Design & Analysis of Wearable Devices for Vital Parameter Monitoring

by

Sangeetha D (22B3310)

Under the Guidance of
Prof. Nirmal Punjabi

Koita Centre For Digital Health (KCDH)
Indian Institute Of Technology, Bombay

Mumbai 400 076

November 29, 2024

Contents

1 Introduction **2**
- 1.1 Aims and objectives . . . 2

2 Literature Review **3**

3 Technical Details **5**
- 3.1 Hardware Overview . . . 5
- 3.2 Software Overview . . . 6

4 Summary Of Progress **8**
- 4.1 Communication Protocols . . . 8
  - 4.1.1 SPI Communication Protocol . . . 8
  - 4.1.2 UART Communication Protocol . . . 9
  - 4.1.3 I2C Communication Protocol: . . . 9
- 4.2 ESP 32-S3 . . . 10
- 4.3 Configuration of TFT Display and Display of ESP32 Internal Temperature 11
  - 4.3.1 Connecting ESP 32-S3 with TFT screen . . . 11
  - 4.3.2 Code . . . 12
  - 4.3.3 Code flow for Internal Temperature plotting algorithm(Loop starts by resetting the screen to black) . . . 14
  - 4.3.4 Results . . . 15
- 4.4 MAX30102 Sensor Setup . . . 15
  - 4.4.1 Hardware Integration . . . 15
  - 4.4.2 Software Configuration . . . 15
  - 4.4.4 Data Processing & Results . . . 19
  - 4.4.5 Troubleshooting . . . 20
  - 4.4.6 Results . . . 22
- 4.5 Alternative Algorithm to Find Heartbeat Values . . . 23

5 Challenges Faced **29**
- 5.1 Challenges Faced . . . 29

6 Key Learnings & Conclusion **31**
- 6.1 Conclusion . . . 32

7 References **33**

8 Appendix **34**

Chapter 1

Introduction

This project focuses on the practical application of embedded systems through the integration of the ESP32-S3 microcontroller with components such as 240x320 TFT display and MAX30102 heartbeat sensor. The initial phase of this project involved studying communication protocols like I2C and SPI, and UART which are commonly used for communication. Later,ESP32-S3 was used to access and display its internal temperature sensor readings using TFT display and the next phase involved integrating the MAX30102 pulse sensor to measure and display heartbeat data on the TFT screen, which required an understanding of the sensor's functionality.

1.1 Aims and objectives

Aims:

- To understand and implement communication protocols like I2C, SPI and UART
- To develop a user-friendly interface that displays real-time data (e.g., internal ESP32-S3 temperature, heart beat values) on a TFT screen.
- To explore the capabilities of the ESP32-S3 in integrating sensors and displays for real-time monitoring applications.

Objectives:

- Understand the functioning and applications of ESP 32-S3
- Set up the ESP32-S3 with a TFT screen using SPI communication.
- Display the internal temperature of the ESP32-S3 in a visually clear and readable format.
- Ensure accurate and smooth data visualization without flickering or lags.
- Understand the functioning of MAX 30102 sensor and it's applications like heartbeat detection
- Implement heartbeat detection algorithm to find out real time heartbeat values
- Plot the heartbeat values on a TFT screen in a readable format

Chapter 2

Literature Review

Communication Protocols

This project uses various communication protocols to interface between hardware components like the ESP32-S3 microcontroller, MAX30102 pulse oximeter sensor, and TFT display. The protocols studied in this project are UART, SPI, and I2C. Below is a simplified review of these protocols, highligting their roles in embedded systems and their application in the current project.

1. UART (Universal Asynchronous Receiver/Transmitter)

Purpose: UART is a widely used protocol for serial communication between microcontrollers and peripheral devices, such as sensors or other microcontrollers. These are the pins used in UART protocol.

- TX (Transmit): This pin is used to send data from the microcontroller to another device.
- RX (Receive): This pin is used to receive data from a device to the microcontroller.

And The "HardwareSerial" library or "Serial" object in the Arduino IDE (for ESP32) is typically used to manage UART communication.

2. SPI (Serial Peripheral Interface)

Purpose: SPI is a high-speed, full-duplex communication protocol used to transfer data between microcontrollers and peripherals like displays, sensors, and memory chips.
Usage in the project: The project uses SPI to communicate with the TFT display, utilizing a library like TFT eSPI to control the graphical output on the display. Requires at least four pins:

- MOSI (Master Out Slave In)
- MISO (Master In Slave Out)
- SCLK (Serial Clock)
- CS (Chip Select)
  These pins are essential for data transfer between the master (ESP32) and the slave devices (such as sensors or displays like TFT screen)

3. I2C (Inter-Integrated Circuit)

Purpose: I2C is a synchronous, multi-master, multi-slave protocol used for communication between multiple devices using just two wires:

- Serial data (SDA)
- Serial clock (SCL)

Usage in the project: I2C is used to transfer data between the microcontroller and the connected device such as MAX 30102 heartbeat sensor, ADXL 345 Accelermeter

4. Choosing the Right Protocol for the Project

The decision to use SPI for the TFT display and I2C for the MAX30102 sensor is based on the need for efficient data transfer:
TFT Display: SPI is preferred due to its higher data rate, which is crucial for displaying real-time data like heart rate.
MAX30102 Sensor: I2C is suitable because of its ability to connect multiple sensors with minimal wiring and the moderate data transfer requirements for sensor readings. These protocols allow the project to balance speed, wiring complexity, and expandability.

5. Challenges in Implementing Communication Protocols

Signal Integrity: Ensuring that data is transmitted without loss or corruption over the communication lines is critical, especially for long-distance communications or high-speed operations.
Timing and Synchronization: Managing the timing for data transfers and ensuring that the microcontroller correctly interprets sensor data without conflicts in multi-device setups (e.g., using I2C with multiple sensors).
Power Consumption: Each protocol has its power consumption characteristics, for eg. I2C consumes the lowest power among all 3 and SPI consumes the highest power and this can be important for battery-operated systems where efficiency is required.

Understanding PPG for Measuring Heart Rate and Blood Oxygen Levels

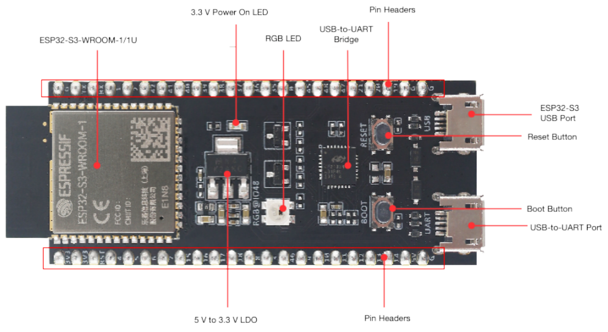
Photoplethysmography (PPG) is a non-invasive optical technique used to monitor physiological parameters like heart rate and blood oxygen levels. PPG measures changes in light absorption or reflection from blood vessels as they expand and contract with each heartbeat. This technique is beneficial because it allows continuous monitoring without the need for invasive procedures. PPG is widely used for both clinical and personal health applications, including heart rate tracking and oxygen saturation measurement. There are other applications of PPG, such as assessing blood volume and circulation efficiency, which play an essential role in diagnosing conditions like heart disease and respiratory problems. Studies have shown that PPG can be used to monitor the effectiveness of treatments for these conditions, making it a valuable tool in medical diagnostics and health monitoring. And PPG sensor can also be used in wearable devices One key feature of the MAX30102 is its compact design, which allows it to be used in small, portable devices. The sensor can accurately measure heart rate and blood oxygen levels by detecting the reflection of light from the bloodstream.It also supports,low power consumption and high signal-to-noise ratios, ensuring that it works effectively even in varying lighting conditions. The following diagram shows important parts of ESP 32-S3

Chapter 3

Technical Details

3.1 Hardware Overview

ESP 32-S3



The ESP32-S3 is a powerful microcontroller designed for edge computing applications, offering robust features such as Xtensa® 32-bit LX7 dual-core microprocessor, 2.4 GHz Wi-Fi (IEEE 802.11b/g/n) and Bluetooth® 5 (LE), and 45 GPIO pins. It has fine-resolution power control through a selection of clock frequency, duty cycle, Wi-Fi operating modes,and individual power control of internal components makes it ideal for IoT and embedded systems requiring real-time data processing.

