OS/2 Programming for the Complete Novice in Liberty BASIC

It may sound incredible, but anyone can program for OS/2 Presentation Manager nowadays. Creating software for IBM's popular OS/2 software was once the domain of the elite C and assembly language programmer. But finally, after years of waiting, tools have arrived for the computer non-guru. This tutorial is written about one of these tools, Liberty BASIC, and about how to make it work for you.

BASIC (Beginners All purpose Symbolic Instruction Code) was created in the 1960's as an easy to learn programming language for computers. Because of BASIC's simple form and because it was an interpreted language and gave the programmer instant feedback, it became the most popular programming language when microcomputers made their debut, and it has a large following even today.

What is Liberty BASIC?

Liberty BASIC is a shareware programming tool that brings BASIC's ease of use to OS/2.

Liberty BASIC includes:

A powerful BASIC language for OS/2 ;
An editor for writing BASIC programs ;
An easy to use tracing debugger ;
A programmable spreadsheet ;
Color graphics capability

This tutorial is meant to be used with the shareware versions of Liberty BASIC. The examples here will also work with the registered versions of these Liberty BASIC with little or no modification.

Overview of this Tutorial

This tutorial was created with Windows Write, the simple word processor that comes with Microsoft Windows (sorry, but OS/2's system editor doesn't cut it). If you want a paper copy just press [Alt] [F] and then [P] to send it to your printer.

This tutorial covers:

Programming: What is it? ;
An Introduction to BASIC ;
GOTO - Doing something more than once ;
IF . . . THEN - Adding smarts to our tax program ;
String Variables ;
Some things to do with Strings ;
Functions ;
Documenting BASIC Code ;
Let's write a program - HILO.BAS ;
Where can I get of copy of Liberty BASIC?

This tutorial is not an operations manual for Liberty BASIC. You should refer to the documentation included with Liberty BASIC for detailed instructions for a complete list of procedures and commands.

Programming: What is it?

There's nothing mystical about programming computers. Although the newest software on the market today begins to look like magic, all software is built from the ground up out of combinations of the simplest software parts. Once you learn what these software parts are and how they're used, hard work and imagination can take you almost anywhere.

Programming is (simply put) the laying out of simple steps to solve a problem, and in a way that a computer can understand. This is a little bit like teaching a person. These steps must be arranged in the correct order. For example:

How to drive a car with automatic transmission:

Get into drivers seat ;
Fasten safety belt ;
Insert ignition key and turn it to start engine ;
Press brake with foot ;
Move transmission selection to D ;
Look around to see if you're safe ;
Remove foot from brake ;
Press accelerator pedal with foot ;
Manuever into traffic ;
Don't crash

Obviously if the above steps are scrambled up (and maybe even if they aren't) you're in for a pretty big insurance claim. Not only that, but if the instructions are given to someone who speaks only, say, Chinese, we will have a similarly spectacular crash! In the same way, computers are particular about both the order and content of the instructions we give them.

A program in its simplest form usually contains three kinds of activity:

INPUT	The program asks the user for some kind of information ;
CALCULATION	The program transforms or manipulates the information ;
OUTPUT	The program displays the final result of CALCULATION

It is the programmer's job to determine exactly how to accomplish these steps.

An Introduction to BASIC

- This tutorial introduces the first principles of Liberty BASIC, but doesn't provide a thorough description of all language features. For more on the full language and command set, refer to the documentation included with your copy of Liberty BASIC.

- You can save yourself some typing by copying the programming examples out of this tutorial and pasting them into Liberty BASIC to try them out.

Now let's create a very simple program to introduce you to the simplest of BASIC's features. We want a BASIC program that:

1 - Asks for a dollar and cent amount for goods ;
2 - Calculates a 5% sales tax amount for the dollar and cent figure ;
3 - Displays the tax amount the the total amount

INPUT - First we need an instruction for the computer that gets information from the user. In BASIC there are several ways to do this but we will choose the **input** command for our program. In this case **input** would be used like so:

```
input "Type a dollar and cent amount "; amount
                                        ^--------- this is a variable
```

This line of BASIC code would display the words **Type a dollar and cent amount ?** and the computer would then stop and wait for the user to type something in. When the [Enter] key is pressed, the typed information would then be stored in the variable* **amount**.

