An overview of the code

September 4, 2023

```
import time
import logging
import collections
import bluetooth
import csv
from datetime import datetime
import os
```

This section imports necessary Python modules:

- `time` - Provides various time-related functions.
- `logging` - For logging messages for debugging and runtime event reporting.
- `collections` - Implements specialized container datatypes.
- `bluetooth` - Used for working with Bluetooth communications.
- `csv` - Implements classes to read from and write to CSV files.
- `datetime` - Supplies classes for working with dates and times.
- `os` - Provides a way of using operating system-dependent functionality.

```
# WiiboardSampling Parameters
#Sampling time
Minutes = 20
Seconds = 0
# Raw data directory
directory = 'BachelorsProject/C.O.MTrial/RawData'
```

This segment sets up sampling parameters for the Wiiboard:

- `Minutes` and `Seconds` set the duration for which the data should be sampled.
- `directory` defines the path where raw data will be saved.

```
N_SAMPLES = 100*(Seconds + 60* Minutes) #Roughly 10ms between samples
SESSION_TIME = Seconds + 60* Minutes #Session time in seconds
N_LOOP = 0
T_SLEEP = 2
BATTERY_MAX = 200.0
TOP_RIGHT = 0
BOTTOM_RIGHT = 1
TOP_LEFT = 2
BOTTOM_LEFT = 3
BLUETOOTH_NAME = "Nintendo RVL-WBC-01"
```

This section defines several constants and parameters:

- `N_SAMPLES` computes the total number of samples based on a rough sampling rate of 10ms.
- `SESSION_TIME` calculates the entire session's duration in seconds.
- `N_LOOP` initializes a loop counter.
- `T_SLEEP` defines a sleep duration.
- `BATTERY_MAX` sets a maximum battery value.
- `TOP_RIGHT`, `BOTTOM_RIGHT`, `TOP_LEFT`, and `BOTTOM_LEFT` are positional constants.
- `BLUETOOTH_NAME` specifies the Bluetooth device's name.

```
# Wiiboard Parameters
CONTINUOUS_REPORTING = b'\x04' #Read reporting mode
COMMAND_REPORTING = b'\x12' #Change reporting mode
COMMAND_LIGHT = b'\x11' #Change light mode
COMMAND_READ_REGISTER = b'\x17' #Setting register read location
COMMAND_REGISTER = b'\x16' #Setting register write location
COMMAND_REQUEST_STATUS = b'\x15' #Request board status
INPUT_STATUS = b'\x20' #Read board status
INPUT_READ_DATA = b'\x21' #Read calibration data
EXTENSION_8BYTES = b'\x32' #Specify 8 bytes of data
BUTTON_DOWN_MASK = 0x08 #Check if button is pressed
LED1_MASK = 0x10 #Check if LED is on
```

This section represents Wiiboard-specific Bluetooth communication commands which consist of both byte sequences and integers. For instance, `CONTINUOUS_REPORTING` with a value of `b'\x04'` prompts the Wiiboard to continuously relay data.

```
# initialize the logger
logger = logging.getLogger(__name__)
handler = logging.StreamHandler()  # or RotatingFileHandler
handler.setFormatter(logging.Formatter('[%(asctime)s][%(name)s][%(levelname)s]/
%(message)s'))
logger.addHandler(handler)
logger.setLevel(logging.DEBUG)  # or DEBUG
```

This initializes a logging utility to capture debug, information, or error messages. The messages are formatted to display the time, logger's name, and the log level.

```
b2i = lambda b: b if isinstance(b, int) else int.from_bytes(b, "big")
```

This lambda function, ‘b2i‘, converts a byte sequence to an integer, but if the input is already an integer, it simply returns the integer.

```
#Search for Wiiboard
def discover(duration=10, prefix=BLUETOOTH_NAME):
    logger.info("Scan Bluetooth devices for %i seconds...", duration)
    devices = bluetooth.discover_devices(duration=duration, lookup_names=True)
    logger.debug("Found devices: %s", str(devices))
    return [address for address, name in devices if name.startswith(prefix)]
#address = '00:24:44:58:C9:AF'
```

