7/4/2014

Half-Byte Tiny Basic

Tiny Basic for Arduino

George Gray
HALF-BYTE

COMMANDS EXPLAINED

So, let's start discussing the statements and how to use them.

NEW

Type NEW and press enter. This will clear out the program space. It does NOT zero out any variables, however. (it is good practice to set your variables at the start of your program. Set them to zero or whatever value you need.)

RUN

Type RUN to start executing your program. There are no parameters.

SAVE and LOAD

Currently, you can SAVE and LOAD your program by typing SAVE to save it to the EEPROM and LOAD to load it back into RAM. As of this writing, you can ONLY save one program. A future release will allow multiple programs to be saved, possibly to SD Cards. Another limitation in this version is that your program can only have 255 lines save. This limitation should be removed in a later version of Tiny Basic.

LIST, LIST number or LIST number-

Typing LIST and hitting Enter will display the program code in its entirety.

Typing LIST followed by a number will display the program code starting with the specified line number. So, LIST 190 will display the program code starting at line 190.

Typing LIST followed by a number and a dash will display ONLY that line.

STATEMENTS EXPLAINED

PRINT and SPRINT PRINT is, perhaps, the most versatile statement in the language. It has many variations and you will figure them all out in time. You can mix formulas, variables and quoted text in many ways. SPRINT is just like PRINT, only it sends the output directly to the SERIAL port (the TX and RX pins on your console are the SERIAL port.) The examples below are all valid:

```
PRINT “Hello, World!”
PRINT “My number was “, N
PRINT D,” days until Christmas.”
PRINT M+(R*2)-L
PRINT “The returned value is “, (x/4)+4-8*64,” is that correct?”
PRINT x,” turns remaining.”
PRINT “Turn Number “;
SPRINT G,” was my answer.”
SPRINT “This will only be seen on the serial port.”
```

So, there’s a few things we haven’t discussed. First, the comma. Placing a comma outside of the quotes tells Tiny Basic to print something first, then the quoted text. You can have multiple commas outside the quotes. They separate items to print. For example, say you want to print the values of several variables and then some text:

```
PRINT A,B,C,” notice how they print together.”
PRINT A,”,”,B,”,”,C,” will print them with a comma between them”
PRINT A,” “,B,” “,C” will print them with a space between them.”
```

Placing a comma at the END of the PRINT statement will result in a SYNTAX ERROR.

Placing a SEMICOLON at the END of the PRINT statement will place the NEXT print space at the end of the current print line. For example:

```
X=1
PRINT “The next value is “;
X=X+1
PRINT X
```

Will result in the value of X showing up after the space in the same line:

```
The next value is 2
```

Also, the parentheses tell Tiny Basic to execute the formula inside them before acting on the rest of the formula. Like in math, parenthesis changes the order of operations.

SERIAL PORT INFO

NOTE: The output of PRINT is ALSO ECHOED on the SERIAL PORT. SPRINT ONLY goes to the serial port. The SERIAL PORT is a special part of the console that allows it to communicate with your computer or another console or device that also contains such a port. A PORT is just another word for connection. On your Half Byte Console and on other Arduino board, it is the TX, RX and GND pins. To connect two of the together, you must remember to reverse the connector pins for TX and RX. For example: on your console you have a green wire on the TX pin, a red wire on the RX pin and a yellow wire on GND. Connect the yellow to GND on BOTH. Connect the other green end to TX and the other red end to RX.

RX---→TX
TX---→RX
GND-→GND

ECHO

To turn off the PRINT echo to the serial port, or to turn it back on, use ECHO. ECHO 1 turns on ECHO, ECHO 0 turns it off.

```
ECHO <value>

100 CLS
110 ECHO 0
120 PRINT “YOU ARE PLAYER 1”
130 SPRINT”YOU ARE PLAYER 2”
140 PRINT “YOU CAN ONLY SEE THIS”
150 ECHO 1
160 PRINT “NOW BOTH CAN SEE.”
```

FOR Loops

A FOR loop is an enclosed set of statements and functions that will executed a number of times before the loop is completed. We use them for things like pausing the program for a certain time, printing the value of something for a number of times as in a countdown or to calculate something. A FOR statement consists of a start point and an endpoint. You can also specify the increment:

```
100 FOR X=1 TO 20
110 PRINT 20-X
120 NEXT X
```

Or

```
100 FOR X=0 TO 100 STEP 10
110 PRINT X
120 NEXT X
```

When the first example is run, it will print 20 minus the value of X and will do so 20 times.

The second example will print the value of X 10 times. Why? Because we told it to STEP by tens. The 'STEP 10' says to ADD 10 to the value of X each time it loops.