variable - In programming, you can give each bit of data (or information) a unique name. This combination of a name and its data is called a **variable because the data part can vary each time the program is used. When you type in a program, you choose a name for each variable. You pick the name for each variable to best fit what kind of data it represents. BASIC doesn't care what kind of name you give to your variables, but you should pick names that make it easy for you to understand what the BASIC program code means and does, especially if you expect to be reading the code some time later or sharing it with others. When running a program, BASIC uses the data part of the variable in its calculations and only uses the name part to fetch the data part or to store new data in that variable if it the value of the data changes. It isn't unusual for a variable to hold many different data values in the single execution of a program.*

CALCULATION - Now we need to calculate the tax for the data in our amount variable:

```
let tax = amount * 0.05
```

This line of code creates a new variable **tax** to hold our computed tax data. The BASIC command **let** tells BASIC to calculate the arithmetic on the right side of the = and set the data of the variable **tax** to be the result. The **let** word is optional (and most programmers leave it out) but I use it here to make it easier to see that the line of code shown is not an exercise in Algebra 101, rather that the expression on the right of the equal sign becomes the data for the variable on the left of it. It could have just as easily been coded:

```
tax = amount * 0.05
```

Now you may be wondering just what that funny little * (called asterisk) is. Since there are no formal arithmetic symbols on a typewriter keyboard, most programming languages use * to denote multiplication, / for division, and the addition and subtraction symbols get lucky and are + and - (what else?).

OUTPUT - Now that we have calculated our tax amount, we will display it with:

```
print "Tax is: "; tax; ". Total is: "; tax + amount
```

The **print** command displays information on the screen for the user to examine it. The line of code above shows how **print** is used to display several items of data, each seperated by a ; (semicolon).

The items are:

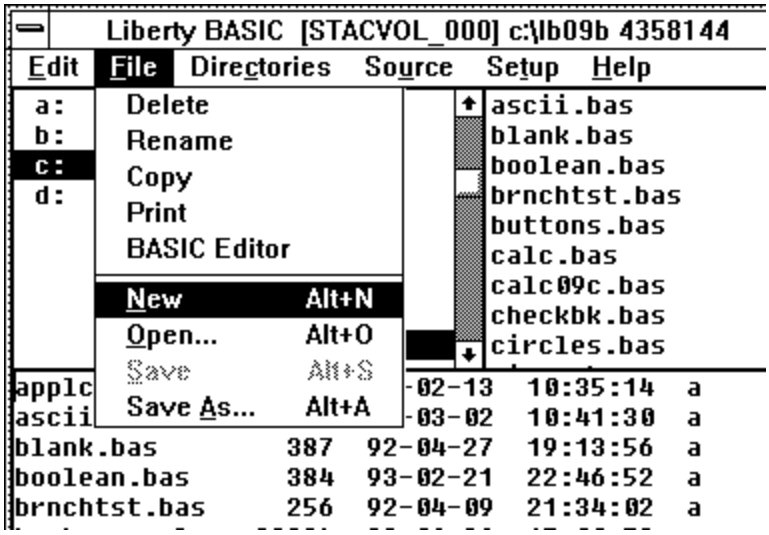
```
"Tax is: "          - This displays on screen as is, but without the quotation marks
;
tax                 - This displays the value of the variable tax ;
". Total is: "      - This also displays on screen as is, but without quotation marks ;
tax + amount        - This displays the sum of the two variables, tax and amount
```

These will all be displayed one after another on the same line. The semicolons are not displayed. After the

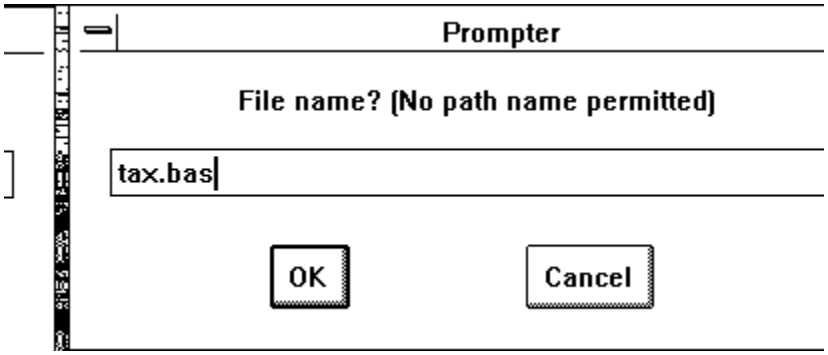
print command is done, the next **print** command will print below it. The result might look like this:

Tax is: 0.05. Total is: 1.05

Now let's run the program. Load Liberty BASIC, pull down the File menu and select the New item, like so:



