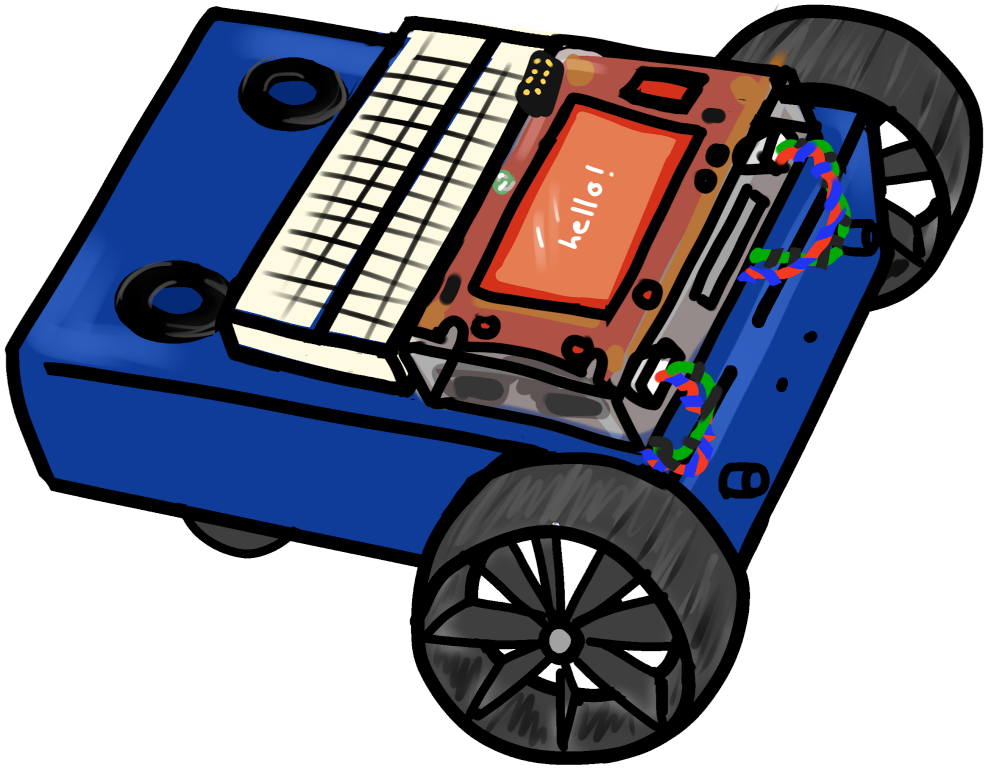
S.T.E.M.BOT 2



User's Manual

Rev. 1.2.1

Table of Contents

1.0.0.0 Introduction

MicroPython is an implementation of the Python programming language optimized for embedded systems. MicroPython is written in C and includes a Python compiler and parser to allow users to program in Python on low-level hardware.

The STEMBoT2 (SB2) uses version 1.10 of MicroPython loaded onto an STM32 32-bit microcontroller unit (MCU), and is intended for education and embedded systems development. Its features include an integrated LCD, speaker, several programmable push buttons, and headers for motors, servos, and serial communication. The three ports for serial communication use the UEXT (universal extension) layout and were included to be used with interchangeable peripherals.

1.1.0.0 The Purpose of the STEMBoT 2

The STEMBoT 2 is the ultimate platform for learning how to program using the Python programming language. With the proliferation of technology in our everyday lives, it's as important as ever to develop an understanding of programming and computer science topics. Python is one of the most popular programming languages in fields such as data science, financial services, and artificial intelligence, but it's also one of the easiest and most forgiving languages for new programmers to learn. The STEMBoT 2 takes this ease-of-use one step further by providing an engaging and interactive programming experience.

1.2.0.0 How to use this manual

This manual should be read start to finish by those new to programming Python. The purpose of this manual is not to teach Python, but to show how the STEMBoT 2 board works. For those already familiar with programming in Python, this manual contains high-level function descriptions at the end.

2.0.0.0 Hardware Overview

The STEMBoT 2 package comes with all of the parts necessary to construct the robot as well as a remote control and starter set of UEXT modules. The remote control and UEXT modules come pre-assembled.

The package contains the following items:

- One STEMBoT 2 Board
- One Aluminum Chassis
- One Laser-cut Acrylic Enclosure
- Two 2-Phase Stepper Motors
- Two Rubberized Wheels
- Two Omnidirectional Wheels
- One Remote Control
- Two Plug and Play Modules

The standard configuration for the SB2 is the way the robot is assembled and delivered to users. The PCB is inside the acrylic enclosure, which is mounted to the front of the aluminum chassis. The stepper motors are mounted underneath the enclosure, and plugged into the motor headers closest to each motor. The battery is affixed to the chassis via velcro tape and a plastic cable hook attached with a screw and nylock nut. The omnidirectional wheels are attached to the rear of the chassis, facing inwards, and at the bottom of the screw slot.

2.1.0.0 STEMBoT 2 Board

The heart of the STEMBoT 2 system is the main printed circuit board (PCB). This board contains 5 programmable switches, a full-color LCD, a tri-color LED, two motor headers, three UEXT headers for plug and plug functionality, and a piezoelectric buzzer. There's also a barrel jack for wall charging, a power indicating LED, a reset button, and a USB-B connector for programming.

2.2.0.0 Battery and Charger

The SB2 comes with a 6-cell, 7.2V, 3800mAh Nickel-Metal Hydride (NiMH) battery , as well as a 12V AC/DC adapter. The battery is connected to the board through an XT-60 adapter.

2.2.1.0 Charging the SB2 Battery

Without the charger connector, the SB2 will notify the user of a low battery condition if the green LED adjacent to the charging connector starts to blink. To charge the SB2 battery, make sure both the battery and charger are plugged in. Charging will begin automatically,

whether the board is turned on or not. One full charge cycle takes a maximum of 4 hours, and a completed cycle is indicated by a solid green light on the LED.

2.3.0.0 Remote Controller

The STEMBoT 2 comes with a remote controller for wirelessly operating the robot. The controller is a standard XBox controller design with custom internal hardware. It uses two AA batteries, operates at 915MHz, and includes an automatic 5-minute shutdown timer. The center green button is used for turning on the remote as well as joystick calibration. The small button at the top next to the left button (LB) is the pairing button, and is used to pair the remote with a STEMBoT. The remaining buttons are user-programmable, and more information can be found in section 4.3.0.0.

2.3.1.0 Pairing the Remote Controller

To pair the remote, navigate to "Tools" on your STEMBoT 2 and then click the B button to enter into the pairing mode. Next press the sync button on the remote, next to the left button (LB). After it says "Successfully paired" you can then drive your bot by selecting the remote control application.