NOTE: FOR loops MUST reside on a line by themselves. So, the following code is NOT legal:

```
110 FOR K=1 TO 100: PRINT K: NEXT K
```

It must be broken out like this:

```
110 FOR K=1 TO 100
120 PRINT K: NEXT K
```

While including NEXT on a combined line is OK, it is better to also put it on its own line, if for, nothing else, only for clarity.

NEXT

The NEXT statement closes the FOR loop and is followed by the loop variable as shown in the two FOR loop examples above.

NOTE: FOR loops MUST HAVE A CORRESPONDING NEXT. Also, exiting a for loop before it is finished and then entering it again can result in a stack overflow. What that means is that the system will forget where it was and try to complete the original loop-which is no longer there. It WILL crash the computer.

IF

To execute a statement based on a condition, use IF followed by an expression and then the statement you want to execute if the expression is true:

```
100 X=RND(20)
110 IF X=15 PRINT"The value of X is 15!"
120 IF X=0 X=1
```

When run, the above code will:
Print
`The value of X is 15!`
If x is 15
If x is zero, then the code makes the value of x to be 1.

Expressions can be as complex or simple as needed. In standard BASIC, the verb 'THEN' must follow the expression, in Tiny Basic, THEN is not needed and not allowed. The format of the IF statement is simply:

IF expression STATEMENT

So, IF the expression is true then execute STATEMENT. You can have multiple STATEMENTS as long as they are separated by a colon:

```
110 IF X=15 PRINT"The value is 15!":GOTO 200
```

GOTO GOTO transfers the execution to the line following the GOTO statement.

```
180 GOTO 300
```

Line 180 will transfer control to line 300 and execution will continue from there.

GOSUB GOSUB will temporarily transfer control to the line number after the GOSUB. Execution will continue from that line until the RETURN statement is encountered.

```
GOSUB line number

100 GOSUB 1000
110 PRINT "Returned from 1000"
….
1000 X=10
1010 RETURN
```

GOSUB's are useful if you have code that you will reuse several times. Rather than typing that same code over again, make a subroutine. Remember to end your subroutine with a RETURN. Say you have code that waits for a user to press a key to continue. You need to do that many times in your program. Rather than typing that code over and over, type it once as a subroutine:

```
200 GOSUB 1500 #wait for input
210 PRINT " "
…
260 GOSUB 1500
270 #more work
…
310 GOSUB 1500
…
1490 STOP
```

```
1500 A=INKEY(0)
1510 IF A=-1 GOTO 1500
1520 RETURN
```

As you can see, there were three times we needed to wait for the user to press a key. If we typed lines 1500 to 1520 each time, that would have added unnecessary code. Using it in a subroutine makes it more efficient. There are things to keep in mind, however…

- You MUST place a RETURN at the end of your subroutine
- You MUST make sure that your code does not 'wander' into the subroutine, so place a STOP before the subroutine (as shown in line 1490 above.)
- Any variables used in the subroutine will also be available everywhere else, so if you have a counter in your subroutine, changing it outside of the subroutine could return some unexpected results.

RETURN Ends your subroutine and returns control to the next line after the GOSUB.

REM or # It is good practice to comment your code so others may follow what you were doing. However, remember that comments, while ignored by Tiny Basic, consume memory. So, when entering code, you may leave them out. If you are giving someone source code, it is best to include them.

INPUT INPUT is a way to get numerical data from your user. The value entered is stored in the variable following INPUT:

```
INPUT <variable>
```

```
100 X=0
110 PRINT "Enter the new value for X";
120 INPUT X
130 PRINT "You entered ",X
```

Will result in:

```
Enter the new value of X? 20  (entered by the user)
You entered 20
```

Legal values are numbers from -32767 to 32767. No decimals or text may be entered using INPUT. You CAN enter text using the INKEY function. See INKEY for more.

POKE POKE allows you to alter the value, directly, of a memory location. For example:

```
100 X=1450  #memory location 1450
```

```
110 POKE X,64
```

```
POKE address, value
```

This example puts the integer 64 in location 1450.
You can use POKE in conjunction with INKEY to accept text from the user.

CLS CLS just clears the screen.

STOP STOP ends execution of the program and returns you to the editor.

SET SET lights up a pixel at location x,y. X and Y values are the height and width of the video screen. On the chip used in the Half-Byte Console, the 328, those values are 84 and 56 as the screen is 84 pixels wide by 56 high.

```
SET x,y
```

```
100 FOR X=1 to 83
110 SET X,10
120 NEXT X
```

This code will draw a line, at the 10th position down from the top, across the screen.

