|  |
| --- |
| from .buzzer import Buzzer |
|  | from .led import Led |
|  | from .led import LedRow |
|  | from .ds18b20 import DS18B20 |
|  | from .digital\_display import DigitalDisplay |
|  | from .dip\_switch\_2bit import DipSwitch2Bit |
|  | from .tact import Tact |
|  | from .tact import TactRow |
|  | from .ic\_74hc595 import IC\_74HC595 |
|  | from .led\_74hc595 import Led74HC595 |
|  | from .ic\_tm1637 import IC\_TM1637 |
|  | from .digital\_display\_tm1637 import DigitalDisplayTM1637 |

|  |
| --- |
| import RPi.GPIO as GPIO |
|  | import time |
|  |  |
|  | class Buzzer(object): |
|  | ''' |
|  | Buzzer class |
|  | ''' |
|  | \_\_pin = 0 |
|  | \_\_real\_true = GPIO.HIGH |
|  | \_\_is\_on = False |
|  |  |
|  | def \_\_init\_\_(self, pin, real\_true = GPIO.HIGH): |
|  | ''' |
|  | Init the buzzer |
|  | :param pin: pin number |
|  | :param real\_true: GPIO.HIGH or GPIO.LOW |
|  | :return: void |
|  | ''' |
|  | self.\_\_pin = pin |
|  | self.\_\_real\_true = real\_true |
|  |  |
|  | #Stauts. |
|  | @property |
|  | def is\_on(self): |
|  | ''' |
|  | Return the status of buzzer |
|  | :return: void |
|  | ''' |
|  | return self.\_\_is\_on |
|  |  |
|  | #Verbs. |
|  | def on(self): |
|  | ''' |
|  | Set buzzer on |
|  | :return: void |
|  | ''' |
|  | GPIO.output(self.\_\_pin, self.\_\_real\_true) |
|  | self.\_\_is\_on = True |
|  |  |
|  | def off(self): |
|  | ''' |
|  | Set buzzer off |
|  | :return: void |
|  | ''' |
|  | GPIO.output(self.\_\_pin, not self.\_\_real\_true) |
|  | self.\_\_is\_on = False |
|  |  |
|  | #functions. |
|  | def beep(self, seconds): |
|  | ''' |
|  | Beep one time |
|  | :param seconds: beep time |
|  | :return: void |
|  | ''' |
|  | self.on() |
|  | time.sleep(seconds) |
|  | self.off() |
|  |  |
|  | def beepAction(self, secs, sleepsecs, times): |
|  | ''' |
|  | Beep in a rhythm |
|  | e.g. beepAction(0.02,0.02,30) |
|  | :param secs: beep time |
|  | :param sleepsecs: break time |
|  | :param times: repeat times |
|  | :return: void |
|  | ''' |
|  | for i in range(times): |
|  | self.beep(secs) |
|  | time.sleep(sleepsecs) |

|  |
| --- |
| import RPi.GPIO as GPIO |
|  | import time |
|  | import re |
|  | from threading import Thread |
|  |  |
|  | class **DigitalDisplay**(object): |
|  | ''' |
|  | Digital display class |
|  | ''' |
|  | \_\_pins = {'seg':[], 'sel':[]} |
|  | \_\_real\_true = GPIO.HIGH |
|  | \_\_numbers = [] |
|  | \_\_is\_flushing = False |
|  | \_\_number\_code = [0x3f, 0x06, 0x5b, 0x4f, 0x66, 0x6d, 0x7d, 0x07, 0x7f, 0x6f, 0x00, 0x40] |
|  |  |
|  | def \_\_init\_\_(self, pins, real\_true = GPIO.HIGH): |
|  | ''' |
|  | Init the digital display |
|  | :param pin: pin numbers in array |
|  | :param real\_true: GPIO.HIGH or GPIO.LOW |
|  | :return: void |
|  | ''' |
|  | self.\_\_pins = pins |
|  | self.\_\_real\_true = real\_true |
|  | try: |
|  | t1 = Thread(target = self.flush\_4bit) |
|  | t1.setDaemon(True) |
|  | t1.start() |
|  | except: |
|  | print("Error: Unable to start thread by DigitalDisplay") |
|  |  |
|  | #Stauts. |
|  | **@**property |
|  | def **numbers**(self): |
|  | ''' |
|  | Get the current numbers array showing |
|  | :return: numbers array |
|  | ''' |
|  | return self.\_\_numbers |
|  |  |
|  | #@numbers.setter |
|  | def **set\_numbers**(self, value): |
|  | ''' |
|  | Set the numbers array to show |
|  | :return: void |
|  | ''' |
|  | pattern = re.compile(r'[-|#|\d]\.?') |
|  | matches = pattern.findall(value) |
|  | #del self.\_\_numbers |
|  | self.\_\_numbers = [] |
|  | for i in range(len(matches)): |
|  | self.\_\_numbers.append(matches[i]) |
|  | #print(self.\_\_numbers) |
|  |  |
|  | #@numbers.deleter |
|  | #def numbers(self): |