In healthcare monitoring, the ESP32-S3 plays a critical role in interfacing with sensors like the MAX30102 for heartbeat detection and onboard sensors for temperature measurement. For heartbeat detection, the microcontroller processes PPG (photoplethysmogram) signals from the MAX30102 by I2C communication protocol. It also enables real-time calculation of beats per minute (BPM).

TFT ILI9341 Display

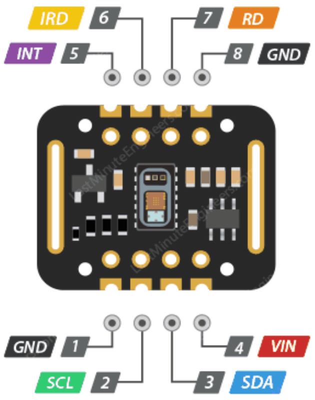
ILI9341 is a 262,144-color single-chip SOC (System-On-Chip) driver for TFT liquid crystal display with resolution of 240RGBx320 dots, comprising a 720-channel source driver, a 320-channel gate driver, 172,800 bytes GRAM for graphic display data of 240RGBx320 dots, and power supply circuit. ILI9341 supports 3-/4-line serial peripheral interface (SPI). The moving picture area can be specified in internal GRAM by window address function. The specified window area can be updated selectively, so that moving picture can be displayed simultaneously, independent of still picture area.

ILI9341 can operate with 1.65V 3.3V I/O interface voltage and an incorporated voltage follower circuit to generate voltage levels for driving an LCD. ILI9341 supports

Figure 3.1: ILI9341 2.4 inch Display

full color, 8-color display mode and sleep mode for precise power control by software and these features make the ILI9341 an ideal LCD driver for medium or small size portable products where long battery life is a major concern.

MAX 30102 Heartbeat and SpO2 sensor



The MAX30102 is a compact pulse oximeter and heart rate sensor ideal for wearable health devices like smartwatches. With ultra-low power consumption—600 A in measurement mode and 0.7 A in standby—it supports real-time monitoring while preserving battery life. Its integrated ambient light cancellation ensures accurate readings in various lighting conditions.

Using red and infrared LEDs and a photodiode, the sensor measures heart rate and blood oxygen levels (SpO2) by analyzing light absorption in blood capillary . Its 32-sample FIFO buffer improves efficiency by storing data before sending it via I2C. An on-chip temperature sensor enhances its versatility, making the MAX30102 a valuable tool for IoT healthcare and portable healthcare

3.2 Software Overview

Software Overview

For this project, I have used 5 softwares namely, Arduino IDE, PuTTy, Jupyter notebook, Excel and Visual Studio Code

Arduino IDE

- Used for coding, testing, and debugging sensor functionalities like MAX30105, TFT screen, and ADXL345.
- Enabled rapid prototyping with its extensive library support and built-in serial monitor.

Visual Studio Code:

- Modified key parameters in UserSetup.h for TFT screen configurations, including resolution, pins, and axis layout.
- VS code Simplified quick edits as it hasfeatures like syntax highlighting and structured code navigation.

PuTTY:

- Logged real-time heartbeat values collected via serial communication from the ESP32
- Exported data to CSV for further analysis in Excel.

Excel:

- Manually calculated heartbeat values by observing peaks in the data.
- Created graphs for analysis and visualization.

Jupyter Notebook:

- Used Python's scipy.signal.find peaks function to automate peak detection in the Filtered IR signal.
- Visualized detected peaks and analyzed heartbeat trends with matplotlib.

Chapter 4

Summary Of Progress

4.1 Communication Protocols

4.1.1 SPI Communication Protocol

SPI0 and SPI1: High-Speed Memory Access SPI0 and SPI1 are optimized for high-speed communication with internal and external memory, such as flash memory and PSRAM. They support multiple SPI modes (Single, Dual, Quad, Octal) and clock frequencies up to 120 MHz in Octal SPI mode. These interfaces are ideal for managing large datasets, ensuring efficient storage and retrieval.

SPI2: Flexible Sensor Communication SPI2 serves as a flexible master-slave interface with a clock frequency of up to 80 MHz, supporting full-duplex communication. It is ideal for connecting to multiple sensors like heart rate and temperature monitors, supporting up to six SPI slave devices for real-time health monitoring and multi-sensor integration in wearable systems.

SPI3: Additional Peripheral Support SPI3 offers similar features to SPI2 but supports fewer SPI-CS (Chip Select) pins, making it suited for handling fewer peripherals. Operating at up to 80 MHz, it provides an additional communication channel for connecting sensors or other peripherals, enhancing the scalability and functionality of wearable health monitoring devices.

Key Features

High-Speed Data Transfer: SPI supports clock frequencies up to 120 MHz (Octal mode) for fast communication.
Full-Duplex Communication: Devices can send and receive data simultaneously.
Master-Slave Architecture: One master can control multiple slave devices using separate chip select (CS) pins.
Multiple SPI Modes: Supports Single, Dual, Quad, and Octal modes for different data transfer needs.
Low Power Consumption: Ideal for battery-operated devices like wearables.
Scalability: Easily supports multiple sensors or peripherals in complex systems.

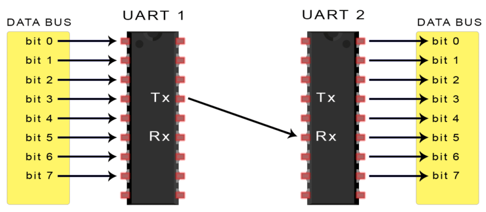
Limitations

Pin Requirement: Requires more pins (MOSI, MISO, SCLK, CS) compared to protocols like I2C.

Short-Range Communication: Not suitable for long distances due to signal degradation.
Master-Slave Limitation: Communication is controlled by the master, which can limit flexibility in some applications.
No Error Checking: Unlike UART, SPI does not include built-in error detection mechanisms. Challenges
Complex Wiring: Adding multiple slave devices increases the complexity of wiring and pin management.
Resource Allocation: Sharing SPI across multiple peripherals may need careful planning to avoid performance bottlenecks
Clock Synchronization: All devices must synchronize with the clock signal for proper communication.

4.1.2 UART Communication Protocol

UART (Universal Asynchronous Receiver/Transmitter) is a hardware component or circuit used for serial communication between two devices. It is commonly found as an embedded feature in microcontrollers or as a standalone IC. Unlike communication protocols like SPI and I2C, UART focuses on transmitting and receiving data through two dedicated pins: TX (transmit) and RX (receive). The transmitting UART converts parallel data into serial form and sends it bit by bit through the TX pin, while the receiving UART converts the serial data back to parallel using the RX pin.



Key Features

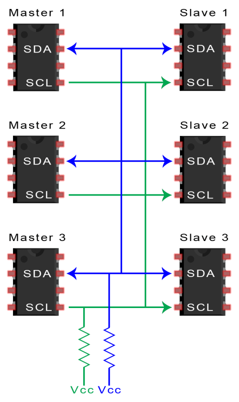
Two-Wire Communication: UART requires only two wires, TX and RX, for data exchange. Asynchronous Data Transfer: It does not use a clock signal, instead relying on ”start” and ”stop” bits to mark the beginning and end of data transmission.
Baud Rate Agreement: Both devices must operate at the same speed, measured in bits per second (bps), to ensure correct data interpretation. Parallel to Serial Conversion: The transmitting UART converts parallel data into a serial stream, and the receiving UART converts it back.

Challenges and Limitations

Asynchronous Nature: Without a clock, accurate communication depends heavily on the baud rate agreement between devices
Limited Range: UART is not ideal for long-distance communication due to signal degradation. Speed Restrictions: Compared to protocols like SPI, UART is slower and better suited for low-speed applications

4.1.3 I2C Communication Protocol:

The Inter-Integrated Circuit (I2C) communication protocol(I2C) allows multiple devices to share a common data bus, making it ideal for applications that require interconnection between a microcontroller and multiple peripherals.



This protocol combines the best features of Serial Peripheral Interface (SPI) and Universal Asynchronous Receiver-Transmitter (UART) by enabling multiple devices to communicate using only two wires: SDA (Serial Data): For bidirectional data transmission. SCL (Serial Clock): For synchronizing data transfer. I2C operates synchronously, meaning that communication is synchronized to a clock signal controlled by a master device. The master can initiate communication with one or more slave devices using unique addresses for each slave, ensuring efficient data transfer without additional slave-select lines.

