and it will be hidden. You should move the sprite to the desired position on the screen (**spritemove** or **spriteplace**) and then show it (**spriteshow**).

spritemove – Move a Sprite from Its Current Location (12)

```
spritemove spritenumber, dx, dy
```

Move the specified sprite x pixels to the right and y pixels down. Negative numbers can also be specified to move the sprite left and up. A sprite's center will not move beyond the edge of the current graphics output window.

You may use the **spritex** and **spritey** functions to determine the current location of the sprite.

You can move a hidden sprite but it will not be displayed until you show the sprite using the **showsprite** statement.

spriteplace – Place a Sprite at a Specific Location (12)

```
spriteplace spritenumber, x, y
```

The spriteplace statement allows you to place a sprite's center at a specific location on the graphics output area.

spriteshow – Show a Sprite (12)

```
spriteshow spritenumber
```

The **spriteshow** statement causes a loaded, created, or hidden sprite to be displayed on the graphics output area.

spriteslice – Capture a Sprite (12)

```
spriteslice spritenumber, x, y, width, height
```

This statement will allow you to create a sprite by copying it from the graphics output area. The arguments *x*, *y*, *width*, and *height* specify a rectangular area to capture and use for the sprite. Pixels that have not been drawn since the last **cls** statement or that were drawn using the color **clear** will be transparent when the sprite is drawn.

By default the sprite will be placed with its center at 0,0 and it will be hidden. You should move the sprite to the desired position on the screen (**spritemove** or **spriteplace**) and then show it (**spriteshow**).

sound – Play a beep on the PC Speaker (3)

```
sound frequency, duration
sound {frequency1, duration1, frequency2, duration2
...}
sound numeric_array
```

The first form of the **sound** statement takes two arguments; (1) the frequency of the sound in Hz (cycles per second) and (2) the length of the tone in milliseconds (ms). The second uses curly braces and can specify several tones and durations in a list. The third form uses an array containing frequencies and durations.

stamp – Put a Polygon Where You Want It (8)

```
stamp x, y, {x1, y1, x2, y2 ...}
stamp x, y, numeric_array
stamp x, y, scale, {x1, y1, x2, y2 ...}
stamp x, y, scale, numeric_array
stamp x, y, scale, rotate, {x1, y1, x2, y2 ...}
stamp x, y, scale, rotate, numeric_array
```

Draw a polygon with it's origin (0,0) at the screen position (x,y). Optionally scale (re-size) it by the decimal scale where 1 is full size. Also you may also rotate the stamp clockwise around it's origin by specifying how far to rotate as an angle expressed in radians (0 to 2π).

system – Execute System Command in a Shell

```
system expression
```

Open a command window and execute the operating system command.

text – Draw text on the Graphics Output Area (8)

```
text x, y, output
```

The **text** command will draw characters on the graphics output area. The *x* and *y* arguments represent the top left corner and will draw the text with the current color and font.

Example:

```
clg
font “Helvetica”, 32, 50
color red
text 100, 100, “Hi Mom.”
```

volume – Adjust Amplitude of Sound Statement

volume *expression*

Adjust the height of the waveform generated by the sound statement.

wavplay – Play a WAV audio file in the background (12)

wavplay *filename*

Load .wav (wave) audio file data from the file name and play. The playback will be synchronous and the next statement in the program will begin immediately as soon as the audio begins playing.

wavstop – Stop playing WAV audio file (12)

wavstop

If there is a currently playing audio file (see **wavplay**) then stop the synchronous playback.

wavwait – Wait for the WAV to finish (12)

```
wavwait
```

If there is a currently playing audio file (see **wavplay**) then wait for it to finish playing.

while / end while – While Loop (7)

```
while condition
    statement(s)
end while
```

Do the statements in the block over and over again while the condition is true. The statements will be executed zero or more times.

write – Write Data to the Currently Open File (16)

```
write expression
write (expression)
write filenumber, expression
write (filenumber, expression)
```

Write the string expression to an open file. Do not add an end of line or a delimiter.

If file number parameter is not specified then file number zero (0) will be used.

writeline – Write a Line to the Currently Open File (16)

```
writeline expression
writeline (expression)
writeline filenumber, expression
writeline (filenumber, expression)
```

Output the contents of the expression to an open file and then append an end of line mark to the data. The file pointer will be positioned at the end of the write so that the next write statement will directly follow.

If file number parameter is not specified then file number zero (0) will be used.

Appendix C: Language Reference - Functions

Functions perform calculations, get system values, and return them to the program.

Each function will return a value of a specific type (integer, Boolean, floating point, or string) and potentially a specific range of values. Chapter number where this function is introduced is shown in parentheses.

abs – Absolute Value (14)

abs (*expression*)

Argument(s):	**Name:**	**Type:**
	expression	floating point
Return Value Type:	floating point	
Return Value Range:	0.0 to ...	

This function returns the absolute value of the expression or numeric value passed to it.

Example:

```
a = -3
print string(a) + " " + string(abs(a))
```

will display the following on the text output area

```
-3 3
```

acos – Return the Arc-cosine (14)

acos(*expression*)

Argument(s):	Name:	Type:
	expression	floating point
Return Value Type:	floating point	
Return Value Range:	0 to π	

The inverse cosine function *acos()* will return an angle measurement in radians for the specified cosine value.

asc – Return the Unicode Value for a Character (11)

asc(*expression*)

Argument(s):	Name:	Type:
	expression	string
Return Value Type:	integer	
Return Value Range:	0 to 65535	

The asc() function will extract the first character of the string *expression* and return the character's Unicode value.