2.3.2.0 Calibrating the Remote Controller

After successfully pairing, if either of your STEMBoT's motors are moving on their own, you can recalibrate your remote by holding the middle green button. Once the red LED flashing sequence begins, move your joysticks in circular patterns until the LED flashing sequence finishes.

4.0.0.0 Programming - Introduction

Programming of the STEMBoT 2 is done through the USB port of the board and the provided cable. With the USB plugged in, the user can view and edit Python files on the board, and program the board through a serial communication program.

4.1.0.0 How to Program the SB2

After starting the SB2, the main menu will appear on the LCD. In this menu, there is an option to Exit to Python REPL. With the USB cable plugged in, this option will automatically mount a drive to your computer called PYBFLASH. The /apps folder in this drive contains the example apps found on the board's main menu, and can be used to store user generated programs. Exiting to Python REPL will also allow the user to access the board's functions through a serial communication program.

Caution: Editing the default boot.py file can make the STEMBoT 2 inoperable.

Warning: The PYBFLASH drive on your computer must be unmounted before the USB cable is removed. Failure to do so could corrupt data in transit, making the STEMBoT 2 inoperable.

4.1.1.0 Programming via Serial Communication

After selecting the Exit to Python REPL option in the main menu, the SB2 can be accessed through a serial communication program. This allows programming in Python on a line-by-line basis. Connect to the SB2 using your serial communication program of choice.

Note: A popular program for this purpose is PuTTY, a free and open-source terminal emulator. This program will be used throughout this document, and can be downloaded at https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html

Type the following three lines, pressing enter after each one. If you are successfully connected, this will turn on the blue LED just above the SB2 display.

```
import pyb
LED=pyb.LED(3)
LED.on()
```

Note: For information on selecting the correct port and properly configuring PuTTY for use with the SB2, see User Guide: Setting Up Serial Communication for the STEMBoT 2, posted on the STEMBoT Inc website. Click here for the Putty Setup Guide.

4.1.2.0 Programming via Mass Storage

Python programs can be added to the SB2 by dragging and dropping files into the /apps folder located on the PYBFLASH drive. Restarting the STEMBot 2 will allow the user to start using their new programs right away, assuming there are no compilation or runtime errors. On your computer, create a new file using the text editor of your choice. In this file, type the following code:

```
import pyb                  #imports LED functions like with the above example
import uasyncio             #necessary for all programs placed in mass storage

async def main():           #necessary header for all programs in mass storage
       pyb.LED(1).off()     #turn off red LED
       pyb.LED(2).off()     #turn off green LED
       pyb.LED(3).on()      #turn on blue LED
```

Program 1: Turning on the Blue LED

Save the file as "new_led.py" to the /apps folder of the PYBFLASH drive. Restart the SB2 by pressing the blue reset button. On the main menu, the new program titled "new led" should appear. Running this program will turn on the blue LED just above the SB2 display.

Note: Due to the nature of the boot.py file, main function headers should start with the keyword `async`, for example, `async def main():`

Caution: When changing data on the SB2 via this method, the red LED will turn on. The SB2 should remain plugged into the computer until this red LED turns off. Unplugging the SB2 before the LED turns off could interrupt the data transfer and result in unintended operation.

4.2.0.0 Programmable Hardware Components

The STEMBoT 2 board itself comes with several programmable components, including switches, a multicolor LED, and a buzzer. These components are common among electronics and serve a variety of purposes. For example on the SB2, the switches are used to navigate the main menu and the LED is used as a status indicator, telling the user when programs are being uploaded or when an error has occurred. What follows is a description of these components and examples on how to access them through Python programming.

There are three modules used for accessing low-level functions of the SB2: `pyb`, `switch` and `buzzer`. The `pyb` module is used for the multicolor LED, as demonstrated in previous examples. The `switch` module can be used for accessing the buttons and other

general-purpose input/output pins (GPIOs). The `buzzer` module is used for accessing the on board buzzer to produce songs and other sounds.

> **Note:** The `pyb` module was developed primarily for a different board which also uses MicroPython, while `sb` contains similar functions generalized for any board. When possible, `sb` should be used over `pyb`.

4.2.1.0 Multicolor LED

The LED is accessible via the `LED()` function of the `pyb` module, and it is programmable for 3 different colors.

```
# This program will turn off the main LED, then cycle through
# its red, green and blue components, blinking each of them
# twice in an infinite loop.

import pyb          #grants access to the LED class
import uasyncio     #grants access to asynchronous delays (very useful!)

async def main():   #standard header for programs in the /apps folder
       x=1
       while x<4:    #turn off LEDs 1 through 3 (all of them)
              pyb.LED(x).off      #turn off the x LED
              x+=1                #increment x

       while(True): #infinite loop
              x = 1
              while x<4:    #cycle through RGB
                     led=pyb.LED(x)
                     led.on()                           #turn on LED
                     await uasyncio.sleep_ms(1000)      #one second delay
                     led.off()                          #turn off LED
                     await uasyncio.sleep_ms(1000)      #one second delay
                     led.on()
                     await uasyncio.sleep_ms(1000)
                     led.off()
                     await uasyncio.sleep_ms(1000)
                     x+=1
```

Program 2: Blinking the LEDs with Delays

> **Note:** The LED is also used for several status indications: it will blink red and green if an error has occurred and will stay solid red while data is being transferred to the SB2.

The three available colors, red, green, and blue, are numbered 1, 2, and 3, respectively, when using the `LED()` function.

4.2.2.0 Switches

As detailed in the Hardware Overview (section 2.1.0.0 of this guide) the SB2 has five programmable switches. These buttons can be used by importing the `machine` Python module. The switches by default are configured as inputs with their pull-up resistors enabled, as the following code demonstrates.

```
pinS8=machine.Pin('E5',machine.Pin.IN,machine.Pin.PULL_UP)
pinS7=machine.Pin('B6',machine.Pin.IN,machine.Pin.PULL_UP)
pinS6=machine.Pin('B5',machine.Pin.IN,machine.Pin.PULL_UP)
pinS5=machine.Pin('B2',machine.Pin.IN,machine.Pin.PULL_UP)
pinS4=machine.Pin('F11',machine.Pin.IN,machine.Pin.PULL_UP)
```

The pins above are named for their label on the PCB. Their values can be read with the `value()` method of the `Switch()` class in the `switch` module.

Caution: The `Pin()` function of the `machine` module can also be used to access other pins on the SB2's main microcontroller. Changing the function or value of certain pins may render critical functions of the SB2 inoperable.

As an example of how the switches work, open a serial connection to the SB2. Type in the following lines, pressing enter after each one.

```
from switch import Switch
button=Switch("up")
button.value()
```

Program 3: Getting the State of the Up Button

After entering the last line, False should appear on the serial terminal. This indicates that the button is not being pressed. Now, hold down the upper left button and send the `button.value()` line again. The 0 that appears on the serial terminal indicates the button is being pressed. Sampling this value is how the buttons are used to scroll through the main menu, but they can also be used for custom programs.

