CodeBug Tether Documentation

Release 0.8.5

Thomas Preston

Mar 29, 2017

Contents

The codebug_tether Python module provides classes for interacting with CodeBug over USB Serial.

Links:

- The CodeBug Website
- CodeBug Activity: "Tethering CodeBug with Python"

Contents:

CHAPTER 1

Installation

Setting up CodeBug

In order to use CodeBug with codebug_tether you need to program CodeBug with `codebug_tether.cbg` ().

To do this, hold down button A and plug in CodeBug via USB — it should appear as a USB drive — then copy the `codebug_tether.cbg` file onto it. CodeBug is now ready to be used via serial USB. Press button B to exit programming mode.

Note: When CodeBug is connected to a computer via USB is should now appear as a serial device. To reprogram CodeBug: hold down button A and (re)plug it into a USB port.

Install codebug_tether on Windows

Note: These instructions are based on The Hitchhikers Guide to Python: Installing Python on Windows

Install Python

Download and install the latest version of Python 3 from here. Make sure you tick the *Add Python 3 to environment variables* checkbox.

Install codebug_tether

To install codebug_tether, open up a command prompt and type:

```
pip install codebug_tether
```

Restart Windows and then open IDLE. Plug in CodeBug and type:

```
>>> import codebug_tether
>>> codebug = codebug_tether.CodeBug()
>>> codebug.set_pixel(2, 2, 1)
```

The middle pixel on your CodeBug should light up.

See *Examples* for more ways to use codebug_tether.

Install codebug_tether on OSX

Note: These instructions are based on The Hitchhikers Guide to Python: Installing Python on Mac OS X

Install Python

Download and install Xcode (if you haven't already) and then enable the command line tools by running (in a terminal):

```
xcode-select --install
```

Now install Homebrew (a package manager for OSX):

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/
↪master/install)"
```

The script will explain what changes it will make and prompt you before the installation begins. Once you've installed Homebrew, insert the Homebrew directory at the top of your **PATH** environment variable. You can do this by adding the following line at the bottom of your ~/.profile file:

```
export PATH=/usr/local/bin:/usr/local/sbin:$PATH
```

Now, we can install Python 3:

```
brew install python3
```

This will take a minute or two.

Install codebug_tether

To install codebug_tether, open up a terminal and type:

```
pip install codebug_tether
```

To test it has worked, plug in CodeBug and open a Python shell by typing:

```
python
```

Your command prompt should have changed to:

```
>>> _
```

Now type:

```
>>> import codebug_tether
>>> codebug = codebug_tether.CodeBug()
>>> codebug.set_pixel(2, 2, 1)
```

The middle pixel on your CodeBug should light up.

See *Examples* for more ways to use codebug_tether.

Install codebug_tether on Linux

Install Python

Python 3 and pip should already be installed but for good measure run:

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install python3
```

If *pip* isn't installed you can securely download it from here get-pip.py.

Then run the following:

```
sudo python3 get-pip.py
```

Install codebug_tether

To install codebug_tether, open up a terminal and type:

```
sudo pip3 install codebug_tether
```

To test it has worked, plug in CodeBug and open a Python shell by typing:

```
python3
```

Your command prompt should have changed to:

```
>>> _
```

Now type:

```
>>> import codebug_tether
>>> codebug = codebug_tether.CodeBug()
>>> codebug.set_pixel(2, 2, 1)
```

The middle pixel on your CodeBug should light up.

See *Examples* for more ways to use codebug_tether.

Troubleshooting

Raspberry Pi - Disable Serial Port Login Shell

CodeBug uses the USB serial port which (on older Raspberry Pi models) is configured to output the login shell by default. You must disable this before CodeBug will work. To do so, run:

```
sudo raspi-config
```

Navigate to *Advanced Options* > *Serial*, disable the login shell and then reboot.

CHAPTER 2

Examples

Basics

```
>>> import codebug_tether

>>> codebug = codebug_tether.CodeBug()  # create a CodeBug object

>>> codebug.set_pixel(2, 1, 1)  # turn on the LED at (2, 1)
>>> codebug.set_pixel(0, 0, 0)  # turn off the LED at (0, 0)
>>> codebug.get_pixel(0, 1)     # return the LED state at (0, 1)
1

>>> codebug.clear()  # turn off all LEDs

>>> codebug.set_row(0, 5)         # set row 0 to 5  (binary: 00101)
>>> codebug.set_row(1, 0x1a)      # set row 1 to 26 (binary: 11010)
>>> codebug.set_row(2, 0b10101)   # set row 2 to 21 (binary: 10101)
>>> bin(codebug.get_row(2))
'0b10101'

>>> codebug.set_col(0, 0x1f)  # turn on all LEDs in column 0
>>> codebug.get_col(0)
31

>>> codebug.get_input('A')  # returns the state of button 'A'
0
>>> codebug.get_input(0)  # returns the state of input 0
0

>>> codebug.set_leg_io(0, 0)  # set leg 0 to output
>>> codebug.set_output(0, 1)  # turn leg 0 'on' (1)
```