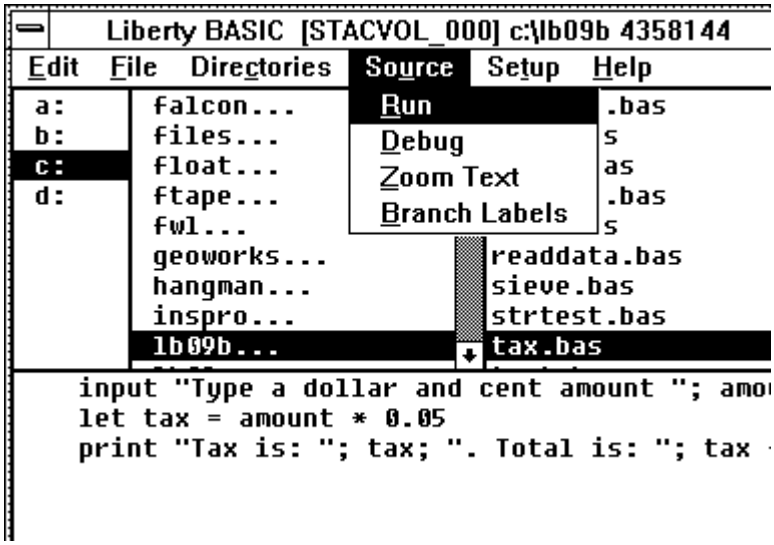
Then type tax.bas into the dialog box as below and press [Enter]:



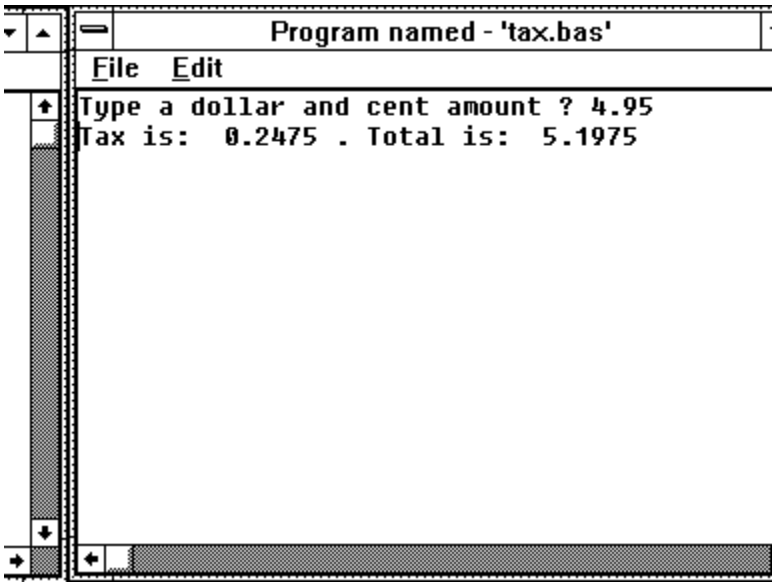
Here is the entire program. Paste this into Liberty BASIC and run it to see for yourself how it works.

```
input "Type a dollar and cent amount "; amount
let tax = amount * 0.05
print "Tax is: "; tax; ". Total is: "; tax + amount
```

Now run the program:



Here is a sample run:



GOTO - Doing something more than once

Assuming that tax.bas does what we want, it still only does it once. Each time you want to use this handy little program you have to run it again. This can get to be tedious and even error prone (say rubber baby buggy bumpers ten times fast). What we need is a way for our program to go to the beginning and do it over. In BASIC (and in some other languages) the command for doing this is called **goto** (surprise!).

Knowing that we have to **goto** some place is not enough. We also need to know where to go. When you hop into your car in a foreign country looking for a food market, you at least know what you are looking for. Liberty BASIC can't ask for directions, so you need to be very precise.

The mechanism that Liberty BASIC uses to mark places that we can **goto** is called a branch label. This is a lot like a mailing address. When you send a letter or package, you mark it with a known mailing address (hopefully). There is a house or building somewhere marked with that address, and that is where your parcel goes to. So in the same way, you mark the place in your BASIC program where you want it to continue running with a branch label (a mailing address of sorts).

There are two ways to define a branch label in Liberty BASIC. You can use any valid integer number as a branch label, or you can use an easier to remember type which uses letters.

Examples of integer branch labels:

10 150 75 900 5400 etc...

