PBasic for Kids Documentation

Release 1.0

Rogelio Negrete

February 09, 2017

This is a beginner's guide for getting started on PBasic which is a programming language created by Parralax as an easy way to interact with their various robots. For new beginners it can be a little hard to grasp various concepts of programming but PBasic does a good job at being a language to power the robots while still being user friendly.

CHAPTER 1

Getting Started

Before we get started here are a couple of things you should consider downloading: (*For Windows*)

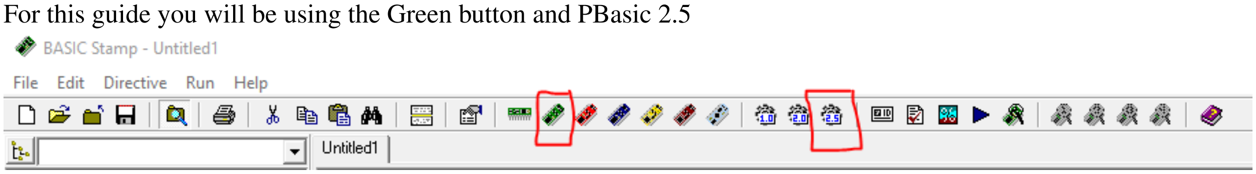
1. Parralax USB Driver - USB Driver to recognize the conncetion between your computer and parralax robot
2. BASIC Stamp Editor - A basic IDE to run your PBasic files on your robot.

We'll be starting off with the easy concepts first as they are part of the core to build upon later on. First on the list is Variables!

1.1 Prepping your Basic Stamp Editor

Before you attempt to run any files make sure to click the top 2 buttons to create type headers at the top of your file. These type headers are required in order to for the Stamp Editor to know which version of PBasic to compile and run on your robot(s).

For this guide you will be using the Green button and PBasic 2.5



Comments in PBasic

Comments in PBasic are ignored when the program runs. Think of it as a way to make notes of what your program does at certain situations so you can remember later. In PBasic, the way comments are denoted is by using the apostrophe symbol: **‘**

For example:

```
' This is a comment!

DEBUG "Hello World!"

' Another comment!!!
```

CHAPTER 3

Variables

Variables are a way to temporarily store data. Think of it as the same as variables in your math class where you can define x = 5

3.1 Defining and Using Variables

Before you can use a variable in a PBASIC program you must declare it. “Declare” means letting the BASIC Stamp know that you plan to use a variable. The format for declaring variables is as follows

```
variable_name    VAR     VarType
```

VarType refers to the following 4 values: **Bit**, **Nib**, **Byte**, and **Word**. Try to think of a variable type as classifying how much space the variable has to store values.

VarType	Value Size
Bit	Value can be 0 or 1
Nib	Value can be 0 to 15
Byte	Value can be 0 to 255
Word	Value can be 0 to 65535

3.1.1 Here’s an analogy of how the size difference works

We can arrange these 4 objects in order by how much they can store: Envelope, Shoebox, Fridge, Room

```
Envelope (Bit) < Shoebox (Nib) < Fridge (Byte) < Room (Word)
```

3.1.2 Some examples of declaring variables

Choose variable names that make sense to you and are not absurd like: ThisVariable_DoessomethingreallyCOOL

```
x               VAR     Bit
dog             VAR     WORD
is_zero         VAR     Nib
someVariable    VAR     Byte
```

3.2 Some Notes on Variable Types

Under certain situations you might use different variable types. However, for the programming problems that you will encounter while undergoing the competition it might be best to just stick to **Byte** and **Word**

Printing output to the Terminal

DEBUG is used to print output to the computer screen while running your program. Think of it as a way to make sure things are are running properly while your program runs.

The easiest use case is regular messages:

```
DEBUG   "Hello, World!"
DEBUG   "I'm learning how to program."
```

4.1 Using the comma seperator

We can have multiple messages added together on the same line by using the comma seperator:

```
DEBUG   "Wow this is a", " multi message!"
```

4.2 Printing on new lines

We can use the keyword (CR) to start on a new line. Think of it like pressing enter in Microsoft Word:

```
DEBUG   "This should be on", CR
DEBUG   "multiple lines."
```

4.3 Printing variables

We can also print variables:

```
x   VAR     Word

Init:
    x = 65

Main:
    DEBUG x
    END
```

Uh oh! When trying to run the above code there should be an issue. It's printing the letter "A"?! This is because by default the BS2 model displays everything as ASCII characters. I won't go into detail what ASCII is but you can follow the link to read more.