Example:

```
# English
print asc("A")
# Russian
print asc("Ы")
```

will display:

```
65
1067
```

asin – Return the Arc-sine (14)

asin(*expression*)

Argument(s):	Name:	Type:
	expression	floating point
Return Value Type:	floating point	
Return Value Range:	$-\frac{1}{2}\pi$ to $\frac{1}{2}\pi$	

The inverse sine function *asin()* will return an angle measurement in radians for the specified sine value.

atan – Return the Arc-tangent (14)

atan(*expression*)

Argument(s):	Name:	Type:

	expression	floating point
Return Value Type:	floating point	
Return Value Range:	- ½ π to ½ π	

The inverse tangent function *atan()* will return an angle measurement in radians for the specified tangent value.

ceil – Round Up (14)

ceil(*expression*)

Argument(s):	**Name:**	**Type:**
	expression	floating point
Return Value Type:	integer	
Return Value Range:		

This function returns an equal or next highest integer value. This method will round up if necessary.

Example:

```
a = ceil(-3.14)
b = ceil(7)
print a
print b
print ceil(9.2)
```

will display the following on the text output area

```
-3
7
10
```

chr – Return a Character (11)

chr(*expression*)

Argument(s):	Name:	Type:
	expression	integer
Return Value Type:	string	

The chr() function will return a single character string that contains the letter or character that corresponds to the Unicode value in the *expression*.

Example:

```
print chr(34) + "In quotes." + chr(34)
```

will display:

```
"In quotes."
```

clickb- Return the Mouse Last Click Button Status (10)

clickb
clickb()

Return Value Type:	integer
Return Value Range:	0 to 7

Returns the state of the last mouse button or combination of buttons that was pressed. If multiple buttons were being pressed at a single time then the returned value will be sum of the button values that were pressed.

Button Value	**Description**
0	Returns this value when no mouse button has been pressed, since the last *clickclear* statement.
1	Returns this value when the “left” mouse button was pressed.
2	Returns this value when the “right” mouse button was pressed.
4	Returns this value when the “center” mouse button was pressed.

clickx- Return the Mouse Last Click X Position (10)

```
clickx
clickx()
```

Return Value Type:	integer
Return Value Range:	0 to *graphwidth()* - 1

Returns the x coordinate of the mouse pointer position on the graphics output window when the mouse button was last clicked.

clicky- Return the Mouse Last Click Y Position (10)

```
clicky
clicky()
```

Return Value Type:	integer
Return Value Range:	0 to *graphheight()* - 1

Returns the y coordinate of the mouse pointer position on the graphics output window when the mouse button was last clicked.

cos – Cosine (14)

```
cos(expression)
```

Argument(s):	**Name:**	**Type:**
	expression	floating point
Return Value Type:	floating point	
Return Value Range:	-1.0 to 1.0	

This function returns the cosine of the expression. The angle should be represented in radians. The result is approximate and may not exactly match expected results.

Example:

```
a = cos(pi/3)
print a
```

will display the following

```
0.5
```

currentdir – Current Working Directory (16)

currentdir
currentdir()

Return Value Type:	string

This function returns a string containing the full path of the application's working directory.

day – Return the Current System Clock – Day (9)

day
day()

Return Value Type:	integer
Return Value Range:	1 to 31

This function returns the current day of the month from the current system clock. It returns the day number from 1 to 28, 29, 30, or 31.

Example:

```
print day
```

On 8/23/2010 it will display the following

```
23
```

dbfloat – Get a Floating Point Value From a Database Set (19)

dbfloat(*column*)

Argument(s):	Name:	Type:
	column	integer
Return Value Type:	floating point	

Return a floating point (decimal value) from the specified column of the current row of the open recordset.

dbint – Get an Integer Value From a Database Set (19)

dbint(*column*)

Argument(s):	Name:	Type:
	column	integer

Return Value Type:	integer

Return an integer (whole number) from the specified column of the current row of the open recordset.

dbrow – Advance Database Set to Next Row (19)

```
dbrow
dbrow()
```

Return Value Type:	boolean

Function that advances the record set to the next row. Returns a true value if there is a row or false if we are at the end of the record set.

dbstring – Get a String Value From a Database Set (19)

```
dbstring(column)
```

Argument(s):	**Name:**	**Type:**
	column	integer
Return Value Type:	string	

Return a string from the specified column of the current row of the open recordset.

degrees – Convert a Radian Value to a Degree Value (14)

```
degrees(expression)
```

Argument(s):	Name:	Type:
	expression	floating point
Return Value Type:	floating point	

The degrees() function does the quick mathematical calculation to convert an angle in radians to an angle in degrees. The formula used is $degrees = radians / 2\pi \times 360$.

eof – Allow Program to Check for End Of File Condition (16)

```
eof
eof()
eof(filenumber)
```

Return Value Type:	Boolean
Return Value Range:	true or false

Returns a Boolean true if the open file pointer is at the end of the file. If file number parameter is not specified then file number zero (0) will be used.

exists – Check to See if a File Exists (16)

```
exists(filename)
exists filename
```

Argument(s):	Name:	Type:
	filename	string
Return Value Type:	Boolean	
Return Value Range:	true or false	

Returns a Boolean value of true if the file exists and false if it does not exist.

Example:

```
if not exists("myfile.dat") then goto fileerror
```

float – Convert a String Value to A Float Value (14)

```
float(expression)
```

Argument(s):	Name:	Type:
	expression	string or integer
Return Value Type:	floating point	

Returns a floating point number from either a string or an integer value. If the expression can not be converted to a floating point number the function returns a zero (0).

Example:

```
a$ = "1.234"
b = float(a$)
print a$
print b
```

will display:

```
1.234
1.234
```

floor – Round Down (14)

floor(*expression*)

Argument(s):	Name:	Type:
	expression	floating point
Return Value Type:	integer	

This function returns an equal or next lowest integer value. This method will round down if necessary.