Key Features of I2C

Addressing: Each slave device is assigned a unique 7-bit or 10-bit address, enabling communication between the master and specific slaves.
Acknowledgment (ACK/NACK): Ensures reliable data transfer by confirming the successful receipt of each data frame.
Start/Stop Conditions: Provides clear indications for initiating or terminating data communication. schematic

4.2 ESP 32-S3

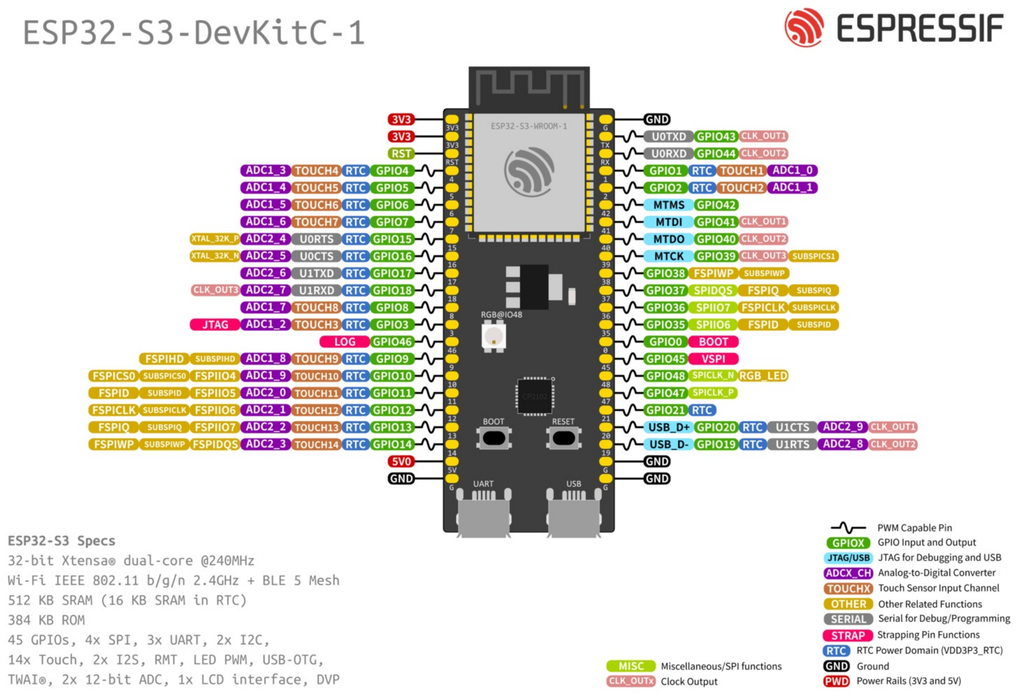
The ESP32-S3 offers a wide range of functions that make it a great option for embedded systems, especially in IoT and healthcare monitoring uses. It is based on the Xtensa® 32-bit LX7 dual-core microprocessor, providing strong processing capabilities reaching speeds of 240 MHz. The microcontroller is compatible with Wi-Fi (2.4 GHz, IEEE 802.11b/g/n) and Bluetooth® 5 (LE), providing effective wireless communication and making it perfect for connected applications. Some important characteristics of the ESP32-S3 are:

- Dual-Core Xtensa® 32-bit LX7 microprocessor offers substantial processing capabilities for managing intricate computations and real-time operations.

- 2.4 GHz Wi-Fi and Bluetooth® 5 (LE) provide rapid and dependable wireless connectivity for IoT purposes, while Bluetooth Low Energy (LE) is compatible with a range of devices.

- 45 GPIO pins allow for versatile hardware connections, enabling the interfacing of a variety of sensors and peripherals.

- Low power consumption is achieved through precise power management, featuring various power modes such as deep sleep, ideal for gadgets reliant on batteries, making it very efficient.
- Incorporated hardware acceleration: This involves cryptographic hardware integrated within the system to ensure secure communication, making it ideal for data-sensitive tasks.
- Supports I2C, SPI, UART communication, thereby providing flexibility for connecting sensors and peripherals in different scenarios.



ESP32-S3-DevKitC-1
ESPRESSIF
ESP32-S3-WROOM-1
RGB@IO48
BOOT
RESET
UART
USB
PWM Capable Pin
GPIOX GPIO Input and Output
JTAG/USB JTAG for Debugging and USB
ADCX_CH Analog-to-Digital Converter
TOUCHX Touch Sensor Input Channel
OTHER Other Related Functions
SERIAL Serial for Debug/Programming
STRAP Strapping Pin Functions
RTC RTC Power Domain (VDD3P3_RTC)
GND Ground
PWD Power Rails (3V3 and 5V)
MISC Miscellaneous/SPI functions
CLK_OUTX Clock Output

ESP32-S3 Specs
32-bit Xtensa® dual-core @240MHz
Wi-Fi IEEE 802.11 b/g/n 2.4GHz + BLE 5 Mesh
512 KB SRAM (16 KB SRAM in RTC)
384 KB ROM
45 GPIOs, 4x SPI, 3x UART, 2x I2C,
14x Touch, 2x I2S, RMT, LED PWM, USB-OTG,
TWAI®, 2x 12-bit ADC, 1x LCD interface, DVP

ESP32-S3 is particularly well-suited for the project due to its high computational power(upto 240MHz), energy efficiency(Power consumption as low as 7 µA) and flexible communication interfaces. In this project, ESP 32-S3 has been used to communicate with sensors and TFT display A 240x320 TFT display is connected to the ESP32-S3, which allows for the graphical representation of heart rate data. The display also serves as an interface to show the current readings and trends, which is essential for users to monitor their health in real time.

4.3 Configuration of TFT Display and Display of ESP32 Internal Temperature

The following is the procedure to monitor and display temperature readings from an ESP32 using a TFT screen

4.3.1 Connecting ESP 32-S3 with TFT screen

- The TFT display is connected using SPI communication.
- Pin configuration for SPI interface:

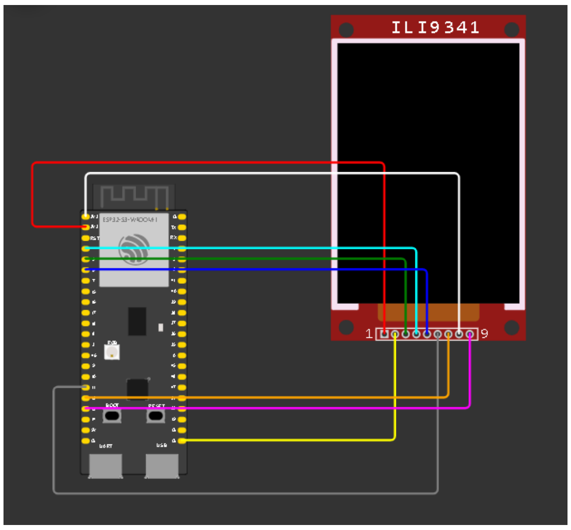


Figure 4.1: Schematic of ESP 32-S3 with TFT display

  - MOSI (Master Out Slave In) on pin 11
  - SCK (Serial Clock) on pin 12
  - CS (Chip Select) on pin 5
  - DC (Data/Command) on pin 6
  - RST (Reset) pin connected to the reset pin of the ESP32-S3.