Examples of alphanumeric (using letters and numbers) branch labels:

[start] [loopBack] [getResponse] [point1] etc...

Examples of unacceptable branch labels:

[loop back]	no spaces allowed
start	must use brackets
(point1)	only use square brackets

Since no spaces are allowed in the alphanumeric branch labels, it works well to capitalize the first letter in each word when multiple words are used in a branch label. For example [gettimedresponse] is valid, but [getTimedResponse] is much more readable.

So let's pick a branch label for our tax.bas program. Since we are going to do it over again from the start, we could pick from several reasonable branch label names like perhaps [start], [begin], or [go]. We will use [start] for our program.

Let's add the branch label as shown:

```
[start]
  input "Type a dollar and cent amount "; amount
  let tax = amount * 0.05
  print "Tax is: "; tax; ". Total is: "; tax + amount
```

Now we need our goto line. Now that we have our branch label, the correct format for **goto** is goto [start]. And here's what our program looks like when both a branch label and a **goto**:

```
[start]
  input "Type a dollar and cent amount "; amount
  let tax = amount * 0.05
  print "Tax is: "; tax; ". Total is: "; tax + amount
  goto [start]
```

Now let's try running this program. It runs over and over and over, right? This programming format is called an unconditional loop because it **always** loops back to repeat the same code no matter what. When we are finished with it, we can close it like any other OS/2 program by double-clicking on the system menu box.

IF . . . THEN - Adding Smarts to Our Tax Program

The program we designed above will only do one thing for us, no frills. Let's learn how to add some smarts to our program. One way that this can be done is with the **if . . . then** statement.

The **if . . . then** statement is a direct descendant of those do-it-yourself style instruction manual texts. For example:

Problem: Your car's engine won't turn over
1) Check your battery for correct voltage.
2) If voltage is less then 11 volts then goto to step 13
3) Clean and tighten ground connection.
4) If this doesn't solve the trouble, continue to step 5.
5) Remove the starter.
6) Connect starter directly to battery
7) If starter does not spin then goto step 18
8) Check starter relay.
. . .
13) Charge battery
. . .
18) See chapter 4 on rebuilding the starter unit

Notice how in the above example how you are led smartly through the troubleshooting procedure. The steps containing the words **if** and **then** make it possible to intelligently work through a complex procedure. In the same way, the **if . . . then** statement in BASIC makes it possible to add a kind of intelligence to your programs.

Let's see how we can apply this. Suppose we want the computer to give us the option to display instructions about how to use our tax program. An easy way to add this ability would be to display instructions whenever a zero value is entered as our dollar amount.

Now whenever **input** is used to get a number from the user of a Liberty BASIC program, and if the user doesn't type a number but only presses the [Enter] key, then Liberty BASIC gives the variable for that **input** statement the value of zero. We can exploit this feature in our tax.bas program. By looking to see if the variable **amount** is zero after the **input** statement, we can decide whether or not to display instructions.

Here's what our new program would look like:

```
[start]
  print "Type a dollar and cent amount."
  input "(Press 'Enter' alone for help) "; amount
  if amount = 0 then goto [help]
  let tax = amount * 0.05
  print "Tax is: "; tax; ". Total is: "; tax + amount
  goto [start]

[help]
  print "This tax program determines how much tax is"
  print "due on an amount entered and also computes"
  print "the total amount.  The tax rate is 5%"
  goto [start]
```

Notice the line **if** amount = 0 **then** goto [help] in the program above. When Liberty BASIC comes to this line to execute it, it checks to see if the value of the variable **amount** is equal to 0. **If** it is, **then** the **goto [help]** statement in the line is executed, and so Liberty BASIC begins following the instructions after the **[help]** branch label. See how the **if . . . then** statement performs exactly what it means when read aloud?

Actually, the **goto** part of the **if . . . then** statement is optional. Either of these two forms is acceptable:

```
        if amount = 0 then goto [help]
- or -
        if amount = 0 then [help]
```

Comparing numbers - The = (equality) operator is only one of several that can be used to make decisions in an **if . . . then** statement. We can use the **if . . . then** statement and the (=, <>, <, >, <=, >=) operators to determine whether:

```
a = b       a is equal to b
a <> b      a is unequal to b
a < b   a is less than b
a > b       a is greater than b
a <= b      a is less than or equal to b
a >= b      a is greater than or equal to b
```