Anyways, in order to properly print the value of *x* we need to use the decimal formatter, **DEC**:

```
x    VAR      Word

Init:
     x = 65

Main:
     DEBUG DEC x
     END
```

4.4 Auto-printing variables

Using the keyword (?) we can auto-print the variable name and value:

```
x    VAR      Word

Init:
     x = 65

Main:
     DEBUG DEC ? x
     END
```

4.5 Example of combining everything together

```
x    VAR      Word
y    VAR      Word

Init:
     x = 65
     y = 99
Main:
     DEBUG DEC "Our value of x is: ", x, CR
     DEBUG DEC ? y
     END
```

CHAPTER 5

Conditional Statements

Conditional statement are used as a way to direct the way things operate. For example, if I say "Please go to the store to buy milk. If they don't have milk then buy apple juice".

Notice how If there isn't milk then we buy apple juice. However if there IS milk then we buy milk.

These types of conditional statements are ordered like this in PBasic:

```
IF (condition) THEN
    statement(s)
ENDIF
```

A condition is made up of comparison symbols

Comparison Operator Symbol	Definition
=	Equal
<>	Not Equal
>	Greater Than
<	Less Than
>=	Greater Than or Equal To
<=	Less Than or Equal To

Here are some examples:

```
IF (4 = 5) THEN
    DEBUG "4 equals 5"
ENDIF

IF (10 <= 100) THEN
    DEBUG "10 is less than or equal to 100"
ENDIF
```

5.1 Chaining mutliple IF.. THEN statements together

You can also call chain multiple IF.. THEN statements together through the use of IF.. ELSEIF.. and/or ELSE..

Structure for Multiple If statements:

```
IF (condition) THEN
    statement(s)
ELSEIF (condition) THEN
    statement(s)
ELSE
```

```
    statement(s)
ENDIF
```

Example:

```
x   VAR     WORD

Main:
    x = 100

    IF (x < 200) THEN
        DEBUG DEC ? x
    ELSEIF (x < 50) THEN
        DEBUG DEC ? x
    ELSE
        DEBUG DEC ? x
    ENDIF
```

5.1.1 Notes about Mutliple If statements

It's not necessary to have an ELSE statement at the end. If it's omitted then the statement will stop at the last ELSEIF statement instead.

Which means that this is also a valid IF Statement:

```
x   VAR     WORD

Main:
    x = 100

    IF (x < 200) THEN
        DEBUG DEC ? x
    ELSEIF (x < 50) THEN
        DEBUG DEC ? x
    ENDIF
```

5.2 Conditional Logic Operators

```
IF (condition) THEN
    statement(s)
ENDIF
```

A condition is also made up of logic operators:

1. NOT
2. AND
3. OR

Logic operators are a little more confusing. The reason to use logic operators is to do multiple comparisons in one IF statement. Take for example:

```
IF (5 < 10) AND (1 < 5) THEN
    DEBUG "Hello there!"
ELSE
```

```
    DEBUG "Goodbye!"
ENDIF
```

Here we have two conditions that we test inside one IF statement **AND** only if they are both true will you see "Hello there!" printed.

The following tables and examples may help make clear how logic operators work together:

5.2.1 Logic Operator: NOT

```
IF NOT (1 > 10) THEN
    DEBUG "Hello World!"
ELSE
    DEBUG "Goodbye"
ENDIF

' Result: True
```

Condition A	NOT A
False	True
True	False

5.2.2 Logic Operator: AND

```
IF (1 > 10) AND (4 = 4) THEN
    DEBUG "Hello World!"
ELSE
    DEBUG "Goodbye"
ENDIF

' Result: False
```

Condition A	Condition B	A AND B
False	False	False
False	True	False
True	False	False
True	True	True

5.2.3 Logic Operator: OR

```
IF (1 > 10) OR (4 = 4) THEN
    DEBUG "Hello World!"
ELSE
    DEBUG "Goodbye"
ENDIF

' Result: True
```

Condition A	Condition B	A OR B
False	False	False
False	True	True
True	False	True
True	True	True

5.3 Nesting IF Statements

You also have the ability to nest IF statements inside of each other like so:

```
x    VAR     WORD

Main:
    x = 7

    IF (x < 10) THEN
        IF (x > 5) THEN
            DEBUG "x is between 5 and 10"
            DEBUG DEC ? x
        ENDIF
    ENDIF
```

Try to think of nesting as asking another question once you received an answer to your previous question. For example:

```
IF (joe went to the store)
    IF (he did buy chocolate)
        "Joe bough chocolate at the store"
    ELSEIF (he did buy milk)
        "Joe bought milk at the store"
    ELSE
        "Joe bought apple juice at the store"
ELSE
    "Joe never went to the store"
```

Do Loops

Lets say you want to do something forever... in programming you would use a do-loop to perform this action!