RESET RESET, like SET, uses the same coordinates for the screen and turns off the pixel at that location.

```
RESET x,y
```

```
100 FOR X=1 to 83
110 SET X,10
120 NEXT X
130 FOR X=35 to 45
140 RESET X,10
150 NEXT X
```

This code will draw the line across the screen, as in the previous example, but then erases part of the line from pixels 35 through 45.

CIRCLE CIRCLE will draw a circle at the specified coordinates on screen. You must supply the X and Y cords and the radius as well as the ‘color’.

```
CIRCLE start_x, start_y, radius, color
```

```
100 CLS
110 CIRCLE 10,10,5,1
```

This example will put a small circle on the screen at 10,10. NOTE: the coordinates you specify will become the CENTER POINT for the circle. That is, it will draw the circle AROUND the x, y co-ordinates going the value of radius out from the point. So, the circle above would actually start at 5,5 since the radius was 5.

LINE

LINE gives you the ability to draw lines on the screen. Specify the start and end points and 'color', and Tiny Basic will draw the line, quickly. To use:

```
LINE start_x, start_y, end_x, end_y, color

100 CLS
110 LINE 0,7,79,7,1
120 CURSOR 0,0
130 PRINT "UNDERLINED"
140 CURSOR 0,3
```

This example will draw a line starting at position 0 on line 7. It will then put some text on the first text line. This will, effectively, underline the whole text line.

BOX

BOX allows you to quickly draw a box on the screen. You can draw or erase a box or invert a section of the screen.

```
BOX start_x, start_ y, end_ x, end_y, color, type

100 CLS
110 BOX 0,0,79,39,1    #DRAW A BOX AROUND THE SCREEN
120 BOX 1,1,78,25,2    #INVERT THE TOP SECTION OF THE BOX
130 CURSOR 5,4
140 PRINT "BOX DEMO"
150 A=INKEY(0)
160 IF A=32 BOX 0,1,78,39,2
170 IF A=13 BOX 1,1,78,39,1
180 IF A=27 STOP
190 GOTO 150
```

This example draws a box around the screen, displays a message and then if the user presses the space bar, a section of the screen is inverted. If they user press ENTER, another box is drawn on top of the current box.

AWRITE, DWRITE AWRITE and DWRITE provide a means to send a signal to either the analog pins or the digital pins. You can use them to blink LED's, turn on or off things like motors or send data to anything you have connected to your console.

```
100 FOR K=1 TO 50
110 DWRITE 13,1 #TURN ON LED ON PIN 13
120 FOR I=1 TO 500    #DELAY
130 NEXT I
140 DWRITE 13,0 #TURN OFF THE LED
150 FOR I=1 TO 500
160 NEXT I
170 NEXT K
```

```
AWRITE <PIN>, VALUE
DWRITE <PIN>, VALUE
```

PIN can be any ATMega328 pin number. VALUE can be from 0 to 255. For most items, a one will 'turn on' and a zero will 'turn off' the item. In the example above, the first DWRITE turned on the 'pin 13' LED that is on your console board (it's the RED LED) while the second DWRITE statement turned it off.

OUT A special version of DWRITE exists but only for the TX pin. This statement allows communication on the first Serial port. If you connect your console to your computer using the FTDI connector, open a terminal window on your computer (you can do this in the Arduino IDE by pressing CTRL SHIFT M) and then typing:

OUT 65

And you should see an uppercase A appear in the terminal window.

To use:

```
OUT <value>
```

```
100 CLS
110 PRINT "Value";
120 INPUT V
130 OUT V
140 GOTO 110
```

MEM MEM displays the remaining free memory on your screen.

TONE

TONE emits a tone through the audio output of the console. It takes two parameters, tone and duration.

```
TONE note, duration
```

```
100 CLS
110 ?”Duration “;:INPUT D
120 IF D<0 D=0
130 FOR T=1 TO 1024
140 TONE T,D
150 IF INKEY(0)=27 STOP
160 NEXT T
```

This example asks the user for the duration of the tones, checks for a valid duration and then loops through 1024 tones, playing them for the specified duration. If the user wishes to exit, they press ESC.

If you look at line 110, you will notice we have used several space saving techniques: use of the question mark for PRINT and putting two statements on one line. The semicolon after the quote tells BASIC NOT to issue a new line, but stay on the current line. This will place the question mark prompt of the INPUT statement right after the space in Duration, so it will appear like this:

Duration ?

If the user enters a number below zero, say ‘-3’, it will become a zero so the tones will not play an infinite amount of time.

