Basic
plain talk for your home computer

Modern Electronics' easy-to-understand primer on how to teach the most popular language to your home computer.

by Peter A. Stark
Contributing Editor

Basic is probably the most popular and widely used computer language for small computer hobbyists, and with good reason—it is powerful, yet simple. I'm sure you're ready to learn more about what Basic is and what it can do.

Originally invented at Dartmouth College in the early 60s, it was intended to bring the computer to the average Dartmouth student in a way that had never been tried before. The traditional approach was to place a large computer center in the corner of the campus and then force students to go to the center to run their computer programs.

Dartmouth tried the exact opposite. It placed computer terminals throughout the campus, even in dormitories, within easy reach of every student and then tempted students to use them, not only by having the terminals easy to get to but also by having a simple computer language to program the computer with. That was the beginning of Basic!

Unlike earlier languages such as Fortran or Cobol, which were intended for large programs, Basic was intended for the small uses. A Fortran and Cobol user had to prepare his programs on punched cards away from the computer. Only when he had the entire set of cards ready, would he go into the computer room and enter the cards into the computer. His problem would be run on the computer, his results printed or punched back into cards, and then he would be encouraged to leave to make room for the next user. In other words, these languages kept the user away from the computer as much as possible.

Basic, on the other hand, was designed for use with terminals, such as teletypewriters, which were connected to the computer and actually using the computer for extended periods of time. A student could sit down at a terminal and play a game against the computer for hours on end.

One popular computer terminal is the CRT or Cathode Ray Tube which displays a program and its results on a screen similar to a tv set.

To make this entire idea practical in the days of million-dollar computers required the use of *time-sharing*, where dozens or perhaps even hundreds of terminals were connected to the computer at the same time. Since the computer is so fast, it easily could take care of many students using the computer at the same time, with each student having the impression he was the only user.

But now, in 1978, the entire approach has changed. For just a few hundred dollars, you can buy a small computer which can run Basic programs. Since it is no longer necessary to time-share, the computer system can be quite simple and cheap, and yet still be power-

ful enough to run sizable programs, although only one at a time.

To see just what Basic is and what we can do with it, let's sit down at a computer terminal and type in some commands, seeing what the computer does. These examples were run on a Southwest Technical Products MP-6800 home computer, but would be the same with any small home computer system.

The first thing we notice is the terminal has a keyboard similar to a typewriter, except some symbols are in new places and some keys have symbols not found on a typewriter.

For instance, above the comma is the symbol <, and above the period is the symbol >. Of special importance is a key labelled CR or RETURN, which means carriage return. This key means you are finished with a line and want to return the carriage, the part that prints on the paper, to the left, ready for the next line. Every line you enter into the computer must be followed by a CR to tell the computer you are done.

Let's sit down at the terminal and start with a CR. The computer responds with

```
READY
```

and returns to the beginning of the next line. Sometimes the computer will print a #, > or ? on the next line. This is the *prompting character* and its purpose is to tell you it's your turn to type something and the computer is waiting.

In our case we get the message

```
READY
#
```

which tells us the computer is ready, and waiting for a command. Let's enter a simple program telling the computer to print something:

```
10 PRINT 2+3
```

A one-line program like this is about as simple as you can get. Don't forget the carriage return or CR at the end. This simply tells the computer to add 2 and 3 and print the result.

Notice the number 10 in front. Every instruction of a Basic program must have a *line number* before it, and this is line number 10. The 10 does not necessarily mean that this is the tenth line of a program; it just means that we have decided to give this line the number 10. We could have just as well numbered it 1 or 500. The point behind line numbers is that every line of a program has a different line number, so at some later time we can go back and remove or change lines at will, referring to them by line numbers.

Once we have typed in a program such as this one, we can do two things with it—get a listing of it on the printer to check that we have typed it correctly, or run it. To get a listing, we type the word

```
LIST
```

and, as soon as we hit the CR, the computer responds with

```
0010 PRINT 2+3
READY
#
```

With minor changes, the computer simply types the program as we have entered it.