|  | # del self.\_\_numbers |
|  |  |
|  | #Verbs. |
|  | def **on**(self): |
|  | ''' |
|  | Set display on |
|  | :return: void |
|  | ''' |
|  | self.\_\_is\_flushing = True |
|  |  |
|  | def **off**(self): |
|  | ''' |
|  | Set display off |
|  | :return: void |
|  | ''' |
|  | self.\_\_is\_flushing = False |
|  | for p in self.\_\_pins['sel'] + self.\_\_pins['seg']: |
|  | GPIO.output(p, not self.\_\_real\_true) |
|  |  |
|  | def **show**(self, str): |
|  | ''' |
|  | Set the numbers array to show and enable the display |
|  | :return: void |
|  | ''' |
|  | self.\_\_is\_flushing = False |
|  | self.set\_numbers(str) |
|  | self.\_\_is\_flushing = True |
|  | #print(self.\_\_numbers) |
|  |  |
|  |  |
|  | def flush\_bit(self, sel, num, dp): |
|  | if num == '#': |
|  | num = 10 |
|  | elif num == '-': |
|  | num = 11 |
|  | else: |
|  | num = int(num) |
|  |  |
|  | GPIO.output(self.\_\_pins['sel'][sel], self.\_\_real\_true) |
|  | n = self.\_\_number\_code[num] |
|  |  |
|  | if dp: |
|  | n = n | 10000000 |
|  |  |
|  | for i in range(8): |
|  | if (n & (1 << i)): |
|  | GPIO.output(self.\_\_pins['seg'][i], self.\_\_real\_true) |
|  |  |
|  | GPIO.output(self.\_\_pins['sel'][sel], not self.\_\_real\_true) |
|  |  |
|  | for i in self.\_\_pins['seg']: |
|  | GPIO.output(i, not self.\_\_real\_true) |
|  |  |
|  | def **flush\_4bit**(self): |
|  | while True: |
|  | if self.\_\_is\_flushing: |
|  | #print(self.\_\_numbers) |
|  | #print(range(min(4, len(self.\_\_numbers)))) |
|  | try: |
|  | for i in range(min(4, len(self.\_\_numbers))): |
|  | self.flush\_bit(i, self.\_\_numbers[i].replace('.',''), True if self.\_\_numbers[i].count('.') > 0 else False) |
|  | time.sleep(0.001) |
|  | except: |
|  | pass |

|  |
| --- |
| import RPi.GPIO as GPIO |
|  | import re |
|  | from .ic\_tm1637 import IC\_TM1637 as IC\_TM1637 |
|  |  |
|  | class DigitalDisplayTM1637(object): |
|  | ''' |
|  | Digital display class |
|  | ''' |
|  |  |
|  | \_\_ic\_tm1637 = None |
|  | \_\_numbers = [] |
|  | \_\_number\_code = [0x3f, 0x06, 0x5b, 0x4f, 0x66, 0x6d, 0x7d, 0x07, 0x7f, 0x6f, 0x00, 0x40] |
|  | \_\_address\_code = [0xc0, 0xc1, 0xc2, 0xc3] |
|  | \_\_is\_on = False |
|  |  |
|  |  |
|  | def \_\_init\_\_(self, pins, real\_true = GPIO.HIGH): |
|  | ''' |
|  | Init the digital display |
|  | :param pin: pin numbers in array |
|  | :param real\_true: GPIO.HIGH or GPIO.LOW |
|  | :return: void |
|  | ''' |
|  | self.\_\_ic\_tm1637 = IC\_TM1637(pins, real\_true) |
|  |  |
|  | #Stauts. |
|  | **@**property |
|  | def **is\_on**(self): |
|  | ''' |
|  | Get the current status of the digital display |
|  | ''' |
|  | return self.\_\_is\_on |
|  |  |
|  | **@**property |
|  | def **numbers**(self): |
|  | ''' |
|  | Get the current numbers array showing |
|  | :return: numbers array |
|  | ''' |
|  | return self.\_\_numbers |
|  |  |
|  | #@numbers.setter |
|  | def **set\_numbers**(self, value): |
|  | ''' |
|  | Set the numbers array to show |
|  | :return: void |
|  | ''' |
|  | pattern = re.compile(r'[-|#|\d]**\.**?') |
|  | matches = pattern.findall(value) |
|  | #del self.\_\_numbers |
|  | self.\_\_numbers = [] |
|  | for i in range(len(matches)): |
|  | self.\_\_numbers.append(matches[i]) |
|  | #print(self.\_\_numbers) |
|  |  |
|  | #@numbers.deleter |
|  | #def numbers(self): |
|  | # del self.\_\_numbers |
|  |  |
|  | **@**property |
|  | def **ic**(self): |
|  | ''' |
|  | Return the instance of ic |
|  | :return: ic |
|  | ''' |
|  | return self.\_\_ic\_tm1637 |
|  |  |
|  | #Verbs. |
|  | def **on**(self): |
|  | ''' |
|  | Set display on |
|  | :return: void |
|  | ''' |
|  | self.\_\_ic\_tm1637.set\_command(0x8f) |
