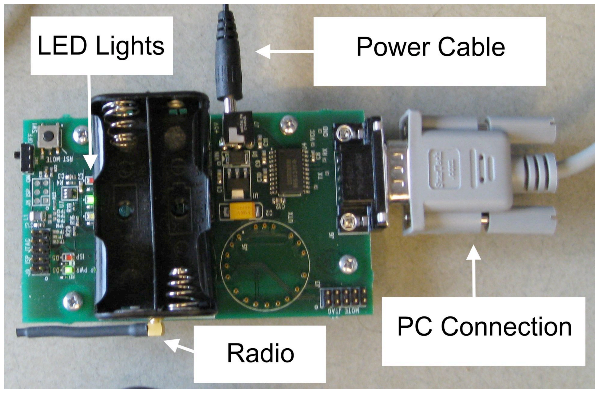
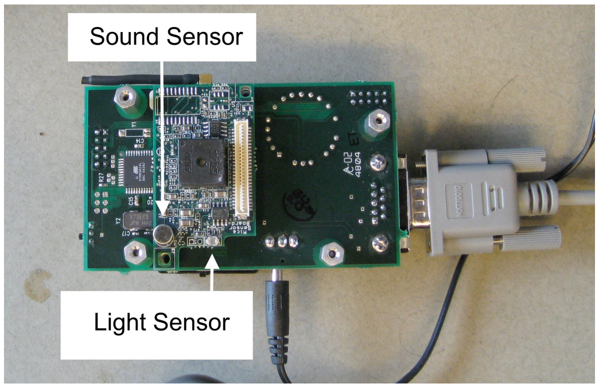
BASIC Tutorial

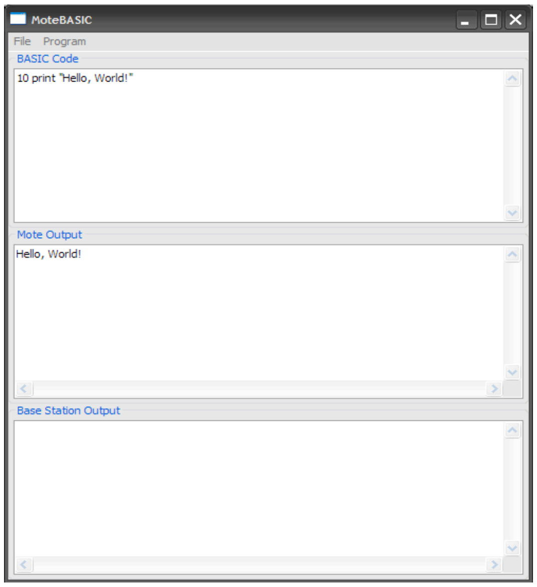
Introduction

In this exercise, you will be programming in a variant of the BASIC programming language designed for small sensors, known as “motes”. Below is an image of the Mote you will be using. On the top side of the mote, there are three lights colored red, yellow and green. On the underside of the mote, there are sensors for detecting light and sound. For this exercise, two motes will be directly connected to a PC.





Programming for the first time can be very challenging, and while BASIC is designed to make writing your first programs as easy as possible some concepts might not be immediately clear. You will be able to make a note of these instances at the end of this exercise.



```
10 print "Hello, World!"
```

```
Hello, World!
```

The programming environment.

The environment you will be using allows you to write and quickly test BASIC programs, with the result being immediately displayed. The programming environment (shown above) consists of three fields: “BASIC Code”, “Mote Output” and “Base Station Output”. Programs are entered in the “BASIC Code” field and are executed by clicking on the “Program” menu and selecting “Run”. Your program will begin executing immediately, displaying any output or errors in the “Mote Output” field. Try entering and running the following program:

```
10 print "Hello, World!"
```

After clicking run, the text “Hello World!” should appear in the “Mote Output” field. If the text does not appear, double check that you have entered in the correct program. If you are still having issues, please ask your proctor for help. If the text appears then you have successfully written your first BASIC program. Throughout this tutorial, you are strongly encouraged to try all of the provided examples.