Here's a basic example that constantly prints to the terminal:

```
DO
    DEBUG "Hi there!", CR
LOOP
```

Here's another example that prints the value of x and increases its value:

```
x    VAR      WORD

Init:
    x = 1
Main:
    DO
        DEBUG DEC ? x, CR
        x = x + 1
    LOOP
```

6.1 DO-WHILE loop

However, more often than not you will want to test some condition to determine whether the loop code should run or continue to run.

To do this we use a DO-WHILE loop like so:

```
x    VAR      WORD

Init:
    x = 1
Main:
    DO WHILE (x <= 5)  ' condition to test before entering loop statements
        DEBUG "#", CR
        x = x + 1
    LOOP
```

6.2 Conclusion

DO loops are useful when you need to run something forever or until some special condition breaks.

For an imaginary example:

```
Main:
    DO WHILE (some_special_condition = 1)
        ' Do some calculations
    LOOP
```

FOR Loops

For loops are a little different than do-loops. For loops were created with the purpose in mind of having a program execute between a range. That range is defined by you!

Here's an example that counts from 0 to 10:

```
x    VAR      WORD

Main:
    FOR x = 0 TO 10
        DEBUG DEC ? x, CR
    NEXT
    END
```

By default, a FOR loop will step through 1 by 1. We can change this behavior by adding a specific value for STEP.

Here the example counts from 0 to 10 but increasing by STEPS of 2:

```
x    VAR      WORD

Main:
    FOR x = 0 TO 10 STEP 2
        DEBUG DEC ? x, CR
    NEXT
    END
```

Notice how only even numbers are being displayed!

We can also make a FOR loop that decreases in range. Here's what I mean:

```
Main:
    FOR 10 TO 5
        DEBUG "Hello!"
    NEXT
    END
```

7.1 Conclusion

FOR loops are very useful when you know there should be a range where a program should run. If we need to run something 10 times then it would be useful to use a FOR loop as its easy to create.

Take this for example, printing 1 to 10 by hand:

```
Main:
    DEBUG "1"
    DEBUG "2"
    DEBUG "3"
    DEBUG "4"
    DEBUG "5"
    DEBUG "6"
    DEBUG "7"
    DEBUG "8"
    DEBUG "9"
    DEBUG "10"
    END
```

VS

Printing 1 to 10 using a for loop:

```
x   VAR     WORD

Main:
    FOR x = 1 to 10
        DEBUG DEC x
    NEXT
    END
```

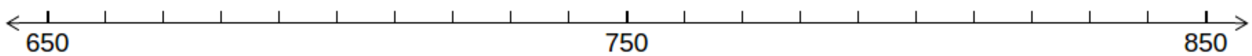
CHAPTER 8

Movement

Moving the wheels of the robot is fairly simple. We will use the **PULSOUT** keyword to send a signal to the wheels to turn. Each wheel has a unique ID and takes in a range of “power values” for how fast the wheel spins and in what direction.

Wheel	ID	Power Value
Right	12	650 <=> 850
Left	13	650 <=> 850

The power values dictate how fast the wheel spins in a certain direction. Think of a number line where 650 and 850 are at the ends and 750 is the center.



Consider 750 to be the neutral value. This means if you set a wheel to a value of 750 it shouldnt move.

If you set a wheel to either 650 or 850 then it will move at full power in a certain direction.

Power Value	Direction
650	Clockwise
750	None
850	Counter-Clockwise

8.1 Moving Forward

In order to move the robot forward we need to spin each wheel either counter-clockwise or clockwise but not the same. Running this code below will make the wheels move in a very short burst.

```
PULSOUT 13, 850
PULSOUT 12, 650
```

To continuously go forwards for a small time we program it like so:

```
i   VAR     WORD

FOR i=1 TO 100
    PULSOUT 13, 850
    PULSOUT 12, 650
NEXT
```

8.2 Moving Backwards

We have the same idea as moving forwards except the values are flipped.

```
PULSOUT 13, 650
PULSOUT 12, 850
```

And again to continuously go backwards for a small time we program it like so:

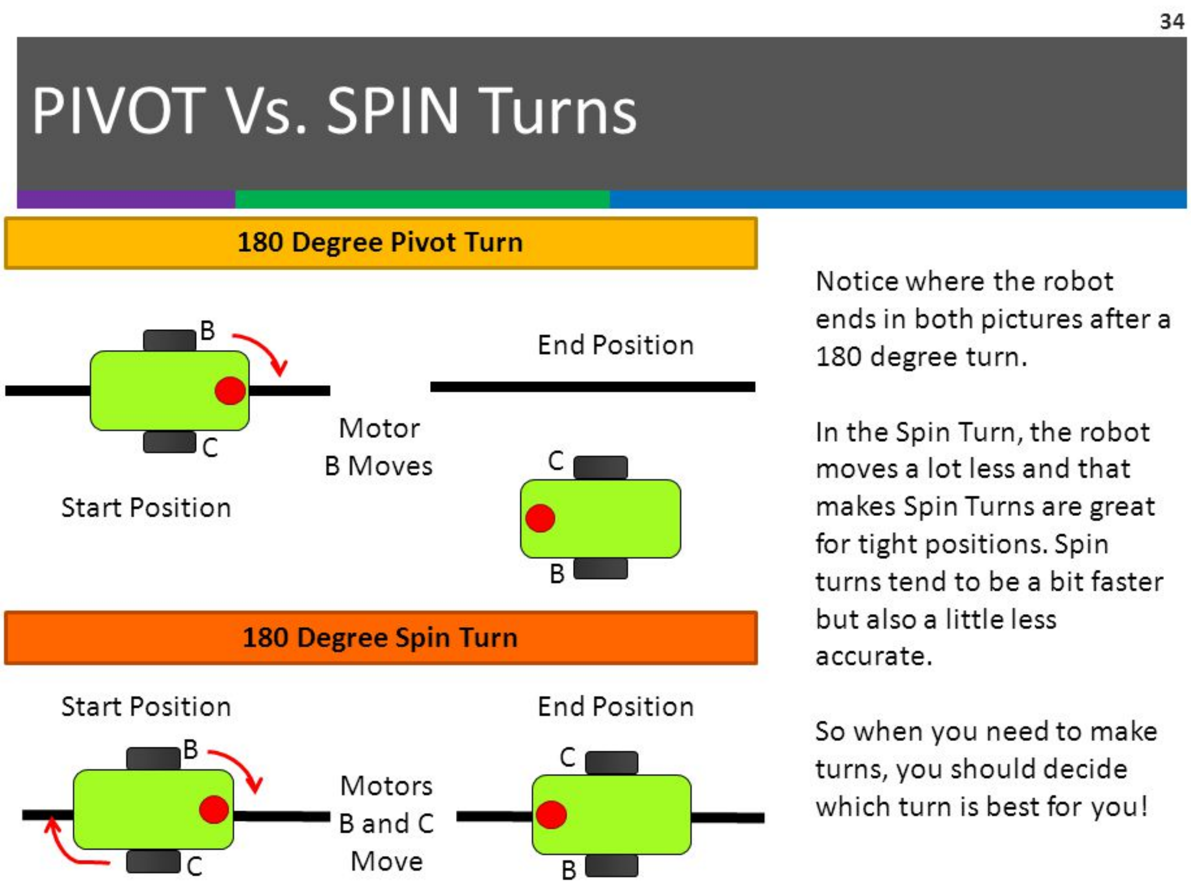
```
i   VAR     WORD

FOR i=1 TO 100
    PULSOUT 13, 650
    PULSOUT 12, 850
NEXT
```

8.3 Turning

There are 2 approaches to turning your robot.

1. Pivot Turn
2. Spin Turn

This diagram helps to explain the key differences:



Ultimately what type of turns you want to perform is up to you. Just make sure you're consistent with the type of turns you perform.

8.3.1 Pivot Turn

Depending on the wheel you want to pivot about influences what code to use.

Pivot about Left Wheel:

```
i   VAR     WORD

FOR i=1 TO 100
    PULSOUT 12, 650
NEXT
```

Pivot about Right Wheel:

```
i   VAR     WORD

FOR i=1 TO 100
    PULSOUT 13, 650
NEXT
```

8.3.2 Spin Turn

Spin turns move both wheels in the same direction either clockwise or counter-clockwise.