The higher the tone value, the higher the pitch. It is possible to create ‘songs’ using the string technique (explained elsewhere) in combination with TONE. Store the tone values (upto 255) in a string and then read the string back but send the output to TONE instead of PRINT:

```
100 M=TOP(0):@M, “ABCDEFGHIJK”
110 FOR K=M TO M+10
120 T=PEEK(K): TONE T,10
130 NEXT T
```

Remember, when using the string method (or POKE in general) you need to make sure you do not clobber your program memory. Always find the top of memory.

DELAY

Sometimes, you need to introduce a delay in your code, to temporarily stop processing to allow something to finish or just slow down your code. You may want to do this to give the user time to read something, allow tones to complete playing, freeze the display for a bit, etc. There are lots of reasons to do this and you can do so with FOR-NEXT loops (as seen in some of the examples) but a better way is to use the DELAY statement.

```
DELAY time
```

```
100 CLS
110 FOR K=1 TO 80
120 SET K, 10          #LIGHT UP THE PIXEL
130 DELAY 1000         #DELAY FOR A SECOND
140 RESET K,10         #TURN OFF THE PIXEL
150 NEXT K
```

This code clears the screen and starts to move a pixel across the screen without leaving a trail. Line 120 sets the pixel, 130 stops execution for about a second and then the pixel is removed in line 140.

SHIFT

Tiny Basic provide a quick and easy way to shift the entire screen in any of four directions: up, down, left or right. This can be useful in writing games. The SHIFT statement provides this capability. To use:

```
SHIFT pixels, direction
```

```
100 CLS
110 FOR K=1 TO 20
120 SET RND(50),RND(40)
130 NEXT K
140 P=PAD(0)
150 IF P>200 SHIFT 1,3
160 IF P<100 SHIFT 1,2
170 IF PAD(2)=1 STOP
180 GOTO 140
```

First, we put twenty random pixels on screen. Then, we read the Nunchuck and if the user moves left, shft the screen left. If they move right, move right.

The direction to shift values:

- `UP = 0`
- `DOWN = 1`
- `LEFT = 2`
- `RIGHT = 3`

INVERT

One thing you can do to get a user’s attention is to flash the screen. Tiny Basic allows you to do this quickly, by using the INVERT statement. It takes no parameters and its syntax is simply:

```
INVERT
```

```
100 CLS
110 CURSOR 3,2
120 PRINT “TO CONTINUE WITH”
130 CURSOR 3,3
140 PRINT “PROCESSING, PRESS C”;
150 A=INKEY(0)
160 IF A=67 GOTO 210
170 INVERT
180 DELAY 500
190 GOTO 150
…
210 #CONTINUE…
```

Starting at line 150, we wait for a key to be pressed. If it is C, goto 210 else, we invert the screen, wait a half second and do it again. If there is no delay, the screen would be just obnoxious. Inserting a delay slows it down a bit.

FUNCTIONS EXPLAINED

ABS

ABS returns the absolute value of a number. All it really does is make a negative number a positive number. Use it when it does not matter if a number is negative or for formatting reasons.

```
110 X=ABS(X)
```

If X is below zero, make x positive.

RND

RND returns a random number from zero to the upper limit specified.

```
100 X=RND(100)
110 IF X<50 X=X+50
120 PRINT “Your random number (from 50 to 100) is “, X
```

This code will generate a random number from 50 to 100. First, it creates the random number from 0 to 100. Then, it checks to see if the number is less than 50. If it is, it will add 50 to the value. This, effectively, makes the number random from 50 to 100.

PEEK

PEEK or @ will return the value of the specified memory location.

```
100 X=@1450
110 IF X=13 PRINT “End of line.”
```

You can use PEEK to examine a line of text entered by the user:

```
1500 #PRINT the name entered
1510 M=N
1520 X=@M
1530 IF X=13 RETURN
1540 PRINT CHR(X);
1550 M=M+1
1560 GOTO 1520
```

Several things in the example to note are:

- Use of the ‘@’ sign instead of PEEK
- Use of the IF statement
- Putting RETURN in a place other than at the end of the subroutine-this is the only legal way to do that.
- How to print the user’s name-provided it is in the memory location at M.
- Use of CHR to print the ASCII representation of the memory location M

INKEY

INKEY will return the ASCII value of the key that was pressed. This function ONLY CHECKS THE KEYBOARD ONE TIME. In other words, it does not wait for a keypress.

```
1600 #Get User Text
1610 M=N
1620 A=INKEY(0)
1630 IF A=-1 GOTO 1620
1640 IF A=13 @M,13: RETURN
1650 PRINT CHR(A);
1660 @M,A
1670 M=M+1
1680 GOTO 1620
```

This subroutine lets you get a line of text from the user. If no key has been pressed, A will get a zero from INKEY. Once a key is pressed, its ASCII value is returned to A. If the user presses ENTER, a carriage return is put in memory to denote the end of the line and control is returned to the calling code. Everything else is stored in memory and the memory location is incremented by one. NOTE: no check is made to make sure the memory location is valid. It is assumed to be when the subroutine is called. You specify the value of the memory location in the 'N' variable. You can use any variable you wish, just change the 'N' in line 1610 accordingly.