For example, instead of checking to see if **amount** was equal to 0 in the above program, we could have just as easily checked to see whether it was less than 0.01 (or one cent). For example:

```
if amount < 0.01 then goto [help]
```

When you run the program above you will probably notice that the things displayed sort of run together. There are things that we can do to neaten up the appearance of a BASIC program. We can add extra blank lines between our **print**ed output to break things up. This is done by using an empty **print** statement, one for each blank line. We can also clear the window at an appropriate time with the **cls** statement. Both of these techniques are applied to our tax program in the listing below:

```
[start]
  cls
  print "Type a dollar and cent amount."
  input "(Press 'Enter' alone for help) "; amount
  if amount = 0 then [help]
  let tax = amount * 0.05
  print "Tax is: "; tax; ". Total is: "; tax + amount
  goto [start]

[help]
  cls
  print "TAX.BAS Help"
  print
  print "This tax program determines how much tax is"
  print "due on an amount entered and also computes"
  print "the total amount.  The tax rate is 5%."
  print
  input "Press [Enter] to continue."; dummyVariable
  print
  goto [start]
```

Notice the line **input "Press [Enter] to continue."; dummyVariable** in the listing. In this example, we are simply using an **input** statement to put a halt on the program so that the instructions can be read. When [Enter] is pressed as instructed, **dummyVariable** receives the value of what is typed. In this case nothing is probably typed before pressing [Enter], so **dummyVariable** gets a value of zero for its data. It really doesn't matter what **dummyVariable's** data is since we don't use the variable in any calculations elsewhere (hence the name **dummyVariable**).

String Variables

So far, the only kind of variables we have used are for holding number values. There are special variables for holding words and other non-numeric character combinations. These variables are called *string* variables (they hold strings of characters*).

**Characters are:*
Letters of the alphabet ;
Digits 0123456789 ;
*Any other special symbols like: , . < > / ? ; : ' " [ ] { } ` ~ ! @ # $ % ^ & * ( ) + - \ | etc . . .*

Let's look at a very simple program using strings:

```
input "Please type your name "; name$
print "It's nice to meet you, "; name$
```

This two-line program asks you for your name. Once you've typed it and pressed [Enter], it says:

It's nice to meet you, *your-name-here*

Notice one special thing about our string variable name. It ends with a $ (dollar sign). In BASIC, when you want to store characters in a variable, you end the variable name with a $. This makes it a string variable. As you can see from our program example, you can both **input** and **print** with string variables, just as we did earlier with our non-string or **numeric** variables.

We've actually been using strings all along, even before this section about string variables. Whenever you saw a BASIC program line with words in quotes (for example: print "It's nice to meet you, ") you were looking at what is called a *string literal*. This is a way to directly express a string in a BASIC program, exactly the way we type numbers directly in, only with characters instead. A string literal **always** starts with a quotation mark and **always** ends with a quotation mark. No quotation marks are allowed in between the starting and ending quotation marks (point: string literals cannot contain quotation marks).

NOTE - A string can have zero characters. Such a string is often called an *empty string*. In BASIC, an empty string can be expressed in a string literal as two quotation marks without any characters between them. For example (noCharactersHere$ is the name of our string variable):

```
let noCharactersHere$ = ""
```

Some things to do with Strings

Just as you can manipulate numbers in a computer programming language by adding, subtracting, multiplying, and dividing, (and more!), we can also manipulate strings

Adding strings - We can add (or concatenate) two or more strings together in BASIC like so:

```
input "What is your first name "; firstName$
input "What is your last name "; lastName$
let fullName$ = firstName$ + " " + lastName$
print "Your full name is: "; fullName$
```

In this short program, we **input** two strings, your first and last name. Then we concatenate the string in **firstName$** with the string literal **" "** (a single space between two quotes) and with **lastName$**. The result is made the data for the string variable **fullName$**, which we then print out.

Comparing strings - We can compare strings with each other just as we can compare numbers. This means that we can use the **if . . . then** statement and the (=, <>, <, >, <=, >=) operators to determine whether:

```
a$ = b$        a$ is equal to b$
a$ <> b$       a$ is unequal to b$
a$ < b$        a$ is less than b$
a$ > b$        a$ is greater than b$
a$ <= b$       a$ is less than or equal to b$
a$ >= b$       a$ is greater than or equal to b$
```

When comparing strings, a string is considered to be equal to another string when all the characters in one string are *exactly* the same in both strings. This means that even if they both print the same onto the screen, they can still be unequal if one has an invisible space on the end, and the other doesn't. For example:

```
a$ = "Hello"
b$ = "Hello "
print "a$ is "; a$
print "b$ is "; b$
if a$ = b$ then goto [areTheSame]
print "a$ and b$ are not the same"
goto [end]
[areTheSame]
  print "a$ and b$ are the same"
[end]
```

When the line **if a$ = b$ then goto [same]** is performed, the result is not to **goto [same]**, because even though if a$ and b$ were printed they would *look* the same, they are not actually the same.

