UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE ENGENHARIA ELÉTRICA

Lucas Martins Primo
Raul Nicolini Rodrigues
Renato Souza Santana Filho

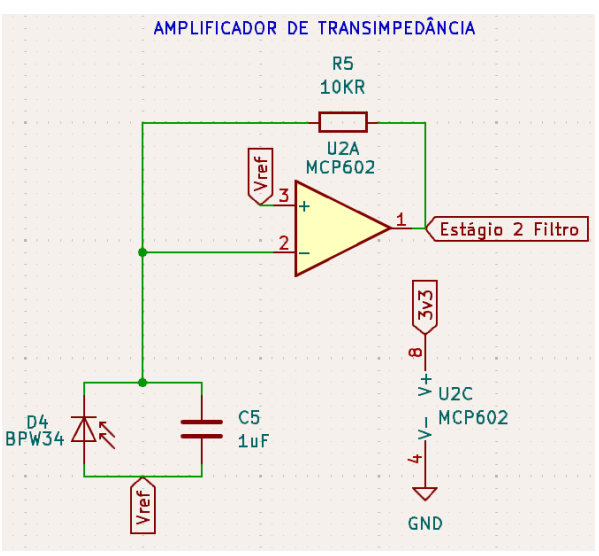
Oxímetro

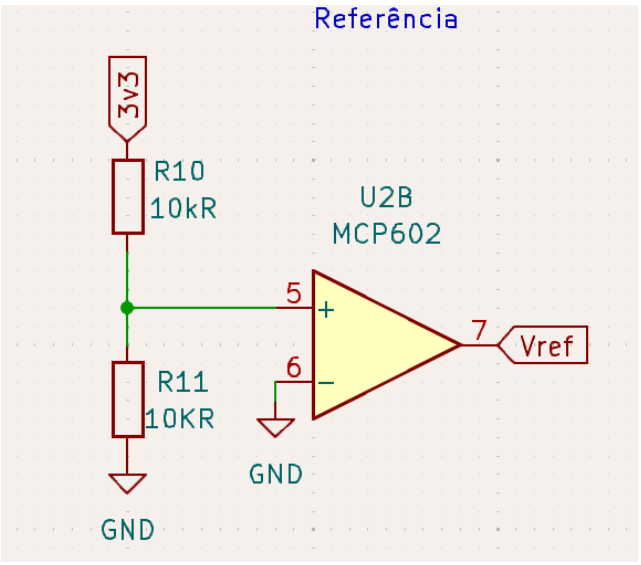
UBERLÂNDIA
2024

Diagrama eletrônico proposto pelo grupo:

O circuito foi idealizado no editor KiCad com base em >>> e pode ser observado na Figura 1 e 2, o circuito de transimpedância e a divisão de tensão com um amplificador operacional para fazer a referência das demais fases. No primeiro circuito é utilizado em um oxímetro de pulso para converter a corrente elétrica gerada pelo fotodiodo em uma tensão mensurável. No caso de um oxímetro de pulso, LEDs emitem luz através da pele, e o fotodiodo detecta a luz transmitida. A corrente gerada pelo fotodiodo devido à absorção diferencial da luz pelo oxigênio no sangue é muito fraca, e o amplificador de transimpedância amplifica essa corrente, transformando-a em um sinal de tensão que pode ser processado e analisado para determinar a saturação de oxigênio no sangue.

Figuras 1 e 2: Etapa de transimpedância e de geração de Referência para o sistema