- Power supply: The TFT display is powered using a 3.3V source from the ESP32-S3.
- Adafruit-GFX and Adafruit- ILI9341 libraries are used to control the TFT display.
- Few example codes are implemented to check the functioning of TFT display
- Later real-time data (temperature values here) are displayed on the TFT screen which are updated dynamically

4.3.2 Code

Step 1: Library Inclusion and Initialization

```
// Include the TFT_eSPI library
#include <TFT_eSPI.h>

// Create an instance of the TFT_eSPI library
TFT_eSPI tft = TFT_eSPI();
```

Step 2: Reading Internal Temperature

The `temperatureRead()` function is used to access the ESP32's internal temperature sensor, which returns the value as a floating-point number.

```
// Function to read the internal temperature sensor
float readInternalTemperature() {
  return temperatureRead();
}
```

Step 3: Setup Function for Initialization

In the setup function:

- Serial communication is initialized to log temperature values for debugging.
- The TFT display is initialized and configured in landscape orientation with a black background for better visibility.

```
void setup() {
  Serial.begin(115200);  // Initialize serial communication
  tft.begin();           // Initialize TFT display
  tft.setRotation(3);    // Set the display to landscape mode
  tft.fillScreen(TFT_BLACK);  // Set black background
}
```

Step 4: Temperature Data Display

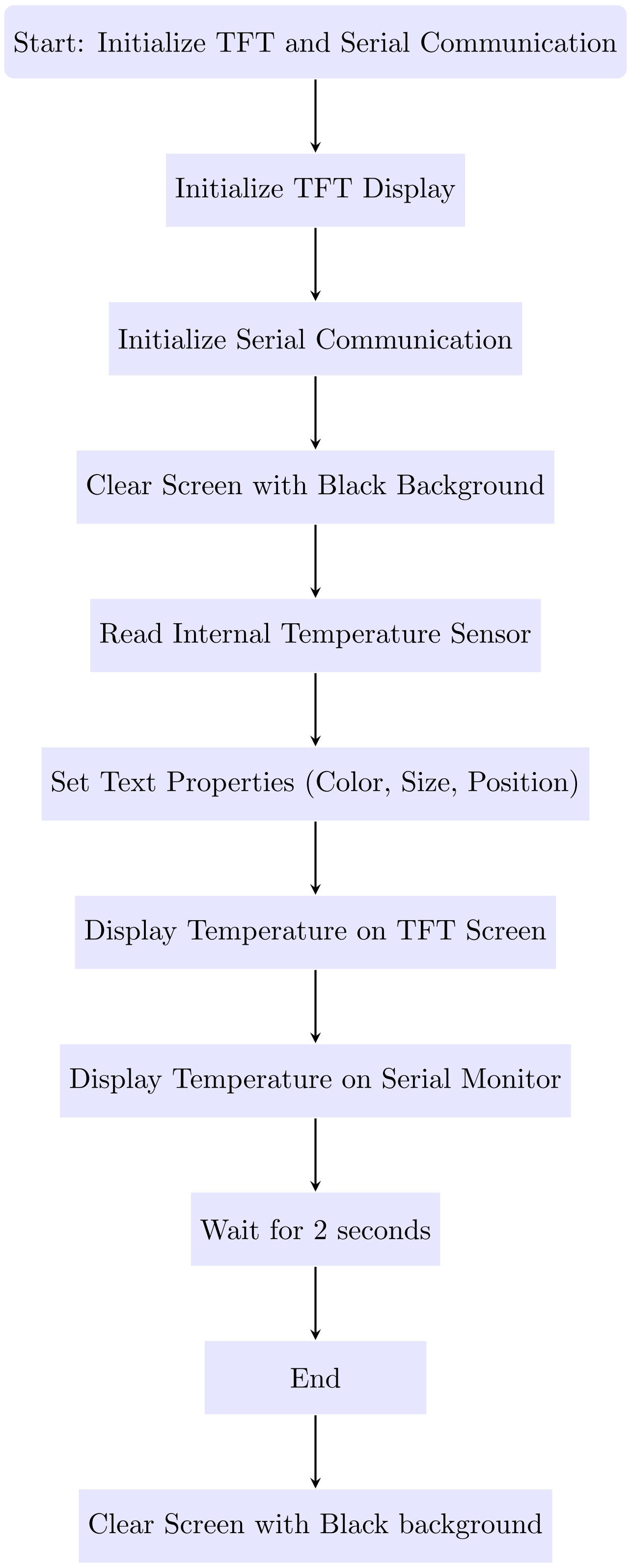
The `loop()` function reads the temperature, clears the screen to avoid text overlap, and displays the current temperature on the TFT. The same value is logged to the serial monitor for verification.

```
void loop() {
  tft.fillScreen(TFT_BLACK); // Clear previous data
  float temperature = readInternalTemperature(); // Read
   temperature

  tft.setTextColor(TFT_WHITE);  // Set text color to white
  tft.setTextSize(2);           // Set text size
  tft.setCursor(38, 120);       // Set cursor position
  tft.printf("Temperature: %.2f C", temperature); // Display
   temperature

  Serial.printf("Temperature: %.2f C\n", temperature); // Log
    to serial
  delay(2000); // Wait for 2 seconds before updating
}
```

4.3.3 Code flow for Internal Temperature plotting algorithm(Loop starts by resetting the screen to black)


Start: Initialize TFT and Serial Communication
Initialize TFT Display
Initialize Serial Communication
Clear Screen with Black Background
Read Internal Temperature Sensor
Set Text Properties (Color, Size, Position)
Display Temperature on TFT Screen
Display Temperature on Serial Monitor
Wait for 2 seconds
End
Clear Screen with Black background

4.3.4 Results

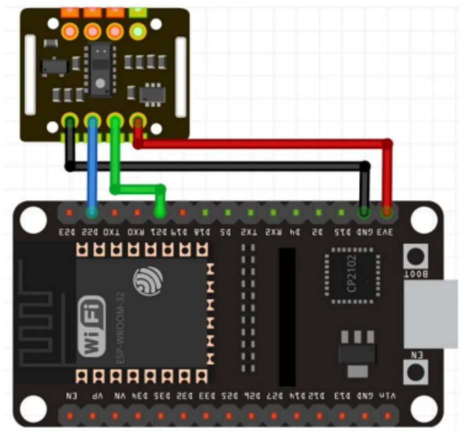
- **Real-Time Display of Internal Temperature:** Successfully displayed the internal temperature of the ESP32-S3 on the 2.4" TFT screen, providing real-time monitoring of the chip's thermal performance.
- **Stable and Accurate Readings:** The temperature readings were stable and updated accurately, indicating correct functionality of the temperature sensor and the display system.
- **Smooth User Interface:** The TFT screen displayed the temperature in a clean, readable format, with minimum delay between screen updates.

4.4 MAX30102 Sensor Setup

This section focuses on dynamically plotting heartbeat values on a TFT screen.

4.4.1 Hardware Integration

- The sensor is powered by a stable 3.3V supply from the microcontroller.
- The SDA and SCL pins of the MAX30102 are connected to the microcontroller's I2C bus with pins8 and 9 respectively.Here as MAX 30102 module has internal pull-up resitors, we don't require any other pull-up resistor



4.4.2 Software Configuration

The following libraries are included in the code to simplify configuration of default modes, LED amplitudes,etc.

Code Snippets and Explanations

```
#include <Wire.h>
```

This library facilitates I2C communication between the MAX30102 sensor and the ESP32 microcontroller. It initializes the I2C bus and ensures smooth data transfer via the SDA and SCL pins, enabling the microcontroller to communicate with the sensor effectively.

```
#include "MAX30105.h"
```

Its a library for the MAX30102 & MAX 30105 optical sensors, this library provides high-level functions for configuration and data retrieval. It allows access to sensor readings such as IR, red, and green light values, simplifying interaction with the sensor.

```
Wire.begin(8, 9);
```

The SDA and SCL pins of the MAX30102 are connected to the microcontroller's I2C bus with pins 8 and 9 respectively.

```
#include "heartRate.h"
```

This library contains algorithms which are essential for detecting heartbeats and calculating heart rate based on the IR sensor data.

particleSensor.setup(); // Configure sensor with default settings This function initializes the sensor with default configurations, including sampling rates and integration times.

```
particleSensor.setPulseAmplitudeRed(0x0A);
 particleSensor.setPulseAmplitudeGreen(0);

```

Turn Red LED to low to indicate sensor is running This command enables the red LED at low brightness which is vital for acquiring optical data for heart rate measurement. // Turn off Green LED

Step 1: Include Libraries and Initialize Variables

- The `Wire.h` library is used to manage I2C communication between the ESP32 and MAX30102.
- The `MAX30105.h` library allows interaction with the MAX30102 sensor.
- Variables like `rates[]` and `rateSpot` are initialized to store and manage heart rate data.

```
#include <Wire.h> // For I2C communication b/w PPG and ESP32
#include "MAX30105.h"
#include "heartRate.h"

MAX30105 particleSensor;
const byte RATE_SIZE = 4;
byte rates[RATE_SIZE];
byte rateSpot = 0;
long lastBeat = 0;
float beatsPerMinute;
int beatAvg;
```