Functions

Now that we've covered bringing data into your programs with **input**, displaying data with **print**, keeping data in string and numeric variables, and controlling program flow with **if . . . then**, we will bring one more way to light to flesh out your programs. Functions provide a means for manipulating program data in meaningful ways.

Look this short program:

```
input "Please type your name "; name$
print "Your name is "; len(name$); " characters long."
```

The second line demonstrates the use of the **len( )** function. The **len( )** function returns the number of characters in a string. The expression inside of the parenthesis must either be a string literal, a string variable, or an expression that evaluates to be a string. This identifies **len( )** as a *string function*. There are other string functions (for example: val( ), trim$( )). The result returns is a number and can be used in any mathematical expression.

There are *numeric functions* as well. For example:

```
[start]
  let count = count + 1
  print "The sine of  "; count; " is "; sin(count)
  if count < 45 then goto [start]
```

This simple program lists the sines for the values from 1 to 45. The **sin( )** function takes the value of **count** enclosed in parenthesis and returns the sine (a function in trigonometry, a branch of mathematics) for that value. Just like the **len( )** function above, **cos( )** and other numeric functions can be used as parts of bigger expressions. We will see how this works just a little further along.

Notice also the way the program counts from 1 to 45. On the first pass, count is equal to zero until it gets to the line **let count = count + 1** which makes sets the data for variable **count** to be one more than its value at that point. Then the program prints the sine of **count** (the sine of one, in other words). After this, the line **if count < 45 goto [start]** checks to see if the data for **count** is less than 45. If it is, then BASIC goes back to the branch label **[start]** to do it again. This happens over and over until **count** reaches a value of 45, and then it doesn't go back to **[start]** again, but instead having no more lines of code to run, the program stops.

Going back to execute code over again is called *looping*. We saw this earlier when we first used the **goto** statement. In our first use of **goto**, the program *always* looped back. In this newest example program we see going back to execute code over again, but based on a condition (in this case whether **count** is less than 45). This is called *conditional looping* (you guessed it, the looping that always happens is called *unconditional looping,* or *infinite looping*).

Documenting BASIC Code

When writing very short and simple BASIC programs, it isn't usually difficult to grasp how they work when reading them days or even weeks later. When a program starts to get large then it can be much harder. There are things that the programmer (yes, you) can do to make BASIC programs more understandable.

VARIABLES - Liberty BASIC makes it easy to give your variables very meaningful names. Since a variable name can be as long as you like and because Liberty BASIC lets you use upper and lower case letters, variable names can be very meaningful.

For example:

```
let c = (a^2 + b^2) ^ 0.5
```

could better be expressed:

```
let lengthOfCable = (distanceFromPole ^ 2 + heightOfPole ^ 2) ^ 0.5
```

Both are valid Liberty BASIC code, but the second is easier to read and maintain.

BRANCH LABELS - Make sure that when you use **goto** that your branch labels describe the kind of activity your BASIC program performs after the label. For example if you are branching to a routine that displays help then use [help] as your branch label. Or if you are branching to the end of your program you might use [endProgram] or [quit] as branch labels.

COMMENTING CODE - BASIC also has a built in documentation feature that lets you add as much commentary as you like in the language of your choice. The **rem** (short for remark) statement lets you type whatever you like after it (you can even misspell or type gobbledy-gook, it doesn't care!). For example:

```
[askForName]

  rem  Ask for the user's name
  input "What is your name" ; yourName$

  rem  If the user didn't type anything, then ask again
  if yourName$ = "" then goto [askForName]
```

Notice how a **rem** statement was added before the **input** statement and before the **if . . . then** statement to describe what they should do. Liberty BASIC just skips over these lines, but a human reader finds this kind of documentation very helpful.

Also see the way that blank lines were added between the different parts of the program? These help to group things together, making the program easier to read.