CHR

CHR, used with PRINT, will display the visual representation of the value specified.

```
100 C=65
110 PRINT CHR(C)
```

Will result in an A being displayed on screen.

TOP

TOP will return the location AFTER the end of your program. This is useful to know when using memory for string storage. This can keep you from stepping on your code.

```
100 A=TOP(0)
110 POKE A,65
120 PRINT PEEK(A),"=",CHR(PEEK(A))
```

This little snippet of code will put the letter 'A' in the first free spot of memory. It then prints out what is in that spot and its visual representation, an 'A' in this case.

PAD

PAD(0) and PAD(1) will read the x and y values from the thumbstick of the Wii Nunchuk. PAD(0) will return the X-axis value and PAD(1) will return the Y-axis value. PAD(2) and PAD(3) return the C and Z buttons while PAD(4), PAD(5) and PAD(6) return the x,y and z positions of the accelerometer.

```
100 X=1: Y=1          #INITIALIZE OUR START
110 SET(X,Y)
120 P=PAD(0)          #GET THE X VALUE
130 Q=PAD(1)          #GET THE Y VALUE
140 IF P>200 X=X+1    #IF VALUE IS MORE THAN 200, GO RIGHT
150 IF P<100 X=X-1    #IF VALUE IS LESS THAN 100, GO LEFT
160 IF Q>200 Y=Y+1
170 IF Q<100 Y=Y-1
180 SET(X,Y)          #LIGHT UP THE PIXEL
190 IF PAD(3) STOP    #IF Z BUTTON IS PRESSED, STOP
200 IF PAD(2) RUN     #IF C BUTTON IS PRESSED, RESTART
210 GOTO 120          #DO IT AGAIN
```

This code will read the x and y axis of the Wii Nunchuk and move either left, right, up or down. Pressing the 'Z' button end and the 'C' button will clear the screen and allow you to start over.

PAD number	Return value
0	Thumbstick x
1	Thumbstick y
2	'C' button
3	'Z'button
4	Accelerometer x
5	Accelerometer y
6	Accelerometer z

GET

GET will return the value of a pixel at a given point. So, if the pixel at, say, 5,5 is lit, the function returns a one. If it is not, it will return a zero. This is useful for detecting a collision in a game.

```
100 CLS
110 x=rnd(40): y=rnd(20)
```

```
120 set x,y
130 x=rnd(40):y=rnd(20)
140 if get(x,y)=1 reset x,y
150 goto 120
```

This example puts random pixels on the screen, generates another set of random locations, tests the location and, if it is lit up, then turn it off. This will keep the screen from filling up.

STRING ASSIGNMENT

You can store a string directly in a memory location, in code, by using a quoted string in your POKE statement. ONLY quoted strings are allowed.

```
100 #QUOTED STRING EXAMPLE
110 A=TOP(0)
120 @A,”MY QUOTED STRING”
130 FOR Z=A TO A+64
140 B=@Z
150 IF B=13 Z=A+64
160 PRINT CHR(B);
170 NEXT Z
```

When you assign a quoted string to a memory location, a carriage return is appended to the string. You can use that carriage return to then look for the end of the string.

Using a quoted string is good if you have several strings you will use over and over.

AREAD, DREAD

AREAD and DREAD will retrieve the current value of the specified PIN. AREAD will return a value from 0 to 1023 that corresponds to the voltage of said pin. DREAD returns a 0 or 1. 0 is LOW, 1 is HIGH. To use:

```
A=DREAD(PIN)
```

```
100 A=DREAD(0)  #GET VALUE OF RX
110 IF A=0 GOTO 100 #LOOP UNTIL THE PIN GOES HIGH
120 PRINT”RX PIN IS HIGH.”
```

Mathematical Operations

Integer math operations (i.e. no decimals, only ‘whole’ numbers) include:

- Addition
- Subtraction
- Multiplication
- Division
- Absolute value
- Power ***(NOTE: there is a math error, not in Tiny Basic, in the compiler that results in incorrect return values. You will need to add one to the results. For example, 2^2 should return a value of 4, however, it returns a value of 3. Add a 1 to the value to get the correct result. We are looking at a fix for this issue.)***