The ‘discover‘ function scans for Bluetooth devices, specifically looking for devices with names starting with the ‘BLUETOOTH$_NAME$‘$prefix$.

```
#Wiiboard actions

class Wiiboard:
    def __init__(self, address=None, csv_writer=None):
        #logger.debug("Entered _init_ method")
        self.csv_writer = csv_writer
        self.controlsocket = bluetooth.BluetoothSocket(bluetooth.L2CAP)
        self.receivesocket = bluetooth.BluetoothSocket(bluetooth.L2CAP)
        self.calibration = [[1e4] * 4] * 3
        self.calibration_requested = False
        self.light_state = False
        self.button_down = False
        self.battery = 0.0
        self.running = True
        if address is not None:
            self.connect(address)

    #Connect to Wiiboard
    def connect(self, address):
        #logger.debug("Entered connect method")
        logger.info("Connecting to %s", address)
        self.controlsocket.connect((address, 0x11))
        self.receivesocket.connect((address, 0x13))
        #logger.debug("Sending mass calibration request")
        # Set read from this particular wiiboard register
        self.send(COMMAND_READ_REGISTER, b"\x04\xA4\x00\x24\x00\x18")
        self.calibration_requested = True
        logger.info("Wait for calibration")
        #logger.debug("Connect to the balance extension, to read mass data")
        # Set write to this particular wiiboard register
```

```
        self.send(COMMAND_REGISTER, b"\x04\xA4\x00\x40\x00")
        #logger.debug("Request status")
        self.status
        self.light(0)
        #logger.debug("Exited connect method")

    #Prepare board to recieve data
    def send(self, *data):
        #logger.debug("Entered send method")
        self.controlsocket.send(b'\x52' + b''.join(data))
        #logger.debug("Exited send method")

    # Setting reporting mode to continuous
    def reporting(self, mode=CONTINUOUS_REPORTING, extension=EXTENSION_8BYTES):
        #logger.debug("Entered reporting method")
        self.send(COMMAND_REPORTING, mode, extension)

    # Switch on LED if on_off is True
    def light(self, on_off=True):
        #logger.debug("Entered light method")
        self.send(COMMAND_LIGHT, b'\x10' if on_off else b'\x00')

    # Request board status
    def status(self):
        #logger.debug("Entered status method")
        self.send(COMMAND_REQUEST_STATUS, b'\x00')


    #Weight in kg after calibration for specified corner
    def calc_mass(self, raw, pos):
        #logger.debug("Entered calc_mass method")
        # Calculates the Kilogram weight reading from raw data at position pos
        # calibration[0] is calibration values for 0kg
        # calibration[1] is calibration values for 17kg
        # calibration[2] is calibration values for 34kg
        if raw < self.calibration[0][pos]:
            return 0.0 #Below 1kg is not possible?
        elif raw < self.calibration[1][pos]:
            return 17 * ((raw - self.calibration[0][pos]) /
                         float((self.calibration[1][pos] -
                                self.calibration[0][pos])))
        else:  # if raw >= self.calibration[1][pos]:
            return 17 + 17 * ((raw - self.calibration[1][pos]) /
                              float((self.calibration[2][pos] -
                                     self.calibration[1][pos])))

    #Indicate if button is pressed
    def check_button(self, state):
        #logger.debug("Entered check_button method")
        if state == BUTTON_DOWN_MASK:
```

```
            if not self.button_down:
                self.button_down = True
                self.on_pressed()
        elif self.button_down:
            self.button_down = False
            self.on_released()

    #Converting weights to kg
    def get_mass(self, data):
        #logger.debug("Entered get_mass method")
        mass_dict = {
            'top_right': self.calc_mass(b2i(data[0:2]), TOP_RIGHT),
            'bottom_right': self.calc_mass(b2i(data[2:4]), BOTTOM_RIGHT),
            'top_left': self.calc_mass(b2i(data[4:6]), TOP_LEFT),
            'bottom_left': self.calc_mass(b2i(data[6:8]), BOTTOM_LEFT),