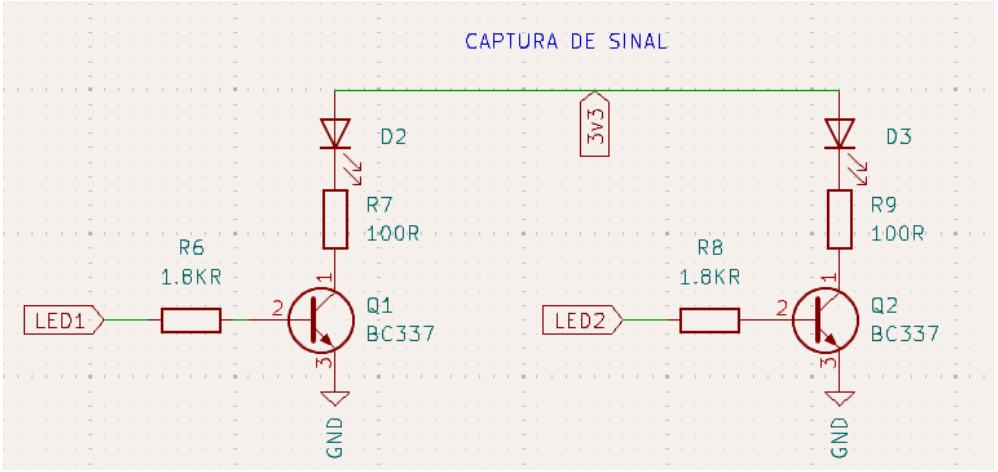


Fonte: Autoria própria

Nesse sentido, foram utilizados dois LEDs de diferentes comprimentos de onda, conforme a Figura 3, um vermelho e um infravermelho para medir a saturação de oxigênio no sangue. Isso ocorre porque a oxihemoglobina (hemoglobina ligada ao oxigênio) e a desoxihemoglobina (hemoglobina sem oxigênio) absorvem luz de maneira diferente em comprimentos de onda distintos. A luz vermelha é mais absorvida pela desoxihemoglobina, que é a forma da hemoglobina sem oxigênio. A luz infravermelha é mais absorvida pela oxihemoglobina, que é a forma da hemoglobina ligada ao oxigênio.

Ao comparar a quantidade de luz absorvida em cada comprimento de onda, o oxímetro pode calcular a proporção de oxihemoglobina em relação à hemoglobina total, o que permite determinar a saturação de oxigênio no sangue.

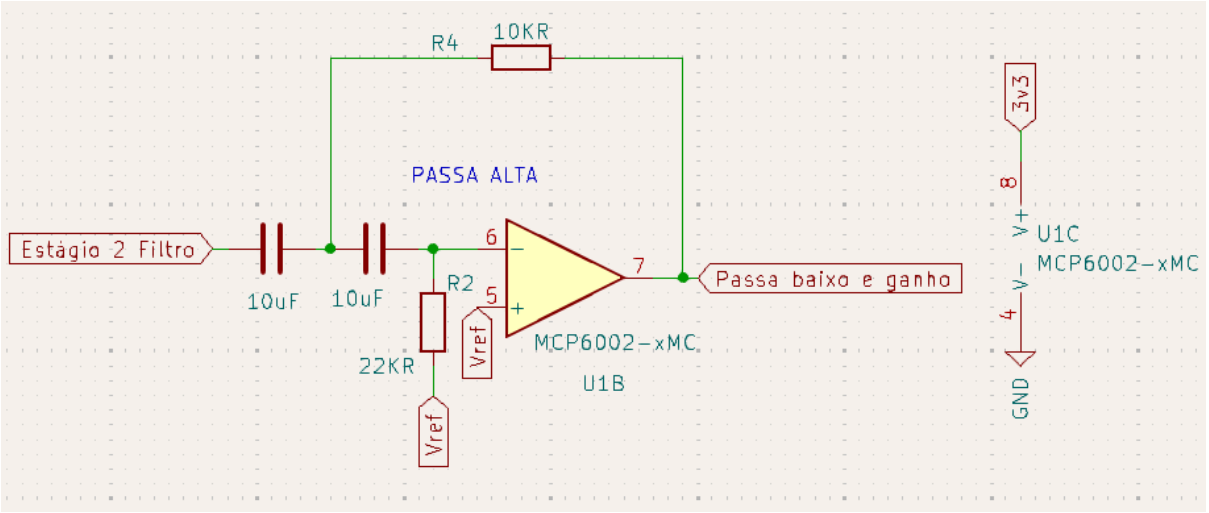
Figura 3: Estágio de captura do sinal para oximetria com um LED vermelho e um LED infravermelho



Fonte: Autoria própria

Um filtro passa-alta de 2ª ordem, ilustrado pela Figura 4, foi utilizado para remover componentes de frequências abaixo de 1Hz que podem interferir na medição precisa do sinal pulsátil de oxigênio no sangue. Essas componentes indesejadas podem incluir: ruído de movimento, variações lentas no ambiente de iluminação e flutuações fisiológicas.