Example:

```
a = floor(-3.14)
b = floor(7)
print a
print b
print floor(9.2)
```

will display:

```
-4
7
9
```

getcolor – Return the Current Drawing Color

```
getcolor
getcolor()
```

Return Value Type:	integer
Return Value Range:	0 to 16777215 or -1

Returns the RGB value of the current drawing color (set by the *color* statement). If the color has been set to CLEAR then this function will return a value of -1.

getslice – Capture Part of the Graphics Output

```
getslice(x, y, width, height)
```

Argument(s):	**Name:**	**Type:**
	x	integer
	y	integer
	width	integer
	height	integer
Return Value Type:	string	

This function returns a string of hexadecimal digits that represent the pixels in the rectangle specified in the parameters. The slice can then be placed back on the screen at it's original location or a new location with the *putslice* statement.

graphheight – Return the Height of the Graphic Display (8)

```
graphheight
graphheight()
```

Return Value Type:	integer
Return Value Range:	0 to ...

The **graphheight()** function will return the height, in pixels, of the current graphics output area.

graphwidth – Return the Width of the Graphic Display (8)

```
graphwidth
graphwidth()
```

Return Value Type:	integer
Return Value Range:	0 to ...

The **graphwidth()** function will return the width, in pixels, of the current graphics output area.

hour – Return the Current System Clock - Hour (9)

```
hour
hour()
```

Return Value Type:	integer
Return Value Range:	0 to 23

This function returns the hour part of the current system clock. It returns the hour number from 0 to 23. Midnight is represented by 0, AM times are represented by 0-11, Noon is represented as 12, and Afternoon (PM) hours are 12-23. This type of hour numbering is known as military time or 24 hour time.

Example:

```
print hour
```

will display at 3:27PM:

```
15
```

instr – Return Position of One String in Another (15)

```
instr(haystack, needle)
```

Argument(s):	**Name:**	**Type:**
	needle	string
	haystack	string
Return Value Type:	integer	
Return Value Range:	0 to length(haystack)	

Return the position of the string *needle* within the string *haystack*. If the *needle* does not exist in the *haystack* then the function will return 0 (zero).

Example:

```
print instr("Hello Jim, How are you?","Jim")
print instr("Hello Jim, How are you?","Bob")
```

will display:

```
7
0
```

int – Convert Value to an Integer (14)

int(*expression*)

Argument(s):	**Name:**	**Type:**
	expression	floating point or string
Return Value Type:	integer	

This function will convert a decimal number or a string into an integer value. When converting a decimal number it will truncate the decimal part and just return the integer part.

When converting a string value the function will return the integer value in the beginning of the string. If an integer value is not found, the function will return 0 (zero).

Example:

```
print int(9)
print int(9.9999)
print int(-8.765)
print int(“ 321 555 foo”)
print int(“I have 42 bananas.”)
```

will display:

```
9
9
-8
321
0
```

key – Return the Currently Pressed Keyboard Key (11)

```
key
key()
```

Return Value Type:	integer
Return Value Range:	0 to ...

Return the key code for the last keyboard key pressed. If no key has been pressed since the last call to the **key** function a zero (0) will be returned. Each key on the keyboard has a unique key code that typically is the upper-case Unicode value for the letter on the key.

lasterror – Return Last Error (18)

```
lasterror
lasterror()
```

Return Value Type:	integer
Return Value Range:	See error code listing in Appendix J

Returns the last runtime error number.

lasterrorextra – Return Last Error Extra Information(18)

```
lasterrorextra
lasterrorextra()
```

Return Value Type:	string

Returns statement specific “extra” information about the last runtime error.

lasterrorline – Return Program Line of Last Error (18)

```
lasterrorline
lasterrorline()
```

Return Value Type:	integer

Returns the line number in the program where the runtime error happened.

lasterrormessage – Return Last Error as String (18)

```
lasterrormessage
lasterrormessage()
```

Return Value Type:	string

Returns a string representing the last runtime error.

left – Extract Left Sub-string (15)

```
left(expression, length)
```

Argument(s):	**Name:**	**Type:**
	expression	string
	length	integer
Return Value	string	

Type:	

Returns a sub-string, the number of characters specified by length, from the left end of the string *expression*. If length is greater than the length of the string *expression* then the entire string is returned.

length – Length of a String (15)

```
length(expression)
```

Argument(s):	**Name:**	**Type:**
	expression	string
Return Value Type:	integer	

Returns the length of the string *expression* in characters.

lower – Change String to Lower Case (15)

```
lower(expression)
```

Argument(s):	**Name:**	**Type:**
	expression	string
Return Value Type:	string	

This function will return a string with the upper case characters changed to lower case characters.

Example:

```
print lower("Hello.")
```

will display:

```
hello.
```

mid – Extract Part of a String (14)

mid(*expression, start, length*)

Argument(s):	**Name:**	**Type:**
	expression	string
	start	integer
	length	integer
Return Value Type:	string	

Return a sub-string from somewhere on the middle of a string. The start parameter specifies where the sub-string begins (1 = beginning of string) and the length parameter specifies how many characters to extract.

minute - Return the Current System Clock - Minute (9)

minute
minute()

Return Value Type:	integer

Return Value Range:	0 to 59

This function returns the number of minutes from the current system clock. Values range from 0 to 59.

Example:

```
print minute
```

will display at 6:47PM:

```
47
```

month - Return the Current System Clock - Month (9)

```
month
month()
```

Return Value Type:	integer
Return Value Range:	0 to 11

This function returns the month number from the current system clock. It returns the month number from 0 to 11. January is 0, February is 1, March is 2, April is 3, May is 4, June is 5, July is 6, August is 7, September is 8, October is 9, November is 10, and December is 11.