In BASIC, each line represents a single operation. An operation might add two numbers, turn on a light on the mote, or send a message. A line in BASIC begins with a number that acts as the line's label.

```
10 a = 1 + 2
20 print a
```

This program consists of two lines: 10 and 20. Lines are executed one after another and should always be entered in increasing order. Entering the above lines in the opposite order (with 10 following 20) will make the program more difficult to understand and might have unintended consequences.

One of the most important operations for someone starting to program is `print`. In BASIC, `print` simply takes all of the values that follow it, transmits them to the PC via the large, beige cable and displays them in the “Mote Output” window. Printing text requires that the text be contained in quotes, while numbers, expressions, and variables can be printed without quotes. Multiple values can be printed by separating each value with a comma.

```
10 print "Two numbers:", 1, 2, "and their sum:", 1 + 2
```

Variables and Expressions
You might now be asking yourself, what is a variable and an expression? A variable in BASIC is a location for storing data, in this case a positive whole number in the range of 0 to 65535. There are 26 variables in BASIC, one for each letter of the alphabet. Values are assigned to variables by using the equal sign. For example, the following line assigns the value one to the variable “a” and prints out the value:

```
10 a = 1
20 print a
```

All data in BASIC can be manipulated using a number of mathematical operators in what is known as an expression. BASIC supports familiar operations including adding, subtracting, multiplying and dividing as well as several others.

Operator	Effect	Example	Result
+	Adds two numbers or variables	a = 1 + 2	a = 3
-	Subtracts two numbers or variables	a = 100 - 90	a = 10
*	Multiplies two numbers or variables	a = 5 * 3	a = 15
/	Divides two numbers of variables	a = 5 / 3	a = 1
%	Generates the remainder of dividing two numbers or variables	a = 5 % 3	a = 2
&	Combines two values with a logical and	a = 1 & 0	a = 0
|	Combines two values with a logical or	a = 1 | 0	a = 1

BASIC Operators.

Many of the operations work as you might expect, however when the numbers exceed the bounds allowed by the BASIC, the resulting values might seem random. For example, consider this program:

```
10 a = 500 * 500
20 print a
```

We would expect to display 250,000, however, because it is beyond BASIC's range, it will print something very different. The other caveat to math in BASIC has to do with division. Because BASIC can only understand whole number values, it is impossible to represent exact fractions. This will lead to some surprises for someone who has never programmed before. For example, try running this program:

```
10 a = 5 / 3
20 print a
```

You may have expected an output of something like one and two-thirds or the approximate decimal 1.6666666, but why just one? Division in BASIC is referred to as integer division which only supplies the quotient. It is possible to capture the remainder using the modulus operator (the percent sign).

```
10 a = 13
20 b = 10
30 print a, "divided by", b, "is", a / b, "remainder", a % b
```

Try writing a few simple programs until you feel comfortable with all of the operators.

Expressions need not be limited to a single operation. In BASIC, you can chain operators and group them using parenthesis. This line first multiplies 5 by 10 and adds 4:

```
10 a = 4 + 5 * 10
20 print a
```

Evaluation proceeds algebraically, with multiplication and division preceding addition and subtraction, with the above code resulting in 54. This precedence can be changed by grouping operations with a pair of parenthesis. Changing the program to the following results in an output of 90.

```
10 a = (4 + 5) * 10
20 print a
```

Special Variables
In BASIC there is one special variable that cannot be assigned to. That variable is called time and stores the current time on the mote. The time is given in seconds and can be accessed using the `time` keyword. You can access the time in any expression:

```
10 print "The time is:", time
20 print "In 1 minute it will be:", time+60
```

Arrays
It is often useful to collect multiple integer values within a single variable. For example, say you wanted to collect 1000 readings from a sensor and store their values. If you attempted to store each value in a variable, you would quickly run out of variables. Also, unless you used some convention such as storing the first value in 'a', the second in 'b', etc., it would be impossible to access the fifth or five-hundredth value. Arrays allow you to maintain large, ordered set of values.