Note: You can specify a different serial device for CodeBug by passing the class a string. For example: `codebug`

```
= codebug_tether.CodeBug(serial_port='/dev/tty.USBmodem').
```

Sprites

You can use the sprites library to quickly draw things on CodeBug's display.

```
>>> import codebug_tether
>>> import codebug_tether.sprites

>>> # create a 3x3 square with the middle pixel off
>>> square_sprite = codebug_tether.sprites.Sprite(3, 3)
>>> square_sprite.set_row(0, 0b111)
>>> square_sprite.set_row(1, 0b101)
>>> square_sprite.set_row(2, 0b111)

>>> # draw it in the middle of CodeBug
>>> codebug = codebug_tether.CodeBug()
>>> codebug.draw_sprite(1, 1, square_sprite)

>>> # write some text
>>> message = codebug_tether.sprites.StringSprite('Hello CodeBug!')
>>> codebug.draw_sprite(0, 0, message)
>>> # move it along
>>> codebug.draw_sprite(-2, 0, message)
>>> # scroll a sprite
>>> codebug.scroll_sprite(message)
```

You can do some more interesting things with Sprites:

```
>>> import codebug_tether.sprites

>>> sprite = codebug_tether.sprites.Sprite(10, 10)

>>> # basic gets and sets
>>> sprite.set_pixel(0, 0, 1)
>>> sprite.get_pixel(0, 0)
1
>>> sprite.set_row(0, 0, 0b1111111111)
>>> sprite.get_row(0)
1023
>>> sprite.set_col(0, 0, 0b1111111111)
>>> sprite.get_col(0)
1023

>>> # transform the sprite
>>> sprite.invert_horizontal()
>>> sprite.invert_vertical()
>>> sprite.invert_diagonal()
>>> sprite.rotate90()
>>> sprite.rotate90(rotation=2)  # rotate 180 degrees

>>> # clone or extract parts of the sprite
>>> dolly_sprite = sprite.clone()
>>> rectangle = sprite.get_sprite(3, 3, 5, 2)
>>> # draw other sprites
>>> sprite.render_sprite(1, 1, rectangle)
```

You can also change the direction text is written in:

```
>>> from codebug_tether.sprites import StringSprite
>>> left_to_right_msg = StringSprite('Hello CodeBug!')
>>> right_to_left_msg = StringSprite('Hello CodeBug!', direction='L')
>>> top_to_bottom_msg = StringSprite('Hello CodeBug!', direction='D')
>>> bottom_to_top_msg = StringSprite('Hello CodeBug!', direction='U')
```

Analogue Input

You can read analogue inputs from all 8 of CodeBug's I/O legs/extension pins:

```
>>> import codebug_tether
>>> from codebug_tether import (IO_DIGITAL_INPUT,
...                              IO_ANALOGUE_INPUT,
...                              IO_PWM_OUTPUT,
...                              IO_DIGITAL_OUTPUT)
...
>>> codebug = codebug_tether.CodeBug()
>>> codebug.set_leg_io(0, IO_ANALOGUE_INPUT)
>>> codebug.read_analogue(0)
128
```

PWM Output

You can drive one synchronised PWM (Pulse Width Modulation) signal out of the first three legs on CodeBug. That is, the same PWM signal will be driven out of legs configured as PWM output:

```
>>> import codebug_tether
>>> from codebug_tether import (IO_DIGITAL_INPUT,
...                              IO_ANALOGUE_INPUT,
...                              IO_PWM_OUTPUT,
...                              IO_DIGITAL_OUTPUT,
...                              T2_PS_1_1,
...                              T2_PS_1_4,
...                              T2_PS_1_16)

>>> codebug = codebug_tether.CodeBug()
>>> # configure legs 0 and 1 to be PWM output
>>> codebug.set_leg_io(0, IO_PWM_OUTPUT)
>>> codebug.set_leg_io(1, IO_PWM_OUTPUT)
>>> # shortcut method to specify a frequency (the note C == 1046 Hz)
>>> codebug.pwm_freq(1046)
>>> time.sleep(2)
>>> codebug.pwm_off()
```