Example:

```
dim months$(12)
```

```
months$ = {"Jan", "Feb", "Mar", "Apr", "May",
"Jun", "Jul", "Aug", "Sept", "Oct", "Nov", "Dec"}
print month + 1
print months$[month]
```

will display on 9/5/2008:

```
9
Sept
```

mouseb- Return the Mouse Current Button Status (10)

```
mouseb
mouseb()
```

Return Value Type:	integer
Return Value Range:	0 to 7

Returns the state of the mouse button or buttons being pressed. If multiple buttons are being pressed at a single time then the returned value will be sum of the button values being pressed.

Button Value	Description
0	Returns this value when no mouse button is being pressed.
1	Returns this value when the “left” mouse button is being pressed.
2	Returns this value when the “right” mouse button

	is being pressed.
4	Returns this value when the "center" mouse button is being pressed.

mousex- Return the Mouse Current X Position (10)

```
mousex
mousex()
```

Return Value Type:	integer
Return Value Range:	0 to *graphwidth()* - 1

Returns the x coordinate of the mouse pointer position on the graphics output window.

mousey- Return the Mouse Current Y Position (10)

```
mousey
mousey()
```

Return Value Type:	integer
Return Value Range:	0 to *graphheight()* -1

Returns the y coordinate of the mouse pointer position on the graphics output window.

netaddress – What Is My IP Address (20)

```
netaddress
netaddress()
```

Return Value Type:	string

Returns a string with the current IPv4 address of this computer. If there are multiple address assigned to this machine only the first one will be returned.

netdata – Is There Network Data to Read (20)

```
netdata
netdata()
netdata(socket)
```

Argument(s):	**Name:**	**Type:**
	socket	integer
Return Value Type:	boolean	

Returns true of there is data to be read from the specified network connection. If there is no data on the socket waiting then false will be returned. If the socket number is omitted the default socket number of zero (0) will be used.

netread – Read Data from Network(20)

```
netread
```

```
netread()
netread(socket)
```

Argument(s):	Name:	Type:
	socket	integer
Return Value Type:	string	

Reads the last packed received on the specified network connection. If there is no data on the socket waiting to be read the program will wait until a message is received. You may use the **netdata** function to detect of there is data waiting to be read. If the socket number is omitted the default socket number of zero (0) will be used.

pixel – Get Color Value of a Pixel

```
pixel(x, y)
```

Argument(s):	Name:	Type:
	x	integer
	y	integer
Return Value Type:	integer	
Return Value Range:	0 to 16777215 or -1	

Returns the RGB color of a single pixel on the graphics output window. If the pixel has not been set since the last **clg** statement or was set to transparent by drawing with the color CLEAR (-1) then this function will return -1.

radians – Convert a Degree Value to a Radian Value (16)

```
radians(expression)
```

Argument(s):	Name:	Type:
	expression	floating point
Return Value Type:	floating point	

The **radians** function does the quick mathematical calculation to convert an angle measured in degrees to an angular measure of radians. The formula used is $radians = degrees / 360 \times 2\pi$.

rand – Random Number (6)

```
rand
rand()
```

Return Value Type:	floating point
Return Value Range:	0.0 to 0.99999$\overline{9}$

This function returns a random decimal number between 0 and 1. To generate random integer values, convert to integer the product of rand and the desired integer value.

Example:

```
print rand
# display a number from 1 to 100
```

```
print int(rand*100)+1
```

will display something like:

```
0.35
22
```

read – Read a Token from the Currently Open File (16)

read
read()
read(*filenumber*)

Return Value Type:	string
Return Value Range:	

Read the next word or number (token) from a file. Tokens are delimited by spaces, tab characters, or end of lines. Multiple delimiters between tokens will be treated as one. If file number parameter is not specified then file number zero (0) will be used.

readline – Read a Line of Text from a File (16)

readline
readline()
readline(*filenumber*)

Return Value Type:	string

Return Value Range:	

Return a string containing the contents of an open file up to the end of the current line. If we are at the end of the file [eof() = *true*] then this function will return the empty string (“”). If file number parameter is not specified then file number zero (0) will be used.

rgb – Convert Red, Green, and Blue Values to RGB (12)

```
rgb(red, green, blue)
```

Argument(s):	**Name:**	**Type:**
	red	integer (0 to 255)
	green	integer (0 to 255)
	blue	integer (0 to 255)
Return Value Type:	integer	
Return Value Range:	0 to 16777215	

The rgb function returns a single number that represents a color expressed by the three color component values. Remember that color component values have the range from 0 to 255. RGB color is calculated by the formula $RGB = \mathrm{RED} \times 256^2 + \mathrm{GREEN} \times 256 + \mathrm{BLUE}$.

right – Extract Right Sub-string (15)

```
right(expression, length)
```

Syntax:		
Argument(s):	**Name:**	**Type:**
	expression	string
	length	integer
Return Value Type:	string	

Returns a sub-string, the number of characters specified by length, from the right end of the string *expression*. If length is greater than the length of the string *expression* then the entire string is returned.

second - Return the Current System Clock - Second (9)

```
second
second()
```

Return Value Type:	integer
Return Value Range:	0 to 59

This function returns the number of seconds from the current system clock. Values range from 0 to 59.

Example:

```
print hour + ":" + minute + ":" + second
```

will display at 5:23:56 PM:

```
17:23:56
```

sin – Sine (16)

sin(*expression*)

Argument(s):	Name:	Type:
	expression	floating point
Return Value Type:	floating point	
Return Value Range:	-1.0 to 1.0	

This function returns the sine of the expression. The angle should be represented in radians. The result is approximate and may not exactly match expected results.

Example:

```
a = sin(pi/3)
print string(a)
```

will display

```
0.87
```

size – Return the size of the open file (15)

size
size()
size(filenumber)

Return Value Type:	integer
Return Value Range:	0 to ...