|  | self.\_\_is\_on = True |
|  |  |
|  | def **off**(self): |
|  | ''' |
|  | Set display off |
|  | :return: void |
|  | ''' |
|  | self.\_\_ic\_tm1637.clear() |
|  | self.\_\_is\_on = False |
|  |  |
|  | def **show**(self, str): |
|  | ''' |
|  | Set the numbers array to show and enable the display |
|  | :return: void |
|  | ''' |
|  | self.set\_numbers(str) |
|  | #print(self.\_\_numbers) |
|  |  |
|  | self.\_\_ic\_tm1637.set\_command(0x44) |
|  |  |
|  | for i in range(min(4, len(self.\_\_numbers))): |
|  | dp = True if self.\_\_numbers[i].count('.') > 0 else False |
|  | num = self.\_\_numbers[i].replace('.','') |
|  | if num == '#': |
|  | num = 10 |
|  | elif num == '-': |
|  | num = 11 |
|  | else: |
|  | num = int(num) |
|  |  |
|  | if dp: |
|  | self.\_\_ic\_tm1637.set\_data(self.\_\_address\_code[i], self.\_\_number\_code[num]|0x80) |
|  | else: |
|  | self.\_\_ic\_tm1637.set\_data(self.\_\_address\_code[i], self.\_\_number\_code[num]) |
|  |  |
|  | self.on() |

|  |
| --- |
| import RPi.GPIO as GPIO |
|  | import time |
|  | from threading import Thread |
|  |  |
|  | class DipSwitch2Bit(object): |
|  | ''' |
|  | Dip switch (2bit) class |
|  | ''' |
|  | \_\_pins = [] |
|  | \_\_real\_true = GPIO.HIGH |
|  | \_\_status = [] |
|  |  |
|  | \_\_observers = [] |
|  |  |
|  | def \_\_init\_\_(self, pins, real\_true = GPIO.HIGH): |
|  | ''' |
|  | Init the dip switch |
|  | :param pin: pin numbers in array |
|  | :param real\_true: GPIO.HIGH or GPIO.LOW |
|  | :return: void |
|  | ''' |
|  | self.\_\_pins = pins |
|  | self.\_\_real\_true = real\_true |
|  | for p in pins: |
|  | self.\_\_status.append(not real\_true) |
|  |  |
|  | if self.\_\_real\_true: |
|  | self.\_\_status[0] = GPIO.input(self.\_\_pins[0]) |
|  | self.\_\_status[1] = GPIO.input(self.\_\_pins[1]) |
|  | else: |
|  | self.\_\_status[0] = not GPIO.input(self.\_\_pins[0]) |
|  | self.\_\_status[1] = not GPIO.input(self.\_\_pins[1]) |
|  |  |
|  |  |
|  | GPIO.add\_event\_detect(self.\_\_pins[0], GPIO.BOTH, callback = self.make\_event, bouncetime = 50) |
|  | GPIO.add\_event\_detect(self.\_\_pins[1], GPIO.BOTH, callback = self.make\_event, bouncetime = 50) |
|  |  |
|  | try: |
|  | t1 = Thread(target = self.watching) |
|  | t1.setDaemon(True) |
|  | #t1.start() |
|  | except: |
|  | print("Error: Unable to start thread by DipSwitch") |
|  |  |
|  | #Stauts. |
|  | @property |
|  | def is\_on(self): |
|  | ''' |
|  | Get the status of each bit |
|  | :return: the status array |
|  | ''' |
|  | return self.\_\_status |
|  |  |
|  | #Events |
|  | def register(self, observer): |
|  | if observer not in self.\_\_observers: |
|  | self.\_\_observers.append(observer) |
|  |  |
|  | def deregister(self, observer): |
|  | if observer in self.\_\_observers: |
|  | self.\_\_observers.remove(observer) |
|  |  |
|  | def notify\_observers(self): |
|  | for o in self.\_\_observers: |
|  | #o.update(self.\_\_status) |
|  | o.on\_dip\_switch\_2bit\_status\_changed(self.\_\_status) |
|  |  |
|  | def status\_changed(self): |
|  | self.notify\_observers() |
|  |  |
|  | def make\_event(self, channel): |
|  | if self.\_\_real\_true: |
|  | if GPIO.input(self.\_\_pins[0]) != self.\_\_status[0]: |
|  | self.\_\_status[0] = GPIO.input(self.\_\_pins[0]) |
|  | self.status\_changed() |
|  |  |
|  | if GPIO.input(self.\_\_pins[1]) != self.\_\_status[1]: |
|  | self.\_\_status[1] = GPIO.input(self.\_\_pins[1]) |
|  | self.status\_changed() |
|  | else: |
|  | if GPIO.input(self.\_\_pins[0]) == self.\_\_status[0]: |
|  | self.\_\_status[0] = not GPIO.input(self.\_\_pins[0]) |
|  | self.status\_changed() |
|  |  |
|  | if GPIO.input(self.\_\_pins[1]) == self.\_\_status[1]: |