Array variables are created using the `dim` keyword, which means "declare in memory." The dim keyword is followed by the desired array size enclosed in angle braces (i.e., "[" and "]"). For example, the following line creates an array that contains 10 elements.

```
10 dim a[10]
```

Array variables are ordered collections of elements. An element in an array is just like a variable. The elements of an array can be accessed by specifying the name of the array variable followed by the index of the value desired. The indices start at zero, so that the first element has an index of zero, the second has an index of one, etc. The following code creates an array, fills it with values, and then prints the entire array along with the first and last elements:

```
10 dim a[5]
20 a[0] = 10
30 a[1] = 20
40 a[2] = 30
50 a[3] = 40
60 a[4] = 50
70 print a
80 print "The first element is", a[0]
90 print "The last element is", a[4]
```

In the preceding example, the largest index you can access is 4. If you attempt to access an index outside the bounds of the array, that is, greater than or equal to the size of the array, the program will print an error.

Note that when an array is created, its elements contain random values. If you attempt to access a value in an array that has not been previously set, the behavior can be unpredictable. For example, try the following program:

```
10 dim a[20]
20 print a
```

Functions
There are common sets of functionality that are shared by a large number of programs. This common functionality is placed into special functions that all applications can access. The following code illustrates the available functions:

```
10 dim a[3]
20 a[0] = 1
30 a[1] = 2
40 a[2] = 3
50 print a
60 print "The minimum is", minimum(a)
70 print "The maximum is", maximum(a)
80 print "The average is", average(a)
```

This code prints out the minimum, maximum, and average of a set of three numbers. To use a function, specify its name, with any arguments to the function enclosed in parenthesis after the function name. Arguments are the data that the function uses or operates over, and functions that take multiple arguments have their arguments separated by commas. Functions can be used anywhere within an expression. The following functions are available in the BASIC interpreter.

Name	Arguments	Description
average	1 array	Returns the average value in the array
minimum	1 array	Returns the minimum value in the array
maximum	1 array	Returns the maximum value in the array

Sensors
Beyond expressions and functions, another source of data is through each mote's built in sensors. The mote you will be using supports reading sound, light, temperature and acceleration values. You can read a sensor input into a variable using the `sense` keyword.

```
10 sense temp into a
20 print a
```

This code takes a temperature measurement and places the value in the “a” variable and then prints that value. The keyword following the `sense` keyword specifies which sensor is to be used, and you can find a list of the sensor mappings in following table.

Sensor keyword	Sensor type
light	Light
temp	Temperature
mic	Sound level
accx	X-axis acceleration
accy	Y-axis acceleration

It is also possible to sense a large number of values directly into in an array. To do this, you must specify a number of extra keywords. Try the following example:

```
10 dim a[5]
20 sense light into a at 10 hz for 5 samples
30 print a
```

This code will print out 5 samples from the light sensor, collected at a rate of 10 hertz, that is, each sample is separated by 100 milliseconds. The first part of the sense statement is the same as above. However, following the variable is the `at` keyword, followed by a number specifying the desired sample rate, given in hertz, followed by `hz`. This is followed by the `for` keyword, then the number of samples desired, followed by the `samples` keyword. Note that the program will produce an error if you try to collect more samples than will fit in the array.

Logic

An important part of any computer program is the ability to perform different actions based on certain conditions. For example, a web-based email service will only let you see your inbox if you enter the correct password, otherwise it might ask you to re-enter your password. At the heart of this is a simple operation comparing two password values that reflects much of the logic used by a programmer to control his or her program.

In BASIC, this kind of control is handled by the if-then statement and is the first statement we have encountered that allows us to control the flow of our program. A typical if-then statement looks like this:

```
10 a = 9
20 if a > 10 then print "greater than 10"
30 if a < 10 then print "less than 10"
```

First things first, what do we mean change the flow? In this statement we conditionally execute two BASIC statements, `print "greater than 10"` and `print "less than 10"`. The first decision of what to execute is based on the code `a > 10`, which is two expressions (a and 10) combined with a relationship operator (>). The table on the next page shows all of the available relationship operators.

Symbol	Operation on a (operator) b
>	True if a is greater than b, false otherwise
<	True if a is less than b, false otherwise
=	True if a equals b, false otherwise

Because the relationship operators work between expressions, it is possible to make more complex logical expressions, for example:

```
10 a = 5
20 if (a % 2) = 0 then print "even"
30 if (a % 2) = 1 then print "odd"
```

Lines 20 and 30 use the modulus operator to check to see if there is any remainder after dividing the variable "`a`" by two. If there is no remainder, then "`a`" is decided to be even, otherwise it is odd.