Spinning in Clockwise direction:

```
i   VAR     WORD

FOR i=1 TO 100
  PULSOUT 13, 650
  PULSOUT 12, 650
NEXT
```

Spinning in Counter-Clockwise direction:

```
i   VAR     WORD

FOR i=1 TO 100
  PULSOUT 13, 850
  PULSOUT 12, 850
NEXT
```

8.4 Practice

I'd like to challenge you to program your robot to move forward, spin in some direction, and then backup with what you've learned so far. In addition, you should try to practice more by programming your own little movement sequence.

CHAPTER 9

Subroutines

Imagine you have a "special piece of code" that's 10 lines long. And you have to use it 7 times in your program. Now, it's not too hard to copy and paste but one can imagine that having to paste 70 lines of the same code can be repetitive and ultimately "ugly". Ugly in the sense that you shouldn't have to repeat yourself.

There is a rule in programming that goes: **DON'T REPEAT YOURSELF (DRY)**

With subroutines, you can use the same piece of code without copy and pasting.

The structure of a subroutine is as follows:

```
YourSubroutineName:
    (Code)
RETURN
```

9.1 Example

```
MySubroutine:
    DEBUG "Hello there!", CR
    DEBUG "This is a subroutine", CR
RETURN
```

To call and execute a subroutine you use the **GOSUB** keyword:

```
Main:
    DEBUG "We're inside Main... Calling MySubroutine", CR
    GOSUB MySubroutine
    END

MySubroutine:
    DEBUG "Hello there!", CR
    DEBUG "This is a subroutine", CR
RETURN
```

9.2 Calculating the area of a square

We can use variables within subroutines like so:

```
sideLength  VAR     WORD
result      VAR     WORD

```

```
Main:
    sideLength = 50
    GOSUB calcSquareArea

    sideLength = 75
    GOSUB calcSquareArea

    sideLength = 100
    GOSUB calcSquareArea

    END

calcSquareArea:
    DEBUG DEC "Calculating Area with side legnth: ", sideLength, CR
    result = sideLength * sideLength ' area = l x w
    DEBUG DEC "Result: ", result, CR
RETURN
```

9.3 Conclusion

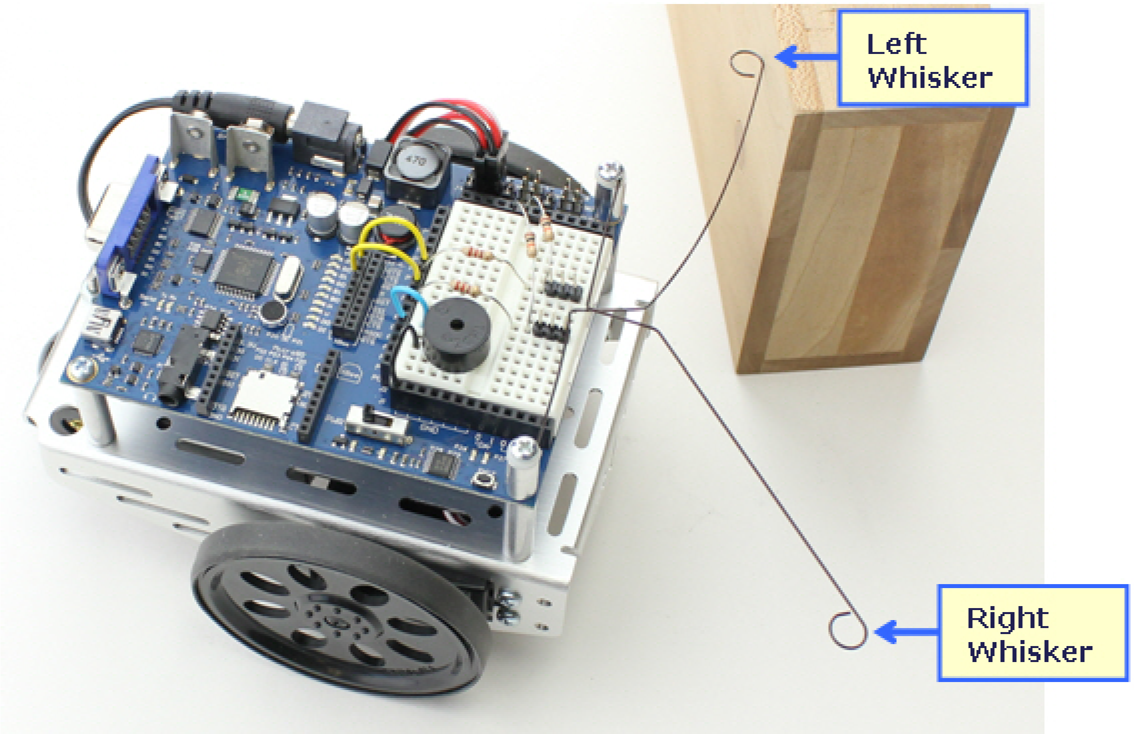
Subroutines are a good way to organize and cleanup your code. If you have parts where you need to constantly repeat yourself than put it into a subroutine! There is no limit to how many times you can call a subroutine.