|  | self.\_\_status[1] = not GPIO.input(self.\_\_pins[1]) |
|  | self.status\_changed() |
|  |  |
|  | def watching(self): |
|  | if self.\_\_real\_true: |
|  | while True: |
|  | if GPIO.input(self.\_\_pins[0]) != self.\_\_status[0]: |
|  | self.\_\_status[0] = GPIO.input(self.\_\_pins[0]) |
|  | self.status\_changed() |
|  |  |
|  | if GPIO.input(self.\_\_pins[1]) != self.\_\_status[1]: |
|  | self.\_\_status[1] = GPIO.input(self.\_\_pins[1]) |
|  | self.status\_changed() |
|  |  |
|  | time.sleep(0.05) |
|  | else: |
|  | while True: |
|  | if GPIO.input(self.\_\_pins[0]) == self.\_\_status[0]: |
|  | self.\_\_status[0] = not GPIO.input(self.\_\_pins[0]) |
|  | self.status\_changed() |
|  |  |
|  | if GPIO.input(self.\_\_pins[1]) == self.\_\_status[1]: |
|  | self.\_\_status[1] = not GPIO.input(self.\_\_pins[1]) |
|  | self.status\_changed() |
|  |  |
|  | time.sleep(0.05) |

|  |
| --- |
| import time |
|  | import subprocess |
|  | import os |
|  | import glob |
|  |  |
|  | class DS18B20(object): |
|  | ''' |
|  | DS18B20 class |
|  | ''' |
|  | \_\_pin = 0 #it's no use |
|  | #\_\_device\_file = '' |
|  | #\_\_temperature = -128 |
|  |  |
|  | def \_\_init\_\_(self, pin = 4): |
|  | ''' |
|  | Init the DS18b20 |
|  | :param pin: pin number |
|  | :return: void |
|  | ''' |
|  | self.\_\_pins = pin |
|  | os.system('sudo modprobe w1-gpio') |
|  | os.system('sudo modprobe w1-therm') |
|  |  |
|  | #Verbs. |
|  | def get\_device\_file(self, index = 0): |
|  | base\_dir = '/sys/bus/w1/devices/' |
|  | #fix "IndexError: list index out of range" |
|  | if not glob.glob(base\_dir + '28\*'): |
|  | return False |
|  | if glob.glob(base\_dir + '28\*')[index] is not None: |
|  | device\_folder = glob.glob(base\_dir + '28\*')[index] |
|  | return device\_folder + '/w1\_slave' |
|  | else: |
|  | return False |
|  |  |
|  | def read\_temp\_raw(self, index = 0): |
|  | df = self.get\_device\_file(index) |
|  | if not df: |
|  | return False |
|  | catdata = subprocess.Popen(['cat', df], stdout = subprocess.PIPE, stderr = subprocess.PIPE) |
|  | out,err = catdata.communicate() |
|  | out\_decode = out.decode('utf-8') |
|  | lines = out\_decode.split('**\n**') |
|  | return lines |
|  |  |
|  | def read\_temp(self, index = 0): |
|  | tr = self.read\_temp\_raw(index) |
|  | if not tr: |
|  | return False |
|  | lines = tr |
|  | while lines[0].strip()[-3:] != 'YES': |
|  | time.sleep(0.2) |
|  | tr = self.read\_temp\_raw(index) |
|  | if not tr: |
|  | return False |
|  | lines = tr |
|  | equals\_pos = lines[1].find('t=') |
|  | if equals\_pos != -1: |
|  | temp\_string = lines[1][equals\_pos+2:] |
|  | temp\_c = float(temp\_string) / 1000.0 |
|  | temp\_f = temp\_c \* 9.0 / 5.0 + 32.0 |
|  | #return temp\_c, temp\_f |
|  | return temp\_c |
|  |  |
|  | #Stauts. |
|  | @property |
|  | def is\_exist(self, index = 0): |
|  | ''' |
|  | Return true if the ds18b20 is exist |
|  | :param index: from 0 to n |
|  | :return: Return true if the ds18b20 is exist |
|  | ''' |
|  | #if not os.path.exists(self.\_\_device\_file): |
|  | # return False |
|  | #else: |
|  | # return True |
|  | return self.get\_device\_file(index) |
|  |  |
|  | @property |
|  | def temperature(self, index = 0): |
|  | ''' |
|  | Get the temperature from ds18b20 |
|  | :param index: from 0 to n |
|  | :return: Return the temperature from ds18b20, return -128 means get a error. |
|  | ''' |
|  | if not self.is\_exist: |
|  | return -128.0 |
|  | else: |
|  | return self.read\_temp(index) |

|  |
| --- |
| import RPi.GPIO as GPIO |
|  | import time |
|  |  |
|  | class IC\_TM1637(object): |
|  | ''' |
|  | IC\_TM1637 class |
|  | ''' |
|  | \_\_pins = {'di' : 0, 'clk' : 0} |