Notice that, up until now, we have not gotten the actual answer of 5, which the computer is supposed to print. We merely have entered the program and checked it. To actually perform it and get our answer, we type

```
RUN
```

and the computer prints

```
5
READY
#
```

Notice that there is a difference between *lines of a program*, which *always* get a line number, and *commands* to the computer telling it what to do with the program, which *never* get a line number. The commands we use most often are LIST and RUN, but each computer system has a number of other commands such as:

- NEW or CLEAR—Erase the program
- SAVE—Save the program on tape or other storage for later use
- LOAD—Load a program previously saved back into the computer

Let's erase the simple program we wrote and enter a new one:

```
NEW
READY
#10 LET I=3
#20 LET J=I+17
#30 PRINT I, J
#
```

With one exception, every program instruction starts with a short word such as LET or PRINT right after the line number. The one exception is that the word LET may be omitted. Notice that each line has a line number. We could have numbered the lines 1, 2, 3 but this is a bad habit to get into. Very often we find, after trying to run the program, we made a mistake and have to add a few lines. With lines numbered 10, 20, 30, and so on, it's easy to slip in extra lines such as line 15 or 18. Even though we may enter them at a later time, giving them a line number between 10 and 20 will automatically tell the computer that we want them placed in that order.

In the above program, lines 10 and 20 mean just what they say. Line 10 says to let a number I be equal to 3. We have to learn the difference between *constants*, which are constant and never change, and *variables* which can vary and change. In this line, the number 3 is a constant while I is a variable. We could, for example, insert another line into the program as follows:

```
22 LET I=5
```

I thus changes—it was equal to 3 at line 10, but becomes equal to 5 at line 22. We could now get a listing of the program as follows:

```
LIST
0010 LET I=3
0020 LET J=I+17
0022 LET I=5
0030 PRINT I, J
READY
#
```

Notice the computer automatically put line 22 in the right place, between 20 and 30.

Constants are plain numbers such as 3, 5, 17, or -12.597. There is a way of expressing very large or very small constants by using powers of 10, but that does

not concern us at this point. By their very nature, they obviously never change.

Variables, on the other hand, are represented by letters such as I or J. In fact, any of the letters A through Z can be used for variables. Since this only would allow 26 different variables, Basic also allows variables to be represented by a letter followed by a number from 0 through 9. This is very convenient for calculations on electrical circuits, since the values of resistors can be represented by the variables R1, R2, and so on.

Let's take the above program and run it.

```
RUN
5     20
READY
#
```

To understand what has happened, we have to examine the above program line by line. Line 10 told the computer to let the variable I equal 3. Line 20 says to add I (which is 3) to 17, and let J be the answer. Thus J becomes equal to 20. Then, line 22 says to let I equal 5. From this point on, I is 5, not 3, so that line 30 prints 5 for I and 20 for J.

As you can see, the computer performs these instructions in the order of their line numbers, not in the order we typed them in. This is another important use of line numbers—they specify the order in which the computer will perform its instructions.

The opposite of a PRINT statement is an INPUT. For an INPUT, the computer stops, prints a ? prompting character, and then waits for you to type in something. Let's write a short program to allow you to type in a number, have the computer multiply it by 3, and print out the answer. First erase the old program:

```
NEW
READY
#
```

Now enter a new program:

```
#10 INPUT N
#20 S = 3 * N
#30 PRINT S
```

Line 10 allows you to type in a number, which becomes the variable N. Line 20 multiplies it by 3; notice how a star * is used to mean *times*. Finally, line 30 prints out the product. If we now type:

```
#RUN
```

the computer prompts with

```
?
```

and we supply a number, such as

```
1.2
```

the computer comes back with

```
3.6
READY
#
```

This would not be much fun if we could only enter and print numbers, but Basic also allows us to use letters and words. For example, let's add the line:

```
#5 PRINT "TYPE IN A NUMBER AND I WILL MULTIPLY IT BY 3"
```

and change line 30 to read

```
#30 PRINT "THE ANSWER IS", S
```

If we list it, we get the printout

```
#LIST
0005 PRINT "TYPE IN A NUMBER AND I WILL MULTIPLY IT BY 3"
0010 INPUT N
0020 S = 3 * N
0030 PRINT "THE ANSWER IS", S
```

Now try running it:

```
#RUN
TYPE IN A NUMBER AND I WILL MULTIPLY IT BY 3
? 7
THE ANSWER IS 21
READY
#
```

As you can see, enclosing a message in quotes " and placing it in the PRINT statement makes the computer print it exactly as it stands.

Another type of variable is the *string variable*. It is signified by a letter A through Z, followed by the $ sign. Its function is to hold a string of letters or other characters from the keyboard, but allow them to be changed, like variables, throughout a program. To illustrate, let's try a new program:

```
#NEW
READY
#10 PRINT "WHAT IS YOUR NAME?"
#20 INPUT N$
#30 PRINT N$, "IS A NICE NAME"
```

Line 20 lets us input a string of letters, while line 30 prints them out again. Watch what happens when we run the program:

```
#RUN
WHAT IS YOUR NAME?
? PETE
PETE      IS A NICE NAME
READY
#
```

After inputing the name PETE, the computer printed it out again, followed by the words IS A NICE NAME. There is a large space after PETE which is put in by the computer because Basic normally prints its output spread out across the page to be in nice columns if numbers are being printed. In this case it makes the output look messy, but that is easy to get around if we use a semicolon ; in line 30 instead of a comma. This is one of the fine points in Basic, which are of little interest to the beginner but are very useful to the expert.

