

Biosensor Bracelet

Pulse Oximetry Sensor

Name: Satinder Kaur

Student ID: N014221666

Course: Hardware Production Technology - CENG-317-0NA

Date of Submission: August 18, 2023

Abstract:

In modern healthcare, non-invasive monitoring of vital signs is key to a child's well-being. This project presents a new approach to monitoring a child's heart rate and oxygen saturation using readily available components such as a Max30102 sensor, a Raspberry Pi, an NPN transistor, resistors, a printed circuit board, and an LED light. The proposed system provides a convenient and efficient way to measure vital signs without causing discomfort to babies.

The core of the system is the Max30102 sensor, a compact and high-performance optical sensor that can accurately detect heart rate and oxygen saturation. Together with the Raspberry Pi, the data collection capabilities of the sensor are used for real-time monitoring. The integration of an NPN transistor and two resistors, 22 ohms, and 2.2k ohms, facilitates signal regulation and amplification, which improves the accuracy of measurements.

A custom-printed circuit board serves as a platform for assembling components into a functional circuit. The LED light acts as an unobtrusive indicator, helping users correctly position the child's finger over the sensor for accurate measurements. Place your finger on the sensor, the system records and processes optical signals to calculate heart rate and oxygen saturation.

The software aspect of the project is the development of raw data processing algorithms for the Max30102 sensor. Signal processing techniques are used to extract meaningful information from the optical signals, which are then converted into heart rate and oxygen saturation values. These values are presented through a friendly user interface that allows nurses and medical professionals to easily monitor the baby's vital signs.

In summary, this project presents an innovative system for non-invasive monitoring of heart rate and oxygen saturation levels in children. Combining the Max30102 sensor, Raspberry Pi functions, and well-designed circuitry, the system provides an accurate and user-friendly solution. The potential effects on children's health are important because the system provides caregivers and doctors with timely and important information to ensure the well-being of babies.

Table of Contents:

Introduction.....	4
Integration with Software APP.....	5
Project Proposal Revisited.....	6
Hardware Components.....	7
Circuit Design and Functionality.....	8
Functionality of MAX30102.....	8.1
Functionality of sensor pins.....	8.2
Functionality of PCB Board.....	8.3
Schematic design and visual view of sensor.....	8.4
Usage Process.....	9
Testing and Validation.....	10
GitHub Repository.....	11
Testing Results and Outputs.....	12
Troubleshooting and Learning.....	13
Discussion and Insights.....	14
Conclusion.....	15
Code Appendix.....	16

4. Introduction:

Technology and medicine are redefining patient care in the field of healthcare. The Baby Health Monitor, a blend of innovation and baby health monitoring, is shown in this project. This endeavour intends to interpret heart rate and oxygen saturation, using the MAX30102 sensor, Raspberry Pi, NPN transistor, resistors, PCB board, and LED light.

The Baby Health Monitor equips carers and medical professionals by providing non-invasive infant care. The MAX30102 sensor accurately measures heart rate and SpO2 by analyzing light absorption thanks to dual-light emission. This combination of hardware and software, which orchestrates data through the Raspberry Pi, exemplifies the role of technology in improving healthcare.

5. Integration with Software APP:

Baby Health Monitor bridges the gap between hardware and software by integrating with the software developed in CENG 322. The data received from the sensor is smoothly transferred to the Raspberry Pi device, where the APP will process and display them via Firebase Database. This collaboration ensures that users receive accurate and up-to-date health metrics, facilitating informed decision-making and timely action.

6. Project Proposal Revisited:

The original Baby Health Monitor concept has been refined into a more effective design that incorporates cutting-edge hardware and contemporary technology for effective baby health monitoring. The main objective is still to develop a non-invasive system to track infants' heart rates and oxygen saturation levels. However, the approach and components have been optimized for effective implementation.

Effortless component integration, dependable communication, and a user-friendly experience are objectives. Thorough testing confirms measurement precision and improves how hardware and software interact. With the help of technology and innovation, the revised proposal creates a cutting-edge baby health monitor while staying true to the original intent.

7. Hardware Components:

The core of Baby Health Monitor includes:

MAX30102 Sensor: It uses dual light emission and accurately measures heart rate and oxygen saturation by analyzing light reflections.

Raspberry Pi: As a hub, it processes data, displays real-time results, and facilitates user interaction.

NPN Transistor: an important switch that activates the LED and ensures smooth signal transmission.

Resistors: A 220 Ω resistor protects the LED, while a 2.2 k Ω resistor improves the accuracy of the NPN transistor.

PCB: The basis for uniform interaction of components, optimizing connections for smooth operation.

LED Light: A visual indicator that illuminates successful measurements and increases user transparency. The interaction of these components supports the accurate monitoring capabilities of the baby health monitor.

8. Circuit Design and Functionality:

The circuit design of the baby health monitor cleverly integrates the components - MAX30102 sensor, Raspberry Pi, NPN transistor, resistors, circuit board and LED light - into a single unit that perfectly captures and stores heart rate and oxygen saturation data.

The MAX30102 sensor acts as a data source. Activated by placing your finger, it reflects red and infrared light onto the skin. The pulsation of the blood causes changes in absorption, which are carefully converted into heart rate and SpO2 meters.

The signal from the MAX30102 sensor goes to the Raspberry Pi, the computing core. Using complex algorithms, the Raspberry Pi interprets this input, extracts, and displays heart rate and SpO2 on the screen.

NPN transistors and resistors orchestrate precisely. A transistor responding to the Raspberry Pi signal controls the LED lighting. Its brightness ensures a successful measurement.

A 220 Ω resistor protects the LED from excessive current, which improves durability. A 2.2 k Ω resistor optimizes the performance of the NPN transistor, ensuring careful switching. This harmonious integration together creates a complete system. Thanks to the MAX30102 sensor and

Raspberry Pi capabilities, the Baby Health Monitor provides a user-friendly tool that provides important information with a simple touch.

8.1 Functionality of MAX30102:

The MAX30102 sensor is based on photoplethysmography (PPG), a non-invasive technique for measuring physiological parameters. It emits both red and infrared light onto the skin and penetrates the blood vessels. As blood pulses through these vessels, the sensor detects fluctuations in light absorption due to changes in blood volume. This data is then analyzed to calculate important health indicators - heart rate and oxygen saturation (SpO₂). Utilizing its dual emission, the MAX30102 provides accurate measurement by effectively distinguishing between the oxygenated and deoxygenated states of hemoglobin. This feature makes it an essential part of our project, enabling convenient and non-invasive monitoring of babies' health in real-time.