Step 2: Setup Function for Initialization

- Initializes serial communication and sets up I2C.
- Configures the MAX30105 sensor for heart rate detection.
- Pulse amplitude for the red LED is set to 0x0A, and the green LED is disabled.

```
void setup() {
  Serial.begin(115200);
  Serial.println("Initializing...");
  Wire.begin(8, 9);
  if (!particleSensor.begin(Wire, I2C_SPEED_FAST)) {
    Serial.println("MAX30105 not found. Please check wiring/
 power.");
    while (1);
  }
  Serial.println("Place your index finger on the sensor with
 steady pressure.");
  particleSensor.setup();
  particleSensor.setPulseAmplitudeRed(0x0A);
  particleSensor.setPulseAmplitudeGreen(0);
}
```

Step 3: Loop Function to Process Heart Rate

- Continuously reads infrared data from the sensor and calculates heart rate.
- The heart rate is averaged over 16 readings to reduce fluctuations.
- The average BPM is printed to the serial monitor.

```
void loop() {
  long irValue = particleSensor.getIR();
  if (checkForBeat(irValue)) {
    long delta = millis() - lastBeat;
    lastBeat = millis();
    beatsPerMinute = 60 / (delta / 1000.0);
    if (beatsPerMinute < 255 && beatsPerMinute > 20) {
      rates[rateSpot++] = (byte)beatsPerMinute;
      rateSpot %= RATE_SIZE;
      beatAvg = 0;
      for (byte x = 0; x < RATE_SIZE; x++) {
        beatAvg += rates[x];
      }
      beatAvg /= RATE_SIZE;
    }
  }
  Serial.println(beatAvg);
  delay(20);
}
```

Step 4: Draw Graph Axes

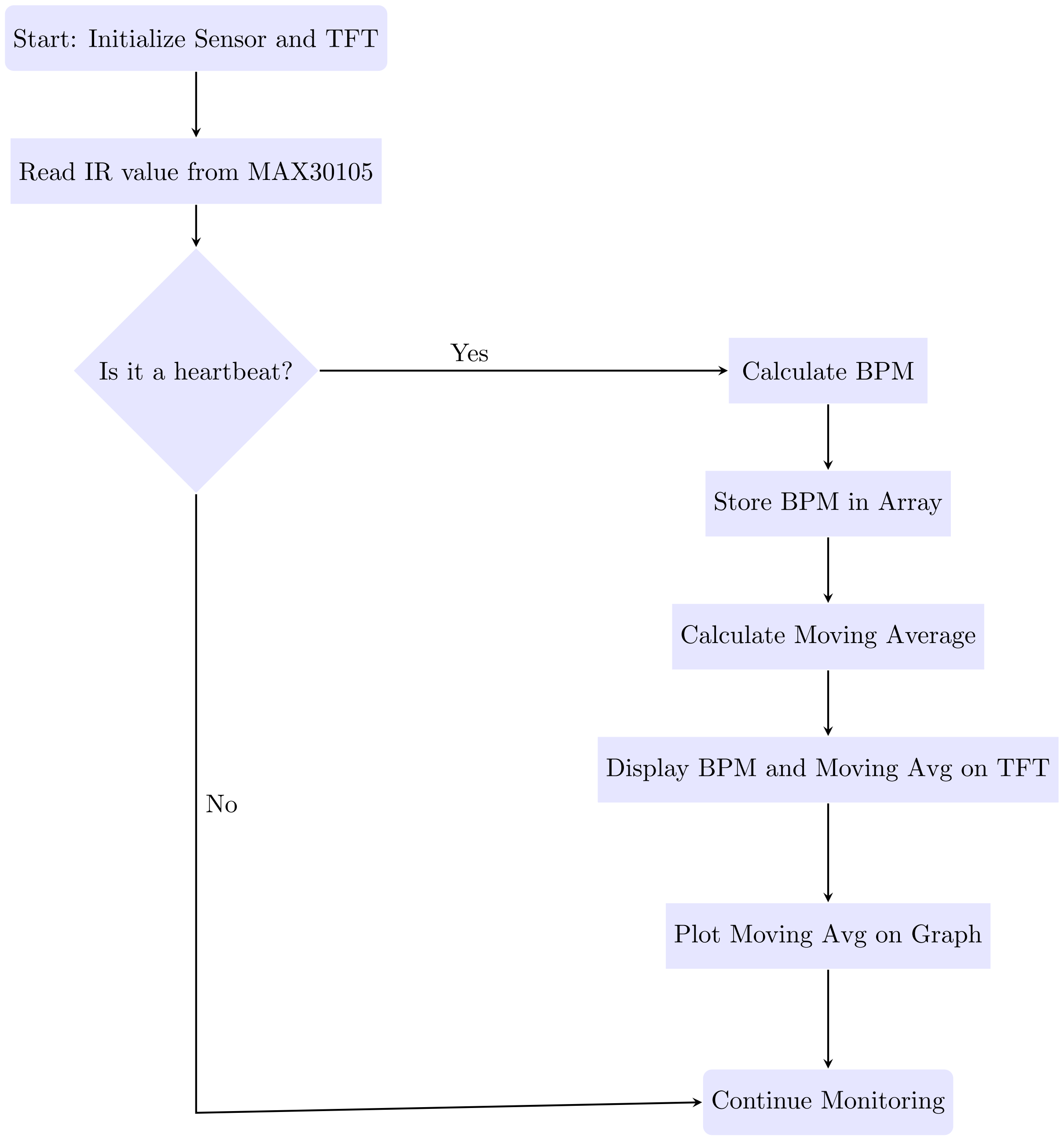
Step 1: Include Libraries and Initialize Variables

- The `Wire.h` library is used to manage I2C communication between the ESP32 and MAX30105.
- The `MAX30105.h` library allows interaction with the MAX30105 sensor.
- Variables like `rates[]` and `rateSpot` are initialized to store and manage heart rate data.

```
#include <Wire.h> // For I2C communication b/w PPG and ESP32
#include "MAX30105.h"
#include "heartRate.h"
MAX30105 particleSensor;
const byte RATE_SIZE = 4;
byte rates[RATE_SIZE];
byte rateSpot = 0;
long lastBeat = 0;
float beatsPerMinute;
int beatAvg;
```

- Clears the screen and draws the axes for the graph.
- Labels the X-axis with time in seconds and the Y-axis with BPM values.
- Adds labels to the Y-axis at intervals of 10 BPM.

- `plotMovingAvg` maps the moving average (BPM) to the graph's Y-coordinates.
- A line is drawn between the previous and current points for smooth plotting when `currentX > graphX`.
- When `currentX = graphX`, a line cannot be drawn from the previous point as there is no previous point.
- Updates lastY to ensure continuity in plotting.
- currentX is incremented, and the graph resets when it reaches the width.



4.4.4 Data Processing & Results

- Values from the IR sensor are measured using the `Particle.getSensor` function, and the inbuilt function `checkForBeat` checks for any beat. If a beat is detected, the time between two peaks is counted towards the heartbeat value.
- BPM value can be calculated using the time between two peaks using the following formula:

$$\frac{60 \times 1000}{\text{Time between two peaks}}$$

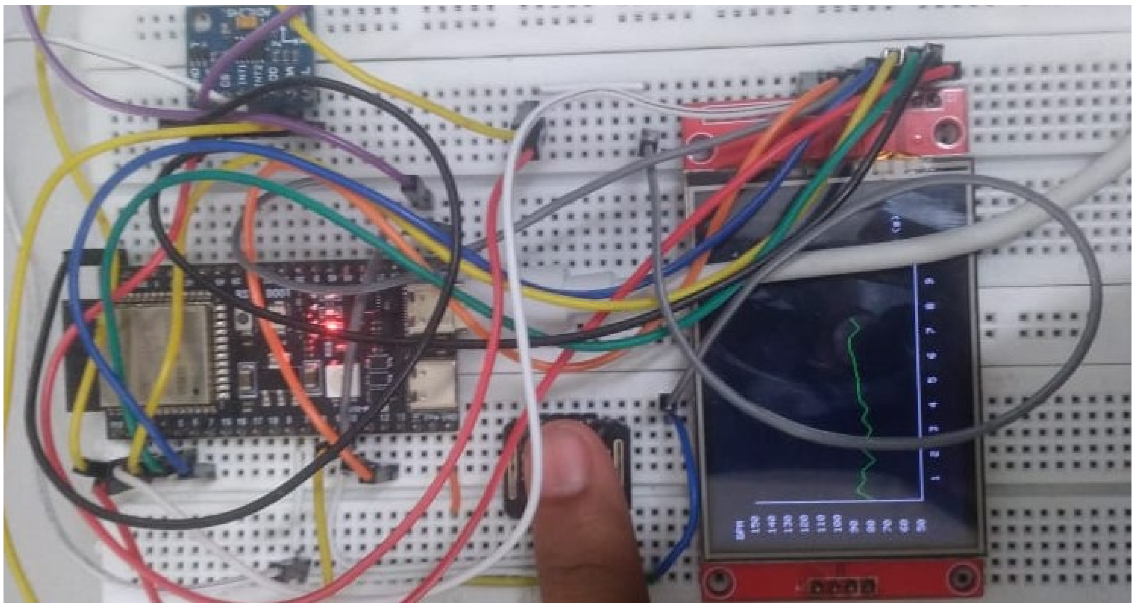


Figure 4.2: TFT screen showing heartbeat values

4.4.5 Troubleshooting

Heartbeat values may fluctuate in the output due to inconsistent pressure of the finger and varying light conditions. To smoothen these heartbeat values, we calculate the average of 4 heartbeat values from the rate array A similar loop adds labels for BPM values (50 to 150), and the `map` function maps BPM values to screen coordinates. This allows us to position the BPM data on the Y-axis by scaling it proportionally, mapping low BPM values to the bottom of the screen and high BPM values to the top. The formula also ensures that the graph fits perfectly within the screen's dimensions.

Heart Rate Detection Using Peak Detection

This section outlines the steps involved in detecting heartbeats using a signal processing technique. The signal data, presumably from an IR sensor, is processed to detect peaks, which are then used to estimate the heart rate in beats per minute (BPM).

Steps Involved

- **Convert Timestamps to Seconds:** Converts the timestamp data (in milliseconds) to seconds for easier calculations and plotting.
- **Peak Detection:** Uses the `find_peaks` function to detect significant peaks in the filtered IR signal, adjusting parameters like height, distance, and prominence.
- **Plotting the Signal and Peaks:** Plots the IR signal and marks the detected peaks to visually analyze the data.
- **Calculate Time Intervals Between Peaks:** Calculates the time between consecutive peaks to measure the intervals between heartbeats.

- **Calculate BPM:** Estimates the heart rate in beats per minute (BPM) using the time intervals between detected peaks.

Code Flow

Python Code for Peak Detection and BPM Calculation

```python
import numpy as np
from scipy.signal import find_peaks
import matplotlib.pyplot as plt

# Ensure timestamps are treated in seconds (80 ms per sample)
data['Time_in_seconds'] = data['Timestamp'] / 1000

# Detect peaks with adjusted parameters
peaks, _ = find_peaks(
    data['Filtered_IR'],
    height=0,           # Detect all peaks above zero
    distance=5,         # Adjusted distance to capture more
  peaks
    prominence=10       # Tune to ignore noise
)
```

Converting Timestamps to Seconds: The line `data['Time_in_seconds'] = data['Timestamp'] / 1000` converts the timestamp from milliseconds to seconds for easier plotting and analysis. **Peak Detection:** The `find_peaks` function detects significant peaks in the filtered IR signal. The parameters used are:

- `height=0`: Detects all peaks above zero.
- `distance=5`: Ensures peaks are at least 5 samples apart, avoiding multiple detections for a single heartbeat.
- `prominence=10`: Filters out noise by only considering prominent peaks.

```python
# Plot the detected peaks
plt.figure(figsize=(12, 6))
plt.plot(data['Time_in_seconds'], data['Filtered_IR'], label=
  'Filtered IR', color='blue')
plt.plot(data['Time_in_seconds'].iloc[peaks], data['
  Filtered_IR'].iloc[peaks], 'rx', label='Detected Peaks')

plt.xlabel('Time (seconds)')
plt.ylabel('Filtered IR Values')
plt.title('Peak Detection with Adjusted Parameters')
plt.legend()
plt.show()
```

Plotting the Signal and Peaks:

- figsize argument defines the size of the plot, in this case, 12 inches by 6 inches, to make the plot easy to read.

- The peaks array contains the indices of the detected peaks, and they are shown as red 'x' symbols on the plot. The label 'Detected Peaks' will appear in the legend.
- The `plt.plot` function is used to plot the IR signal and mark the detected peaks with red 'x' symbols for visual analysis.
- The signal is drawn in blue, and the label 'Filtered IR' will be used in the legend.
- plt.legend() call displays the labels for the plotted data, and plt.show() renders the plot for visualization.

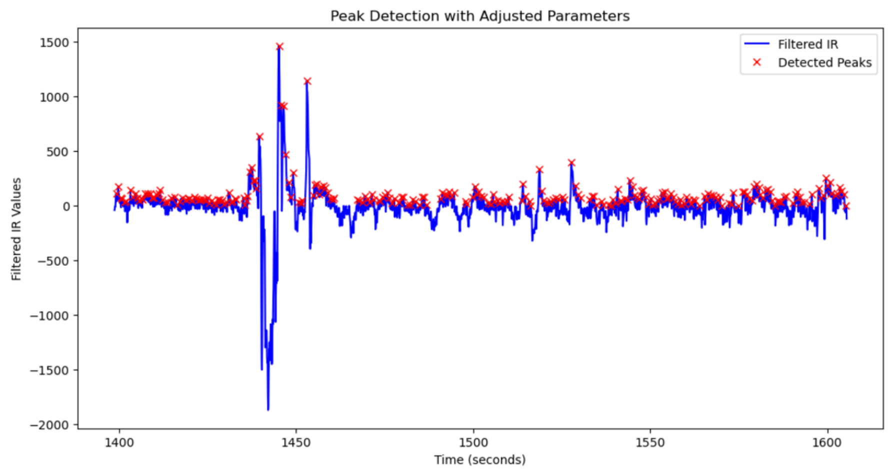
```python
# Calculate time intervals between consecutive peaks (in seconds)
peak_intervals = np.diff(data['Time_in_seconds'].iloc[peaks])

if len(peak_intervals) > 0:
    # Compute average interval and BPM
    average_interval = np.mean(peak_intervals)
    bpm = 60 / average_interval
    print(f"Estimated Heart Rate: {bpm:.2f} BPM")
else:
    print("No valid peaks detected.")
```

- **Time Intervals Between Peaks:** The `np.diff` function computes the time differences between consecutive peaks, which correspond to the intervals between heartbeats.
- **BPM Calculation:** The BPM is calculated by dividing 60 (seconds per minute) by the average interval between peaks, giving the number of beats per minute.

4.4.6 Results

- The Filtered IR signal is plotted against time to show the heart rate data.
- The detected peaks are marked with red 'x' symbols, representing the heartbeats identified in the signal.



```
Estimated Heart Rate: 74.79 BPM
```

4.5 Alternative Algorithm to Find Heartbeat Values

Another algorithm can also be implemented which is based on derivatives of the plot of IR values

1. Collecting raw IR data from the MAX30102 sensor.
2. Applying a smoothing algorithm to reduce noise in the signal.
3. Calculating the derivative of the signal to identify positive-to-negative transitions, which indicate peaks.
4. Validating peaks based on a predefined threshold and minimum distance between peaks.
5. Computing the heart rate in beats per minute (BPM) using the time difference between detected peaks.
6. Visualizing the results on a 2.4” TFT screen.

The steps are outlined below with corresponding code snippets and explanations.

Step 1: Sensor Initialization and Data Collection

Overview: The MAX30102 sensor is initialized to collect raw IR data at a sampling rate of 100Hz

```
#include <MAX30105.h>
#include <TFT_eSPI.h>

MAX30105 particleSensor;
TFT_eSPI tft = TFT_eSPI();

const int sampleRate = 100;
long irValues[100];
int sampleCount = 0;

void setup() {
    Serial.begin(115200);
    Wire.begin(8, 9);

    tft.init();
    tft.setRotation(1);
    tft.fillScreen(TFT_BLACK);

    if (!particleSensor.begin(Wire)) {
        Serial.println("MAX30105 not found");
        while (1);
    }

    particleSensor.setup(0x1F, 4, 2, sampleRate, 411, 4096);
}

```

```
void loop() {
    long irValue = particleSensor.getIR();
    irValues[sampleCount % 100] = irValue;
    sampleCount++;
}
```

Listing 4.1: Initialization code

Explanation:

- The MAX30102 sensor is set up with specific LED brightness, sample average, and other configurations.
- IR values are stored in an array for further processing.
- The ‘sampleRate‘ determines the frequency of data collection.
- TFT display is initialized

Step 2: Signal Smoothing

Overview: A moving average filter smooths the IR data, reducing noise.

```
long smoothedIR[100];

void smoothSignal() {
    for (int i = 1; i < sampleCount; i++) {
        smoothedIR[i % 100] = (irValues[(i-1) % 100] +
  irValues[i % 100]) / 2;
    }
}
```

Listing 4.2: Smoothing Algorithm

Explanation:

- The filter averages consecutive IR values to create a smoother signal.
- This prepares the data for derivative calculation.

