PBASIC Programming

Program Structure

Humans often think of computers and microcontrollers as “smart” devices and yet, they will do nothing without a specific set of instructions. This set of instructions is called a program. It is our job to write the program. Stamp programs are written in a programming language called PBASIC, a Parallax-specific version of the BASIC (Beginners All-purpose Symbolic Instruction Code) programming language. BASIC is very popular because of its simplicity and English-like syntax.

A working program can be as simple as a list of statements. Like this:

```
statement1
statement2
statement3
END
```

Comments

Comments are included in a program for readability and notes. These statements are not loaded into the microcontroller and do not take up any program space.

Each comment starts with an apostrophe. Comments can be on lines by themselves, or to the right of PBASIC programming statements.

Here’s an example of using comments:

```
' Program 1: LED
' This program lights up an LED

' Hardware Setup:
' Active-low LED connected to Stamp Pin P7

HIGH 7                                          ' Turn LED on
PAUSE 500                                       ' for half second
LOW 7                                           ' Turn LED off

END
```

Capitalization

PBASIC is not case-sensitive. PBASIC keywords, variables, and constants can be entered in either upper or lower case, with no change in meaning.

```
debug “Hello”              ‘ All these commands are interpreted the same
DEBUG “Hello”
DeBuG “Hello”
Debug “Hello”

counter      VAR    Byte

counter = 1   ‘ All these variable names are interpreted the same
Counter = 1
COUNTER = 1
```

The DEBUG Statement

DEBUG outputdata {, outputdata}

```
DEBUG "Hello World!"
```

Output:

```
Hello World!
```

```
DEBUG "Hello World!"
DEBUG "and Hello again”
```

Output:

```
Hello World!and Hello again
```

```
DEBUG "Hello World!", CR
DEBUG "and Hello again”
```

Output:

```
Hello World!
and Hello again
```

```
DEBUG "Hello World!", CR, “and hello again”
```

Output:

```
Hello World!
and Hello again
```

```
DEBUG CR, CR, “Hello World”, CR, “and hello again”
```

Output:

```


Hello World!
and Hello again
```

Debug Control Characters

CR	Carriage Return
CLS	Clear screen, cursor positioned in upper left
HOME	Moves cursor to upper left corner but doesn’t clear screen
BELL	Makes a sound
BKSP	Backspace
TAB	Tab

Variables

Variable Sizes

The BASIC Stamp supports four variable types. For the most efficient use of the Stamp's memory, a variable should be defined based on program requirements. Using a Word variable, for example, when the value will never exceed 15 is an inefficient use of the Stamp's variable memory space.

Type	Bits	Range
Bit	1	0 .. 1
Nib	4	0 .. 15
Byte	8	0 .. 255
Word	16	0 .. 65,535

Variable Declaration

Variables must be declared before they can be used in a program. All variable declarations are usually placed together at the top of a program. To declare a variable, enter the name of the variable, the keyword VAR, and the size of the variable.

Here's an example of declaring several variables:

```
flag          VAR    Bit
counter       VAR    Nib
status        VAR    Byte
```

Printing Variables using DEBUG

```
x             VAR    Byte
x = 65

DEBUG "Printing variables", CR
DEBUG ? x                               ' Shorthand to print "x = ", value, CR
DEBUG x, CR                     ' ASCII value!
DEBUG DEC x, CR                 ' Decimal
DEBUG IBIN x, CR                ' Indicated Binary, starts with % sign
DEBUG IHEX x, CR                ' Indicated Hex, starts with $ sign
                                ' Printing variables along with text
DEBUG "The temperature is ", DEC x, " degrees", CR
```

Debug Formatters

?	Shorthand notation. Prints "var = <value>", CR
DEC	Decimal
IHEX	Indicated hexadecimal
IBIN	Indicated binary
STR	String from BYTEARRAY
ASC	ASCII

Note: The default is ASCII! To print a number, you must include a formatter.

Using Variables

```
x            VAR    Byte
y            VAR    Byte
result       VAR    Byte

  x = 25
  y = 2

  result = x - y                                    ' Subtraction
  DEBUG "x - y = ", DEC result, CR

  result = x / y                                    ' Division
  DEBUG "x / y  = ", DEC result, CR

  result = x + y * 10                               ' Order of operations
  DEBUG "x + y * 10 = ", DEC result, CR
```

Output:

```
x - y = 23
x / y = 12
x + y * 10 = 14
```

Constants

Constants are values that cannot change during the execution of program. Named constants will usually assist the reader in understanding the nature of the program and, in many cases, assist the original programmer that has set the program aside for some time. The value of a constant can only be changed by editing the source code.