This function returns the length of an open file in bytes. If file number parameter is not specified then file number zero (0) will be used.

spritecollide – Return the Collision State of Two Sprites (12)

spritecollide(*expression1, exression2*)

Argument(s):	**Name:**	**Type:**
	expression1	integer
	expression2	integer
Return Value Type:	boolean	

This function returns true of the two sprites collide with or overlap each other. The collision detection is done by

spriteh – Return the Height of Sprite (12)

spriteh(*expression*)

Argument(s):	**Name:**	**Type:**
	expression	integer
Return Value	integer	

Type:	
Return Value Range:	0 to ...

This function returns the height, in pixels, of a loaded sprite. Pass the sprite number in expression.

Spritev – Return the Visible State of a Sprite (12)

```
spritev(expression)
```

Argument(s):	**Name:**	**Type:**
	expression	integer
Return Value Type:	boolean	

This function returns a true value if a loaded sprite is currently displayed on the graphics output area. Pass the sprite number in expression.

spritew – Return the Width of Sprite (12)

```
spritew(expression)
```

Argument(s):	**Name:**	**Type:**
	expression	integer
Return Value Type:	integer	
Return Value Range:	0 to ...	

This function returns the width, in pixels, of a loaded sprite. Pass the sprite number in expression.

spritex – Return the X Position of Sprite (12)

```
spritex(expression)
```

Argument(s):	Name:	Type:
	expression	integer
Return Value Type:	integer	
Return Value Range:	0 to ...	

This function returns the position on the x axis of the center, in pixels, of a loaded sprite. Pass the sprite number in expression.

spritey – Return the Y Position of Sprite (12)

```
spritey(expression)
```

Argument(s):	Name:	Type:
	expression	integer
Return Value Type:	integer	
Return Value Range:	0 to ...	

This function returns the position on the y axis of the center, in pixels, of a loaded sprite. Pass the sprite number in expression.

string – Convert a Number to a String (14)

string(*expression*)

Argument(s):	**Name:**	**Type:**
	expression	floating point or integer
Return Value Type:	string	

Returns a string representation of an integer or floating point number.

Example:

```
a = 1.234
b$ = string(a)
print a
print b$
```

will display:

```
1.234
1.234
```

tan – Tangent (16)

tan(*expression*)

Argument(s):	**Name:**	**Type:**
	expression	floating point

Return Value Type:	floating point

This function returns the tangent of the expression. The angle should be represented in radians. The result is approximate and may not exactly match expected results.

Example:

```
a = tan(pi/3)
print string(a)
```

will display:

```
1.73
```

upper – Change String to Upper Case (15)

upper(*expression*)

Argument(s):	Name:	Type:
	expression	string
Return Value Type:	string	

This function will return a string with the lower case characters changed to upper case characters.

Example:

```
print upper("Hello.")
```

will display:

```
HELLO.
```

year - Return the Current System Clock - Year (9)

```
year
year()
```

Return Value Type:	integer

This function returns the year part the current system clock. It returns the full 4 digit Julian year number.

Example:

```
print year
```

will display on 1/3/2009:

```
2009
```

Appendix D: Language Reference – Operators and Constants

Mathematical Operators:

Mathematical operators take one or more numeric values, do something, and return a number.

+ - Adds Two Numbers or Concatenates Two Strings (1)

- - Subtracts Two Numbers (1)

*** - Multiplies Two Numbers (1)**

/ - Divides Two Numbers (1)

% - Returns the Remainder of Integer Division of Two Numbers (13)

\ - Integer Division (14)

^ - Exponent (14)

() - Groups Operators (1)

Mathematical Constants or Values:

A mathematical constant is sort of like a variable. It returns a predefined value so that you do not need to remember what it is.

Constant:	***Value:***
pi	3.141593

Color Constants or Values:

BASIC-256 also includes a list of constants defining a simple pallet of colors. The color constants are integers that represent the RGB value required to draw that color on the screen.

Constant:	***Value:***	***Same as:***
black	0	rgb(0, 0, 0)
white	16,316,664	rgb(248, 248, 248)
red	16,711,680	rgb(255, 0, 0)
darkred	8,388,608	rgb(128, 0, 0)
green	65,280	rgb(0, 255, 0)
darkgreen	32,768	rgb(0, 128, 0)
blue	255	rgb(0, 0, 255)
darkblue	128	rgb(0, 0, 128)
cyan	65,535	rgb(0, 255, 255)
darkcyan	32,896	rgb(0, 128, 128)
purple	16,711,935	rgb(255, 0, 255)
darkpurple	8,388,736	rgb(128, 0, 128)
yellow	16,776,960	rgb(255, 255, 0)
darkyellow	8,421,376	rgb(128, 128, 0)
orange	16,737,792	rgb(255, 102, 0)
darkorange	11,154,176	rgb(170, 51, 0)
gray /grey	10,790,052	rgb(164, 164, 164)
darkgray / darkgrey	8,421,504	rgb(128, 128, 128)

clear	-1	

Logical Operators:

Logical operators return a true/false value that can then be used in the IF statement. They are used to compare values or return the state of a condition in your program.

= - Test if Two Values are Equal (6)

<> - Test if Two Values are Not Equal (6)

< - Test if One Value is Less Than Another Value (6)

<= - Test if One Value is Less Than or Equal Another Value (6)

> - Test if One Value is Greater Than Another Value (6)

>= - Test if One Value is Greater Than or Equal Another Value (6)

and – Returns True if Both Values are True (6)

not – Changes True to False and False to True (6)

or – Returns True if One or Both Values are True (6)

Logical Constants or Values:

A logical constant is sort of like a variable. It returns a predefined value so that you do not need to remember what it is. You can not change a constant's value in your program.