|  | \_\_real\_true = GPIO.HIGH |
|  |  |
|  | def \_\_init\_\_(self, pins, real\_true = GPIO.HIGH): |
|  | ''' |
|  | Init the ic |
|  | :param pin: pin number |
|  | :param real\_true: GPIO.HIGH or GPIO.LOW |
|  | :return: void |
|  | ''' |
|  | self.\_\_pins = pins |
|  | self.\_\_real\_true = real\_true |
|  |  |
|  | #Verbs. |
|  | def bus\_delay(self): |
|  | ''' |
|  | Delay |
|  | :return: void |
|  | ''' |
|  | time.sleep(0.001) |
|  |  |
|  | def start\_bus(self): |
|  | ''' |
|  | Start bus |
|  | :return: void |
|  | ''' |
|  | GPIO.output(self.\_\_pins['clk'], self.\_\_real\_true) |
|  | GPIO.output(self.\_\_pins['di'], self.\_\_real\_true) |
|  | self.bus\_delay() |
|  | GPIO.output(self.\_\_pins['di'], not self.\_\_real\_true) |
|  | self.bus\_delay() |
|  | GPIO.output(self.\_\_pins['clk'], not self.\_\_real\_true) |
|  | self.bus\_delay() |
|  |  |
|  | def stop\_bus(self): |
|  | ''' |
|  | Stop bus |
|  | :return: void |
|  | ''' |
|  | GPIO.output(self.\_\_pins['clk'], not self.\_\_real\_true) |
|  | self.bus\_delay() |
|  | GPIO.output(self.\_\_pins['di'], not self.\_\_real\_true) |
|  | self.bus\_delay() |
|  | GPIO.output(self.\_\_pins['clk'], self.\_\_real\_true) |
|  | self.bus\_delay() |
|  | GPIO.output(self.\_\_pins['di'], self.\_\_real\_true) |
|  | self.bus\_delay() |
|  |  |
|  | def set\_bit(self, bit): |
|  | ''' |
|  | Set a bit |
|  | :param bit: bit |
|  | :return: void |
|  | ''' |
|  | GPIO.output(self.\_\_pins['clk'], not self.\_\_real\_true) |
|  | self.bus\_delay() |
|  | GPIO.output(self.\_\_pins['di'], bit) |
|  | self.bus\_delay() |
|  | GPIO.output(self.\_\_pins['clk'], self.\_\_real\_true) |
|  | self.bus\_delay() |
|  |  |
|  | def set\_byte(self, data): |
|  | ''' |
|  | Set a byte |
|  | :param data: data |
|  | :return: void |
|  | ''' |
|  | for i in range (0, 8): |
|  | self.set\_bit((data >> i) & 0x01) |
|  |  |
|  | GPIO.output(self.\_\_pins['clk'], not self.\_\_real\_true) |
|  | self.bus\_delay() |
|  |  |
|  | GPIO.output(self.\_\_pins['di'], self.\_\_real\_true) |
|  | self.bus\_delay() |
|  |  |
|  | GPIO.output(self.\_\_pins['clk'], self.\_\_real\_true) |
|  | self.bus\_delay() |
|  |  |
|  | def set\_command(self, command): |
|  | ''' |
|  | Set command |
|  | :param command: command code |
|  | :return: void |
|  | ''' |
|  | self.start\_bus() |
|  | self.set\_byte(command) |
|  | self.start\_bus() |
|  |  |
|  | def set\_data(self, address, data): |
|  | ''' |
|  | Set data with address and data |
|  | :param address: address |
|  | :param data: data |
|  | :return: void |
|  | ''' |
|  | self.start\_bus() |
|  | self.set\_byte(address) |
|  | self.set\_byte(data) |
|  | self.start\_bus() |
|  |  |
|  | def clear(self): |
|  | ''' |
|  | Clear the data |
|  | :return: void |
|  | ''' |
|  | self.set\_command(0x80) |

|  |
| --- |
| import RPi.GPIO as GPIO |
|  | import time |
|  | from threading import Thread |
|  |  |
|  | class Led(object): |
|  | ''' |
|  | Led class |
|  | ''' |
|  | \_\_pin = 0 |
|  | \_\_real\_true = GPIO.HIGH |
|  | \_\_pwm = None |
|  | \_\_is\_on = False |
|  | \_\_is\_pulse = None |
|  |  |
|  | def \_\_init\_\_(self, pin, real\_true = GPIO.HIGH): |
|  | ''' |
|  | Init the led |
|  | :param pin: pin numbers in array |
|  | :param real\_true: GPIO.HIGH or GPIO.LOW |
|  | :return: void |
|  | ''' |
|  | self.\_\_pin = pin |
|  | self.\_\_real\_true = real\_true |
|  |  |
|  | #Stauts. |
|  | @property |
|  | def is\_on(self): |
|  | ''' |
|  | Get the current status of the led |
|  | ''' |
|  | return self.\_\_is\_on |
|  |  |
|  | #Verbs. |
|  | def on(self): |
|  | ''' |
|  | Set the led on |
|  | ''' |
|  | if not self.\_\_is\_pulse: |