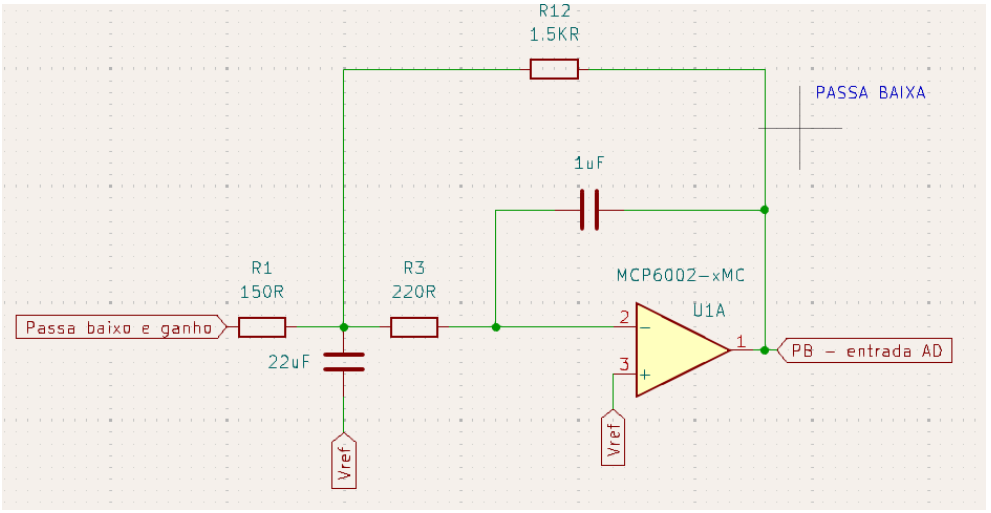
Figura 4: Filtro Passa Alta de 2ª ordem



Fonte: Autoria própria

Um filtro passa-baixa de 60 Hz com um ganho de 10x foi utilizado para melhorar a qualidade do sinal medido pelo dispositivo. O filtro passa-baixa, conforme a Figura 5, é projetado para atenuar ruídos de alta frequência, especialmente a interferência eletromagnética de 60 Hz proveniente de linhas de energia elétrica, que pode distorcer a leitura. Além de rejeitar esse ruído, o filtro também amplifica o sinal útil captado pelo fotodiodo, aumentando sua amplitude em 10 vezes. Isso facilita a detecção e a análise precisa do sinal correspondente ao pulso cardíaco, melhorando a confiabilidade da medição da saturação de oxigênio no sangue.

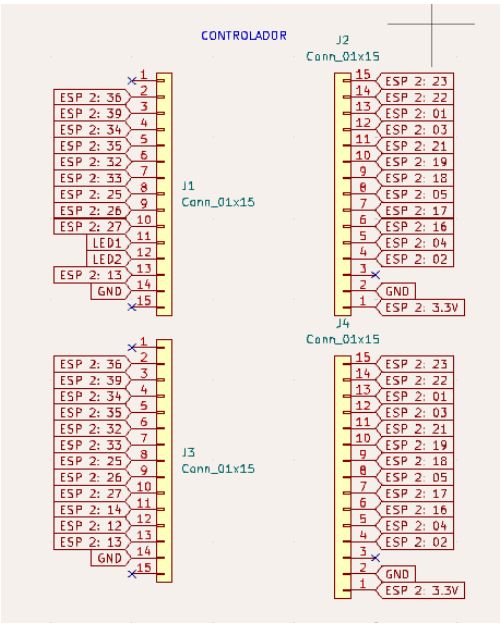
Figura 5: Filtro Passa Baixa de 2ª ordem com ganho



Fonte: Autoria própria

Na Figura 6, ilustra-se o esquemático das portas do microcontrolador, nele são configurados 3 portas GPIO, duas para alternar o acionamento dos LEDs e uma para receber a leitura do sinal captado pelo fotodiodo.

Figura 6: Portas de um microcontrolador ESP32-WROOM-DEVKIT-V1



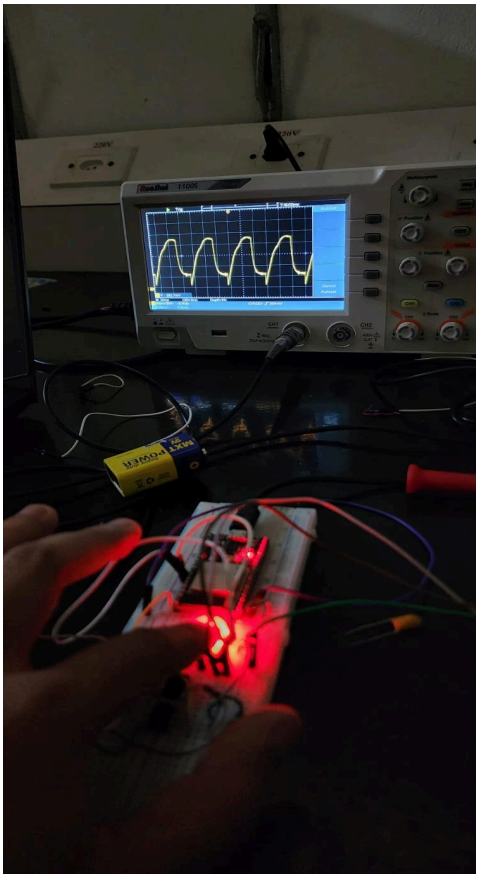
Fonte: Autoria própria

Lista de componentes:

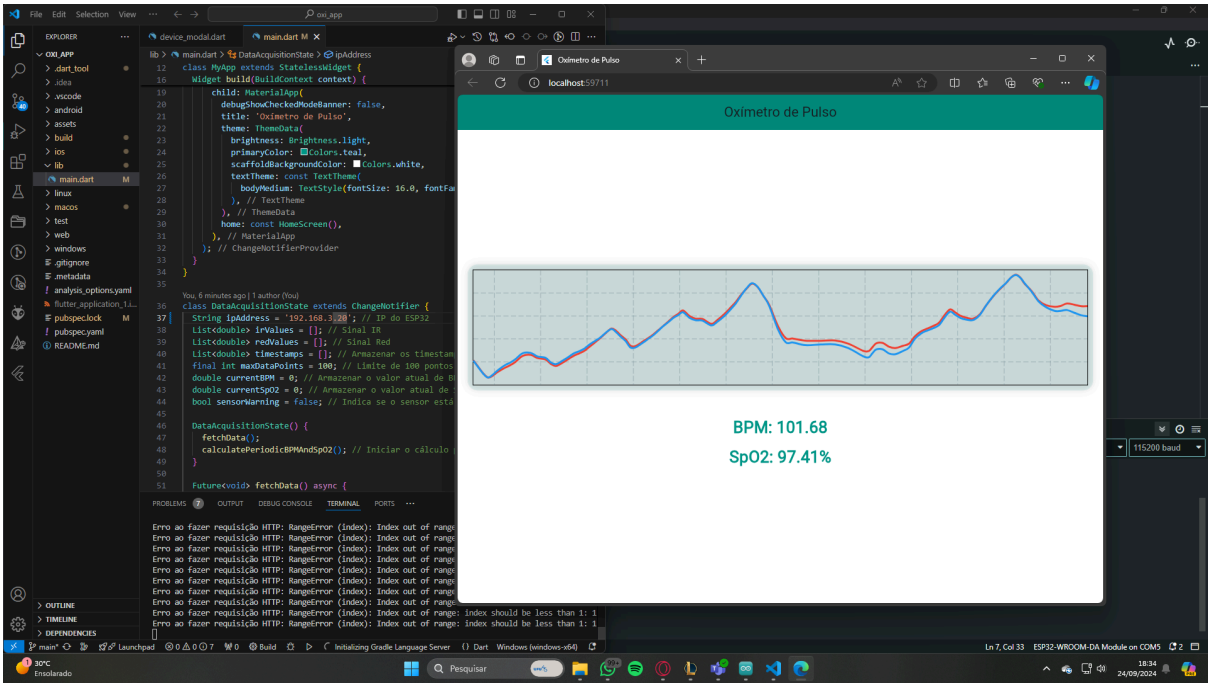
Nome	“Valor”	Quantidade
C1,C2	10uF	2
C3	22uF	1
C4,C5	1uF	2
D2,D3	MTE8120CP 805nm	2
D4	BPW34	1
J1,J2,J3,J4	Conn_01x15	4
Q1,Q2	BC337	2
R1	150R	1
R2	22KR	1
R3	220R	1
R4,R5,R11	10KR	3