Note: In the serial terminal, the last line of code can be quickly accessed by pressing the up button on your keyboard.

4.2.3.0 Buzzer

Use of the piezoelectric buzzer requires the `buzzer` Python module. The buzzer is used for creating sounds, from simple tones to more complex musical pieces

4.2.3.1 Playing Songs from the Board

The SB2 comes with several songs pre loaded. These songs can be accessed using the `play_song()` function of the `buzzer` module.

```
import buzzer
buzzer.play_song('NationalAnthem')
```

Program 4: Playing the National Anthem

A complete list of available songs can be found in the document User Guide: List of Available Buzzer Songs.

4.2.3.2 Creating Songs using RTTTL

```
RTTTL('C Major Scale:d=4,o=5,b=180:c,d,e,f,g,a,b,c6')       #C major scale
```

The first parameter is the title of the tune. The next set of parameters are d, o, and b, which represent the default note, the default octave, and the tempo, respectively. For d, 1 represents a whole note, 2 is a half note, 4 is a quarter note, etc. The o sets the default octave between 4 and 7, but individual notes can be programmed for different octaves if need be. The b is the tempo given in beats per minute (bpm).

The last set of parameters defines the tune itself, and uses a duration-pitch-octave structure for its strings. For example, the string `2f#6` would play the F# note in the sixth octave for two intervals (defined by the d parameter).

```
# This program uses RTTTL to play the national anthem.

import rtttl
import buzzer

async def main():

    anthem=RTTTL('Anthem:d=16,o=5,b=90:2g,e,4c,4e,4g,2c6,8e6,d6,4c6,4e,4f#,2g')
    buzzer.play(anthem)
```

Program 5: Playing the National Anthem

4.3.0.0 Liquid Crystal Display (LCD)

Liquid crystal displays (LCDs) are used virtually anywhere modern electronics are found. From TVs to computers to cell phones, they allow people to connect with technology in an intuitive and seamless way. The LCD on the SB2 is 320x240 pixels wide, and manipulating pixels is done by defining their location, and then the data to be written.

The origin of the LCD, that is, where x and y are equal to 0, is at the upper left hand corner. Functions and methods that use x and y for drawing or printing to the LCD are thus measured from this point. For example, drawing an arrow that points to the x,y coordinate 160,120 will point to the center of the LCD (half of the total width and height).

The SB2's LCD is accessible through the `graphics` and `color` modules. These modules allow the user to create colors, write text to the LCD, and draw shapes.

4.3.1.0 Colors

The LCD uses standard RGB values to determine which color should be displayed. These colors can be created by using the `RGB()` function of the `color` module. This function takes three parameters, numbers between 0-255, corresponding to red, green, and blue..

Open a serial communication with the SB2. Input the following lines of code to demonstrate the basic functions of the `lcd` module.

```
import graphics
import color
red=color.RGB(255,0,0)
graphics.paint(red)
```

Program 6: Coloring the LCD Red

This set of code will color the entire LCD with the color red. The `screen()` class is used to access the LCD's functions, so it should be instantiated before using those functions. The `paint()` function of the `screen()` class fills the entire LCD with a given color. This can be used to set a solid colored background or to erase the entire screen.

> **Note:** While the entire screen will be colored by the paint() function, using the mass storage method of programming will allow the SB2 status bar to override that effect on the very top of the screen. For this reason, it is recommended that programs using the LCD avoid using the upper **(rewrite)** pixels of the screen.

> **Note:** An online RGB color wheel can be found here: https://www.colorspire.com/rgb-color-wheel/

4.3.2.0 Text

Writing text to the LCD is done through the `print()` function of the `graphics()` module. The `print()` function takes four parameters: the string to be written, the x location, the y location, and the color. For ease of use, three additional functions are provided: printTop(), printMiddle(), and printBottom(). These functions accept only two arguments, the string and color, and print to the top of the LCD, the middle, and the bottom, respectively.

```
# This program is for demonstrating the operation of the LCD by
# printing "Hello, world!" in white text on a blue background.

import graphics     #imports the module used for LCD operations
import color        #imports the module for LCD colors
import uasyncio     #imports the module used for concurrent programming

bgColor=color.RGB(0,0,255)                      #blue
textColor=color.RGB(255,255,255)                #white

async def main():                               #main function header
       graphics.paint(bgColor)                  #fill screen with color
       graphics.print("Hello, World!",40,120,textColor)     #print to LCD
```

Program 7: Hello, world!

Text that has been written using the `print()` function can be erased using the `erase()` function. This function accepts four arguments as well: the number of characters to be erased, the x location, the y location, and the color. The x and y locations should be the same as those sent to the print() function. The color should be the background color on which the characters have been written.

For an example using the `print()` and `erase()` functions, set up a serial connection to the SB2, and type in the following lines:

```
import graphics
import color
red=color.RGB(255,0,0)
black=color.RGB(0,0,0)
graphics.paint(red)
graphics.print("Hello, world!",40,120,black)
```

Now, after making sure the text has shown up, send the following command to the SB2:

```
graphics.erase(13,40,120,red)
```

This line tells the SB2 how many characters there are in "Hello, world!" (be sure to include spaces), the placement of the text, and the background color. The `erase()` function works by drawing a filled rectangle (section 4.3.3.1) around the previously printed text. If a color other than the background color is sent, the LCD will simply print a box of that color over the text.

Text that is printed to the LCD will not automatically clear if new text is printed, so it's up to the programmer to "clean up" after printing text.

```
# The following program is a basic second counter. It prints to the LCD how
many
# seconds have passed.

import lcd          #import lcd functions
import gfx          #this module can be used to draw shapes
import uasyncio     #import timing functions

lcdWidth=320        #define the LCD dimensions
lcdHeight=240

LCD=lcd.screen()                        #instantiate a screen object
bgColor=LCD.colorRGB(0,0,255)           #blue
textColor=LCD.colorRGB(0xFF,0xFF,0xFF)        #white

#the GFX function allows shapes and other objects to be printed with ease
graphics = gfx.GFX(lcdWidth, lcdHeight, LCD.lcd.pixel)

async def main():               #standard header
    LCD.paint(bgColor)          #fill the screen
    x=1                         #set up variable to count
    while(True):                #infinite loop
       # Print the time to the LCD: the curly brackets allow numbers to be
       # printed as strings when using the format() method
        LCD.lcd.text("Time = {}".format(x),40,90,textColor)
       # Wait for 1 second to pass
        await uasyncio.sleep_ms(1000)
       # Increment the counter
        x+=1
       # "Erase" the previous number by filling in the same space with the
       # background color. The parameters are x, y, width, height, color.
       # Choosing 96 for the x value starts coloring in after the "Time = "
       # text. You'll notice starting with 10 seconds, only the first number is
       # erased.
        LCD.erase(5,96,90,bgColor)
```