|  | GPIO.output(self.\_\_pin, self.\_\_real\_true) |
|  | self.\_\_is\_on = True |
|  |  |
|  | def off(self): |
|  | ''' |
|  | Set the led off |
|  | ''' |
|  | if self.\_\_is\_pulse: |
|  | self.\_\_is\_pulse = False |
|  | self.\_\_pwm.stop() |
|  | time.sleep(0.1) |
|  | GPIO.output(self.\_\_pin, not self.\_\_real\_true) |
|  | self.\_\_is\_on = False |
|  |  |
|  | #functions. |
|  | def flash(self, seconds): |
|  | ''' |
|  | Flash one time |
|  | :param seconds: on time |
|  | :return: void |
|  | ''' |
|  | self.on() |
|  | time.sleep(seconds) |
|  | self.off() |
|  |  |
|  | #e.g. flashAction(0.02,0.02,30) |
|  | def flashAction(self, secs, sleepsecs, times): |
|  | ''' |
|  | Flash in a rhythm |
|  | e.g. flashAction(0.02,0.02,30) |
|  | :param secs: on time |
|  | :param sleepsecs: break time |
|  | :param times: repeat times |
|  | :return: void |
|  | ''' |
|  | for i in range(times): |
|  | self.flash(secs) |
|  | time.sleep(sleepsecs) |
|  |  |
|  | def pulse(self, hertz=50, pause\_time=0.01): |
|  | ''' |
|  | Breath until led off |
|  | :param hertz: GPIO PWM hertz |
|  | :param pause\_time: breath pause time |
|  | :return: void |
|  | ''' |
|  | if self.\_\_pwm == None: |
|  | self.\_\_pwm = GPIO.PWM(self.\_\_pin, hertz) |
|  | else: |
|  | self.\_\_pwm.ChangeFrequency(hertz) |
|  | self.\_\_pwm.start(0) |
|  | if self.\_\_is\_pulse == None: |
|  | def pulse\_worker(): |
|  | while True: |
|  | if self.\_\_is\_pulse: |
|  | try: |
|  | for i in xrange(0, 101, 1): |
|  | self.\_\_pwm.ChangeDutyCycle(i) |
|  | # off |
|  | time.sleep(pause\_time) |
|  | time.sleep(1) |
|  | for i in xrange(100, -1, -1): |
|  | self.\_\_pwm.ChangeDutyCycle(i) |
|  | # on |
|  | time.sleep(pause\_time) |
|  | except: |
|  | continue |
|  | try: |
|  | pulse\_thread = Thread(target = pulse\_worker) |
|  | pulse\_thread.setDaemon(True) |
|  | pulse\_thread.start() |
|  | except: |
|  | print('Error: Unable to start thread by Led') |
|  | self.\_\_is\_pulse = True |
|  | self.\_\_is\_on = True |
|  |  |
|  | class LedRow(object): |
|  | ''' |
|  | Class of leds in row |
|  | ''' |
|  | \_\_leds = [] |
|  | \_\_pins = [] |
|  | \_\_real\_true = GPIO.HIGH |
|  |  |
|  | def \_\_init\_\_(self, pins, real\_true = GPIO.HIGH): |
|  | ''' |
|  | Init the leds |
|  | :param pin: pin numbers in array |
|  | :param real\_true: GPIO.HIGH or GPIO.LOW |
|  | :return: void |
|  | ''' |
|  | self.\_\_pins = pins |
|  | self.\_\_real\_true = real\_true |
|  | for p in pins: |
|  | self.\_\_leds.append(Led(p, real\_true)) |
|  |  |
|  | #Stauts. |
|  | #@property |
|  | def is\_on(self, index): |
|  | ''' |
|  | Get status of led in ledrow by index |
|  | :param index: index of the led |
|  | :return: status in boolean |
|  | ''' |
|  | if index >= len(self.\_\_leds): |
|  | return False |
|  | return self.\_\_leds[index].is\_on |
|  |  |
|  | @property |
|  | def row\_status(self): |
|  | ''' |
|  | Get status array of the ledrow |
|  | :return: status array |
|  | ''' |
|  | r = [] |
|  | for l in self.\_\_leds: |
|  | r.append(l.is\_on) |
|  | return r |
|  |  |
|  | @property |
|  | def items(self): |
|  | ''' |
|  | Get the instances of the leds in ledrow |
|  | :return: instances array |
|  | ''' |
|  | return self.\_\_leds |
|  |  |
|  | #Verbs. |
|  | #@multimethod() |
|  | def on(self): |
|  | ''' |
|  | Set all the leds on |
|  | :return: void |
|  | ''' |
|  | for l in self.\_\_leds: |
|  | l.on() |
|  |  |
|  | #@multimethod() |
|  | def off(self): |
|  | ''' |
|  | Set all the leds off |
|  | :return: void |
|  | ''' |
|  | for l in self.\_\_leds: |
|  | l.off() |
|  |  |
|  | #@multimethod(int) |
|  | def on\_for\_index(self, index): |
|  | ''' |