The tremendous power of the computer comes from the fact that programs, or portions of them, can be repeated over and over. Suppose we add one more line to the above program:

```
#40 GO TO 30
```

and run it again:

```
#RUN
WHAT IS YOUR NAME?
? PETE
PETE        IS A NICE NAME
PETE        IS A NICE NAME
PETE        IS A NICE NAME
PETE        IS A NICE NAME
PETE        IS A NICE NAME
PETE        IS A NICE NAME
PETE        IS A NICE NAME
PETE        IS A NICE NAME
```

Computer experts would now say the computer is *stuck in a loop*. It would keep on printing out the same line over and over if we didn't stop it by pushing a

button on the control panel. Our last line, line 40, is the culprit. It told the computer to go back to line 30 and repeat from there. Thus the computer does the printout in line 30, and the very next line sends it right back to do another printout, and so on. This is an *infinite loop*, since it never stops—unless we push a button to stop it, that is.

A better way of controlling a GO TO is with an IF instruction. For example, we can say IF X=3 GO TO 30, and the GO TO will only be done by the computer if the value of the variable X happens to be 3.

Let's change the above program so it will ask for a name, and will only print out "IS A NICE NAME" if the name happens to be PETE; otherwise, the computer will answer that the name is a poor one:

```
#10 PRINT "WHAT IS YOUR NAME?"
#20 INPUT N$
#30 IF N$ = "PETE" GO TO 60
#40 PRINT N$, "IS A POOR NAME"
#50 GO TO 10
#60 PRINT N$, "IS A NICE NAME"
#70 GO TO 10
```

A computer terminal's keyboard has several added keys you won't find on an ordinary typewriter. Otherwise it's similar. An important key always used is the CR or Carriage Return key, shown at right.

As before, the computer asks WHAT IS YOUR NAME. If you answer PETE, then line 30 tells the computer to go to line 60, so that it will print the name again, followed by the words IS A NICE NAME. For any other name, the computer will *not* go to line 60, but will instead continue to line 40 and print IS A POOR NAME. Either way, a GO TO 10 returns to the top, so the computer asks for another name. Let's run it to see what happens:

```
#RUN
WHAT IS YOUR NAME?
? SAM
SAM         IS A POOR NAME
WHAT IS YOUR NAME?
? GEORGE
GEORGE      IS A POOR NAME
WHAT IS YOUR NAME?
? PETE
PETE        IS A NICE NAME
WHAT IS YOUR NAME?
?
```

As before, the computer is stuck in a loop since it keeps returning to step 10. This is usually not quite what we want. A good loop is one which has an end to it. In some way, we like to tell the computer when to get out of the loop. One common way is to count the repetitions of the loop, and stop at some predetermined number of them. For example, the following program prints out the numbers from 1 to 12 and their squares:

```
#NEW
READY
#10 LET N = 1
#20 LET S = N * N
#30 PRINT N, S
#40 LET N = N + 1
#50 IF N < 13 GO TO 20
```

Line 10 starts the number N at 1; line 20 squares it by multiplying it by itself; line 30 then prints the number N and its square S. Now, line 40 says something a bit different from what a mathematician would expect from N = N + 1 (which is not really a good equation after all.) What it means is that the computer should take the value of N, add 1 to it, and then place the result back as a new N. In other words, line 40 adds 1 to N. Since N started at 1, it is now 2. But since this is in a loop, in a little while N will go to 3, and then 4, and so on, all the way up to 12.

The symbol < in line 50 means *less than*, so this line says "if N is less than 13, go back to line 20." But eventually N will go from 12 to 13, and when that happens, line 50 no longer sends the computer back to line 20. So we have here a loop which is repeated exactly 12 times.

The IF statement is very useful, since it allows checking whether two things are equal or not. In addition to the less than or < symbol, we also use > which means *greater than*. The combination <> means *less than or greater than*, which is the same as saying *not equal*, so IF X <> 5 GO TO 300 means that if X is not equal to 5 the computer should go to line 300. Moreover, instead of ending the IF with a GO TO, we can also end with the word THEN followed by any other valid Basic instruction. Our program to judge whether a name is nice or not could have been written with these two IFs:

```
#40 IF N$ = "PETE" THEN PRINT N$, "IS A NICE
NAME"
#50 IF N$ <> "PETE" THEN PRINT N$, "IS A POOR
NAME"
```

Two other combinations are <= which means *less than or equal*, and >= which means *greater than or equal*.