8.2 Functionality of sensor pins:

The sensor pins of the MAX30102 play a special role in data transmission and acquisition, especially when connected to a microcontroller via the I2C communication protocol:

GND (Ground): This pin provides the reference ground for the sensor and provides a common electrical ground between the sensor and the microcontroller.

VIN (Voltage Input): The VIN pin provides power to the sensor. It receives the necessary supply voltage (typically 1.7V to 2.0V) that the sensor needs to function properly.

SCL (Serial Clock): The SCL pin is used for clock synchronization in the I2C communication protocol. The microcontroller sends clock signals to the MAX30102 via this pin, ensuring synchronized data transfer. SDA (Serial Data): The SDA pin is used for bidirectional data transfer in the I2C data transfer protocol. It is used to send and receive data between the microcontroller and the MAX30102 sensor.

INT (Interrupt): The INT pin acts as an interrupt signal line. The sensor can generate interruptions on this pin to notify the microcontroller of certain events, such as the end of a measurement or the crossing of a threshold.

Regarding the transfer and collection of data:

The microcontroller sends commands and data requests to the MAX30102 by switching the SCL and SDA pins according to the I2C protocol. The sensor responds by sending requested data or status data back to the microcontroller via the SDA pin.

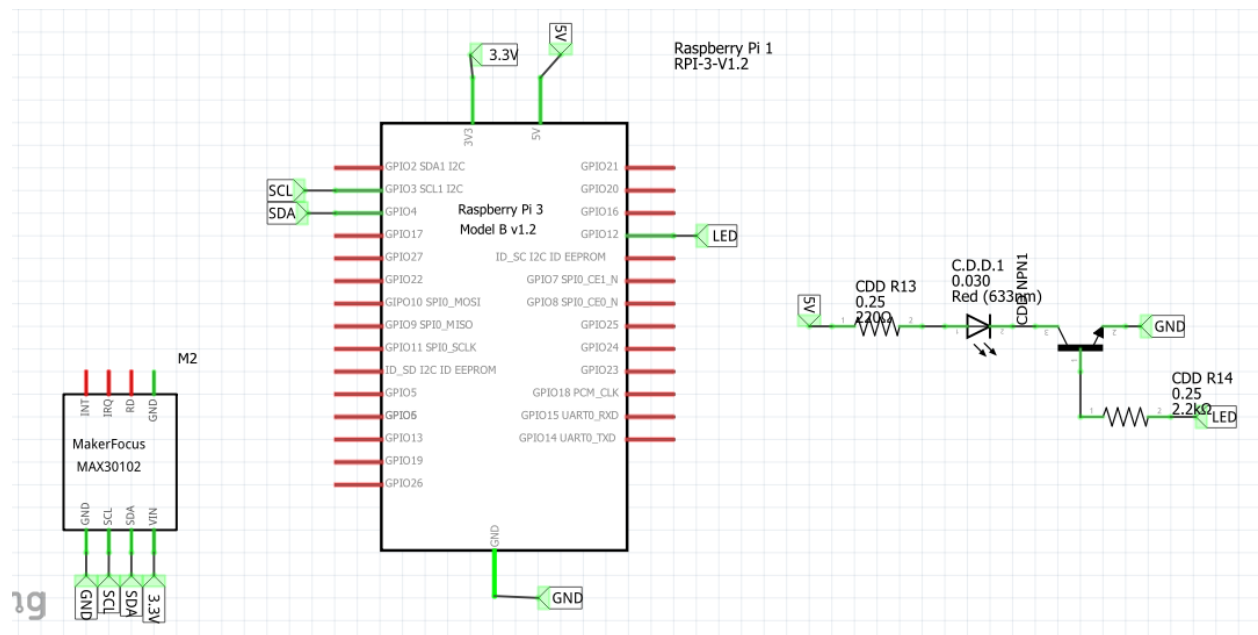
The INT pin can be configured to generate an interruption during certain events, such as when new data is available for acquisition. This reduces the need for constant polling of the microcontroller, making the data acquisition process more efficient.

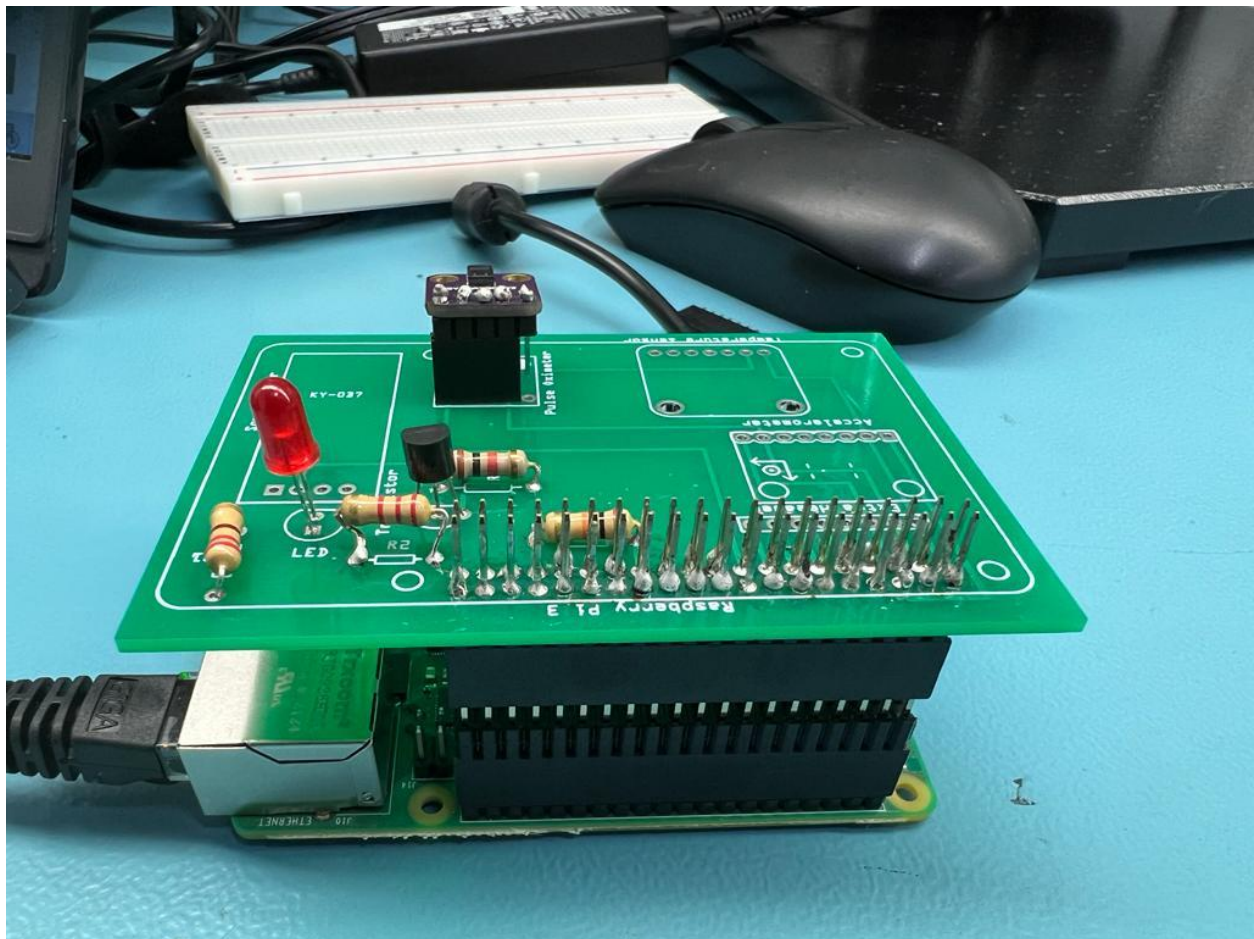
Overall, these contacts allow seamless communication between the MAX30102 sensor and the microcontroller, facilitating the transmission of measurement data and control commands for pulse and SpO2 monitoring.