|  | Set the led on by index in the ledrow |
|  | :return: void |
|  | ''' |
|  | self.\_\_leds[index].on() |
|  |  |
|  | #@multimethod(int) |
|  | def off\_for\_index(self, index): |
|  | ''' |
|  | Set the led off by index in the ledrow |
|  | :return: void |
|  | ''' |
|  | self.\_\_leds[index].off() |
|  |  |
|  | def set\_row(self, status): |
|  | ''' |
|  | Set the ledrow's status in boolean array |
|  | :param status: boolean array |
|  | :return: void |
|  | ''' |
|  | for i in range(len(status)): |
|  | #print(str(i) + str(status[i])) |
|  | if status[i] is None: |
|  | continue |
|  | if status[i]: |
|  | self.\_\_leds[i].on() |
|  | else: |
|  | self.\_\_leds[i].off() |

|  |
| --- |
| import RPi.GPIO as GPIO |
|  | from .ic\_74hc595 import IC\_74HC595 as IC\_74HC595 |
|  |  |
|  | class **Led74HC595**(object): |
|  | ''' |
|  | Class of leds in 74HC595 |
|  | ''' |
|  | \_\_ic\_74hc595 = None |
|  |  |
|  | def \_\_init\_\_(self, pins, real\_true = GPIO.HIGH): |
|  | ''' |
|  | Init the leds |
|  | :param pin: pin numbers in array |
|  | :param real\_true: GPIO.HIGH or GPIO.LOW |
|  | :return: void |
|  | ''' |
|  | self.\_\_ic\_74hc595 = IC\_74HC595(pins, real\_true) |
|  |  |
|  | #Stauts. |
|  | **@**property |
|  | def **ic**(self): |
|  | return self.\_\_ic\_74hc595 |
|  |  |
|  | def **is\_on**(self, index): |
|  | ''' |
|  | Get status of led in ledrow by index |
|  | :param index: index of the led |
|  | :return: status in boolean |
|  | ''' |
|  | if index >= 8: |
|  | return False |
|  | return self.\_\_ic\_74hc595.data >> index & 0x01 |
|  |  |
|  | **@**property |
|  | def **row\_status**(self): |
|  | ''' |
|  | Get status array of the ledrow |
|  | :return: status array |
|  | ''' |
|  | r = [] |
|  | for i in range (0, 8): |
|  | r.append(self.\_\_ic\_74hc595.data >> i & 0x01) |
|  | return r |
|  |  |
|  | #Verbs. |
|  | def **on**(self): |
|  | ''' |
|  | Set all the leds on |
|  | :return: void |
|  | ''' |
|  | self.\_\_ic\_74hc595.set\_data(0xff) |
|  |  |
|  | def **off**(self): |
|  | ''' |
|  | Set all the leds off |
|  | :return: void |
|  | ''' |
|  | self.\_\_ic\_74hc595.clear() |
|  |  |
|  | def **on\_for\_index**(self, index): |
|  | ''' |
|  | Set the led on by index in the ledrow |
|  | :return: void |
|  | ''' |
|  | self.\_\_ic\_74hc595.set\_data(self.\_\_ic\_74hc595.data | (0x01 << (index))) |
|  |  |
|  | def **off\_for\_index**(self, index): |
|  | ''' |
|  | Set the led off by index in the ledrow |
|  | :return: void |
|  | ''' |
|  | arr = [0xfe, 0xfd, 0xfb, 0xf7, 0xef, 0xdf, 0xbf, 0x7f] |
|  | self.\_\_ic\_74hc595.set\_data(self.\_\_ic\_74hc595.data & arr[index]) |
|  |  |
|  | def **set\_row**(self, status): |
|  | ''' |
|  | Set the ledrow's status in boolean array |
|  | :param status: boolean array |
|  | :return: void |
|  | ''' |
|  | for i in range(len(status)): |
|  | #print(str(i) + str(status[i])) |
|  | if status[i] is None: |
|  | continue |
|  | if status[i]: |
|  | self.on\_for\_index(i) |
|  | else: |
|  | self.off\_for\_index(i) |

|  |
| --- |
| import RPi.GPIO as GPIO |
|  | import time |
|  | from threading import Thread |
|  |  |
|  | class Tact(object): |
|  | ''' |
|  | Tact class |
|  | ''' |
|  | \_\_pin = 0 |
|  | \_\_real\_true = GPIO.HIGH |
|  | \_\_status = False |
|  |  |
|  | \_\_observers = [] |
|  |  |
|  | def \_\_init\_\_(self, pin, real\_true = GPIO.HIGH): |
|  | ''' |
|  | Init the tact |
|  | :param pin: pin number in array |
|  | :param real\_true: GPIO.HIGH or GPIO.LOW |
|  | :return: void |
|  | ''' |
|  | self.\_\_pin = pin |
|  | self.\_\_real\_true = real\_true |
|  |  |
|  | if self.\_\_real\_true: |
|  | self.\_\_status = GPIO.input(self.\_\_pin) |