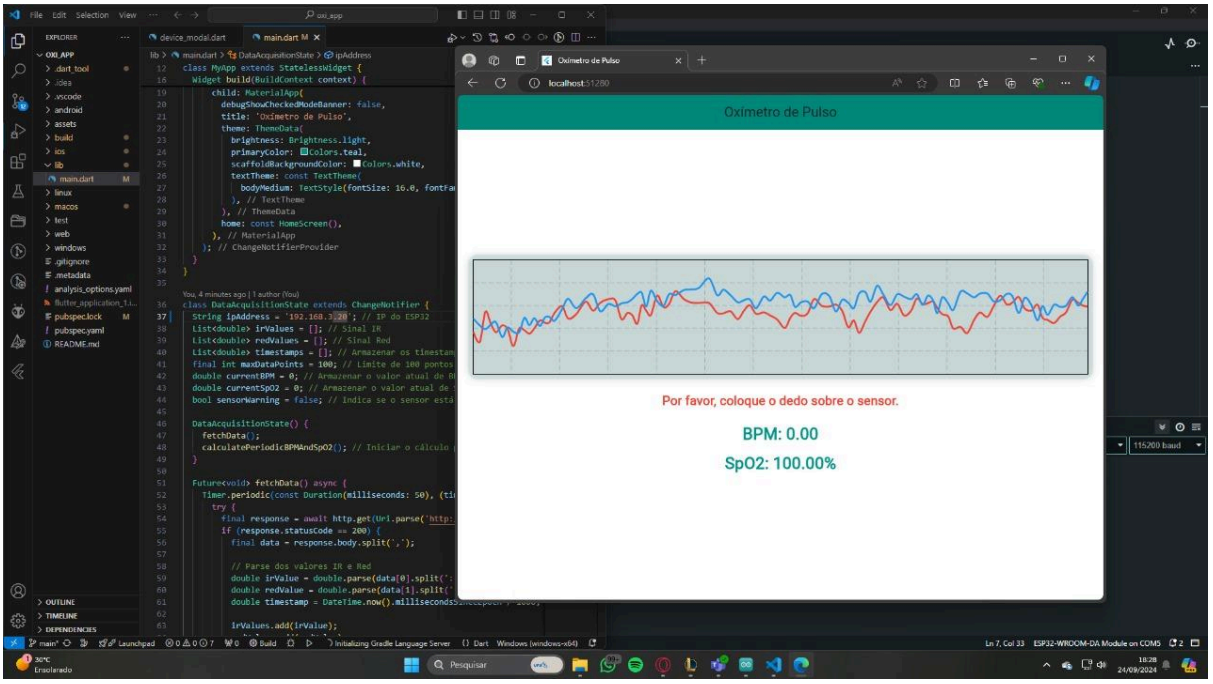
R6,R8	**1.8KR**	**2**
R7,R9	**100R**	**2**
R10	**10kR**	**1**
R12	**1.5KR**	**1**
U1	**MCP6002-xMC**	**1**
U2	**MCP602**	**1**

Imagens do circuito funcionando:



Imagens do Software Funcionando:





Software Embarcado:

```
#include <WiFi.h>
#include <WebServer.h>

// Configurações Globais
#define verm_pin 18
#define infra_pin 19
#define leitura_pin 33
```

```
// Variáveis com o estado dos pinos, se estão ou não ativos
bool verm_estado = false;
bool infra_estado = false;

// Variáveis para os valores recebidos
float infra_recebido, verm_recebido;

// Índices para contagem e resultados
float verm_soma, infra_soma; // Soma dos valores dos vetores
float verm_media, infra_media;
float R; // Relação entre os resultados das médias
float spo2; // Saturação de oxigênio no sangue

const int intervalo = 500; // Intervalo ajustado para quantidade de
dados recebidos (ajuste conforme necessário)

// Vetores para salvar as variáveis
float verm_valores[intervalo], desl_valores[intervalo];
float infra_valores[intervalo];
float verm_subtracao[intervalo], infra_subtracao[intervalo];
float verm_filtrado[intervalo], infra_filtrado[intervalo];

// Variáveis para controle de tempo
unsigned long tempo_anterior_leitura = 0; // Tempo da última leitura
unsigned long tempo_anterior_calculo = 0; // Tempo do último cálculo de
SpO2
const unsigned long intervalo_leitura = 20; // Intervalo em
milissegundos entre leituras (20 ms)
const unsigned long intervalo_calculo = 1000; // Intervalo em
milissegundos para cálculo de SpO2 (1 segundo)

// Configuração Wi-Fi
const char* ssid = "Net do lucas";
const char* password = "12345678";

// Instância do servidor web
WebServer server(80);
void setup() {
    Serial.begin(115200);

    pinMode(verm_pin, OUTPUT);
    pinMode(infra_pin, OUTPUT);
```

```
    configurarLEDs(false, false); // Ambos LEDs desligados

    // Conectando à rede Wi-Fi
    WiFi.begin(ssid, password);
    while (WiFi.status() != WL_CONNECTED) {
        delay(1000);
        Serial.println("Conectando ao WiFi...");
    }
    Serial.println("Conectado ao WiFi");
    Serial.println(WiFi.localIP()); // Exibe o IP atribuído

    // Configuração do servidor web
    server.on("/dados", [](){
    String dados = "SpO2:" + String(spo2);
    server.sendHeader("Access-Control-Allow-Origin", "*");  // Adiciona
o cabeçalho CORS
    server.send(200, "text/plain", dados);
});

    server.begin();
    Serial.println("Servidor iniciado");
}

// Loop principal
void loop() {
    server.handleClient(); // Lida com clientes do servidor

    unsigned long tempo_atual = millis();

    // Verifica se o tempo desde a última leitura é maior que o
intervalo desejado
    if (tempo_atual - tempo_anterior_leitura >= intervalo_leitura) {
        tempo_anterior_leitura = tempo_atual; // Atualiza o tempo da
última leitura

        // Realizar leitura dos LEDs
        realizar_leitura();
        alternar_leds(); // Alterna o estado dos LEDs
    }

    // Verifica se é hora de realizar os cálculos de SpO2
    if (tempo_atual - tempo_anterior_calculo >= intervalo_calculo) {
```

```
        tempo_anterior_calculo = tempo_atual; // Atualiza o tempo do
último cálculo

        // Chamada de funções para cálculo de SpO2
        filtro_modocomum();
        calculo_media();
        calculo_final();

        // Plotagem no serial monitor dos resultados finais
        Serial.print("SpO2: ");
        Serial.println(spo2);
        Serial.println();
    }

    delay(1); // Pequeno delay para aliviar o processamento
}

// Função para realizar leitura, dependendo do estado dos LEDs
void realizar_leitura() {
    static int i = 0; // Índice estático para manter a posição entre
chamadas

    if (verm_estado) { // Quando o LED vermelho está ligado
        verm_recebido = analogRead(leitura_pin);
        verm_valores[i] = verm_recebido;
        infra_valores[i] = 0;
    } else if (infra_estado) { // Quando o LED infravermelho está
ligado
        infra_recebido = analogRead(leitura_pin);
        infra_valores[i] = infra_recebido;
        verm_valores[i] = 0;
    } else { // Quando os dois LEDs estão desligados
        desl_valores[i] = analogRead(leitura_pin);
    }

    i = (i + 1) % intervalo; // Incrementa o índice e reseta quando
atinge o intervalo
}

// Função de filtro de modo comum com filtragem adicional de suavização
void filtro_modocomum() {
    for (int k = 0; k < intervalo; k++) {
        verm_subtracao[k] = verm_valores[k] - desl_valores[k];
```

```
        infra_subtracao[k] = infra_valores[k] - desl_valores[k];

        // Filtro passa-baixa simples para suavização
        verm_filtrado[k] = 0.9 * abs(verm_subtracao[k]) + 0.1 *
verm_filtrado[k];
        infra_filtrado[k] = 0.9 * abs(infra_subtracao[k]) + 0.1 *
infra_filtrado[k];
    }
}

// Função para calcular a média dos vetores
void calculo_media() {
    verm_soma = 0;
    infra_soma = 0;

    for (int i = 0; i < intervalo; i++) {
        verm_soma += verm_filtrado[i];
        infra_soma += infra_filtrado[i];
    }

    verm_media = verm_soma / intervalo;
    infra_media = infra_soma / intervalo;

    // Verifica as médias calculadas
    Serial.print("Vermelho Médio: ");
    Serial.println(verm_media);
    Serial.print("Infravermelho Médio: ");
    Serial.println(infra_media);
}

// Função para fazer o cálculo final usando a relação de Beer-Lambert
void calculo_final() {
    if (infra_media != 0) { // Evitar divisão por zero
        R = verm_media / infra_media;
        spo2 = 110 - 25 * R; // Ajuste conforme necessário
        if (spo2 < 70) spo2 = 70;
        if (spo2 > 100) spo2 = 100;
    } else {
        spo2 = 0; // Caso infra_media seja zero, SpO2 é indefinido
        Serial.println("Aviso: Média infravermelha zero, SpO2
indefinido.");
    }
}
```

```
// Função para configurar o estado dos LEDs
void configurarLEDs(bool estadoVermelho, bool estadoInfra) {

    digitalWrite(verm_pin, estadoVermelho ? HIGH : LOW);
    digitalWrite(infra_pin, estadoInfra ? HIGH : LOW);
    verm_estado = estadoVermelho;
    infra_estado = estadoInfra;
}

// Função para alternar os estados de LEDs com intervalo mais longo
para estabilidade
void alternar_leds() {
    static unsigned long ultimo_troca = 0;
    if (millis() - ultimo_troca > 100) { // Alterna a cada 100ms para
estabilidade
        if (!verm_estado && !infra_estado) {
            configurarLEDs(true, false); // Liga vermelho
        } else if (verm_estado && !infra_estado) {
            configurarLEDs(false, true); // Liga infravermelho
        } else {
            configurarLEDs(false, false); // Desliga ambos
        }
        ultimo_troca = millis();
    }
}
```