Or you can be more specific with the duty cycle and timing:

```
>>> # pwm on with 1:4 prescaler and 75% duty cycle @ ~977Hz
>>> # Timer 2 prescale: 4Mhz clock / 4 = 1MHz timer speed
>>> # full_period: 255 << 2 = 1024  (timer resets at this count; PWM = 1)
>>> # on_period: 765 (PWM goes to zero at this count; PWM = 0)
>>> # therefore duty cycle here is 75%
>>> codebug.pwm_on(T2_PS_1_4, 255, 765)
>>> time.sleep(2)
>>> codebug.pwm_off()
```

Servos

It is possible to drive up to eight servos from CodeBug. Servos typically operate by sending them a PWM (Pulse Width Modulation) signal with a 20ms period and a 1-2ms duty cycle which controls the angle of the servo. For example:

```
 Pulse Length
  <-1-2ms->
1+---------+
 |         |
0+         +---------------------------------------+
  <------------------20ms (50Hz)------------------>
                     PWM Period
```

A duty cycle of 1ms will typically correspond to 0° rotation and a duty cycle of 2ms will typically correspond to 180° rotation. Although the precise values may differ depending on the type of servo.

In order to drive servos from CodeBug you can call the *servo_set()* method which takes the servo index (which leg you are driving the servo from) and the the pulse length specified in N 0.5μs. For example:

```
>>> import codebug_tether
>>> from codebug_tether import (IO_DIGITAL_OUTPUT, scale)

>>> # init CodeBug and configure leg 0 to be digital output
>>> codebug = codebug_tether.CodeBug()
>>> codebug.set_leg_io(0, IO_DIGITAL_OUTPUT)

>>> # set servo on leg 0 with pulse length of 1ms (2000 * 0.5μs)
>>> codebug.servo_set(0, 2000)

>>> # stop driving the servo on leg 0
>>> codebug.servo_set(0, 0)
```

You can use the scale function to easily calculate the required pulse length value like so:

```
>>> import codebug_tether
>>> from codebug_tether import (IO_DIGITAL_OUTPUT, scale)

>>> # scale 50 in the range 0-100 to the range 0-255
>>> scale(50, 0, 100, 0, 255)
127

>>> # scale 10 in the range 0-30 to the range 100-400
>>> scale(10, 0, 30, 100, 400)
200

>>> # scale 90° in the range 0-180° to the range 2000-4000 * 0.5μs
>>> scale(90, 0, 180, 2000, 4000)
3000
```

```
>>> # init CodeBug and configure leg 0 to be digital output
>>> codebug = codebug_tether.CodeBug()
>>> codebug.set_leg_io(0, IO_DIGITAL_OUTPUT)

>>> # drive the servo to be at 90 degrees
>>> codebug.servo_set(0, scale(90, 0, 180, 2000, 4000))
```

Colour Tail

You can control Colour Tails (WS2812's) attached to CodeBug. By default, ColourTails attach to the CS pin on the extension header. You can also configure ColourTails to be driven from leg 0.

```
>>> import codebug_tether
>>> import codebug_tether.colourtail

>>> codebug = codebug_tether.CodeBug()
>>> colourtail = codebug_tether.colourtail.CodeBugColourTail(codebug)

>>> # draw arrow pointing to the Colour Tail
>>> codebug.set_row(4, 0b00100)
>>> codebug.set_row(3, 0b00100)
>>> codebug.set_row(2, 0b10101)
>>> codebug.set_row(1, 0b01110)
>>> codebug.set_row(0, 0b00100)

>>> # make sure the extension header is configured as I/O
>>> codebug.config_extension_io()

>>> # initialise the colourtail (using EXT_CS pin)
>>> colourtail.init()
>>> colourtail.set_pixel(0, 255, 0, 0)  # red
>>> colourtail.set_pixel(1, 0, 255, 0)  # green
>>> colourtail.set_pixel(2, 0, 0, 255)  # blue
>>> colourtail.update()  # turn on the LEDs

>>> # initialise the colourtail (using LEG_0 pin)
>>> colourtail.init(use_leg_0_not_cs=True)
>>> colourtail.set_pixel(0, 255, 0, 0)  # red
>>> colourtail.set_pixel(1, 255, 0, 0)  # red
>>> colourtail.set_pixel(2, 0, 255, 0)  # green
>>> colourtail.set_pixel(3, 0, 255, 0)  # green
>>> colourtail.set_pixel(4, 0, 0, 255)  # blue
>>> colourtail.set_pixel(5, 0, 0, 255)  # blue
>>> colourtail.update()  # turn on the LEDs
```

Extension Header

You can use the extension header to drive SPI and I2C buses.