A more elegant form of the **rem** statement uses the **'** (apostrophe, the key just to the left of the Enter key). Instead of typing **rem**, substitute the **'** like so:

```
[askForName]

  '  Ask for the user's name
  input "What is your name" ; yourName$

  '  If the user didn't type anything, then ask again
  if yourName$ = "" then goto [askForName]
```

Most people consider this to be more readable than using **rem**, and it works the same. One extra thing that you can do only with the apostrophe version of **rem** is to hang it off the end of whatever line you are commenting. For example:

```
[askForName]

  input "What is your name" ; yourName$           '  Ask for the user's name

  if yourName$ = "" then goto [askForName]          '  If the user didn't type anything, then ask again
```

This optional, but it saves screen space and many prefer it.

Learning to document the programs you write takes practice. Try to develop a consistent style. Everyone does it differently and there isn't a right or wrong way to do it. Smaller programs may not need any documentation at all. Programs that you intend to share with others should probably be thoroughly documented.

Let's Write a Program - HILO.BAS

Now we will write a simple game using all of the concepts described in this chapter. These include:

input statement
print statement
let statement
variables
goto statement
conditional branching with **if . . . then**
functions
documenting

THE GAME - HILO.BAS

Hi-Lo is a simple guessing game. The computer will pick a number between 1 and 100. Our job is to guess the number in as few guesses as we can. When we guess, the computer will tell us to guess higher or to guess lower depending on whether we guessed too high or too low. When we finally get it, the computer will tell us how many guesses it took.

Let's outline how our program will work before we begin to write code:

(1) Pick a number between 1 and 100
(2) Print program title and give some instructions
(3) Ask for the user to guess number
(4) Tally the guess
(5) If the guess is right go to step (9)
(6) If the guess is too low tell the user to guess higher
(7) If the guess is too high tell the user to guess lower
(8) Go back to step (3)
(9) Beep and tell the user how many guess it took to win
(10) Ask the user whether to play again
(11) If the user answers yes then clear the guess tally and goto step (1)
(12) Give the user instructions on how to close the game window
(13) End the program

When we write an outline for a computer program like we did here, the resulting outline is often called *pseudocode*, which is a fancy name for *false code*. This can be a useful tool for planning out software before it is written, and it can be very helpful in developing ideas before code is actually written.

Now we are going to take each of the steps above and write BASIC code for each step. We will document the code to explain its purpose:

(1) Pick a number between 1 and 100

```
  ' Here is an interactive HI-LO
  ' Program

[start]
  guessMe = int(rnd(1)*100) + 1
```

The first couple of lines are just **'** remark statements to give a brief desciption for the program. The **[start]** branch label is equivalent to calling this part of the program step (1).

Now we have the code that picks the number. We use two functions here to accomplish this task:

The **rnd( )** function is the key to this line of code. It picks a random (or nearly random) number greater than 0 and less than 1 (for example 0.3256). Then we multiply this by 100 with the * operator to get a number between greater than 0 but less then 100 (0.3256 times 100 would be 32.56) ;

The **int( )** function removes the fractional part to leave only the integer part of the number (the 0.56 part of 32.56 would be removed to leave only 32).

Then we add 1 to this. This is necessary because we want to pick a number as small as 1 and as large as 100. The **rnd( )** function only gives us a number as large as 0.9999999 and not as large as 1. If you multiply 0.9999999 by 100, the biggest number you can get is 99.99999, and this is not big enough so we add one.

When we have picked the number, we assign its value to the variable **guessMe**.

2) Print program title and give some instructions

```
' Clear the screen and print the title and instructions
cls
print "HI-LO"

print

print "I have decided on a number between one"
print "and a hundred, and I want you to guess"
print "what it is.  I will tell you to guess"
print "higher or lower, and we'll count up"
print "the number of guesses you use."

print
```

This very simple part of the program wipes the window clean and prints the title **HI-LO** and some instructions. Notice the use of blank **print** statements to add space between the title and the instructions and after the instructions also.

3) Ask for the user to guess number

```
[ask]
  ' Ask the user to guess the number and tally the guess
  input "OK.  What is your guess"; guess
```

Here the branch label **[ask]** will let us go back here later if the user needs to be asked to guess again. Then we use the **input** statement to ask the user for a guess. The user's guess is then placed in the **guess** (what else?) variable.

```
  ' check to see if the guess is right
  if guess = guessMe then goto [win]
```

4) Tally the guess

```
  ' Now add one to the count variable to count the guesses
  let count = count + 1
```

Now we take the value of **count** and add one to it. Each time this code is performed, **count's** value will increase by one.