Delay

For many applications in which sensor motes are used it is often necessary to stop program execution for some fixed amount of time. For example, say you want to take several readings, each one second apart and report the average. With the `sleep` statement, you can do exactly this.

```
10 sense temp into a
20 sleep 1000 ms
30 sense temp into b
40 c = (a + b)/2
50 print "Average temperature is", c
```

The `sleep` keyword is followed by an expression indicating how long to sleep, followed by the units of the sleep time, in this case 1000 milliseconds. The available units are milliseconds (`ms`), seconds (`sec`), minutes (`min`), and hours (`hr`). When the mote is asleep, it uses substantially less power because it does not need to process instructions. Sensor applications can use this fact to reduce power consumption when it is not necessary to meet an immediate deadline, and many applications can tolerate some delays in reading data.

Lights

The sensor mote used in this tutorial has three LED lights attached to it, with one colored red, yellow and green. The lights can be illuminated using the `led` keyword.

```
10 led red 1
```

```
20 sleep 5 sec
```

This program turns on the red LED for five seconds. The `led` statement is followed by the color of the LED followed by one expression (in this case, one). The possible colors are `red`, `green`, and `yellow`. The second expression determines whether the light is turned on or off. If the expression results in 0, the light is turned off. If the expression is greater than 0, the light is turned on.

Using expressions to determine whether or not to turn on the lights allows us to control the lights with more complex logic. When your application is finished, any lights enabled during your program will be turned off. If you omit the `sleep` line above, the light will not appear to turn on because the application will quickly end.

GOTO and Loops

At this point, you should feel comfortable reading a value from a sensor, displaying that value on the screen and perhaps blinking the lights based on that value. While this is fundamentally not so different from how real sensor code operates, it has one serious drawback: we cannot repeat our program or reuse parts of it. In the programs we have seen so far, each line is executed one right after another and the program terminates after all of the lines are completed. We can change this by introducing a new statement to our language: `goto`.

If you were wondering why each line requires its own, increasing number, `goto` should provide an explanation. The `goto` statement allows your program to jump from one line to another. For example, the following program uses `goto` to skip the execution of a line.

```
10 print "this line is executed"
20 goto 40
30 print "this line is skipped"
40 print "this line is also executed"
```

With `goto`, there is a new potential danger in our programs: they might never finish execution. If in the preceding code we had typed `goto 10` instead of `goto 40`, the mote would endlessly execute line 10 until the program was forced to stopped, either by either selecting “Stop” from the “Program” menu or resetting the mote. However, we can use this functionality to our advantage.

BASIC gives us another means of controlling our programs: for loops. A for loop allows you to execute a number of lines of code a specified number of times.

```
10 s = 0
20 for i = 1 to 10
30 print i
40 s = i + s
50 next i
60 print "sum is", s
```

This program uses a for loop print all of the values between one and ten and their sum (55). The program works by first clearing out the variable s, which will be used to accumulate our sum. The next line is perhaps the most important. The `for` keyword is followed by the iteration variable and that variable's initial value, in this case one, the `to` keyword and finally the terminating value, ten. All of the lines following the `for` statement up until the `next` statement are considered to be the body of the for loop. The body of the for loop is executed until the value of the iteration variable is equal to the terminating value. After each iteration, that is, at the `next` statement, the value of the iteration variable is increased by one. Both the initial value of the iteration variable and the terminating value can be initialized using expressions, allowing for more complex loops.

SLEEP Loop

A common pattern in sensor network applications is to sleep most of the time, waking up periodically to perform some computation, such as sensing a value. For this operation, there is a special BASIC control structure. Consider the following program:

```
10 sleep period 3 sec
20 print "I’m awake!"
30 resume
```

This program causes the mote to print out “I’m awake!” every three seconds. The `sleep` statement has a new keyword, `period`, followed by the usual time duration. The addition of the `period` statement causes the sleep statement to repeat any code between the `sleep` statement and the next `resume` statement at the time interval specified.