9.4 Practice

Create a movement sequence again but this time using subroutines. You should notice you're code should look a lot cleaner this time around.

Whiskers

Whiskers are one of the components included in the robot kit. Whisker sensors allow the robot to detect obstacles when it bumps into them.



Whisker values are accessed via **IN5** and **IN7**

IN id	Whisker
IN5	Left Whisker
IN7	Right Whisker

Whisker State	Value
Unpressed	1
Pressed	0

10.1 Example: Outputting values when pressed

```
left_whisker    VAR     Bit
right_whisker   VAR     Bit

Main:
    DO
        left_whisker = IN5
        right_whisker = IN7

        IF (left_whisker = 0) AND (right_whisker = 0) THEN
            DEBUG "Both Whiskers were pressed!"
        ELSEIF (left_whisker = 0) THEN
            DEBUG "Left Whisker was pressed!"
        ELSEIF (right_whisker = 0) THEN
            DEBUG "Right Whisker was pressed!"
        ELSE
            DEBUG "No Whiskers are pressed..."
        ENDIF
    LOOP
```

10.2 Example: Utilizing the whiskers

```
left_whisker    VAR     Bit
right_whisker   VAR     Bit
pulse_count     VAR     Byte

Main:
    left_whisker = IN5
    right_whisker = IN7

    DO
        IF (left_whisker = 0) AND (right_whisker = 0) THEN
            ' Left and Right whiskers are pressed so we back up and make a U-turn by default
            ' A U-turn is just 2 left turns
            GOSUB Back_Up
            GOSUB Spin_Turn_Left
            GOSUB Spin_Turn_Left
        ELSEIF (left_whisker = 0) THEN
            GOSUB Back_Up
            GOSUB Spin_Turn_Right
        ELSEIF (right_whisker = 0) THEN
            GOSUB Back_Up
            GOSUB Spin_Turn_Left
        ELSE
            ' here the whiskers are NOT in contact with a wall so we pulse forward
            GOSUB Pulse_Forward
        ENDIF
    LOOP


Pulse_Forward:
    PULSOUT 13,850
    PULSOUT 12,650
RETURN
```

```

Spin_Turn_Left:
    FOR pulse_count = 0 TO 50
        PULSOUT 13, 650
        PULSOUT 12, 650
    NEXT
RETURN

Spin_Turn_Right:
    FOR pulse_count = 0 TO 50
        PULSOUT 13, 850
        PULSOUT 12, 850
    NEXT
RETURN

Back_Up:
    FOR pulse_count = 0 TO 50
        PULSOUT 13, 650
        PULSOUT 12, 850
    NEXT
RETURN
```

10.3 Conclusion

Whiskers are a good way to detect obstacles in front of the robot. However, whiskers aren't the best way to detect obstacles. There are some quirks of the whiskers bending in weird ways and which makes them less reliable. In the next section we will cover Infrared Sensors which offer much more in terms of depth perception and field of view (fov).

Infrared Sensors

Infrared sensors are the best sensors included in the robot kit. They offer more reliability since they don't bend or lose shape over time like the Whiskers. They work in the same way that whiskers work in terms of whether an obstacle is detected or not detected.

IN id	Sensor
IN9	Left Sensor
IN0	Right Sensor

Sensor State	Value
Undetected	1
Detected	0

11.1 Example: Outputting values when detected

```
left_ir_sensor      VAR     Bit
right_ir_sensor     VAR     Bit

Main:
    DO
        FREQOUT 8, 1, 38500
        left_ir_sensor = IN9

        FREQOUT 2, 1, 38500
        right_ir_sensor = IN0

        IF (left_ir_sensor = 0) AND (right_ir_sensor = 0) THEN
            DEBUG "Both sensors detected something!"
        ELSEIF (left_ir_sensor = 0) THEN
            DEBUG "Left IR sensor detected something!"
        ELSEIF (right_ir_sensor = 0) THEN
            DEBUG "Right IR sensor detected something!"
        ELSE
            DEBUG "No detection..."
        ENDIF
    LOOP
```