5) If the guess is right go to step (9)

This line compares the variable **guess** with the variable **guessMe**. If they are equal, then we **goto [win]**, which is equivalent to **. . . go to step (9)** in our outline.

6) If the guess is too low tell the user to guess higher

```
  ' check to see if the guess is too low
  if guess < guessMe then print "Guess higher."
```

This line compares the variable **guess** with the variable **guessMe**. If **guess** is less than **guessMe**, then display the text "Guess higher."

7) If the guess is too high tell the user to guess lower

```
  ' check to see if the guess is too high
  if guess > guessMe then print "Guess lower."
```

This line compares the variable **guess** with the variable **guessMe**. If **guess** is greater than **guessMe**, then display the text "Guess lower."

8) Go back to step 3

```
  ' go back and ask again
  goto [ask]
```

9) Beep and tell the user how many guess it took to win

```
[win]
  ' beep once and tell how many guesses it took to win
  beep
  print "You win!  It took"; count; "guesses."

  ' reset the count variable to zero for the next game
  let count = 0
```

This is the code our game executes when the player wins. The **beep** statement rings the terminal bell once. Then the **print** statement says that the game is won and how many guesses it took. Finally, the **let** statement resets the **count** variable to zero for the next game.

10) Ask the user whether to play again

```
' ask to play again
input "Play again (Y/N)"; play$
```

This **input** statement asks whether or not to play again. The resulting string is stored in the string variable called **play$**.

11) If the user answers yes then goto step 1

```
if instr("YESyes", play$) > 0 then goto [start]
```

This **if . . . then** statement uses the **instr( )** function to determine whether the player answered "Y", "y", "YES", or "yes". The **instr( )** function checks to see if the string in **play$** is found anywhere in the string literal "YESyes". If it is, then **instr( )** returns the position, which is then compared with 0 using the **>** operator. If the contents of **play$** are found in "YESyes", then the value returned from **instr( )** will be greater than zero, so that **goto [start]** will be executed, and the game will be restarted.

12) Give the user instructions on how to close the game window

```
print "Press ALT-F4 to close this window."
```

Since the player did not wish to play again, we will display instructions on how to close the game window.

13) End the program

```
end
```

It is good practice place the **end** statement as the end of your BASIC programs. It can also be used at any place where you want the program to stop running.

THE COMPLETE LISTING FOR HILO.BAS - Here is the complete listing for HILO.BAS so that you can just copy and paste it into Liberty BASIC to run it.

```
  ' Here is an interactive HI-LO
  ' Program

[start]
  guessMe = int(rnd(1)*100) + 1

  ' Clear the screen and print the title and instructions
  cls
  print "HI-LO"

  print

  print "I have decided on a number between one"
  print "and a hundred, and I want you to guess"
  print "what it is.  I will tell you to guess"
  print "higher or lower, and we'll count up"
  print "the number of guesses you use."

  print

[ask]
  ' Ask the user to guess the number and tally the guess
  input "OK.  What is your guess"; guess

  ' Now add one to the count variable to count the guesses
  let count = count + 1

  ' check to see if the guess is right
  if guess = guessMe then goto [win]
  ' check to see if the guess is too low
  if guess < guessMe then print "Guess higher."
  ' check to see if the guess is too high
  if guess > guessMe then print "Guess lower."

  ' go back and ask again
  goto [ask]

[win]
  ' beep once and tell how many guesses it took to win
  beep
  print "You win!  It took"; count; "guesses."

  ' reset the count variable to zero for the next game
  let count = 0

  ' ask to play again
  input "Play again (Y/N)"; play$
  if instr("YESyes", play$) > 0 then goto [start]

  print "Press ALT-F4 to close this window."

  end
```

WHERE CAN I GET A COPY OF LIBERTY BASIC?

Liberty BASIC can be found on most major BBS systems in the USA, and on many in Europe. The filename to look for is LB07.ZIP. The Windows version is LB11W.ZIP or LB12W.ZIP.

Compuserve members can find it in the OS2USER forum as LB07.ZIP in the Library 4. The Windows version is in the PCPROG forum as LB11W.ZIP or LB12W.ZIP in Library 4.

Shoptalk Systems can be reached at:

Shoptalk Systems
P.O. Box 1062
Framingham, MA 01701

Phone: 508-872-5315
Compuserve ID: 71231,1532

The official Liberty BASIC support BBS is:
The Metro-West Free BBS, 508-626-2158