Program 8: LCD Timer

4.3.3.0 Shapes

The SB2 includes functions for drawing various shapes, either filled or unfilled, to the LCD. The functions require defined coordinates as well as the desired color of the shape.

```
# This program demonstrates the SB2's ability to draw shapes by drawing a green
# square, red circle, and blue triangle to the middle of the screen

import graphics
import color
import uasyncio

black=color.RGB(0,0,0)
red=color.RGB(255,0,0)
green=color.RGB(0,255,0)
blue=color.RGB(0,0,255)

async def main():
	graphics.paint(black)
	graphics.fill_rectangle(100,60,120,120,green)		#green square
	graphics.fill_circle(160,120,60,red)			#red circle
	graphics.fill_triangle(160,60,100,180,220,180,blue)	#blue triangle
```

Program 9: Drawing Filled Shapes

4.3.3.1 Drawing Rectangles

The two functions available for drawing rectangles are `rectangle()` and `fill_rectangle()`. The first function draws the outline of the shape, while the second fills in the shape with a given color. Both of these functions take five parameters: the starting x and y coordinates, the desired width, the desired height, and the color.

4.3.3.2 Drawing Circles

The two functions available for drawing circles are `circle()` and `fill_circle()`. The first function draws the outline of the shape, while the second fills in the shape with a given color. Both of these functions take four parameters: the x and y coordinate for the center of the circle, the radius of the circle, and the color.

4.3.3.3 Drawing Triangles

The two functions available for drawing triangles are `triangle()` and `fill_triangle()`. The first function draws the outline of the shape, while the second fills in the shape with a given color. Both of these functions take seven parameters: three pairs of x and y coordinates which define the corners of the triangle, and the desired color.

4.4.0.0 Radio

The SB2 has a built-in radio for communicating with the remote control. An SB2 remote controller can be paired with one robot at a time, and will continuously send signals while on. The remote's signals can be captured and interpreted using the `remote` module. For more information on the remote controller, see section 2.3.0.0.

4.4.1.0 Initialization and Configuration

To access the remote control methods, the `Remote()` class of the `remote` module must be instantiated. The RemoteData() class should also be instantiated and assigned to a variable. The pair() method is used to pair the remote controller to the SB2.

```
import remote
rdata=remote.RemoteData()
r=remote.Remote()
r.pair()
```

In this example, rdata will hold the remote's signal, r is the Remote() object, and r.pair() is used to pair the remote to the SB2. The pairing button on the controller should be pressed as soon as possible after sending the pair command.

4.4.2.0 Reading Radio Signals

Once the remote control is paired with the SB2, it starts transmitting button presses and joystick information. These transmissions can be captured with the read() method of the Remote() class. This method captures the most recent transmission sent by the controller, which includes button and joystick states. The read() method accepts two parameters, the data buffer (which should be the RemoteData() class) and a timeout (measured in milliseconds).The default timeout is set to 2 seconds.

> **Note:** To make the most of the remote control, signals should be captured frequently and repeatedly. Loops (such as while loops) are a good way of accomplishing this. See the lib/RC.py file on the SB2 for an example.

4.4.2.1 Interpreting Joystick States

After a signal has been received and placed in the rdata variable, the joystick value can be read by accessing the relevant variables of the `RemoteData()` object. These variables are `ljoy_up_down`, `ljoy_left_right`, `rjoy_up_down`, and `rjoy_left_right`. The variables represent the joystick direction and each have a range of -127 to 128. For the left joystick, the positive values represent up and right while the negative values represent down and left. On the right

joystick, the positive values represent down and right, and the negative values represent up and left.

```
left_updown=rdata.ljoy_up_down
left_leftright=rdata.ljoy_left_right
right_updown=rdata.rjoy_up_down
right_leftright=rdata.rjoy_left_right
```

The remote will send joystick values if they are actuated in any direction. For instance, if the left joystick is held up and to the right, the `ljoy_up_down` and `ljoy_left_right` values will be around 128.

4.4.2.2 Interpreting Button States

Button values are returned in binary, that is, either 1 (pressed) or 0 (not pressed), and are stored in the button variable of the RemoteData() class. These states can be determined with a process called *masking*, whereby all bits but the relevant one are set to 0, and the remaining bit is determined to be either 1 or 0. This is accomplished with the & operator.

```
start = rdata.buttons & remote.STRT_BIT
```

In this example, start will be 0 if the start button was not pressed when capturing the signal, and will be nonzero otherwise.

4.5.0.0 Motors

The SB2 comes with two stepper motors for driving the robot. Stepper motors get their name from the fact that they move incrementally, or in steps. The standard motors for the SB2 take 200 steps to complete a full rotation.

4.5.1.0 Initialization and Configuration

Use of the two stepper motors for movement requires the `sb` module. The Motor() class contains a set of methods that turn the motors on or off, and set their speed and acceleration. There's also a distance() function that commands the motors to move a given number of steps at a given speed. To begin using the motors, turn them on by setting the sleep mode to false with sleep(False), as shown in the next program.

4.5.2.0 Speed and Acceleration

The `speed()` method is used for setting a constant, immediate speed, or for accelerating to a given speed in a given period of time. Passing no parameters with the `speed()` method will return the current speed. Passing the `speed()` method one parameter (between 50 and 500) will start the motor immediately at that speed. Passing two parameters will cause the motor to rev up to a given speed (the first parameter) in a given time (the second parameter, in

milliseconds). The motor can be stopped by sending 0 speed, and sending negative speeds will cause the motor to move in reverse.

```
import uasyncio as asyncio
import sb

async def main():

	#create the two motor objects
	leftMotor=sb.Motor(1)
	rightMotor=sb.Motor(2)

	#motor control logic is initially disabled. Let's enable it
	leftMotor.sleep(False)
	rightMotor.sleep(False)

	while(True):

		#left motor moves forward at 200 steps per second
		#and will take 1 second to transition to that speed
		leftMotor.speed(200,1000)
		#right motor moves backwards at 200 steps per second
		#and will take 1 second to transition to that speed
		rightMotor.speed(-200,1000)

		#do the above (spin) for 5 seconds
		await asyncio.sleep_ms(5000)

		#left motor moves backward at 200 steps per second
		#and will take 1 second to transition to that speed
		leftMotor.speed(-200,1000)
		#right motor moves forward at 200 steps per second
		#and will take 1 second to transition to that speed
		rightMotor.speed(200,1000)

		#do the above (spin) for 5 seconds
		await asyncio.sleep_ms(5000)
```

Program 11: Spinning Robot

4.5.3.0 Distance and Time

The `distance()` method accepts 4 parameters: the distance to move (in steps), the speed, the acceleration, and blocking mode. The speed and acceleration are the same as in the `speed()` method. If the blocking mode is set to True, the code will not continue until the motor has finished rotating. This can be useful if there are several lines of `distance()` methods placed one after the other.