11.2 How IR detection works

I want to explain what this block of code does inside the DO-LOOP:

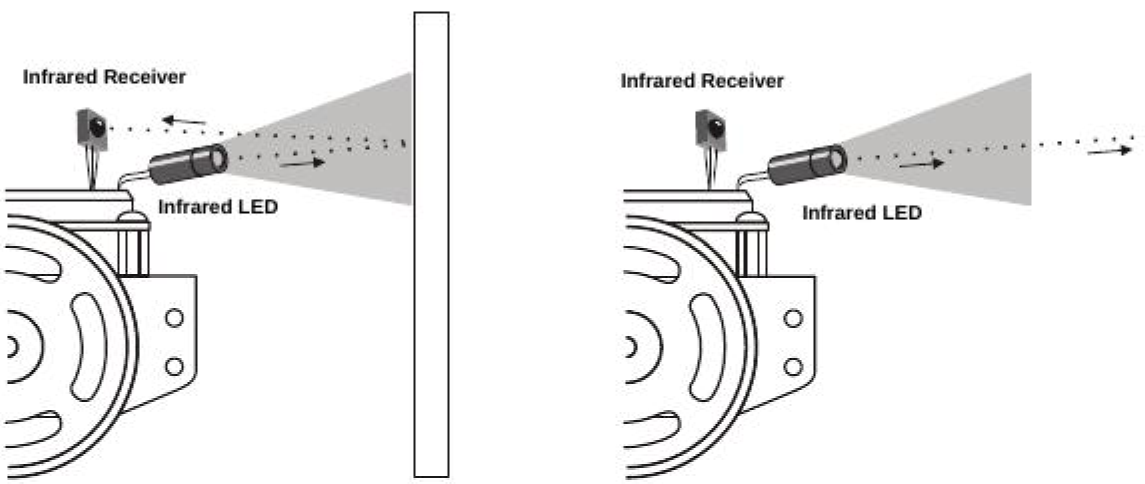
```
FREQOUT 8, 1, 38500
left_ir_sensor = IN9

FREQOUT 2, 1, 38500
right_ir_sensor = IN0
```

FREQOUT makes the IR LED shoot a 38.5 kHz IR signal outwards. Think of it like laser blasters from star wars.

Now, lets say that signal bounces off a wall like deflecting the laser in star wars.

The last thing to do is catch the signal in the IR Reciever. Which now makes so we can detect if there is an object ahead of us or not!

Here's a pretty good diagram of what I mean:



11.3 Example: Utilizing the IR Sensors

```
left_ir_sensor      VAR     Bit
right_ir_sensor     VAR     Bit
pulse_count         VAR     Byte
```

```

Main:
    DO
        FREQOUT 8, 1, 38500
        left_ir_sensor = IN9

        FREQOUT 2, 1, 38500
        right_ir_sensor = IN0

        IF (left_ir_sensor = 0) AND (right_ir_sensor = 0) THEN
            ' Left and Right IR sensors detected so we back up and make a U-turn by default
            ' A U-turn is just 2 left turns
            GOSUB Back_Up
            GOSUB Spin_Turn_Left
            GOSUB Spin_Turn_Left
        ELSEIF (left_ir_sensor = 0) THEN
            GOSUB Back_Up
            GOSUB Spin_Turn_Right
        ELSEIF (right_ir_sensor = 0) THEN
            GOSUB Back_Up
            GOSUB Spin_Turn_Left
        ELSE
            ' here the IR Sensors DONT detect anything so we pulse forward
            GOSUB Pulse_Forward
        ENDIF
    LOOP


Pulse_Forward:
    PULSOUT 13,850
    PULSOUT 12,650
RETURN

Spin_Turn_Left:
    FOR pulse_count = 0 TO 50
        PULSOUT 13, 650
        PULSOUT 12, 650
    NEXT
RETURN

Spin_Turn_Right:
    FOR pulse_count = 0 TO 50
        PULSOUT 13, 850
        PULSOUT 12, 850
    NEXT
RETURN

Back_Up:
    FOR pulse_count = 0 TO 50
        PULSOUT 13, 650
        PULSOUT 12, 850
    NEXT
RETURN
```

11.3.1 Important notes about Example: Utilizing the IR Sensors

The way the subroutines are coded is that they have set amounts for how much the robot will turn or backup. This isn't the most optimized way to navigate through a maze. You run the risk of either overshooting your turn or not turning enough. These risks should be very concerning to you even if they aren't!