|  | else: |
|  | self.\_\_status = not GPIO.input(self.\_\_pin) |
|  |  |
|  | GPIO.add\_event\_detect(pin, GPIO.BOTH, callback = self.make\_event, bouncetime = 1) |
|  |  |
|  | try: |
|  | t1 = Thread(target = self.watching) |
|  | t1.setDaemon(True) |
|  | #t1.start() |
|  | except: |
|  | print("Error: Unable to start thread by Tact") |
|  |  |
|  |  |
|  | #Stauts. |
|  | @property |
|  | def is\_on(self): |
|  | ''' |
|  | Get current of tact |
|  | ''' |
|  | if self.\_\_real\_true: |
|  | if self.\_\_status != GPIO.input(self.\_\_pin): |
|  | self.\_\_status = GPIO.input(self.\_\_pin) |
|  | else: |
|  | if self.\_\_status == GPIO.input(self.\_\_pin): |
|  | self.\_\_status = not GPIO.input(self.\_\_pin) |
|  |  |
|  | return self.\_\_status |
|  |  |
|  | #Events |
|  | def register(self, observer): |
|  | if observer not in self.\_\_observers: |
|  | self.\_\_observers.append(observer) |
|  |  |
|  | def deregister(self, observer): |
|  | if observer in self.\_\_observers: |
|  | self.\_\_observers.remove(observer) |
|  |  |
|  | def notify\_observers(self, status): |
|  | for o in self.\_\_observers: |
|  | o.on\_tact\_event(self.\_\_pin, status) |
|  |  |
|  | def event(self, action): |
|  | self.notify\_observers(action) |
|  |  |
|  | def make\_event(self, channel): |
|  | self.notify\_observers(self.\_\_real\_true if GPIO.input(self.\_\_pin) else not self.\_\_real\_true) |
|  | if self.\_\_real\_true: |
|  | if self.\_\_status != GPIO.input(self.\_\_pin): |
|  | self.\_\_status = GPIO.input(self.\_\_pin) |
|  | #self.notify\_observers(self.\_\_real\_true if self.\_\_status else not self.\_\_real\_true) |
|  | else: |
|  | if self.\_\_status == GPIO.input(self.\_\_pin): |
|  | self.\_\_status = not GPIO.input(self.\_\_pin) |
|  | #self.notify\_observers(self.\_\_real\_true if not self.\_\_status else not self.\_\_real\_true) |
|  |  |
|  | def watching(self): |
|  | if self.\_\_real\_true: |
|  | while True: |
|  | if GPIO.input(self.\_\_pin) != self.\_\_status: |
|  | self.\_\_status = GPIO.input(self.\_\_pin) |
|  | self.notify\_observers(self.\_\_real\_true if self.\_\_status else not self.\_\_real\_true) |
|  | time.sleep(0.05) |
|  | else: |
|  | while True: |
|  | if GPIO.input(self.\_\_pin) == self.\_\_status: |
|  | self.\_\_status = not GPIO.input(self.\_\_pin) |
|  | self.notify\_observers(self.\_\_real\_true if not self.\_\_status else not self.\_\_real\_true) |
|  | time.sleep(0.05) |
|  |  |
|  | class TactRow(object): |
|  | ''' |
|  | Class of tacts in row |
|  | ''' |
|  | \_\_tacts = [] |
|  | \_\_pins = [] |
|  | \_\_real\_true = GPIO.HIGH |
|  |  |
|  | def \_\_init\_\_(self, pins, real\_true = GPIO.HIGH): |
|  | ''' |
|  | Init the tacts |
|  | :param pin: pin numbers in array |
|  | :param real\_true: GPIO.HIGH or GPIO.LOW |
|  | :return: void |
|  | ''' |
|  | self.\_\_pins = pins |
|  | self.\_\_real\_true = real\_true |
|  | for p in pins: |
|  | self.\_\_tacts.append(Tact(p, real\_true)) |
|  |  |
|  | #Stauts. |
|  | def is\_on(self, index): |
|  | ''' |
|  | Get status of tact in tactrow by index |
|  | :param index: index of the tact |
|  | :return: status in boolean |
|  | ''' |
|  | if index >= len(self.\_\_tacts): |
|  | return False |
|  | return self.\_\_tacts[index].is\_on |
|  |  |
|  | @property |
|  | def row\_status(self): |
|  | ''' |
|  | Get status array of the tactrow |
|  | :return: status array |
|  | ''' |
|  | r = [] |
|  | for l in self.\_\_tacts: |
|  | r.append(l.is\_on) |
|  | return r |
|  |  |
|  | @property |
|  | def items(self): |
|  | ''' |
|  | Get the instances of the tacts in tactrow |
|  | :return: instances array |
|  | ''' |
|  | return self.\_\_tacts |