Constant:	***Value:***	***Notes:***
true	1	Represents a true event with the number one.
false	0	A false condition is expressed with the integer zero.

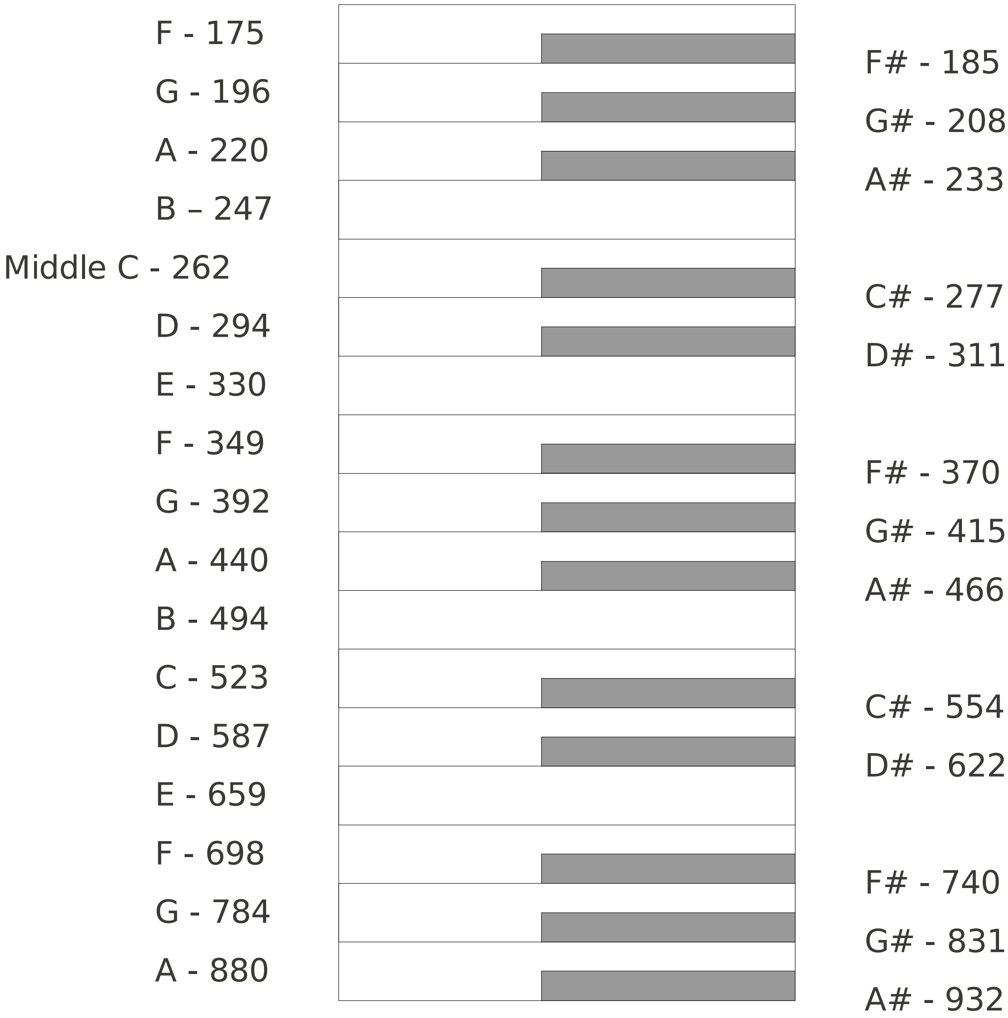
Appendix E: Color Names and Numbers

Listing of standard color names used in the *color* statement. The corresponding RGB values are also listed.

Color	RGB Values	Swatch
black	0, 0, 0	
white	255, 255, 255	
red	255, 0, 0	
darkred	128, 0, 0	
green	0, 255, 0	
darkgreen	0, 128, 0	
blue	0, 0, 255	
darkblue	0, 0, 128	
cyan	0, 255, 255	
darkcyan	0, 128, 128	
purple	255, 0, 255	
darkpurple	128, 0, 128	
yellow	255, 255, 0	
darkyellow	128, 128, 0	
orange	255, 102, 0	
darkorange	176, 61, 0	
gray /grey	160, 160, 164	
darkgray / darkgrey	128, 128, 128	
clear		

Appendix F: Musical Tones

This chart will help you in converting the keys on a piano into frequencies to use in the **sound** statement.



Appendix G: Key Values

Key values are returned by the *key()* function and represent the last keyboard key pressed since the key was last read. This table lists the commonly used key values for the standard English keyboard. Other key values exist.

English (EN) Keyboard Codes										
Key	**#**		**Key**	**#**		**Key**	**#**		**Key**	**#**
Space	32		A	65		L	76		W	87
0	48		B	66		M	77		X	88
1	49		C	67		N	78		Y	89
2	50		D	68		O	79		Z	90
3	51		E	69		P	80		ESC	16777216
4	52		F	70		Q	81		Backspace	16777219
5	53		G	71		R	82		Enter	16777220
6	54		H	72		S	83		Left Arrow	16777234
7	55		I	73		T	84		Up Arrow	16777235
8	56		J	74		U	85		Right Arrow	16777236
9	57		K	75		V	86		Down Arrow	16777237

Appendix H: Unicode Character Values – Latin (English)

This table shows the Unicode character values for standard Latin (English) letters and symbols. These values correspond with the ASCII values that have been used since the 1960's. Additional character sets are available at http://www.unicode.org.