        }
        #print(b2i(data[0:2]))
        #print(b2i(data[2:4]))
        #print(b2i(data[4:6]))
        #print(b2i(data[6:8]))
        TR = mass_dict['top_right']
        BR = mass_dict['bottom_right']
        TL = mass_dict['top_left']
        BL = mass_dict['bottom_left']
        self.csv_writer.writerow([TR, BR, TL, BL])


        return mass_dict


    def loop(self):
        #logger.debug("Entered loop method")
        while self.running and self.receivesocket:
            data = self.receivesocket.recv(25)
            #logger.debug("socket.recv(25): %r", data)
            if len(data) < 2:
                continue
            input_type = data[1]
            #Checking communication is status
            if input_type == ord(INPUT_STATUS):
                self.battery = b2i(data[7:9]) / BATTERY_MAX
                # 0x12: on, 0x02: off/blink
                self.light_state = data[4] & LED1_MASK == LED1_MASK
                self.on_status()

             #Checking communication is factory calibration data
             #self.calibration = [[5944, 3314, 12024, 4221], [7688, 4994, 13794, 5967], [9
            elif input_type == ord(INPUT_READ_DATA):
                #logger.debug("Got calibration data")
                if self.calibration_requested:
```

```
                    #logger.debug("self.cal true")
                    length = b2i(data[4]) // 16 + 1 #Finding length encoded in first 4 bit
                    data = data[7:7 + length] #Data is now Data[7] to Data[7+length]
                    cal = lambda d: [b2i(d[j:j + 2]) for j in [0, 2, 4, 6]] #Lambda functi
                    #logger.debug("cal calculated")

                #Creating 3x4 nested list of calibration data.
                if length == 16:  # First packet of calibration data (16 bytes)
                    self.calibration = [cal(data[0:8]), cal(data[8:16]), [1e4] * 4] #Split
                elif length < 16:  # Second packet of calibration data inserted into 3rd l
                    self.calibration[2] = cal(data[0:8])
                    self.calibration_requested = False
                    self.on_calibrated()
                    #Checks if data is 8byte extension data.
            elif input_type == ord(EXTENSION_8BYTES):
                self.check_button(b2i(data[2:4]))  # Check button press
                self.on_mass(self.get_mass(data[4:12])) # Goes to on_mass method in Wiiboa

    #Read battery level
    def on_status(self):
        #logger.debug("Entered on_status method")
        self.reporting()  # Must set the reporting type after every status report
        logger.info("Status: battery: %.2f%% light: %s", self.battery * 100.0,
                    'on' if self.light_state else 'off')
        self.light(1)

    #Print factory calibration
    def on_calibrated(self):
        #logger.debug("Entered on_calibrated method")
        logger.info("Board calibrated: %s", str(self.calibration))
        print("\n \n MEASUREMENT IN PROGRESS")
        print("\n \n PLEASE STAND ON THE BOARD FOR FIRST TRIAL (CALIBRATION)")
        self.light(1)

    def on_mass(self, mass):
        #logger.debug("Entered on_mass method")
        logger.debug("New mass data: %s", str(mass))

    #Indicate button is pressed
    def on_pressed(self):
        #logger.debug("Entered on_pressed method")
        logger.info("Button pressed")

    #Indicate button is released
    def on_released(self):
        #logger.debug("Entered on_released method")
        logger.info("Button released")


    #Shutdown
```

```
def close(self):
    csv_file.close()
    #logger.debug("Entered close method")
    self.running = False
    if self.receivesocket:
        self.receivesocket.close()
    if self.controlsocket:
        self.controlsocket.close()

def __del__(self):
    #logger.debug("Entered __del__ method")
    self.close()

#### with statement ####
def __enter__(self):
    #logger.debug("Entered __enter__ method")
    return self

def __exit__(self, exc_type, exc_val, exc_tb):
    #logger.debug("Entered __exit__ method")
    self.close()
    return not exc_type  # re-raise exception if any
```