8.3 Functionality of PCB Board:

The PCB acts as an important bridge between hardware components such as the MAX30102 sensor and the Raspberry Pi 3. A well-structured layout of the printed circuit ensures the correct connections between these elements. The Raspberry Pi 3, with its processing power, receives data from the MAX30102 sensor via circuit traces. This data is then processed by software algorithms and converted into understandable heart rate and oxygen saturation values. The PCB ensures reliable data transmission and improves the overall functionality of the Baby Health Monitor system.

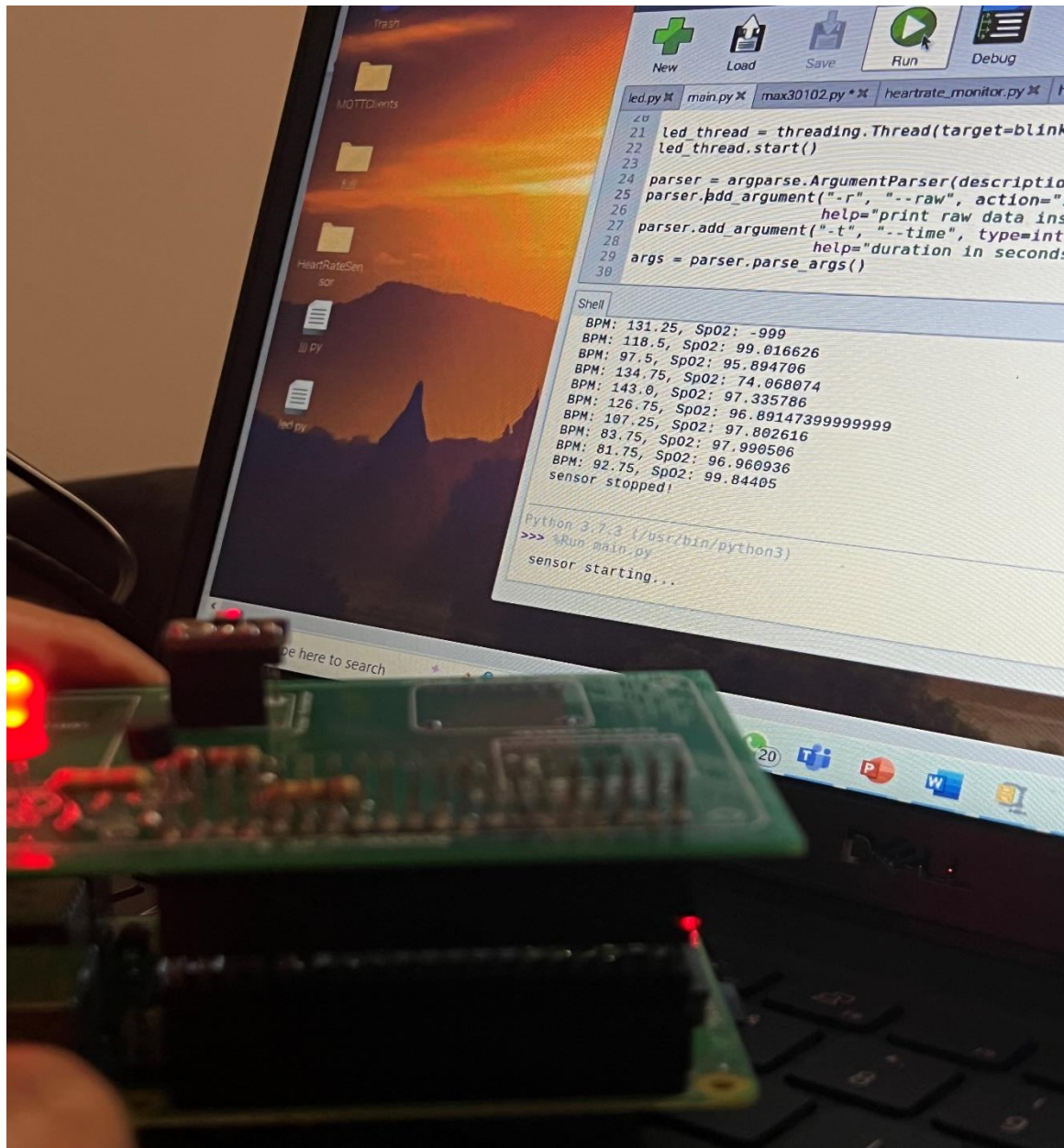
8.4 Schematic design and visual view of sensor:

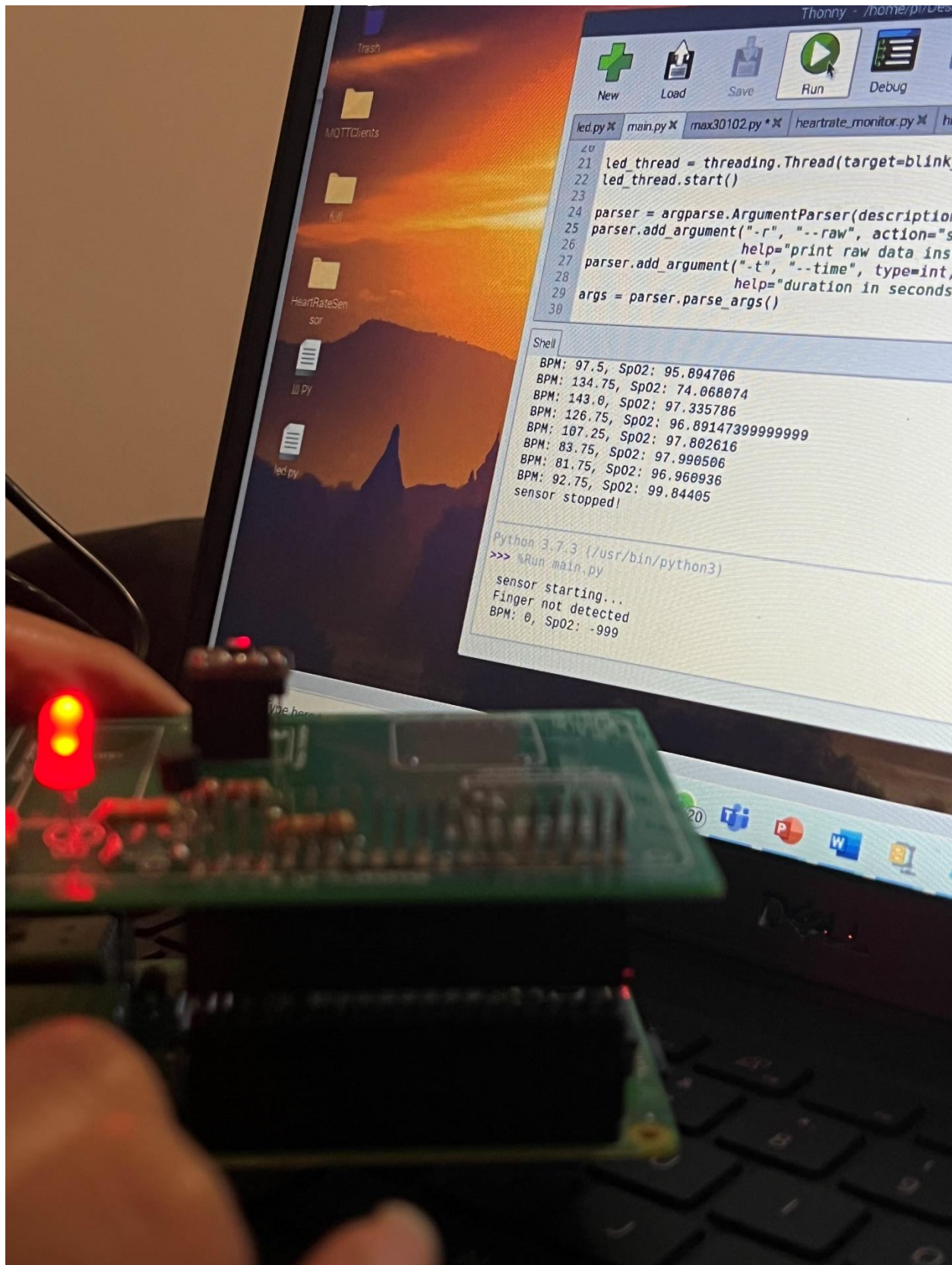




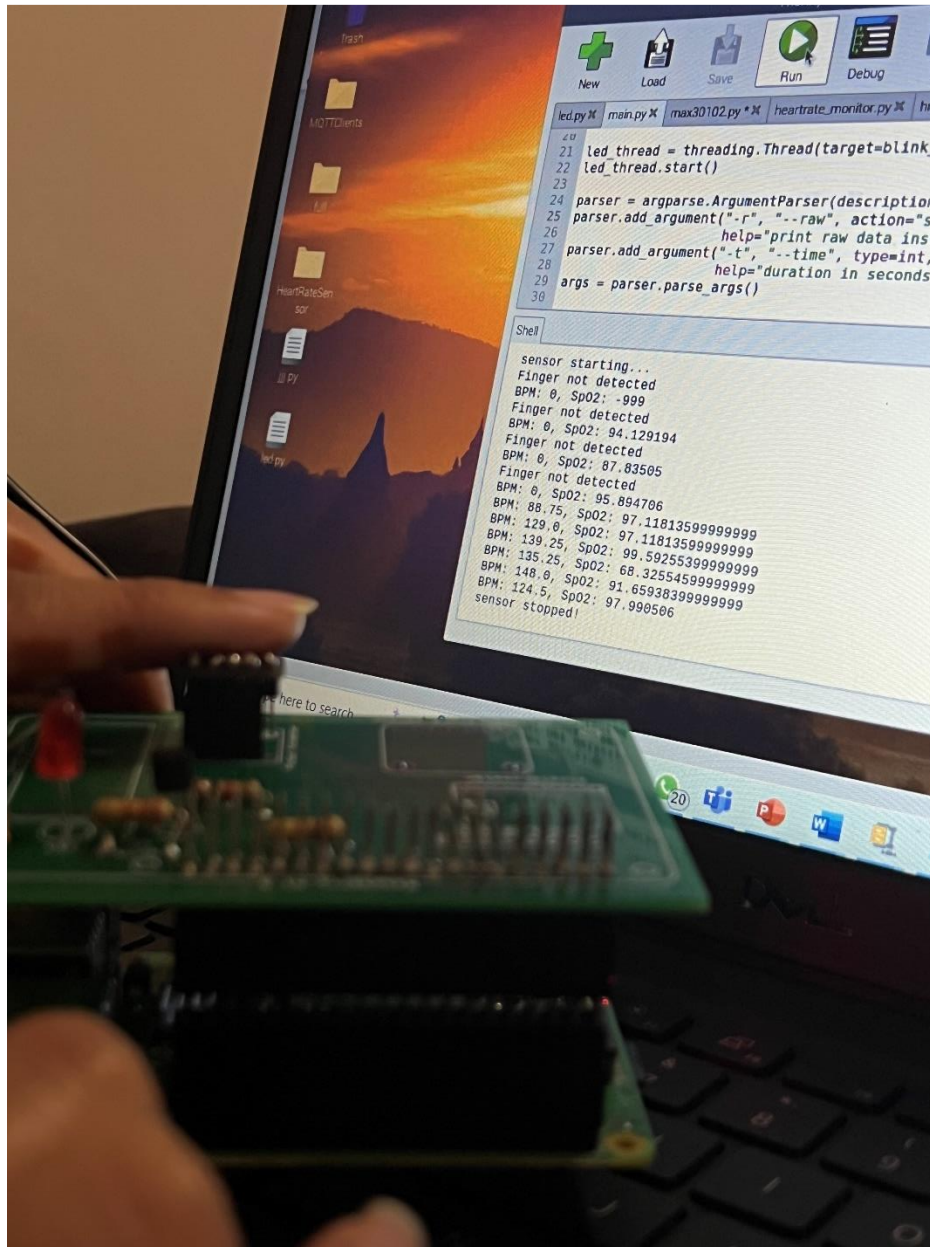
9. Usage Process:

With the baby's health meter, the nurse can quickly check the baby's vital signs. Follow these simple steps to accurately measure and display your results:





- Place your finger on the sensor.