CHR	#	CHR	#	CHR	#	CHR	#	CHR	#	CHR	#
NUL	0	SYN	22	,	44	B	66	X	88	n	110
SOH	1	ETB	23	-	45	C	67	Y	89	o	111
STX	2	CAN	24	.	46	D	68	Z	90	p	112
ETX	3	EM	25	/	47	E	69	[	91	q	113
ET	4	SUB	26	0	48	F	70	\	92	r	114
ENQ	5	ESC	27	1	49	G	71	]	93	s	115
ACK	6	FS	28	2	50	H	72	^	94	t	116
BEL	7	GS	28	3	51	I	73	_	95	u	117
BS	8	RS	30	4	52	J	74	`	96	v	118
HT	9	US	31	5	53	K	75	a	97	w	119
LF	10	Space	32	6	54	L	76	b	98	x	120
VT	11	!	33	7	55	M	77	c	99	y	121
FF	12	“	34	8	56	N	78	d	100	z	122
CR	13	#	35	9	57	O	79	e	101	{	123
SO	14	$	36	:	58	P	80	f	102	|	124
SI	15	%	37	;	59	Q	81	g	103	}	125
DLE	16	&	38	<	60	R	82	h	104	~	126
DC1	17	'	39	=	61	S	83	i	105	DEL	127
DC2	18	(	40	>	62	T	84	j	106		
DC3	19	)	41	?	63	U	85	k	107		
DC4	20	*	42	@	64	V	86	l	108		
NAK	21	+	43	A	65	W	87	m	109		

0-31 and 127 are non-printable.
Adapted from the Unicode Standard 5.2 – Available from http://www.unicode.org/charts/PDF/U0000.pdf

Appendix I: Reserved Words

These are the words that the BASIC-256 language uses to perform various tasks. You may not use any of these words for variable names or labels for the GOTO and GOSUB statements

```
#
abs
acos
and
asc
asin
atan
black
blue
ceil
changedir
chr
circle
clear
clg
clickb
clickclear
clickx
clicky
close
cls
color
colour
cos
currentdir
cyan
darkblue
darkcyan
darkgray
darkgrey
darkgeeen
darkorange
darkpurple
darkred
darkyellow
day
dbclose
dbcloseset
dbexecute
dbfloat
dbint
dbopen
dbopenset
dbrow
dbstring
decimal
degrees
dim
do
else
end
endif
endwhile
eof
exists
false
fastgraphics
float
floor
font
for
getcolor
getslice
gosub
goto
graphheight
graphsize
graphwidth
gray
grey
green
hour
if
imgload
input
instr
int
key
kill
lasterror
lasterrorextra
lasterrorline
lasterrormessage
left
length
line
log
log10
lower
mid
minute
month
mouseb
mousex
mouseynetaddress
netclose
netconnect
netdata
netlisten
netread
netwritenext
not
offerror
open
onerror
or
orange
pause
```

pi
pixel
plot
poly
print
purple
putslice
radians
rand
read
readline
rect
red
redim
refresh
rem
reset
return
rgb
right
say
second
seek
sin
size
sound
spritecollide
spritedim
spriteh
spritehide
spriteload
spritemove
spriteplace
spriteshow
spriteslice
spritev
spritew
spritex
spritey
stamp
step
string
system
tan
text
then
to
true
until
upper
volume
wavplay
wavstop
wavwait
while
white
write
writeline
xor
year
yellow

Appendix J: Error Numbers

Error #		Error Description (EN)
0	ERROR_NONE	
1	ERROR_NOSUCHLABEL	“No such label”
2	ERROR_FOR1	“Illegal FOR – start number > end number”
3	ERROR_FOR2	“Illegal FOR – start number < end number”
4	ERROR_NEXTNOFOR	“Next without FOR”
5	ERROR_FILENUMBER	“Invalid File Number”
6	ERROR_FILEOPEN	“Unable to open file”
7	ERROR_FILENOTOPEN	“File not open.”
8	ERROR_FILEWRITE	“Unable to write to file”
9	ERROR_FILERESET	“Unable to reset file”
10	ERROR_ARRAYSIZELARGE	“Array dimension too large”
11	ERROR_ARRAYSIZESMALL	“Array dimension too small”
12	ERROR_NOSUCHVARIABLE	“Unknown variable”
13	ERROR_NOTARRAY	“Not an array variable”
14	ERROR_NOTSTRINGARRAY	“Not a string array variable”
15	ERROR_ARRAYINDEX	“Array index out of bounds”
16	ERROR_STRNEGLEN	“Substring length less that zero”
17	ERROR_STRSTART	“Starting position less than zero”
18	ERROR_STREND	“String not long enough for given starting character”
19	ERROR_NONNUMERIC	“Non-numeric value in numeric expression”
20	ERROR_RGB	“RGB Color values must be in the range of 0 to 255.”

21	ERROR_PUTBITFORMAT	“String input to putbit incorrect.”
22	ERROR_POLYARRAY	“Argument not an array for poly()/stamp()“
23	ERROR_POLYPOINTS	“Not enough points in array for poly()/stamp()“
24	ERROR_IMAGEFILE	“Unable to load image file.”
25	ERROR_SPRITENUMBER	“Sprite number out of range.”
26	ERROR_SPRITENA	“Sprite has not been assigned.”
27	ERROR_SPRITESLICE	“Unable to slice image.”
28	ERROR_FOLDER	“Invalid directory name.”
29	ERROR_DECIMALMASK	“Decimal mask must be in the range of 0 to 15.”
30	ERROR_DBOPEN	“Unable to open SQLITE database.”
31	ERROR_DBQUERY	“Database query error (message follows).”
32	ERROR_DBNOTOPEN	“Database must be opened first.”
33	ERROR_DBCOLNO	“Column number out of range.”
34	ERROR_DBNOTSET	“Record set must be opened first.”
35	ERROR_EXTOPBAD	“Invalid Extended Op-code.”
36	ERROR_NETSOCK	“Error opening network socket.”
37	ERROR_NETHOST	“Error finding network host.”
38	ERROR_NETCONN	“Unable to connect to network host.”
39	ERROR_NETREAD	“Unable to read from network connection.”
40	ERROR_NETNONE	“Network connection has not been opened.”
41	ERROR_NETWRITE	“Unable to write to network connection.”
42	ERROR_NETSOCKOPT	“Unable to set network socket options.”
43	ERROR_NETBIND	“Unable to bind network socket.”