Danger: Powering CodeBug from 5V USB means that the VCC pin on the extension header will also be at 5V. Do not use this pin to power devices which require less than 5V.

Connect your SPI/I2C device onto the SPI/I2C lines:

```
+                                 +
 +        Back of CodeBug       +
  +                           +
   +--------------------------+
   | CodeBug Extension Header |
   +--------------------------+
    |    |    |    |    |    |
    CS  GND  SDO  SCL  SDI/A VCC

+----------+---------------------+
| Pin Name | Function            |
+----------+---------------------+
| CS       | Chip Select         |
| GND      | Ground (0v)         |
| SDO      | SPI MOSI            |
| SCL      | SPI/I2C Clock       |
| SDI/A    | SPI MISO / I2C data |
| VCC      | V+ (3V3, 5V)        |
+----------+---------------------+
```

You can configure the extension header mode with the following methods:

```
>>> import codebug_tether

>>> codebug = codebug_tether.CodeBug()

>>> codebug.config_extension_spi()    # configure extension as SPI
>>> codebug.config_extension_i2c()    # configure extension as I2C
>>> codebug.config_extension_uart()   # configure extension as UART
>>> codebug.config_extension_io()     # reset extension as normal I/O
```

SPI

```
>>> import codebug_tether

>>> codebug = codebug_tether.CodeBug()
>>> codebug.config_extension_spi()

>>> # send three bytes (get three bytes back -- SPI is duplex)
>>> codebug.spi_transaction(bytes((0x12, 0x34, 0x56)))
b'\xff\xff\xff'
```

I2C

```
>>> import codebug_tether
>>> from codebug_tether.i2c import (reading, writing)
>>>
>>> # example I2C address
>>> i2c_addr = 0x1C
>>>
>>> # setup
>>> codebug = codebug_tether.CodeBug()
>>> codebug.config_extension_i2c()
```

Single byte read transaction (read reg 0x12):

```
>>> codebug.i2c_transaction(writing(i2c_addr, 0x12), # reg addr
                            reading(i2c_addr, 1))     # read 1 reg
(42,)
```

Multiple byte read transaction (read regs 0x12-0x17):

```
>>> codebug.i2c_transaction(writing(i2c_addr, 0x12), # reg addr
                            reading(i2c_addr, 6))     # read 6 reg
(65, 87, 47, 91, 43, 60)
```

Single byte write transaction (write value 0x34 to reg 0x12):

```
>>> codebug.i2c_transaction(writing(i2c_addr, 0x12, 0x34))
```

Multiple byte write transaction (write values 0x34, 0x56, 0x78 to reg 0x12):

```
>>> codebug.i2c_transaction(
        writing(i2c_addr, 0x12, 0x34, 0x56, 0x78))
```

UART

Sending data:

```
>>> import codebug_tether
>>> codebug = codebug_tether.CodeBug()
>>> codebug.config_extension_uart()
>>>
>>> # send 0xAA, 0xBB over UART
>>> codebug.uart_tx(bytes((0xAA, 0xBB)))
>>>
>>> # send 0xAA, 0xBB over UART at 300 baud
>>> codebug.uart_tx(bytes((0xAA, 0xBB)), baud=300)
```

You can also write to the buffer first and then send data from within it:

```
>>> import codebug_tether
>>> codebug = codebug_tether.CodeBug()
>>> codebug.config_extension_uart()
>>>
>>> codebug.uart_tx_set_buffer(bytes((0xAA, 0xBB)))
>>>
>>> codebug.uart_tx_start(1, offset=0)  # send 0xAA over UART
>>> codebug.uart_tx_start(1, offset=1)  # send 0xBB over UART
>>>
>>> # send 0xAA over UART at 300 baud
>>> codebug.uart_tx_start(1, offset=0, baud=300)
```

Receiving data:

```
>>> import codebug_tether
>>> codebug = codebug_tether.CodeBug()
>>> codebug.config_extension_uart()
>>>
>>> codebug.uart_rx_start(2)  # ready to receive 2B over UART
>>>
>>> # wait until data ready (alternatively, sleep X seconds)
>>> while not codebug.uart_rx_is_ready():
...     pass
...
>>> codebug.uart_rx_get_buffer(2)  # read out the two bytes
```

CHAPTER 3

Indices and tables

- genindex
- modindex
- search