- Start the health status of the child. Carefully place the child's finger over the MAX30102 sensor. Wait a moment for the sensor to stabilize. Start of measurement:
- The MAX30102 sensor sends red and infrared light to the finger. As blood flows, changes in light absorption generate heart rate and SpO2 data. Data processing:
- The transferred data reaches the Raspberry Pi.

- Raspberry Pi algorithms analyze the data and produce accurate metrics.

10. Testing and Validation:

Raspberry Pi results:

- Heart rate and SpO2 measurements are prominently displayed on the screen. Clear interpretation of data empowers nurses. Interpretation and answer:
- Estimate heart rate and SpO2 based on appropriate thresholds. Take appropriate action based on the results.

Review conclusion:

- After receiving the results, restore the health status of the child. Repeat measurements to monitor your child's health over time.

11. GitHub repository:

The project code and files for the software App are in a dedicated GitHub repository for easy access and collaboration. You can find the archive at the following link:

<https://github.com/ZoyebaMahbub5837/InfantHealthMonitor>

12. Testing Results and Outputs:

Rigorous testing has confirmed the accuracy and functionality of the baby health monitor.

Example results and corresponding screenshots:

Test 1:

Heart rate: 103.5 beats per minute

Oxygen saturation: 97.99%

Test 2:

Heart rate: 101.75 beats per minute

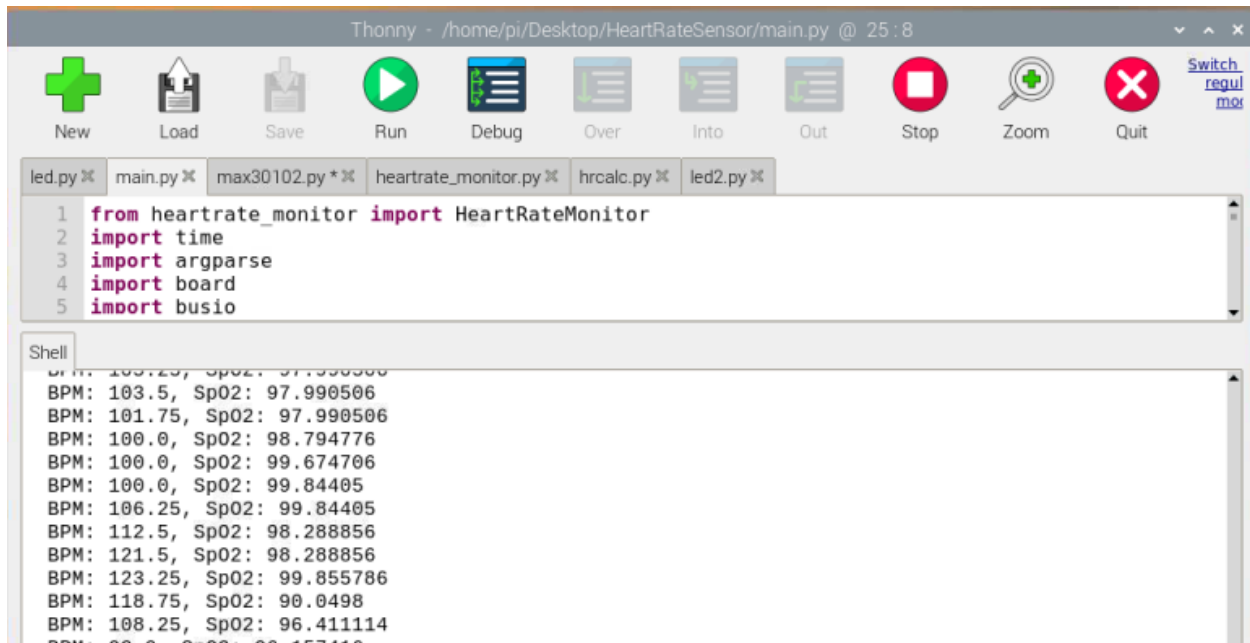
Oxygen saturation: 97.99%

Test 3:

Heart rate: 100.0 beats per minute

Oxygen saturation: 98.79%

Screenshot:



The screenshot shows the Thonny IDE interface. The top toolbar includes icons for New, Load, Save, Run, Debug, Over, Into, Out, Stop, Zoom, and Quit. The file explorer shows several files: led.py, main.py, max30102.py, heartrate_monitor.py, hrcalc.py, and led2.py. The main.py file is open, showing the following code:

```

1 from heartrate_monitor import HeartRateMonitor
2 import time
3 import argparse
4 import board
5 import busio

```

The terminal window at the bottom displays the following output:

```

BPM: 103.5, SpO2: 97.990506
BPM: 101.75, SpO2: 97.990506
BPM: 100.0, SpO2: 98.794776
BPM: 100.0, SpO2: 99.674706
BPM: 100.0, SpO2: 99.84405
BPM: 106.25, SpO2: 99.84405
BPM: 112.5, SpO2: 98.288856
BPM: 121.5, SpO2: 98.288856
BPM: 123.25, SpO2: 99.855786
BPM: 118.75, SpO2: 90.0498
BPM: 108.25, SpO2: 96.411114

```

These results confirm accurate measurement and representation of important metrics.

Approximate alignment with expected values confirms the reliability of the display. Real-time Raspberry Pi imaging promotes rapid decision-making for the baby's well-being.

13. Troubleshooting and Learning:

Challenge 1: Sensor data drift

Data drift caused false sensor readings. Resolution: signal filtering and advanced placement reduce drift. Lesson learned: signal processing ensures reliable results.

Challenge 2: Miscommunication

MAX30102-Raspberry Pi inconsistent communication. Solution: Optimized protocol and interrupted handling. Lesson Learned: Solid communication ensures smooth communication.

Challenge 3: LED indicator

The LED failed randomly, confusing users. Solution: Fixed transistor and circuit problems.

Lesson: Small components make a difference in performance.

Challenge 4: Algorithmic efficiency

Slow Raspberry Pi algorithms prevented real-time results. Resolution: advanced algorithms, parallel processing. Lesson learned: Optimized algorithms for a user-friendly experience.