Constant Declaration

Named constants must be declared before they can be used in a program. All constant declarations are usually placed together at the top of a program. To declare a constant, enter the name of the constant, the keyword CON, and the value of the constant.

Here’s an example of declaring several constants:

```
MaxTemp       CON    212
MidPoint      CON    750
```

Using Constants

```
ScaleFactor   CON    100
degreeF       VAR    Byte

degreeF = degreeF / ScaleFactor
```

Declaring Pin Numbers as Constants

One very useful application of constants is to stand for BASIC Stamp pin numbers. This practice makes a program more understandable.

For example, suppose we have an LED connected to the BASIC Stamp's pin number P8, and a servo motor connected to pin number P12. Instead of using the numerals "8" and "12" throughout the program, we can declare them as constants.

```
Led           CON    8                    ' LED is connected to Pin P8
Servo         CON    12                   ' Servo motor connected to Pin P12
```

Branching – Redirecting the Flow of a Program

A branching command is one that causes the flow of the program to change from its linear path. In other words, when the program encounters a branching command, it will, in almost all cases, not be running the next [linear] line of code. The program will usually go somewhere else. There are two categories of branching commands: *unconditional* and *conditional*. PBASIC has two commands, **GOTO** and **GOSUB** that cause unconditional branching.
Here's an example of an unconditional branch using **GOTO**:

GOTO and Labels

```
Greeting:
  DEBUG "Hello World!", CR
  PAUSE 500
  GOTO Greeting
```

We call this an *unconditional* branch because it always happens. **GOTO** redirects the program to another location. The location is specified as part of the **GOTO** command and is called an address. Remember that addresses start a line of code and are followed by a colon (:). You'll frequently see **GOTO** at the end of the main body of code, forcing the program statements to run again.

Conditional branching will cause the program flow to change under a specific set of circumstances. The simplest conditional branching is done with **IF-THEN** construct. The PBASIC **IF-THEN** construct is different from other flavors of BASIC. In PBASIC, **THEN** is always followed by a valid program address (other BASICs allow a variety of programming statements to follow **THEN**). If the condition statement evaluates as TRUE, the program will branch to the address specified. Otherwise, it will continue with the next line of code.

IF-THEN

General format:

```
Start:
  statement 1
  statement 2
  statement 3
  IF condition THEN Start
```

The statements will be run and then the condition is tested. If it evaluates as TRUE, the program will branch back to the line called Start. If the condition evaluates as FALSE, the program will continue at the line that follows the **IF-THEN** construct.

IF condition THEN label

```
IF (controlVar = 0) THEN Label_0
IF (controlVar = 1) THEN Label_1
IF (controlVar = 2) THEN Label_2
```

Looping – Running Code Again and Again

GOTO LOOP

Looping causes sections of the program to be repeated. Looping often uses unconditional and conditional branching to create the various looping structures. Here's an example of *unconditional looping*:

```
Greeting:
  DEBUG "Hello World!", CR
  PAUSE 500
  GOTO Greeting
```

By using **GOTO** the statements are unconditionally repeated, or looped. By using **IF-THEN**, we can add a conditional statement to the loop. The next few examples are called *conditional looping*. The loops will run under specific conditions. Conditional programming is what gives microcontrollers their "smarts."

CONDITIONAL LOOPING WITH IF-THEN

```
Label:
  statement 1
  statement 2
  statement 3
  IF condition THEN Label
```

With this loop structure, statements will be run so long as the condition evaluates as TRUE. When the condition is evaluated as FALSE, the program will continue at the line following the **IF-THEN** statement. It's important to note that in the previous listing the statements will always run at least once, even if the condition is FALSE.

FOR-NEXT LOOP

The final example of conditional looping is the programmed loop using the **FOR-NEXT** construct.

```
FOR controlVar = startVal TO endVal STEP stepSize
  statement 1
  statement 2
  statement 3
NEXT
```

The **FOR-NEXT** construct is used to cause a section of code to execute (loop) a specific number of times. **FOR-NEXT** uses a control variable to determine the number of loops. The size of the variable will determine the upper limit of loop iterations. For example, the upper limit when using a byte-sized control variable would be 255.

```
counter       VAR    Nib

PAUSE 250                               ' let DEBUG window open

FOR counter = 1 TO 5
  DEBUG "Loop Number: ", DEC counter
  DEBUG "  Hello World!", CR
  PAUSE 500
NEXT
```

Output:

```
Loop Number: 1 Hello World!
Loop Number: 2 Hello World!
Loop Number: 3 Hello World!
Loop Number: 4 Hello World!
Loop Number: 5 Hello World!
```

The **STEP** option of **FOR-NEXT** is used when the loop needs to count increments other than one. If, for example, the loop needed to count even numbers, the code would look something like this:

```
FOR controlVar = 2 TO 20 STEP 2
  statement 1
  statement 2
  statement 3
NEXT
```

Subroutines – Reusable Code that Saves Program Space

The final programming concept we'll discuss is the subroutine. A subroutine is a section of code that can be called (run) from anywhere in the program. **GOSUB** is used to redirect the program to the subroutine code. The subroutine is terminated with the **RETURN** command. **RETURN** causes the program to jump back to the line of code that follows the calling **GOSUB** command.