The sleep loop makes sure that the code between the sleep and resume statements is executed exactly once in the period specified. In the previous example, the first time the code executes, line 20 is immediately executed followed by line 30 which causes the program to sleep, then line 20, etc. If the period is set to 3000 milliseconds and the code takes 500 milliseconds to execute, the mote will sleep for 2500 milliseconds after the code executes, so that the total time between the start of two subsequent executions of the code is 3000 milliseconds.

If the mote is awake longer than the sleep period, then “Warning: Sleep period exceeded” Will be displayed in “Mote Output” window. For example, the code in the following program will generate the error:

```
10 sleep period 1 sec
20 sleep 2 sec
30 resume
```

In this example, the sleep loop attempts to execute once every second. However, the sleep statement within the loop causes the mote to stop executing code for 2 seconds. Thus, by the time the resume statement is executed, it has been 2 seconds since the sleep loop started. In this case, the warning message with be displayed and the loop will immediately execute again.

It is also possible to make the program sleep until a special external sensor passes a given threshold. The special hardware that is currently not connected to the mote. For now, do not worry about the actual hardware implementation, just know that they can be generated at any time. In BASIC, you can treat the interrupt as a method of waking up the sleeping mote. For example, the following code will print “Woken up!” on the first execution, then only wakeup when an interrupt occurs:

```
10 sleep channel 1 thresh 512
20 print "Woken up!"
30 resume
```

In this example, the code will execute the print statement whenever an interrupt occurs on channel 1. The syntax of the sleep statement is slightly different when waiting on an interrupt. Following the `sleep` keyword is the `channel` keyword, followed by the channel number. Interrupts can occur on several different channels so it is necessary to designate the channel number. Finally, it is not sufficient to specify only the channel number; you must also indicate the threshold for that interrupt. In this example, the mote is waiting for a signal of 512.

You can also mix interrupts and periodic sleep. For example, the following code will print out “Woken up!” every five seconds or when an interrupt is triggered:

```
10 sleep period 5 sec or channel 1 thresh 512
20 print “Woken up!”
30 resume
```

To wait on either condition, separate the sleep period and interrupt channel specification by the `or` keyword.

GOSUB

Another mechanism for controlling flow is `gosub`. The `gosub` statement functions almost identically to `goto` with one distinction: it is possible to return from where the `gosub` statement directed execution using the `return` keyword. Consider the following example.

```
10 print "Executing line 10"
20 gosub 50
30 print "Executing line 30"
40 end
50 print "Executing line 50"
60 return
```

There are a few new concepts here. First, notice from the output that line 50 is executed after line 10 but before line 30. This is because we have directed the program to jump from 20 to line 50 with a `gosub` statement, as though we had used a `goto`. The distinction occurs on line 60, where we issue the `return` statement, which causes the application to execute the line following the last call to `gosub`. Also, notice that after line 40, we do not execute line 50 a second time. This is because we use the `end` statement to immediately halt our program. With `gosub`, it is possible to call the same part of an application from different locations. For example, the following program uses gosub to double the value in variable i from different locations in the code:

```
10 i = 5
20 gosub 80
30 print i
40 i = 30
50 gosub 80
60 print i
70 end
80 i = i * 2
90 return
```

Communication

A fundamental aspect of sensor motes is the way in which they communicate the data they collect. In our BASIC environment, we provide one such means of communication with the `send` statement. The send function will wirelessly transmit the value of a variable or expression to the base station. The base station is effectively a mote that is always listening for data coming in wirelessly form other motes. You can see all of the data coming into the base station in the “Base Station Output” window. The following code should display the sequence of numbers from 1 to 3 in the “Base Station Output” window:

```
10 dim a[3]
20 a[0] = 1
30 a[1] = 2
40 a[2] = 3
50 send a
```

Not that the send statement will only send the value of a single variable or expression. Unlike print, which is sent from the cable directly connected to the mote to the computer, send transmits the sensor value wirelessly from your mote to another mote directly connected to the computer. However, just as your cell phone occasionally loses signal or drops calls, the wireless communication is not always reliable, unlike the print statement. Also, it can take some time to send all of the data back to the base station. For example, it takes roughly 20-30 seconds to send 1000 values.