Aplicativo do celular:

```
import 'dart:async';
import 'dart:math';
import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;
import 'package:provider/provider.dart';
import 'package:fl_chart/fl_chart.dart'; // Para exibir gráficos no
Flutter

void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});
```

```
  @override
  Widget build(BuildContext context) {
    return ChangeNotifierProvider(
      create: (context) => DataAcquisitionState(),
      child: MaterialApp(
        debugShowCheckedModeBanner: false,
        title: 'Oxímetro de Pulso',
        theme: ThemeData(
          brightness: Brightness.light,
          primaryColor: Colors.teal,
          scaffoldBackgroundColor: Colors.white,
          textTheme: const TextTheme(
            bodyMedium: TextStyle(fontSize: 16.0, fontFamily:
'Roboto'),
          ),
        ),
        home: const HomeScreen(),
      ),
    );
  }
}

class DataAcquisitionState extends ChangeNotifier {
  String ipAddress = '192.168.3.20'; // IP do ESP32
  List<double> irValues = []; // Sinal IR
  List<double> redValues = []; // Sinal Red
  List<double> timestamps = []; // Armazenar os timestamps
  final int maxDataPoints = 100; // Limite de 100 pontos
  double currentBPM = 0; // Armazenar o valor atual de BPM
  double currentSpO2 = 0; // Armazenar o valor atual de SpO2
  bool sensorWarning = false; // Indica se o sensor está com problema

  DataAcquisitionState() {
    fetchData();
    calculatePeriodicBPMAndSpO2(); // Iniciar o cálculo periódico de
BPM e SpO2
  }

  Future<void> fetchData() async {
    Timer.periodic(const Duration(milliseconds: 50), (timer) async {
      try {
```

```
        final response = await
http.get(Uri.parse('http://$ipAddress/dados'));
        if (response.statusCode == 200) {
          final data = response.body.split(',');

          // Parse dos valores IR e Red
          double irValue = double.parse(data[0].split(':')[1]);
          double redValue = double.parse(data[1].split(':')[1]);
          double timestamp = DateTime.now().millisecondsSinceEpoch /
1000;

          irValues.add(irValue);
          redValues.add(redValue);
          timestamps.add(timestamp);

          // Verificar se os valores estão abaixo de 40.000
          if (irValue < 40000 || redValue < 40000) {
            sensorWarning = true; // Ativar o aviso de sensor
          } else {
            sensorWarning = false; // Desativar o aviso se os valores
forem normais
          }

          // Remover os dados antigos quando o limite de pontos for
ultrapassado
          if (irValues.length > maxDataPoints) {
            irValues.removeAt(0);
            redValues.removeAt(0);
            timestamps.removeAt(0);
          }

          notifyListeners();
        } else {
          print('Erro na resposta HTTP: ${response.statusCode}');
        }
      } catch (e) {
        print('Erro ao fazer requisição HTTP: $e');
      }
    });
  }

  // Função para calcular o BPM e SpO2 a cada 10 segundos
  void calculatePeriodicBPMAndSpO2() {
```

```
    Timer.periodic(const Duration(seconds: 5), (timer) {
      if (irValues.isNotEmpty && redValues.isNotEmpty) {
        currentBPM = calculateBPM();
        currentSpO2 = calculateSpO2();
        notifyListeners(); // Atualizar os valores exibidos na
interface
      }
    });
  }

  // Função para calcular o BPM com base nos picos do sinal IR
  double calculateBPM() {
    if (timestamps.length < 2) return 0;

    // Detectar picos no sinal IR para identificar batimentos
    List<int> peakIndices = [];
    for (int i = 1; i < irValues.length - 1; i++) {
      if (irValues[i] > irValues[i - 1] &&
          irValues[i] > irValues[i + 1] &&
          irValues[i] > 40000) {
        peakIndices.add(i);
      }
    }

    // Calcular BPM com base nos intervalos de tempo entre os picos
    if (peakIndices.length >= 2) {
      List<double> intervals = [];
      for (int i = 1; i < peakIndices.length; i++) {
        double interval = timestamps[peakIndices[i]] -
timestamps[peakIndices[i - 1]];
        intervals.add(interval);
      }

      double averageInterval = intervals.reduce((a, b) => a + b) /
intervals.length;
      double bpm = 60 / averageInterval; // Converter intervalo para
BPM
      return bpm;
    } else {
      return 0; // Não há batimentos suficientes para calcular BPM
    }
  }
```

```
  // Função para calcular o SpO2
  double calculateSpO2() {
    if (irValues.isEmpty || redValues.isEmpty) return 0;

    // Calcular a razão R = (AC_red/DC_red) / (AC_ir/DC_ir)
    double irAC = irValues.reduce((a, b) => max(a - irValues[0], b -