4.6.0.0 UEXT (Plug-and-Play) Headers

On the STEMBoT 2, there are three headers for UEXT connections. UEXT is a standard means of connecting to modules via three types of serial interfaces, UART, SPI, and I2C. The header also contains pins for power (3.3V) and ground.

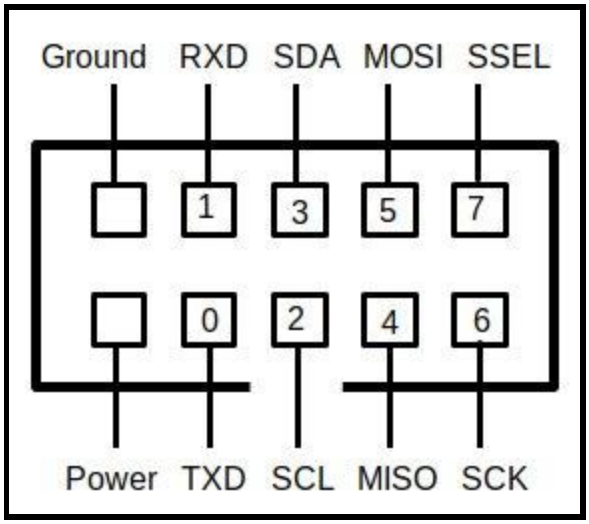


Figure 1: UEXT Header Pin Labels

4.6.1.0 General Purpose Use

The pins used for serial communication can also be used as GPIO (general purpose input/output) pins. These pins can be accessed by including the `pnp` module, which contains the `GPIO` class.

```
#the following statements will return a list of pins
#according to the parameter in the GPIO function
import uext
port=pnp.GPIO("J11")
```

The `port` object in the example above will give access to the GPIOs on header J11, which are set as outputs by default. A parameter of "top" can also be used to access this port. The other ports, J8 and J10 are also called "left" and "right". The individual pins can be accessed through the `GPIO` class like in the following example:

```
port.pin[3].on()     #turn on pin 3 (SDA)
port.pin[3].off()    #turn off pin 3
port.pin[3]()        #returns either 1 or 0 if the pin if high or low
```

The GPIO class of the uext module also contains two functions for turning all of the pins on or off. These functions are called allOn() and allOff(), respectively. Toggling of individual pins can be done with the togglePin() function, which takes the target pin as its only parameter.

```
port.allOn()        #activate all pins
port.allOff()       #deactivate all pins
port.togglePin(port.pin[3])      #activate if off, deactivate if on
```

The pins can be also be assigned individually by importing the `machine` module. In the case of J11, the header on the top of the STEMBoT 2, the pins have been defined below.

```
pinTX=machine.Pin('C12',machine.Pin.OUT)
pinRX=machine.Pin('D2',machine.Pin.OUT)
pinSDA=machine.Pin('B11',machine.Pin.OUT)
pinSCL=machine.Pin('B10',machine.Pin.OUT)
pinSSEL=machine.Pin('A5',machine.Pin.OUT)
pinMOSI=machine.Pin('B15',machine.Pin.OUT)
pinMISO=machine.Pin('B14',machine.Pin.OUT)
pinCLK=machine.Pin('B13',machine.Pin.OUT)
```

The MOSI, MISO, and CLK pins are common among all three UEXT connectors. That is, turning one on or off does the same for all three. The SDA and SCL pins are common between the J8 and J10 UEXT ports, and are on by default. Toggling the pins is done by using the `on()` or `off()` functions. The value() function can be used to determine if the pin is high (1) or low (0).

```
pinTX.on()
pinTX.off()
pinTx.value()       #this will return 0, since it was just turned off
```

Note: Toggling can be done at 35.7kHz (T/2=13.3us). [Initial test]

4.6.2.0 Serial Communication

4.6.2.3 Inter-Integrated Circuit (I2C)

I2C objects are created using the `machine` module. Initially, the user must select the bus and baud rate (or bit rate) of the I2C object. The SB2 has two busses for I2C communication: bus 1 corresponds to headers J8 and J10, located below the LCD, and bus 2 corresponds to header J11, just above the LCD.

```
import machine
#create I2C object on bus 1 with baud rate of 9600 bits per second
i2c=machine.I2C(1,freq=9600)
```

Note: The baud/bit rate is the rate at which data will be sent serially, measured in bits per second. A rate of 9600 is fairly common, but the SB2 is capable of transmitting at standard rates up to 115200 bits/second.

I2C based UEXT modules can either accept commands directly or consist of a set of registers which can be accessed individually. Writing via I2C can be done using the `writeto()` function by using an integer for the address and a bytes() type value for the data to be sent. A third boolean (True/False) parameter can be sent to include or exclude a stop condition (check the appropriate datasheet for more information). Reading can be done with the `readfrom` function, again using an integer for the address, but using a bytesarray() type buffer for the received data. The address for both of these functions should be 7-bits, as MicroPython will append the integer with either a binary 1 or 0 for reading or writing.

```
#initialize variables (these types must be used for I2C to work properly)
address=64                  #address for SHT21 temperature module
fetchTemp=bytes([0xF3])     #SHT21 temperature command
tempData=bytearray(3)       #3 byte buffer for SHT21 temperature data

#create I2C object (remember to import machine first)
i2c=machine.I2C(1,freq=9600)

i2c.writeto(address,fetchTemp,False)     #send command to address
tempData=i2c.readfrom(address,3)         #read three bytes, store in tempData
```

Program 12: Controlling an SHT21 PnP Module Manually

Writing and reading via I2C to a register-based device is done similarly. The two relevant functions, `writeto_mem` and `readfrom_mem_into` require an integer for the device and register addresses, and bytes() or bytearray() type objects for data sending and storage.

4.6.3.0 Plug and Play (PnP) Modules

The SB2 Plug and Play modules can be used by importing the pnp module in Python. This module contains classes for controlling the available modules as well as control over the UEXT ports as GPIOs.