Step 3: Peak Detection Using Derivatives

Overview: The derivative is computed to detect transitions, which are potential peaks.

```
const float epsilon = 1000.0;
const int peakThreshold = 100000;
int peaksDetected = 0;

void detectPeaks() {
    for (int i = 1; i < sampleCount - 1; i++) {
        float derivative = smoothedIR[i] - smoothedIR[i - 1];
        if (derivative > epsilon && smoothedIR[i] >
    smoothedIR[i + 1]) {
            if (smoothedIR[i] > peakThreshold) {
                peaksDetected++;
            }
        }
    }
}
```
Listing 4.3: Peak Detection

Explanation:

- Here we are using the fact that time values are discrete as in time can only take values 1,2,3 and so on
- Here we take derivative= smoothedIR [i] - smoothedIR [i - 1]
- If derivative is negative and smoothedIR [i + 1]) then we can say that a peak is detected
- But a peak is validated only if its value exceeds 'peakThreshold'.

Step 4: Heart Rate Calculation

Overview: The heart rate is calculated using the time difference between peaks.

```
unsigned long firstPeakTime, lastPeakTime;

float calculateBPM() {
    if (peaksDetected > 1) {
        float timeBetweenPeaks = (lastPeakTime -
    firstPeakTime) / 1000.0;
        return (60.0 * (peaksDetected - 1)) /
    timeBetweenPeaks;
    }
    return 0.0;
}
```
Listing 4.4: Heart Rate Calculation

Explanation:

- BPM is derived from the number of peaks and the time elapsed.
- the formula we are using for BPM is BPM= 60.0 * (peaksDetected - 1)) / timeBetweenPeaks

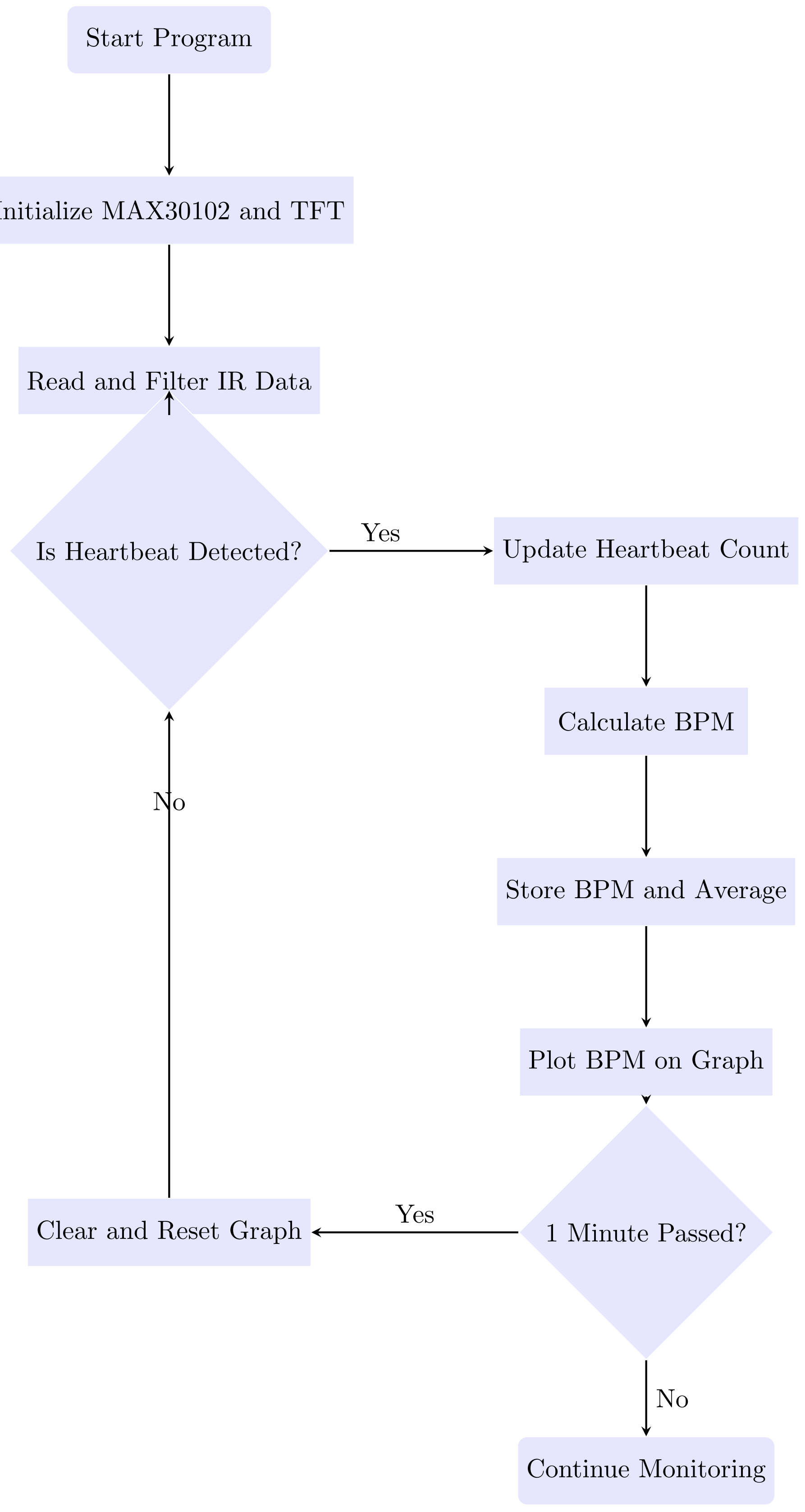
Step 5: Visualization

Overview: The calculated BPM is displayed on the TFT screen.

```
void displayBPM(float bpm) {
    tft.setTextColor(TFT_GREEN, TFT_BLACK);
    tft.setCursor(10, 10);
    tft.printf("BPM: %.1f", bpm);
}
```

Listing 4.5: Visualization

Code flow for the Derivative based algorithm



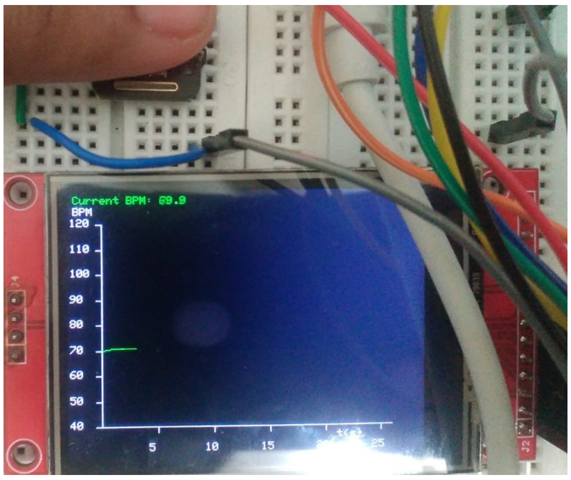


Figure 4.3: TFT Screen showing the heartbeat plot

Results

- Heartbeat peaks detected reliably with sufficient smoothing.
- BPM calculated and updated on the TFT screen in real-time.

Challenges and Limitations

- **Missed Peaks:** The first detected peak may not represent the actual heartbeat peak, leading to inaccuracies.
- **Noise Sensitivity:** High noise in IR readings can result in false positives or missed detections.
- **Inaccurate heartbeat**:The code may not determine the actual peak when —derivative— is less than epsilon leading to inaccurate heartbeat values
- **Threshold Tuning:** The peak detection threshold (‘peakThreshold‘ and ‘epsilon‘) requires careful tuning for different environments.

Chapter 5

Challenges Faced

5.1 Challenges Faced

Throughout this project I faced a lot of challenges which are listed as follows Noise in Raw Sensor Data

- Initially, the infrared (IR) signal from the MAX30105 sensor was highly noisy, making it difficult to identify clear peaks corresponding to heartbeats. This issue was addressed by implementing a smoothing algorithm, but fine-tuning the parameters of the algorithm (like the window size) was a challenge as it impacted both noise reduction and signal clarity.