44	ERROR_NETACCEPT	“Unable to accept network connection.”
45	ERROR_NETSOCKNUMBER	“Invalid Socket Number”
9999	ERROR_NOTIMPLEMENTED	“Feature not implemented in this environment.”

Appendix K: Glossary

Glossary of terms used in this book.

algorithm – A step-by-step process for solving a problem.

angle – An angle is formed when two line segments (or rays) start at the same point on a plane. An angle's measurement is the amount of rotation from one ray to another on the plane and is typically expressed in radians or degrees.

argument – A data value included in a statement or function call used to pass information. In BASIC-256 argument values are not changed by the statement or function.

array – A collection of data, stored in the computer's memory, that is accessed by using one or more integer indexes. See also numeric array, one dimensional array, string array, and two dimensional array.

ASCII – (acronym for American Standard Code for Information Interchange) Defines a numeric code used to represent letters and symbols used in the English Language. See also Unicode.

asynchronous – Process or statements happening at one after the other.

Boolean Algebra – The algebra of true/false values created by Charles Boole over 150 years ago.

Cartesian Coordinate System – Uniquely identify a point on a plane by a pair of distances from the origin (0,0). The two distances are measured on perpendicular axes.

column (database) – defines a single piece of information that will be common to all rows of a database table.

constant – A value that can not be changed.

data structure – is a way to store and use information efficiently in a computer system

database – An organized collection of data. Most databases are computerized and consist of tables of similar information that are broken into rows and columns. See also: column, row, SQL, and table.

degrees – A unit of angular measure. Angles on a plane can have measures in degrees of 0 to 360. A right angle is 90 degrees. See also angle and radians.

empty string – A string with no characters and a length of zero (0). Represented by two quotation marks (“”). See also string.

false – Boolean value representing not true. In BASIC-256 it is actually short hand for the integer zero (0). See also Boolean Algebra and true.

floating point number – A numeric value that may or may not contain a decimal point. Typically floating point numbers have a range of $\pm 1.7 \times 10^{\pm 308}$ with 15 digits of precision.

font – A style of drawing letters.

frequency – The number of occurrences of an event over a specific period of time. See also hertz.

function – A special type of statement in BASIC-256 that may take zero or more values, make calculations, and return information to your program.

graphics output area – The area on the screen where drawing is displayed.

hertz (hz) – Measure of frequency in cycles per second. Named for German physicist Heinrich Hertz. See also frequency.

integer – A numeric value with no decimal point. A whole number. Typically has a range of –2,147,483,648 to 2,147,483,647.

IP address – Short for Internet Protocol address. An IP address is a numeric label assigned to a device on a network.

label – A name associated with a specific place in the program. Used for jumping to with the **goto** and **gosub** statements.

list – A collection of values that can be used to assign arrays and in some statements. In BASIC-256 lists are represented as comma (,) separated values inside a set of curly-braces ({}).

logical error – An error that causes the program to not perform as expected.

named constant – A value that is represented by a name but can not be changed.

numeric array – An array of numbers.

numeric variable – A variable that can be used to store integer or floating point numbers.

one dimensional array - A structure in memory that holds a list of data that is addressed by a single index. See also array.

operator – Acts upon one or two pieces of data to perform an action.

pixel – Smallest addressable point on a computer display screen.

point – Measurement of text – 1 point = 1/72”. A character set in 12 point will be 12/72” or 1/6” tall.

port – A software endpoint number used to create and communicate on a socket.

pseudocode – Description of what a program needs to do in a natural (non-computer) language. This word contains the prefix “pseudo” which means false and “code” for programming text.

radian - A unit of angular measure. Angles on a plane can have measures in radians of 0 to 2π. A right angle is $\pi/2$ degrees. See also angle and degrees.

radius – Distance from a circle to it's center. Also, ½ of a circle's diameter.

RGB – Acronym for Red Green Blue. Light is made up of these three colors.

row (database) – Also called a record or tuple. A row can be thought of as a single member of a table.

socket – A software endpoint that allows for bi-directional (2 way) network communications between two process on a single computer or two computers.

sprite – An image that is integrated into a graphical scene.

SQL – Acronym for Structured Query Language. SQL is the most widely used language to manipulate data in a relational database.

statement – A single complete action. Statements perform something and do not return a value.

string – A sequence of characters (letters, numbers, and symbols). String constants are surrounded by double quotation marks (“).

string array – An array of strings.

string variable – A variable that can be used to store string values. A string variable is denoted by placing a dollar sign ($) after the variable name.

sub-string – Part of a larger string.

subroutine – A block of code or portion of a larger program that performs a task independently from the rest of the program. A piece that can be used and re-used by many parts of a program.

syntax error – An error with the structure of a starement so that the program will not execute.

synchronous – Happening at the same time.

table (database) – Data organized into rows and columns. A table has a specific number of defined columns and zero or more rows.

transparent – Able to see through.

text output area – The area of the screen where plain text and errors is displayed.

true – Boolean value representing not false. In BASIC-256 it is actually short hand for the integer one (1). See also Boolean Algebra and false.

two dimensional array – A structure in memory that will hold rows and columns of data. See also array.

Unicode – The modern standard used to represent characters and symbols of all of the world's languages as integer numbers.

variable – A named storage location in the computer's memory that can be changed or varied.