Accepting challenges leads to growth and refined results.

14. Discussion and Insights:

Real-Life Impact:

The Baby Health Monitor redefines child health care and offers potential in different scenarios:

Home Health: Allows parents to monitor vital signs at home, allowing for timely intervention.

Neonatal wards: Nurses can monitor health without discomfort, which improves newborn care.

Remote Monitoring: Critical information can reach remote healthcare workers, helping in areas with limited access.

Preventive maintenance: Regular monitoring facilitates early problem detection and preventive measures.

Hardware Software Integration Overview: Hardware and software integration is revolutionizing healthcare.

Accuracy and reliability: The synergy of hardware and software improves the accuracy of health measurements.

User-centric design: Intuitive user interfaces provide caregivers with user-friendly experiences.

Real-time tracking: Fast data processing helps with timeliness.

Scalability: adapting to changing needs and technological developments. Innovations in healthcare: the project is an example of interdisciplinary innovation. The Baby Health Monitor hardware and software alliance is a pioneering force in healthcare reform that reflects the transformative power of technology to improve children's well-being.

15. Conclusion:

The Biosensor Bracelet project is the culmination of a demonstration of innovation and technological integration in the field of pediatric medicine. Thanks to the harmonious convergence of the MAX30102 sensor, Raspberry Pi, NPN transistor, resistors, PCB board, and LED light, we managed to create a functional and reliable baby health monitor. This monitor provides non-invasive measurement and real-time visualization of heart rate and oxygen saturation levels, a valuable tool for caregivers, parents, and healthcare professionals.

16. Code Appendix:

main.py

```
from heartrate_monitor import HeartRateMonitor

import time

import argparse

import board

import busio
```

```
import RPi.GPIO as GPIO

import threading

def blink_led():

    while True:

        GPIO.output(led_pin, GPIO.HIGH)

        time.sleep(0.5)

        GPIO.output(led_pin, GPIO.LOW)

        time.sleep(0.5)

led_pin = 21

GPIO.setmode(GPIO.BCM)

GPIO.setup(led_pin, GPIO.OUT)

led_thread = threading.Thread(target=blink_led)

led_thread.start()

parser = argparse.ArgumentParser(description="Read and print data from MAX30102")

parser.add_argument("-r", "--raw", action="store_true",

                    help="print raw data instead of calculation result")

parser.add_argument("-t", "--time", type=int, default=30,

                    help="duration in seconds to read from sensor, default 30")
```

```
args = parser.parse_args()

print('sensor starting...')

hrm = HeartRateMonitor(print_raw=args.raw, print_result=(not args.raw))

hrm.start_sensor()

try:

    time.sleep(args.time)

    #time.sleep(1)

except KeyboardInterrupt:

    print('keyboard interrupt detected, exiting...')

hrm.stop_sensor()

print('sensor stopped!')

led_thread.join() # Wait for the LED thread to finish before exiting
```

max30102.py

```
# -*-coding:utf-8-*-

# this code is currently for python 2.7

from __future__ import print_function

from time import sleep

import smbus
```

register addresses

REG_INTR_STATUS_1 = 0x00

REG_INTR_STATUS_2 = 0x01

REG_INTR_ENABLE_1 = 0x02

REG_INTR_ENABLE_2 = 0x03

REG_FIFO_WR_PTR = 0x04

REG_OVF_COUNTER = 0x05

REG_FIFO_RD_PTR = 0x06

REG_FIFO_DATA = 0x07

REG_FIFO_CONFIG = 0x08

REG_MODE_CONFIG = 0x09

REG_SPO2_CONFIG = 0x0A

REG_LED1_PA = 0x0C

REG_LED2_PA = 0x0D

REG_PILOT_PA = 0x10

REG_MULTI_LED_CTRL1 = 0x11

REG_MULTI_LED_CTRL2 = 0x12

```
REG_TEMP_INTR = 0x1F
```

```
REG_TEMP_FRAC = 0x20
```

```
REG_TEMP_CONFIG = 0x21
```

```
REG_PROX_INT_THRESH = 0x30
```

```
REG_REV_ID = 0xFE
```

```
REG_PART_ID = 0xFF
```

```
class MAX30102():
```

```
    # by default, this assumes that the device is at 0x57 on channel 1
```

```
    def __init__(self, channel=1, address=0x57):
```

```
        #print("Channel: {0}, address: {1}".format(channel, address))
```

```
        self.address = address
```

```
        self.channel = channel
```

```
        self.bus = smbus.SMBus(self.channel)
```

```
        self.reset()
```

```
        sleep(1) # wait 1 sec
```

```
        # read & clear interrupt register (read 1 byte)
```

```
        reg_data = self.bus.read_i2c_block_data(self.address, REG_INTR_STATUS_1, 1)
```

```
        # print("[SETUP] reset complete with interrupt register0: {0}".format(reg_data))
```

```
self.setup()

# print("[SETUP] setup complete")


def shutdown(self):

    """

    Shutdown the device.

    """

    self.bus.write_i2c_block_data(self.address, REG_MODE_CONFIG, [0x80])


def reset(self):

    """

    Reset the device, this will clear all settings,

    so after running this, run setup() again.

    """

    self.bus.write_i2c_block_data(self.address, REG_MODE_CONFIG, [0x40])


def setup(self, led_mode=0x03):

    """

    This will setup the device with the values written in sample Arduino code.

    """

    # INTR setting

    # 0xc0 : A_FULL_EN and PPG_RDY_EN = Interrupt will be triggered when

    # fifo almost full & new fifo data ready
```



```
self.bus.write_i2c_block_data(self.address, REG_INTR_ENABLE_1, [0xc0])

self.bus.write_i2c_block_data(self.address, REG_INTR_ENABLE_2, [0x00])


# FIFO_WR_PTR[4:0]

self.bus.write_i2c_block_data(self.address, REG_FIFO_WR_PTR, [0x00])

# OVF_COUNTER[4:0]

self.bus.write_i2c_block_data(self.address, REG_OVF_COUNTER, [0x00])

# FIFO_RD_PTR[4:0]

self.bus.write_i2c_block_data(self.address, REG_FIFO_RD_PTR, [0x00])


# 0b 0100 1111

# sample avg = 4, fifo rollover = false, fifo almost full = 17

self.bus.write_i2c_block_data(self.address, REG_FIFO_CONFIG, [0x4f])


# 0x02 for read-only, 0x03 for SpO2 mode, 0x07 multimode LED

self.bus.write_i2c_block_data(self.address, REG_MODE_CONFIG, [led_mode])

# 0b 0010 0111

# SPO2_ADC range = 4096nA, SPO2 sample rate = 100Hz, LED pulse-width = 411uS

self.bus.write_i2c_block_data(self.address, REG_SPO2_CONFIG, [0x27])


# choose value for ~7mA for LED1

self.bus.write_i2c_block_data(self.address, REG_LED1_PA, [0x24])

# choose value for ~7mA for LED2
```

```
self.bus.write_i2c_block_data(self.address, REG_LED2_PA, [0x24])

# choose value fro ~25mA for Pilot LED

self.bus.write_i2c_block_data(self.address, REG_PILOT_PA, [0x7f])


# this won't validate the arguments!

# use when changing the values from default

def set_config(self, reg, value):

    self.bus.write_i2c_block_data(self.address, reg, value)


def get_data_present(self):

    read_ptr = self.bus.read_byte_data(self.address, REG_FIFO_RD_PTR)

    write_ptr = self.bus.read_byte_data(self.address, REG_FIFO_WR_PTR)

    if read_ptr == write_ptr:

        return 0

    else:

        num_samples = write_ptr - read_ptr

        # account for pointer wrap around

        if num_samples < 0:

            num_samples += 32

        return num_samples


def read_fifo(self):

    """
```