4.6.3.1 LED Module

The LED module is controlled through the methods provided by the GPIO class. As well as the on() and off() methods described in section 4.6.1.0, the GPIO class also contains allOff() and allOn() methods which turn all of the pins off or on, respectively. This class also contains the togglePin() method, which turns a pin on if it was previously off, and vice versa.

```
port.allOff()                    # deactivate all pins
```

```
port.togglePin(port.pin[3])       # activates pin 3, since it was just
                                  # deactivated
```

4.6.3.2 Seven Segment Display Module

The Seven Segment Display Plug and Play module can be controlled through the `SevenSegmentDisplay()` class of the pnp module. This class takes one parameter, and that is the port into which the module is plugged in ("top", "right", or "left"). This class contains 3 primary methods: `clear()`, `toggleDP()`, and `displayNumber()`. The `displayNumber()` method accepts one parameter, a number between (and including) 0 and 9, and it activates the appropriate pins to display the passed number. The `clear()` method will turn off all of the LEDs, and the `toggleDP()` method will turn the decimal point either on or off, depending on its previous state.

```
import pnp
ssd=pnp.SevenSegmentDisplay("top")      # module is plugged into the top port
ssd.displayNumber(7)                    # display the number 7
ssd.clear()                             # clear the display
ssd.toggleDP()                          # turn on the decimal point (off by
                                        # default)
```

Program 13: Displaying the Number 7 on the SSD PnP Module

Note: Using the `displayNumber()` method automatically clears the previous number display, but does not affect the decimal point.

4.6.3.8 Temperature/Humidity Sensor

This module uses an SHT21 sensor for detecting temperature and relative humidity. The SB2 automatically converts data to either Celsius or Fahrenheit for temperature, and percentage (from 0% to 100%) for relative humidity. Users can access these measurements through the `SHT21()` class of the `pnp` module.

The `SHT21()` class accepts only one argument, either "top", "right", or "left", depending on the port the module is plugged into. The class contains three main methods: `getTempC()`, `getTempF()`, and `getRH()`, which return the temperature in Celsius, the temperature in Fahrenheit, and the relative humidity, respectively. None of these methods accept parameters.

Note: Be aware that all three of the methods included in the SHT21() class use a 100ms delay.

4.6.3.9 Motion Tracking Sensor (Accelerometer/Gyroscope)

The motion tracking sensor uses an MPU6050 chip which has an on-board accelerometer and gyroscope to read acceleration and angular velocity, respectively. Acceleration is measured in meters per second squared, and angular velocity is measured in degrees per second. The conversion to these units is performed automatically by the SB2 through use of the `MPU6050()` class of the `pnp` module. This class only accepts one argument, either "top", "right", or "left", depending on which port the module is plugged into.

The two primary methods available through this class are `getAcceleration()` and `getAngularVelocity()`. These take no parameters and return three values, the acceleration or angular velocity along the x, y, and z axes.

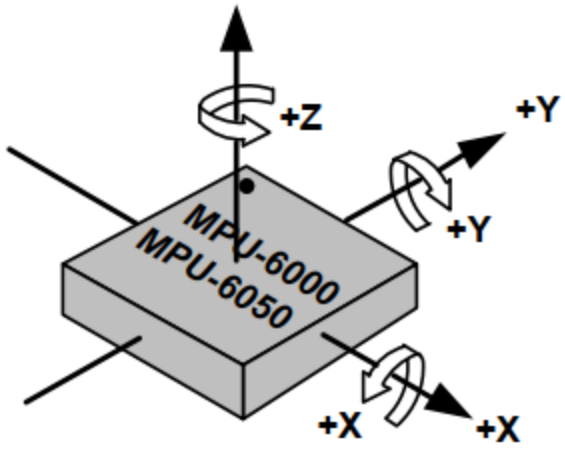


Figure 2: MPU-6050 Orientation, from InvenSense

Example (serial):

```
import pnp
mpu=pnp.MPU("top")  #make sure the module is plugged into the top port
ax,ay,az=mpu.getAcceleration()          #place values in ax, ay, and az
gx,gy,gz=mpu.getAngularVelocity()       #place values in gx, gy, and gz
```

Program 13: Getting Motion Data from MPU6050 PnP Module

Since this device, like all measuring devices, is imperfect, it will not always return 0 when getting measurements, even if it's perfectly still. One way to compensate for this is by taking an average of several or many readings in your code and subtracting those averages from your desired measurements. See the PnP_MPU6050 sample code for an example.

Note: When reading acceleration, there should always be one axis that reads around 9.8m/s^2. This is caused by acceleration due to Earth's gravity.

4.7.0.0 Servos

The SB2 comes with a 3x4 pin header to be used with servos. From left to right, the pins are the control pin, the power (5V) pin, and the ground (0V) pin. Servos, unlike motors, are used for precise control, typically between 0 and 180 degrees. This is accomplished by sending a pulse-width modulated (PWM) signal which the servo can interpret as a specific angle.

The servos connected to the SB2 can be accessed by importing the pyb module, and using the Servo() class. This class contains three methods: `calibration()`, `angle()`, and `speed()`.

4.7.1.0 Calibration Procedure

Servos require calibration to ensure that programs will operate as intended. This is due to both the wide array of servos available on the market, as well as variations between servos of the same type. On the SB2, calibration is done with the calibration() function. This function takes 3 parameters: the minimum pulse width, the maximum pulse width, and the center pulse width (for a 0 degree angle). These values are measured in microseconds and should be provided with the servo's datasheet. Regardless, calibration should be completed to achieve optimal results.

> Note: Although the SB2 comes pre-calibrated, it is highly recommended that this procedure is followed for any new servo. Once new calibration values are found, they should be recorded for repeated use.

1. Create a serial connection between your computer and the SB2.
2. Without connecting any servo arms, connect the servo to the top row of pins . This is servo 1 in the SB2 firmware.
3. Using a serial terminal, use the following commands to access the servo:
    a. import pyb
    b. S1=pyb.Servo(1)
4. S1 is the new servo object. Calibrate the object with the following command where the three values are given by the servo's datasheet:
    a. S1.calibration(minimumPulseWidth,maximumPulseWidth,centerPulseWidth)
5. This calibration procedure assumes the center pulse width is accurate. Center the servo by entering the following command:
    a. S1.angle(0)
6. Attach a servo arm so it is oriented parallel to one of the servo's dimensions. (For example, if the servo is longer than it is wide, attach the arm along the long part of the servo.)
7. Send the following command:
    a. S1.angle(-90)
8. If the servo arm has moved 90 degrees from its original (center) position, the first parameter of the calibration() function is correct. If not, either increase (if the arm moved

too far) or decrease (if the arm hasn't moved far enough) the value of the first parameter. Repeat steps 7 and 8 until the servo arm is 90 degrees from the center position.

9. Repeat step 5 to re-center the servo arm.
10. Send the following command:
    a. S1.angle(90)
11. If the servo arm has moved 90 degrees from its original (center) position, the second parameter of the calibrate() function is correct. If not, either increase (if the arm hasn't moved far enough) or decrease (if the arm moved too much) the value of the first parameter.
12. Repeat steps 10 and 11 until the servo arm has moved 90 degrees from the center position. Record the final values for use with the servo.

    Note: The servo used for testing this procedure (smraza S51) gave good results with S1.calibration(550,2475,1500)

4.7.2.0 Setting Angles

As demonstrated in the calibration procedures, setting angles can be done with the angle() function. This function will accept parameters between -90 and 90 which determines the servo angle, with 0 being the center position. Using the function without parameters returns the current angle of the servo.