GOSUB and RETURN

```
PAUSE 250

Main:
  GOSUB Hello
  GOSUB Goodbye
  END

Hello:
  DEBUG "Hello there!", CR
  RETURN

Goodbye:
  DEBUG "Bye now!", CR
  RETURN
```

BASIC Stamp Memory

The BASIC Stamp has two kinds of memory; RAM (for variables used by your program) and EEPROM (for storing the program itself). EEPROM may also be used to store long-term data in much the same way that desktop computers use a hard drive to hold both programs and files.

An important distinction between RAM and EEPROM is this:

- RAM loses its contents when the BASIC Stamp loses power; when power returns, all RAM locations are cleared to 0s.
- EEPROM retains the contents of memory, with or without power, until it is overwritten (such as during the program-downloading process or with a WRITE instruction.)

The BS2, BS2e, BS2sx and BS2p have 32 bytes of Variable RAM space. Of these, the first six bytes are reserved for input, output, and direction control of the I/O pins. The remaining 26 bytes are available for general-purpose use as variables.

I/O Registers

- Occupy first 3 words RAM (6 bytes)
- 16-bit registers (Stamp has 16 I/O pins)
- Are all initialized to zero
- All pins set to inputs by default

Names of I/O Registers

- **INS** Shows state of I/O pins regardless whether input or output
- **OUTS** Write values into here to make pin high (1) or low (0)
- **DIRS** 0=Input, 1 = Output

Reserved Names for Referring to I/O Registers

INS REGISTER

Name	**Size**
IN0 - IN15	Bit
INA, INB, INC, IND	Nibble
INL, INH	Byte
INS	Word

INS REGISTER REFERENCE																
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
INA																
INB																
INC																
IND																
INL																
INH																
INS																

OUTS REGISTER

Name	**Size**
OUT0 – OUT15	Bit
OUTA, OUTB, OUTC, OUTD	Nibble
OUTL, OUTH	Byte
OUTS	Word

OUTS REGISTER REFERENCE																
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OUTA																
OUTB																
OUTC																
OUTD																
OUTL																
OUTH																
OUTS																

DIRS REGISTER
0 = INPUT, 1 = OUTPUT

Name	**Size**
DIR0 .. DIR15	Bit
DIRA, DIRB, DIRC, DIRD	Nibble
DIRL, DIRH	Byte
DIRS	Word

DIRS REGISTER REFERENCE																
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DIRA																
DIRB																
DIRC																
DIRD																
DIRL																
DIRH																
DIRS																

To Specify a Pin as Output

Since all pins default to inputs, you must specify which pins you wish to use as outputs. There are a number of ways to do this:

1. Use DIRS register
   Write a "1" for "Output"

```
DIRS = %00110000000000000000          ' Outputs:  13, 12
DIR4 = 1                              ' Outputs: 4
```

2. Use OUTPUT keyword

```
OUTPUT 7                              ' Outputs: 7
```

3. Use HIGH or LOW keywords
   These set the direction, and write a value

```
HIGH 5         '  Specifies that pin P5 is an output, and sets it high
LOW 3          '  Specifies that pin P3 is an output, and sets it low
```

4. Use keywords that do it for you
   No need to use OUTPUT or DIRS first
   FREQOUT, PULSOUT, SEROUT, ...

Examples of I/O Register Usage

```
DIRS = %0011000000000000                     ' Outputs:  13, 12

if (IN1 = 1) THEN Do_Something
```

Aliases

An alias is an alternate name for an existing variable. One of the most useful applications of aliases is to create alternate names for the Stamp's built-in variable names used for input and output. This can greatly increase a program's readability and understandability. To declare an alias, enter the alias, the keyword VAR, and the name of the existing variable.

Here's an example of creating two aliases, called "btn" and "LED", which refer to BASIC Stamp pins P7 and P8, respectively.

```
btn           VAR    IN7                             ' name (alias) the input
LED           VAR    OUT8                            ' name (alias) the output
```