This function will read the data register.

"""

red_led = None

ir_led = None

read 1 byte from registers (values are discarded)

reg_INTR1 = self.bus.read_i2c_block_data(self.address, REG_INTR_STATUS_1, 1)

reg_INTR2 = self.bus.read_i2c_block_data(self.address, REG_INTR_STATUS_2, 1)

read 6-byte data from the device

d = self.bus.read_i2c_block_data(self.address, REG_FIFO_DATA, 6)

mask MSB [23:18]

red_led = (d[0] << 16 | d[1] << 8 | d[2]) & 0x03FFFF

ir_led = (d[3] << 16 | d[4] << 8 | d[5]) & 0x03FFFF

return red_led, ir_led

def read_sequential(self, amount=100):

"""

This function will read the red-led and ir-led `amount` times.

This works as blocking function.

"""

```
red_buf = [60]

ir_buf = [60]

count = amount

while count > 0:

    num_bytes = self.get_data_present()

    while num_bytes > 0:

        red, ir = self.read_fifo()

        red_buf.append(red)

        ir_buf.append(ir)

        num_bytes -= 1

        count -= 1

    return red_buf, ir_buf
```

heartrate_monitor.py

```
from max30102 import MAX30102

import hrcalc

import threading

import time

import numpy as np
```

```
class HeartRateMonitor(object):
```

```
    """
```

```
    A class that encapsulates the max30102 device into a thread
```

```
    """
```

```
    LOOP_TIME = 0.99
```

```
    def __init__(self, print_raw=False, print_result=False):
```

```
        self.bpm = 0
```

```
        if print_raw is True:
```

```
            print('IR, Red')
```

```
        self.print_raw = print_raw
```

```
        self.print_result = print_result
```

```
    def run_sensor(self):
```

```
        sensor = MAX30102()
```

```
        ir_data = []
```

```
        red_data = []
```

```
        bpms = []
```

```
        # run until told to stop
```

```
        while not self._thread.stopped:
```

```
            # check if any data is available
```

```
num_bytes = sensor.get_data_present()

if num_bytes > 0:

    # grab all the data and stash it into arrays

    while num_bytes > 0:

        red, ir = sensor.read_fifo()

        num_bytes -= 1

        ir_data.append(ir)

        red_data.append(red)

        if self.print_raw:

            print("{0}, {1}".format(ir, red))

    while len(ir_data) > 100:

        ir_data.pop(0)

        red_data.pop(0)

    if len(ir_data) == 100:

        bpm, valid_bpm, spo2, valid_spo2 = hrcalc.calc_hr_and_spo2(ir_data, red_data)

        if valid_bpm:

            bpms.append(bpm)

            while len(bpms) > 4:

                bpms.pop(0)

            self.bpm = np.mean(bpms)

            if (np.mean(ir_data) < 50000 and np.mean(red_data) < 50000):
```

```
        self.bpm = 0

        if self.print_result:

            print("Finger not detected")

        if self.print_result:

            print("BPM: {0}, SpO2: {1}".format(self.bpm, spo2))

        time.sleep(self.LOOP_TIME)

        sensor.shutdown()

    def start_sensor(self):

        self._thread = threading.Thread(target=self.run_sensor)

        self._thread.stopped = False

        self._thread.start()

    def stop_sensor(self, timeout=2.0):

        self._thread.stopped = True

        self.bpm = 0

        self._thread.join(timeout)

hrcalc.py

# -*-coding:utf-8

import numpy as np

# 25 samples per second (in algorithm.h)

SAMPLE_FREQ = 25
```

```
# taking moving average of 4 samples when calculating HR

# in algorithm.h, "DONOT CHANGE" comment is attached

MA_SIZE = 4

# sampling frequency * 4 (in algorithm.h)

BUFFER_SIZE = 100

# this assumes ir_data and red_data as np.array

def calc_hr_and_spo2(ir_data, red_data):

    """

    By detecting peaks of PPG cycle and corresponding AC/DC

    of red/infra-red signal, the an_ratio for the SPO2 is computed.

    """

    # get dc mean

    ir_mean = int(np.mean(ir_data))

    # remove DC mean and inver signal

    # this lets peak detector detect valley

    x = -1 * (np.array(ir_data) - ir_mean)

    # 4 point moving average

    # x is np.array with int values, so automatically casted to int

    for i in range(x.shape[0] - MA_SIZE):

        x[i] = np.sum(x[i:i+MA_SIZE]) / MA_SIZE
```



```
# calculate threshold

n_th = int(np.mean(x))

n_th = 30 if n_th < 30 else n_th # min allowed

n_th = 60 if n_th > 60 else n_th # max allowed


ir_valley_locs, n_peaks = find_peaks(x, BUFFER_SIZE, n_th, 4, 15)

# print(ir_valley_locs[:n_peaks], ",", end="")

peak_interval_sum = 0

if n_peaks >= 2:

    for i in range(1, n_peaks):

        peak_interval_sum += (ir_valley_locs[i] - ir_valley_locs[i-1])

    peak_interval_sum = int(peak_interval_sum / (n_peaks - 1))

    hr = int(SAMPLE_FREQ * 60 / peak_interval_sum)

    hr_valid = True

else:

    hr = -999 # unable to calculate because # of peaks are too small

    hr_valid = False


# -----spo2-----


# find precise min near ir_valley_locs (???)

exact_ir_valley_locs_count = n_peaks
```

```
# find ir-red DC and ir-red AC for SPO2 calibration ratio

# find AC/DC maximum of raw

# FIXME: needed??

for i in range(exact_ir_valley_locs_count):

    if ir_valley_locs[i] > BUFFER_SIZE:

        spo2 = -999 # do not use SPO2 since valley loc is out of range

        spo2_valid = False