6.0.0.0 Troubleshooting

Sometimes technology doesn't work as expected. This section covers common troubleshooting issues involving the hardware and software, as well as issues based on your operating system. For any other issues not found here, please contact support@stembots.com.

6.1.0.0 Power/Charging System

Checking the fuse: Take the top of the enclosure off of the SB2. Plug in the battery and no other power source, and turn on the SB2. With a DMM, check the voltage across capacitor C54. If it is 0V, the fuse likely needs to be replaced.

6.2.0.0 Motors
6.3.0.0 Troubleshooting by Operating Systems

6.3.1.0 Linux

Creating/editing .py files in Linux: In terminal, type "findmnt" to determine the SB2 filepath. As a root user, type "mount -o remount,rw /PATH" where path is the previously identified filepath. This will change the SB2 from a read-only filesystem to a read/write filesystem.

Finding serial port in Linux: dmesg | grep tty

6.3.2.0 MacOS

MacOS creates hidden files on external drives. Oftentimes this will fill up the SB2 memory. TO clean it up, look for any files prepended with a period and delete them.

6.4.0.0 Notes for REPL -- Special Key Combinations

CTRL-A - enter REPL mode

Enter raw REPL mode. Raw REPL is like REPL except for the following:
- there is no >>> prompt
- characters which are typed are not echoed back
- there is no auto indent (i.e. ... prompt)
This mode is extremely convenient for programs (as opposed to humans) to use.

CTRL-B - enter normal REPL mode

Gets you back to what's referred to as a the friendly REPL in the code. This is the REPL that you normally see (with the >>> prompt).

CTRL-C - interrupt a running program

Can be used to interrupt a running program. Try entering the following program at the REPL:
CODE: SELECT ALL

```
for i in range(1000000):
    print(i)
```

You see you probably don't want to wait for it to reach 1000000. So you can press Control-C:

```
19243
19244
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyboardInterrupt:
>>>
```

CTRL-D - soft reset

This basically reset the python interpreter without resetting the processor. It wipes any progam in memory and reruns boot.py and main.py and then returns you to the REPL. So for example, if when you first boot up the board and you use the dir() command:

```
>>> dir()
['__name__', 'pyb']
```

Now create some variables and then redo the dir() command:

```
>>> i = 1
>>> j = 23
>>> x = 'abc'
>>> dir()
['j', 'x', '__name__', 'pyb', 'i']
>>>
```

dir() shows that you have some variables in the local scope. Press Control-D and they all get wiped out:

```
PYB: sync filesystems
PYB: soft reboot
MicroPython v1.5-51-g6f70283-dirty on 2015-10-30; PYBv1.0 with STM32F405RG
Type "help()" for more information.
>>> dir()
['__name__', 'pyb']
>>>
```

CTRL-E - paste mode

This allows you to paste in functions from your computer without getting messed up by the auto-indent facility. So suppose I have the following function:

```
def foo():
    print('This is a test to show paste mode')
    print('Here is a second line')
foo()
```

At the regular REPL if you tried to paste that in you'd see something like this:

```
>>> def foo():
...         print('This is a test to show paste mode')
...             print('Here is a second line')
...             foo()
...
Traceback (most recent call last):
  File "<stdin>", line 3
IndentationError: unexpected indent
```

Whereas with paste mode you'd see:

```
paste mode; CTRL-C to cancel, CTRL-D to finish
=== def foo():
===     print('This is a test to show paste mode')
===     print('Here is a second line')
=== foo()
===
This is a test to show paste mode
Here is a second line
>>>
```

7.0.0.0 Module/Class/Function Quick Reference

The purpose of this section is to provide a high level overview of the SB2 API's functions for quick reference. Modules need to be included in the code using the **import** statement. Classes can be accessed by appending the module with the class name using a period. Methods and functions can be accessed the same way. The follow program demonstrates the use of modules, classes, and functions to turn on the red (1) and green(2) LEDs:

```
import pyb
redLED=pyb.LED(1)
redLED.on()
pyb.LED(2).on()
```

Be mindful of both spelling and capitalization; the names of the class, methods, functions, and objects must be used exactly as seen below. Python ignores whitespaces within lines of code, so for example if in the program above you replaced the second line with "`redLED = pyb.LED( 1 )`" it would function exactly the same. The names of objects however must be kept in tact, that is, no spaces. The line "`red LED=py b.LE D(1)`" would not work.

(Module) - **pyb**

(Class) - **LED(i)** - Used to access the tri-color LED. The parameter i can be either 1, 2, or 3 to access the red, green, and blue LEDs, respectively.

(Method) - **on()** - Turns the LED on

(Method) - **off()** - Turns the LED off

(Module) - **switch**

(Class) - **Switch(s)** - Used to access the five programmable buttons on the SB2. Allowed values for parameter s are 'a', 'b', 'c', 'up', and 'down', to access the respective buttons.

(Method) - **value()** - Returns True if the associated button is pressed, and False otherwise

(Module) - **color**

(Function) - **RGB(r, g, b)** - Returns a color object that can be used with the graphics module. The parameters are integer values of 0-255 which represent the amount of red, green, and blue in the returned color.

(Object) - **RED, GREEN, BLUE, WHITE, BLACK** - Predefined colors

(Module) - **graphics**

(Function) - **paint(color)** - Fills the LCD with a given color. Colors can be created using the RGB() function of the color class
(Function) - **print(string, x, y, color)** - Prints the given string to the LCD starting at the x,y coordinate. The new text will be the given color.
(Function) - **printTop(string, color)** - Prints the given string at the top of the LCD with the given color.
(Function) - **printMiddle(string, color)** - Prints the given string to the middle of the LCD with the given color.
(Function) - **printBottom(string, color)** - Prints the given string to the bottom of the LCD with the given color.
(Function) - **erase(i, x, y, o)** - Fills the space starting x pixels from the left of the screen and y pixels from the top of the screen the given color o. The integer i is the number of characters that will be replaced with the color (used to "erase" text)
(Function) - **line(x1, y1, x2, y2, color)** - Draws a line with the given color from x1,y1 to x2,y2. THe line has a width of 1 pixel.
(Function) - **rectangle(x, y, width, height, color)** - Draws a rectangle starting at coordinate x,y with a width and height of w and h. The edges of the rectangle are 1 pixel wide and the color o.
(Function) - **fill_rectangle(x, y, width, height, color)** - Draws a rectangle starting at coordinate x, y with a given width and height. The shape is filled in with the given color.
(Function) - **circle(x, y, r, color)** - Draws a circle centered around coordinate x, y with a given radius r. The circle will be drawn with the given color and have a thickness of 1 pixel.
(Function) - **fill_circle(x, y, r, color)** - Draws a circle centered around coordinate x, y with a radius r. THe circle will be filled in with the given color.
(Function) - **triangle(x1, y1, x2, y2, x3, y3, color)** - Draws a triangle with the three points at the given coordinates. The triangle is drawn with the given color and has an edge thickness of 1 pixel.
(Function) - **fill_triangle(x1, y1, x2, y2, x3, y3, color)** - Draws a triangle with the three points at the given coordinates. The triangle is filled in with the given color.
(Function) - **arrow(angle, x, y, length, color)** - Draws an arrow with a given color pointing to a given x,y coordinate with a given length. The function accepts angles in 45 degree increments (i.e., 0, 45, 90, 135, etc.).
(Object) - **lcdheight** - The height of the LCD in pixels.
(Object) - **lcdwidth** - The width of the LCD in pixels.