The idea of using a variable to count the repetitions of a loop is so common and useful that Basic has a special pair of instructions just for that purpose—the FOR and NEXT pair. These always go together, the FOR at the start of the loop and the NEXT at the end. To see how they work, let's rewrite the program to square the numbers from 1 to 12:

```
#NEW
READY
```

```
#10 FOR N = 1 TO 12
#20 LET S = N * N
#30 PRINT N, S
#40 NEXT N
```

Line 10 tells the computer that N is the counter, and it is supposed to vary from 1 to 12. Initially, N starts at 1, and the computer continues down through the following steps until it gets to NEXT N. Now it adds 1 to N, and goes back to the first statement inside the loop, which is line 20. It will repeat the loop, adding 1 to N each time, until N reaches 12. When N tries to go to 13, the loop ends.

There is a variation on the FOR which lets N change in different ways; this is done by adding one more word to the line:

```
#10 FOR N = 1 TO 12 STEP 1
```

This specifies that N is supposed to go from 1 to 12 in steps of 1. If we said

```
#10 FOR N = 1 TO 12 STEP 3
```

then N would go up in steps of 3. Or if we said

```
#10 FOR N = 12 TO 1 STEP -1
```

it would go from 12 back to 1 in steps of -1. That is, N would go 12, 11, 10, 9, 8, and so on, all the way to 1. Just to see what happens, let's try running the program:

```
RUN
12   144
11   121
10   100
9    81
8    64
7    49
6    36
5    25
4    16
3    9
2    4
1    1
READY
#
```

Basic has several more possible instruction types. Some, like REM (remark) and STOP, are useful to the beginner and we will see them later in some of the demonstration programs. Others are for more advanced users and we will skip them here.

In addition to the various instruction types, Basic also has *functions* which perform specific math calculations or some other operations. For example, a mathematician or engineer might use the SIN or COS functions when working with angles. The functions likely to be used by the beginner, out of the dozen or more most computers have, are these:

■INT ( ) converts whatever is placed inside the parentheses into the next lower integer (whole number). For example, saying

```
#10 LET J = INT(3.14)
```

would make J equal to 3.

■RND (0) makes the computer invent a random number between 0 and 1. This is usually used in games for coming up with random moves or random numbers. For instance,

```
#10 LET J = RND(0)
```

would result in J becoming equal to some unknown value between 0 and 1.

Sometimes we combine the RND and INT functions to generate other random numbers. For instance, suppose we are writing a game where the computer is supposed to pick a card from a deck of cards and print out what it is. Since there are 13 cards in a suit, we need a random number which is a whole number between 1 and 13.

If we use RND to make a number from 0 to 1, and then multiply it by 13, the result will be a number from 0 to 13. Add 1 to this, and you have a random number between 1 and 14, but always just a bit smaller than 14. Convert it to an integer with INT, and you have a whole number ranging from 1 to 13 (and never equal to 14.) The result of putting all this into one line is

```
#100 LET C=INT( RND(0) * 13+1)
```

One more function useful to beginners is the TAB( ); which makes the terminal's printer or display move over to the right to the position indicated by whatever is inside the parenthesis. For example

```
#50 PRINT TAB(15); I
```

would print the value of I fifteen places from the left end of a line on the printer. Note that the TAB is used in a PRINT statement, and that it is usually followed by a semicolon.

Finally we are ready to put all this together into several simple programs. How about a program to pick five cards at random and print out what they are? We will program it as a loop which is repeated five times, use the RND function to pick a random number, and use IF statements to print out words like JACK or KING:

```
#NEW
READY
#10 FOR I=1 TO 5
#20 LET C=INT(RND(0)*13+1)
#30 IF C<11 THEN PRINT C
#40 IF C=11 THEN PRINT "JACK"
#50 IF C=12 THEN PRINT "QUEEN"
#60 IF C=13 THEN PRINT "KING"
#70 NEXT I
#RUN
1
7
QUEEN
7
2
READY
#
```

Now let's add a few more steps to add the suit. We will use RND again to pick a number between 1 and 4, and use it to print out the suit. Add the following steps:

```
#25 LET S=INT(RND(0)*4+1)
#62 IF S=1 THEN PRINT TAB(6); "OF HEARTS"
#63 IF S=2 THEN PRINT TAB(6); "OF
DIAMONDS."
#64 IF S=3 THEN PRINT TAB(6); "OF CLUBS"
#65 IF S=4 THEN PRINT TAB(6); "OF SPADES"
```