- Sometimes, port and the microcontroller did not show up in IDE, for that I had to go uninstall and reinstall the drivers again

- Threshold Selection for Peak Detection: Setting a threshold to distinguish between actual heartbeat peaks and noise proved tricky.As, threshold value would depend on the lighting and pressure applied by the finger, etc.So,If the threshold was too low, it resulted in false positives, and if too high, genuine peaks were missed. Testing different threshold values for varying heart rates added complexity.

- Missed Peaks Due to High Variation in Signals: Even after smoothing, variations in individual heartbeat amplitudes caused some peaks to be missed. Adjusting the algorithm to account for such variability while avoiding overfitting was an iterative and time-consuming process.

- Real-Time Processing on ESP32-S3: There is a significant delay in the code which calculates heartbeat values as it requires to average 18 heartbeat values everytime, but this also will show impact on actual peaks as they might not be detected

- Display Constraints on the 2.4" TFT Screen: Designing a clear and readable graph for the heart rate and displaying average value and other things like finger not placed on the small TFT screen was challenging. Ensuring that both the axes and real-time BPM value were visible required careful planning of screen layout and font size adjustments.

- Environmental Interference: External factors like ambient light and movement have impacted signal quality alot. Adjusting sensor placement or Applying additional filtering techniques, required repeated testing in various conditions.

- First Peak Detection Error: The algorithm occasionally misidentified the first peak due to initialization errors or noise. Adding checks to verify peak validity and refining the initialization process helped, but required multiple iterations to balance accuracy and responsiveness. In conclusion most of the problems /challenges have occured as heartbeat values depend on a lot of factors such as physical activity, emotional state, and environmental conditions can cause fluctuations in heart rate measurements, which can complicate the accuracy and consistency of sensor readings.

Chapter 6

Key Learnings & Conclusion

- Understanding Sensor Limitations: Heart rate values can fluctuate due to various factors like physical activity, stress, or environment, which affects the accuracy of sensor readings.
- Importance of Calibration: Proper calibration of sensors is essential to reduce errors and improve the reliability of heart rate readings, especially in dynamic conditions.
- Real-time Data Processing: Handling live data from sensors and displaying it on the TFT screen taught me how to process and visualize real-time information in embedded systems.
- User Interface Design: Ensuring the display is intuitive and user-friendly can significantly improve the overall experience for users, making the data easier to understand.
- Troubleshooting and Debugging: Working through issues such as unstable sensor readings or communication errors deepened my understanding of how embedded systems work and how to troubleshoot effectively.
- Code Optimization: Efficient coding practices, such as minimizing delays(in some cases adding delays) and using correct data types, are important for optimizing the performance of embedded systems.
- Code Testing: Testing the entire code under different conditions (e.g., varying temperature or interference) emphasized the need for comprehensive testing to ensure reliability.
- Hardware-Software Integration: Successfully interfacing sensors with displays and microcontroller, especially handling communication protocols like I2C, SPI, and UART, is crucial for effective real-time data monitoring.

6.1 Conclusion

In conclusion, this project enhanced my skills in both hardware interfacing and software development. I was able to successfully initialize the TFT display, adjust the rotation settings, and clear the screen before rendering graphical data. The experience taught me how to work with different types of hardware, especially in terms of sensor data display and communication protocols.

I also gained a deeper understanding of the importance of setting up the system properly, ensuring the hardware works seamlessly, and testing each part of the system for robustness. The project reinforced how critical it is to plan the user interface carefully, as it directly impacts the user experience. Going forward, I am more confident in using embedded systems for various applications, particularly those involving real-time data processing and display.

Chapter 7

References

- https://www.analog.com/en/products/max30102.html
- https://docs.espressif.com/projects/
  esp-idf/en/latest/esp32s3/api-reference/peripherals/temp$_{s}ensor.html$
- https://pmc.ncbi.nlm.nih.gov/articles/PMC7146569/
- https://www.circuitbasics.com/basics-of-the-spi-communication-protocol/
- https://www.circuitbasics.com/basics-uart-communication/
- https://www.circuitbasics.com/basics-of-the-i2c-communication-protocol/
- https://microcontrollerslab.com/esp32-heart-rate-pulse-oximeter-max30102/
- https://wokwi.com/projects/new/esp32-s3
- https://github.com/atomic14/esp32-s3-pinouts
- https://eshelp.org/what-is-esp32-its-features-applications-and-limitations/
- https://forum.arduino.cc/t/com-port-issues-and-odd-serial-monitor-behavior-adafruit-esp32-feather/1272497/3
- https://randomnerdtutorials.com/lvgl-esp32-tft-touchscreen-display-ili9341-arduino/config-file-windows-pc
- https://www.youtube.com/watch?v=rq5yPJbX$_{u}k$
- https://forum.arduino.cc/t/difference-between-sensors-modules-and-sensors/402542
- https://www.dnatechindia.com/basic-working-pulse-oximeter-sensor.html
- https://www.youtube.com/watch?v=V5UvNVQsUsY

Chapter 8

Appendix

Additionally, these were some of the codes which were implemented during the course of project

Code to check for I2C device address

```
#include <Wire.h>

void setup() {
  Serial.begin(115200);
  while (!Serial); // Wait for serial to connect
  Serial.println("\nI2C Scanner");
  Wire.begin();
}

void loop() {
  byte error, address;
  int nDevices = 0;

  Serial.println("Scanning...");
  for (address = 1; address < 127; address++) {
    Wire.beginTransmission(address);
    error = Wire.endTransmission();
    if (error == 0) {
      Serial.print("I2C device found at address 0x");
      if (address < 16) {
        Serial.print("0");
      }
      Serial.print(address, HEX);
      Serial.println("  !");
      nDevices++;
      delay(500); // Wait a moment to be able to see the result
    } else if (error == 4) {
      Serial.print("Unknown error at address 0x");
      if (address < 16) {
        Serial.print("0");
      }
      Serial.println(address, HEX);
    }
  }
  if (nDevices == 0) {
    Serial.println("No I2C devices found\n");
  } else {
    Serial.println("done\n");
  }
  delay(5000); // Wait 5 seconds for the next scan
}
```

Code for gathering Raw and Filtered IR values along with time

```
    #include <Wire.h>
#include "MAX30105.h"

MAX30105 particleSensor;

float alpha = 0.95;  // DC removal constant
long w = 0;          // DC removal memory

long dcRemoval(long x) {
  long w_new = x + alpha * w;
  long y = w_new - w;
  w = w_new;
  return y;
}

void setup() {
  Serial.begin(115200);
  Serial.println("Initializing...");

  // Initialize the sensor
  if (!particleSensor.begin(Wire, I2C_SPEED_FAST)) {
    Serial.println("MAX30105 was not found. Please check
   wiring/power.");
    while (1);  // Halt if the sensor is not found
  }

  // Configure the MAX30105 sensor
  byte ledBrightness = 0x1F;
  byte sampleAverage = 8;
  byte ledMode = 3;
  int sampleRate = 100;
  int pulseWidth = 411;
  int adcRange = 4096;

  particleSensor.setup(ledBrightness, sampleAverage, ledMode,
    sampleRate, pulseWidth, adcRange);

  // Take an average of IR readings at startup for baseline
  const byte avgAmount = 64;
  long baseValue = 0;
  for (byte x = 0; x < avgAmount; x++) {
    baseValue += particleSensor.getIR();
  }
  baseValue /= avgAmount;

  // Pre-populate plotter with baseline values to stabilize Y-scale
  for (int x = 0; x < 500; x++) {
    Serial.println(baseValue);
  }

  // Print CSV header
  Serial.println("Millis,Raw IR Value,Filtered IR Value");
}

void loop() {
  long currentMillis = millis();  // Get the current time in milliseconds
  long rawIrValue = particleSensor.getIR();  // Get raw IR value from the sensor
  long filteredIrValue = dcRemoval(rawIrValue);  // Apply DC removal filter

  // Print CSV-formatted output
  Serial.print(currentMillis);
  Serial.print(",");
  Serial.print(rawIrValue);
  Serial.print(",");
  Serial.println(filteredIrValue);

  delay(10);  // Small delay to avoid flooding the serial output
}
```

References