(Module) - **utime**
(Function) - **sleep(s)** - Delay of s seconds
(Function) - **sleep_ms(s)** - Delay of s milliseconds
(Function) - **sleep_us(s)** - Delay of s microseconds

(Module) - **sb**

(Class) - **Motor(x)** - Used for interacting with the motors. Accepts either 1 or 2 as a parameter.

(Method) - **sleep(bool)** - Forces the associated motor to enter or exit sleep mode. True sets the motor to sleep and False wakes it up. Leaving the parameter blank returns either True or False depending on the sleep state.

(Method) - **brake_mode(bool)** - Sets the motor brake mode. If true, the motor will "brake" at the end of movement, otherwise the motor will be allowed to roll. Leaving the parenthesis blank will return either True or False depending on the state of the brake mode.

(Method) - **speed(x)** - Sets the stepper motor to move continuously. Values above 500 may cause the motor to stall. Leaving the parenthesis blank will return the current speed.

(Method) - **distance(steps, speed, acceleration, blocking)** - Causes the motor shaft to rotate a given number of steps at a certain speed. Speeds above 500 may cause the motor to stall. Acceleration is the time it takes to wind up to the given speed (a value of 10 is recommended for most purposes). Setting blocking to True prevents the next line of code from being executed until the motor finishes moving.

(Method) - **stop()** - Causes the motor to stop moving.

(Function) - **setWheelDiameter(x)** - Sets the wheel diameter in millimeters. This value is used for distance conversions. Default is 85.

(Function) - **getWheelDiameter()** - Returns the current wheel diameter in millimeters.

(Function) - **inches_to_steps(x)** - Converts the given number of inches to steps based on the wheel's diameter. Used for the distance() method.

(Function) - **mm_to_steps(x)** - Converts the given number of millimeters to steps based on the wheel's diameter. Used for the distance() method.

(Function - **decirevs_to_steps(x)** - Converts the given number of decirevs to steps. Used for the distance() method.

(Function) - **steps_to_inches(x)** - Converts a given number of steps to inches based on the wheel's diameter.

(Function) - **steps_to_mm(x)** - Converts the given number of steps to millimeters based on the wheel's diameter.

(Function) - **steps_to_decirevs(x)** - Converts the given number of steps to decirevs.

(Class) - **Servo(x)** - Used for programming the servo ports.

(Method) - **pulse_width(x)** - Sets the pulse width of the servo signal to the given number of microseconds

(Module) - **remote**

(Class) - **Remote()** - Used to access remote controller methods

(Method) - **pair()** - Pairs the SB2 with a remote controler

(Method) - **read(RemoteData, timeout)** - Reads the current radio data and puts the information into a RemoteData object. If data cannot be read before the timeout, nothing happens.

(Class) - **RemoteData()** - Used to store remote controller data

(Object) - **buttons** - A length 15 bytearray object that stores data on which remote controller buttons are being pressed. Used with bitwise operators and the "_BIT" objects to determine the state of single buttons.

(Object) - **ljoy_up_down** - Contains a numerical value representing the up/down position of the left joystick.

(Object) - **ljoy_left_right** - Contains a numerical value representing the left/right position of the left joystick.

(Object) - **rjoy_up_down** - Contains a numerical value representing the up/down position of the right joystick.

(Object) - **rjoy_left_right** - Contains a numerical value representing the left/right position of the right joystick.

(Object) - **SLCT_BIT**, **STRT_BIT** - Numerical value representing the select and start buttons. To be compared with the RemoteData().buttons object.

(Object) - **L1_BIT**, **L2_BIT**, **L3_BIT**, **R1_BIT**, **R2_BIT**, **R3_BIT** - Numerical value representing the left and right buttons (L3 and R3 are the joystick buttons). Used in conjunction with the RemoteData().buttons object.

(Object) - **DPAD_UP_BIT**, **DPAD_RT_BIT**, **DPAD_DN_BIT**, **DPAD_LT_BIT** - Numerical value representing the buttons on the directional pad. Used in conjunction with the RemoteData().buttons object.

(Object) - **A_BUTTON**, **B_BUTTON**, **X_BUTTON**, **Y_BUTTON** - Numerical values representing the colored buttons. Used in conjunction with the RemoteData().buttons object.

(Module) - **pnp**

(Class) - **GPIO(position)** - Used to access the UEXT port as pins (also for the LED plug and play modules). Accepts one parameter, either "top", "left", or "right".

(Method) - **togglePin(pin)** - Toggles the given pin between on and off states.

(Method) - **allOn()** - Turns on all of the pins.

(Method) - **allOff()** - Turns off all of the pins.

(Class) - **SevenSegmentDisplay(position)** - Used to access the methods of the seven segment display (SSD) plug and play module. Accepts one parameter, either "top", "left", or "right".

(Method) - **clear()** - Turns off all of the LEDs on the SSD.

(Method) - **displayNumber(x)** - Displays a given number. Only integers from 0 to 9 are allowed.

(Method) - **toggleDP()** - Toggles the decimal point.

(Class) - **SHT21(position)** - Used to access the methods of the SHT21 temperature/humidity sensor plug and play module. Accepts one parameter, either "top", "left", or "right".

(Method) - **getTempC()** - Returns the current temperature in degrees Celsius.

(Method) - **getTempF()** - Returns the current temperature in degrees Fahrenheit.

(Method) - **getRH()** - Returns the relative humidity in percent.

(Class) - **MPU6050(position)** - Used to access the methods of the MPU6050 accelerometer/gyroscope plug and play module. Accepts one parameter, either "top", "left", or "right".

(Method) - **getAcceleration()** - Returns x, y, and z axis acceleration data in meters per second squared.

(Method) - **getAngularVelocity()** - Returns angular velocity about the x, y, and z axes. Data is returned in units of degrees per second.

(Class) - **OPT3001(position)** - Used to access the method of the OPT3001 plug and play module. Accepts one parameter, either "top", "left", or "right".

(Method) - **getLux()** - Returns the ambient brightness in Lux.