11.4 Example: Optimizing the use of IR Sensors

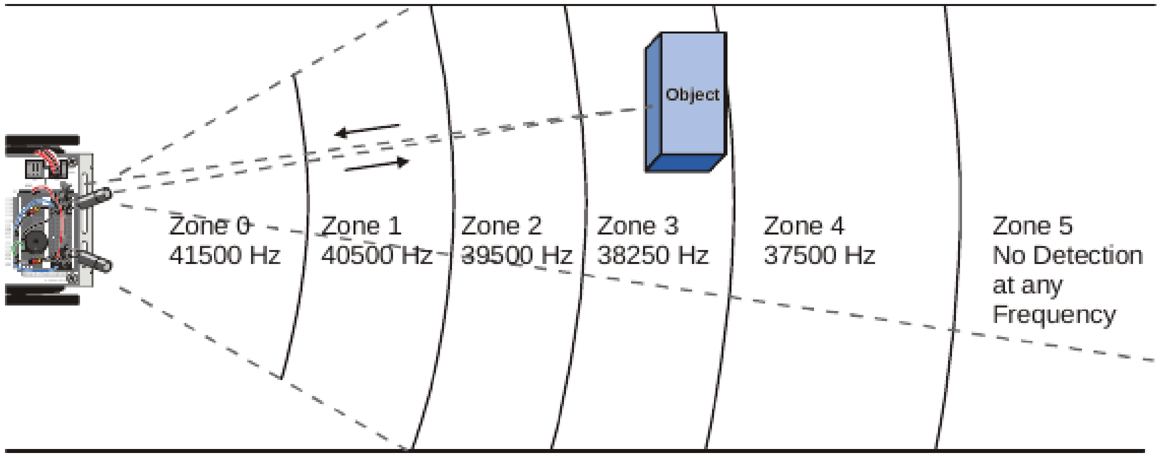
```
left_ir_sensor      VAR     Bit
right_ir_sensor     VAR     Bit
pulse_left          VAR     Word
pulse_right         VAR     Word

Main:
    DO
        FREQOUT 8, 1, 38500
        left_ir_sensor = IN9

        FREQOUT 2, 1, 38500
        right_ir_sensor = IN0

        IF (left_ir_sensor = 0) AND (right_ir_sensor = 0) THEN
            ' Both sensors detect something so we back up
            pulse_left = 650
            pulse_right = 850
        ELSEIF (left_ir_sensor = 0) THEN
            ' We pulse spin-turn the wheels to the right
            pulse_left = 850
            pulse_right = 850
        ELSEIF (right_ir_sensor = 0) THEN
            ' We pulse spin-turn the wheels to the left
            pulse_left = 650
            pulse_right = 650
        ELSE
            ' We pulse forward
            pulse_left = 850
            pulse_right = 650
        ENDIF

        ' Apply the pulse to the wheels
        PULSOUT 13, pulse_left
        PULSOUT 12, pulse_right
    LOOP
```

11.4.1 Notes about Example: Optimizing the use of IR Sensors

This is a much more accurate way to traverse a maze. Since changes to the direction the robot is moving is now done in single pulses. We get a much more reliable way to move throughout the maze. Now we don't have to worry about turning too much or too little!

11.5 Conclusion

The IR sensors are reliable and are the ones I encourage you to use. One thing that I'd like to take a moment to address is that you can change the signal frequency at which the IR transmitter sends. Increasing or decreasing has effects on the distance at which an object can be detected.



You might be wondering why the value of zone 4 is 37.5 kHz and not 38.5 kHz. The reason they are not the values that you would expect based on the % sensitivity graph is because the **FREQOUT** command transmits a slightly more powerful signal at 37.5 kHz than it does at 38.5 kHz. The frequencies listed in Figure 8-2 are frequencies you will program the BASIC Stamp to use to determine the distance of an object.

For example:

```
FREQOUT 8, 1, 40500
left_ir_sensor = IN9
```

CHAPTER 12

Competition Files

Here you can find all the files used in the competion from previous years. Click to download the files!

12.1 2016

1. `Cha-Cha-Slide.bs2`
2. `Dance.bs2`

CHAPTER 13

Contact

If you have any questions then feel free to send me an email and I'll try to get back to you as soon as possible:

```
rogelio_negrete@live.com
```

CHAPTER 14

Indices and tables

- genindex
- modindex
- search