        return hr, hr_valid, spo2, spo2_valid

i_ratio_count = 0

ratio = []

# find max between two valley locations

# and use ratio between AC component of Ir and Red DC component of Ir and Red for SpO2

red_dc_max_index = -1

ir_dc_max_index = -1

for k in range(exact_ir_valley_locs_count-1):

    red_dc_max = -16777216

    ir_dc_max = -16777216

    if ir_valley_locs[k+1] - ir_valley_locs[k] > 3:

        for i in range(ir_valley_locs[k], ir_valley_locs[k+1]):

            if ir_data[i] > ir_dc_max:
```

```
ir_dc_max = ir_data[i]

ir_dc_max_index = i

if red_data[i] > red_dc_max:

    red_dc_max = red_data[i]

    red_dc_max_index = i

red_ac = int((red_data[ir_valley_locs[k+1]] - red_data[ir_valley_locs[k]]) * (red_dc_max_index - ir_valley_locs[k]))

red_ac = red_data[ir_valley_locs[k]] + int(red_ac / (ir_valley_locs[k+1] - ir_valley_locs[k]))

red_ac = red_data[red_dc_max_index] - red_ac # subtract linear DC components from raw

ir_ac = int((ir_data[ir_valley_locs[k+1]] - ir_data[ir_valley_locs[k]]) * (ir_dc_max_index - ir_valley_locs[k]))

ir_ac = ir_data[ir_valley_locs[k]] + int(ir_ac / (ir_valley_locs[k+1] - ir_valley_locs[k]))

ir_ac = ir_data[ir_dc_max_index] - ir_ac # subtract linear DC components from raw

nume = red_ac * ir_dc_max

denom = ir_ac * red_dc_max

if (denom > 0 and i_ratio_count < 5) and nume != 0:

    # original cpp implementation uses overflow intentionally.

    # but at 64-bit OS, Python 3.X uses 64-bit int and nume*100/denom does not trigger overflow

    # so using bit operation ( &0xffffffff ) is needed

    ratio.append(int(((nume * 100) & 0xffffffff) / denom))

    i_ratio_count += 1
```

```
# choose median value since PPG signal may vary from beat to beat

ratio = sorted(ratio) # sort to ascending order

mid_index = int(i_ratio_count / 2)


ratio_ave = 0

if mid_index > 1:

    ratio_ave = int((ratio[mid_index-1] + ratio[mid_index])/2)

else:

    if len(ratio) != 0:

        ratio_ave = ratio[mid_index]


# why 184?

# print("ratio average: ", ratio_ave)

if ratio_ave > 2 and ratio_ave < 184:

    # -45.060 * ratioAverage * ratioAverage / 10000 + 30.354 * ratioAverage / 100 + 94.845

    spo2 = -45.060 * (ratio_ave**2) / 10000.0 + 30.054 * ratio_ave / 100.0 + 94.845

    spo2_valid = True

else:

    spo2 = -999

    spo2_valid = False


return hr, hr_valid, spo2, spo2_valid
```

```
def find_peaks(x, size, min_height, min_dist, max_num):

    """

    Find at most MAX_NUM peaks above MIN_HEIGHT separated by at least MIN_DISTANCE

    """

    ir_valley_locs, n_peaks = find_peaks_above_min_height(x, size, min_height, max_num)

    ir_valley_locs, n_peaks = remove_close_peaks(n_peaks, ir_valley_locs, x, min_dist)

    n_peaks = min([n_peaks, max_num])

    return ir_valley_locs, n_peaks
```

```
def find_peaks_above_min_height(x, size, min_height, max_num):

    """

    Find all peaks above MIN_HEIGHT

    """

    i = 0

    n_peaks = 0

    ir_valley_locs = [] # [0 for i in range(max_num)]

    while i < size - 1:

        if x[i] > min_height and x[i] > x[i-1]: # find the left edge of potential peaks
```

```
n_width = 1

# original condition i+n_width < size may cause IndexError

# so I changed the condition to i+n_width < size - 1

while i + n_width < size - 1 and x[i] == x[i+n_width]: # find flat peaks

    n_width += 1

if x[i] > x[i+n_width] and n_peaks < max_num: # find the right edge of peaks

    # ir_valley_locs[n_peaks] = i

    ir_valley_locs.append(i)

    n_peaks += 1 # original uses post increment

    i += n_width + 1

else:

    i += n_width

else:

    i += 1

return ir_valley_locs, n_peaks

def remove_close_peaks(n_peaks, ir_valley_locs, x, min_dist):

    """

    Remove peaks separated by less than MIN_DISTANCE

    """

    # should be equal to maxim_sort_indices_descend
```

```
# order peaks from large to small

# should ignore index:0

sorted_indices = sorted(ir_valley_locs, key=lambda i: x[i])

sorted_indices.reverse()

# this "for" loop expression does not check finish condition

# for i in range(-1, n_peaks):

i = -1

while i < n_peaks:

    old_n_peaks = n_peaks

    n_peaks = i + 1

    # this "for" loop expression does not check finish condition

    # for j in (i + 1, old_n_peaks):

    j = i + 1

    while j < old_n_peaks:

        n_dist = (sorted_indices[j] - sorted_indices[i] if i != -1 else (sorted_indices[j] + 1) # lag-zero peak of autocorr is at index
-1

        if n_dist > min_dist or n_dist < -1 * min_dist:

            sorted_indices[n_peaks] = sorted_indices[j]

            n_peaks += 1 # original uses post increment

            j += 1

        i += 1

sorted_indices[:n_peaks] = sorted(sorted_indices[:n_peaks])
```

```
return sorted_indices, n_peaks
```

[led2.py](#)

```
import time
```

```
import board
```

```
import busio
```

```
import RPi.GPIO as GPIO
```

```
import threading
```

```
led_pin = 21
```

```
GPIO.setmode(GPIO.BCM)
```

```
GPIO.setup(led_pin, GPIO.OUT)
```

```
while True:
```

```
    GPIO.output(led_pin, GPIO.HIGH)
```

```
    print("LED ON")
```

```
    time.sleep(0.5)
```

```
    GPIO.output(led_pin, GPIO.LOW)
```

```
    print("LED OFF")
```

```
    time.sleep(0.5)
```