This class encapsulates the entire functionality related to the Wiiboard:

- The constructor initializes various attributes and establishes Bluetooth connections if an address is provided.
- The `connect` method establishes a connection to the Wiiboard and initializes it for communication.
- The `send` method sends data to the Wiiboard.
- The `reporting` method sets the board's reporting mode.
- The `light` method toggles the board's LED.
- The `status` method requests the board's status.
- The `calc_mass` method calculates weight in kilograms based on raw sensor readings.
- The `check_button` method checks if the Wiiboard's button has been pressed or released.
- The `get_mass` method computes the weight distribution on the board's four corners.
- The `loop` method is the main loop that constantly receives data from the board and processes it.
- The `on_status`, `on_calibrated`, `on_mass`, `on_pressed`, and `on_released` methods are event handlers for various board events.

- The `close` method closes all Bluetooth connections.
- The destructor ensures that connections are closed when the object is destroyed.
- The `__enter__` and `__exit__` methods make the class usable with the `with` statement in Python.

```
# Processing Nsamples

class WiiboardSampling(Wiiboard):
    def __init__(self, address=None, nsamples=N_SAMPLES):
        #logger.debug("Entered __initsample__ method")
        super().__init__(address)
        #Storing most recent samples up to nsamples
        self.samples = collections.deque([], nsamples)
        #logger.debug("Exited __initsample__ method")

    def on_mass(self, mass):
        #logger.debug("Entered on_masssample method")
        self.samples.append(mass)
        self.on_sample()

    def on_sample(self):
        logger.debug("Entered on_samplesample method")
        #time.sleep(0.01)
```

The `WiiboardSampling` class inherits from the `Wiiboard` class. It focuses on sampling mass data from the Wiiboard.

- The constructor initializes the sampling by setting up a deque to store the most recent samples up to `nsamples`.
- The `on_mass` method is overridden to append mass data to the samples and then call `on_sample`.
- The `on_sample` method is a placeholder which will be implemented in derived classes.

```
# Print sample data

class WiiboardPrint(WiiboardSampling):
    def __init__(self, address=None, nsamples=N_SAMPLES):
        #logger.debug("Entered _init__print method")
        super().__init__(address, nsamples)
        self.nloop = 0
        #logger.debug("Exited _init__print method")

#Print sample data
    def on_sample(self):
        #logger.debug("Entered on_sampleprint method")
        #If Nsamples are reached, print the time and average mass
        if len(self.samples) == N_SAMPLES:
            samples = [sum(sample.values()) for sample in self.samples]
            print(f"\n\n SESSION OF LENGTH {Minutes} MINUTES AND {Seconds} SECONDS \
            HAS BEEN COMPLETED \n ")
            self.samples.clear() #Clear samples
            self.status() # Stop the board from publishing mass data
            self.nloop += 1 #Increment loop counter
            if self.nloop > N_LOOP:
                return self.close()
            self.light(0)
            #Wait T_SLEEP seconds before starting again
            time.sleep(T_SLEEP)
```

The `WiiboardPrint` class inherits from `WiiboardSampling` and focuses on printing the sampled data.

- In its constructor, it initializes a loop counter.
- The `on_sample` method is overridden to check if the desired number of samples (`N_SAMPLES`) has been reached. Once reached, it calculates the average mass, prints it, clears the samples, and manages board status, light, and potential loop repetitions.

```
# Main
if __name__ == '__main__':
    import sys
    if '-d' in sys.argv:
        logger.setLevel(logging.DEBUG)
        sys.argv.remove('-d')
    if len(sys.argv) > 1:
        address = sys.argv[1]
    else:
        wiiboards = discover()
        logger.info("Found wiiboards: %s", str(wiiboards))
        if not wiiboards:
```

```
            raise Exception("Press the red sync button on the board")
        address = wiiboards[0]

    # Open CSV file
    current_datetime = datetime.now()
    date_string = current_datetime.strftime("%Y-%m-%d@%H-%M")
    filename = f"mass {date_string}.csv"
    file_path = os.path.join(directory, filename)
    with open(file_path, "w") as csv_file:
        csv_writer = csv.writer(csv_file)
        with WiiboardPrint(address) as wiiprint:
            wiiprint.csv_writer = csv_writer
            wiiprint.loop()
```

This is the main execution block of the script:

- It first checks for command-line arguments to set debug logging and to potentially get the Bluetooth address of the Wiiboard.
- If no address is provided, it uses the `discover` function to find available Wiiboards and selects the first one found.
- It then opens a CSV file with a timestamped filename to save the sampled data.
- Finally, it creates an instance of the `WiiboardPrint` class, sets its CSV writer, and initiates its main loop to start sampling and printing data.