To see what the program now is, we list it:

```
#LIST
0010 FOR I=1 TO 5
0020 LET C=INT(RND(0)*13+1)
0025 LET S=INT(RND(0)*4+1)
0030 IF C<11 THEN PRINT C
0040 IF C=11 THEN PRINT "JACK"
0050 IF C=12 THEN PRINT "QUEEN"
0060 IF C=13 THEN PRINT "KING"
0062 IF S=1 THEN PRINT TAB(6); "OF HEARTS"
0063 IF S=2 THEN PRINT TAB(6); "OF
DIAMONDS"
0064 IF S=3 THEN PRINT TAB(6); "OF CLUBS"
0065 IF S=4 THEN PRINT TAB(6); "OF SPADES"
READY
#RUN
KING
      OF HEARTS
5
      OF DIAMONDS
JACK
      OF CLUBS
6
      OF DIAMONDS
JACK
      OF CLUBS
READY
#
```

We could neaten the output so each card is printed on one line, but that's more complicated. Let's do another example. How about a program to input the names of two people and print them out in alphabetical order?

```
#NEW
READY
#10 PRINT "ENTER TWO NAMES"
#20 INPUT A$, B$
#30 IF A$<B$ THEN PRINT A$, B$
#40 IF B$<A$ THEN PRINT B$, A$
#RUN
ENTER TWO NAMES
? SMITH,JONES
JONES          SMITH
READY
#
```

Notice how we are comparing two strings of letters as if they were two numbers; whichever is less is printed first. Although this example only sorts two names, we could do it for more names with a more complicated program.

Suppose a math student needs to plot an equation for his homework. The equation is $y=x^2-10x+26$, and he is supposed to find y for x going from 0 to 10. This program would do it:

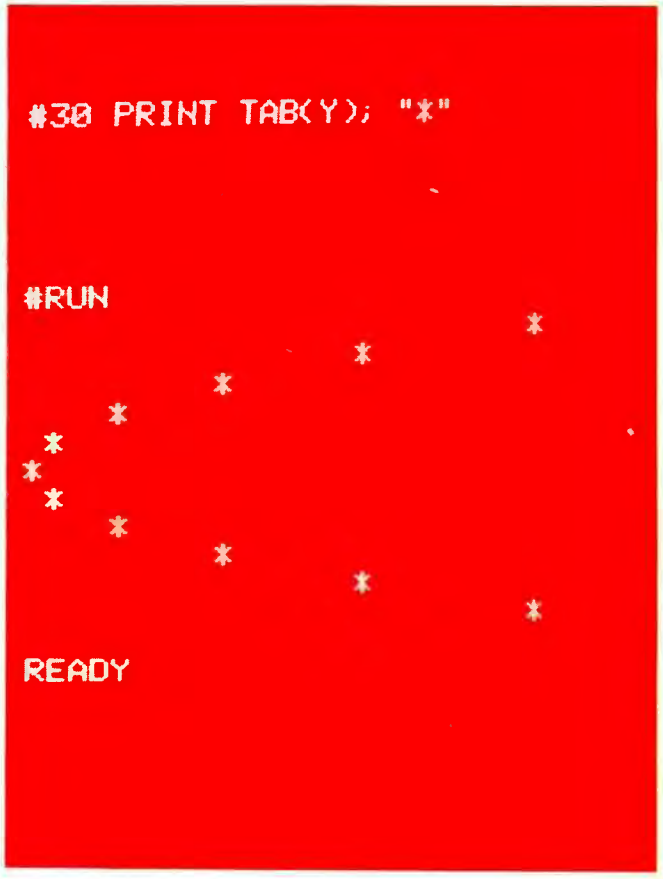
```
#NEW
READY
#5 REM THIS IS A REMARK
#7 REM LET X GO FROM 0 TO 10
#10 FOR X=0 TO 10
#20 LET Y=X*X -10*X+26
#25 REM PRINT BOTH X AND Y
#30 PRINT X, Y
#35 REM END OF LOOP
#40 NEXT X
#50 REM WHEN LOOP IS DONE, STOP
#60 STOP
#RUN
0              26
1              17
2              10
3              5
4              2
5              1
6              2
7              5
8              10
9              17
10             26
READY
#
```

Better yet, why not have the computer draw a picture? Change line 30 to

```
#30 PRINT TAB(Y); "*"
```

and run:

```
#RUN
```



The computer draws a picture, as instructed in a program, to solve a math equation.

```
READY
#
```

The graph may be sideways and a little coarse, but it certainly gives the picture.

With this introduction to Basic, you're on the way to writing your own programs.