irValues[0]));
    double irDC = irValues.reduce((a, b) => a + b) / irValues.length;

    double redAC = redValues.reduce((a, b) => max(a - redValues[0], b -
redValues[0]));
    double redDC = redValues.reduce((a, b) => a + b) /
redValues.length;

    if (irDC == 0 || redDC == 0) return 0;

    double R = (redAC / redDC) / (irAC / irDC);

    // Fórmula aproximada para calcular SpO2 a partir de R
    double spO2 = 110 - 25 * R;

    // Ajuste: Limitar o SpO2 entre 95% e 100%
    return spO2.clamp(95, 100); // Modificação aqui para garantir
valores entre 95 e 100
  }

  // Normalizar os valores do sinal entre 0 e 100 para o gráfico
  List<double> normalizeValues(List<double> values) {
    if (values.isEmpty) return [];

    double minValue = values.reduce((a, b) => a < b ? a : b);
    double maxValue = values.reduce((a, b) => a > b ? a : b);

    if (maxValue == minValue) return List<double>.filled(values.length,
50.0);

    return values.map((value) {
      return 100 * (value - minValue) / (maxValue - minValue);
    }).toList();
  }

  List<double> get normalizedIrValues => normalizeValues(irValues);
  List<double> get normalizedRedValues => normalizeValues(redValues);
```

```dart
}

class HomeScreen extends StatefulWidget {
  const HomeScreen({super.key});

  @override
  State<HomeScreen> createState() => _HomeScreenState();
}

class _HomeScreenState extends State<HomeScreen> {
  @override
  Widget build(BuildContext context) {
    final dataState = Provider.of<DataAcquisitionState>(context);

    return Scaffold(
      appBar: AppBar(
        title: const Text('Oxímetro de Pulso'),
        backgroundColor: Colors.teal[600],
        centerTitle: true,
      ),
      body: Padding(
        padding: const EdgeInsets.all(16.0),
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            Container(
              decoration: BoxDecoration(
                borderRadius: BorderRadius.circular(16),
                color: Colors.teal.withOpacity(0.1),
                boxShadow: [
                  BoxShadow(
                    color: Colors.black12,
                    blurRadius: 8.0,
                  ),
                ],
              ),
              child: SizedBox(
                height: 200,
                child: Padding(
                  padding: const EdgeInsets.all(8.0),
                  child: LineChart(
                    LineChartData(
                      minY: 0,  // Limitar o valor mínimo do eixo Y a 0
```

```
                    maxY: 100,  // Limitar o valor máximo do eixo Y a
100
                    titlesData: const FlTitlesData(
                      show: true,
                      bottomTitles: AxisTitles(
                        sideTitles: SideTitles(showTitles: false), //
Remover completamente as legendas do eixo X
                      ),
                      leftTitles: AxisTitles(
                      ),
                      topTitles: AxisTitles(
                        sideTitles: SideTitles(showTitles: false), //
Remover qualquer título superior
                      ),
                      rightTitles: AxisTitles(
                      ),
                    ),
                    lineBarsData: [
                      // Gráfico do sinal IR
                      LineChartBarData(
                        spots: List.generate(
                          dataState.normalizedIrValues.length,
                          (index) => FlSpot(
                            dataState.timestamps[index],
                            dataState.normalizedIrValues[index],
                          ),
                        ),
                        isCurved: true,
                        color: Colors.red, // Sinal IR em vermelho
                        barWidth: 3,
                        belowBarData: BarAreaData(show: false),
                        dotData: FlDotData(show: false), //
Desabilitar pontos no gráfico
                      ),
                      // Gráfico do sinal Red
                      LineChartBarData(
                        spots: List.generate(
                          dataState.normalizedRedValues.length,
                          (index) => FlSpot(
                            dataState.timestamps[index],
                            dataState.normalizedRedValues[index],
                          ),
                        ),
```

```
                        isCurved: true,
                        color: Colors.blue, // Sinal Red em azul
                        barWidth: 3,
                        belowBarData: BarAreaData(show: false),
                        dotData: FlDotData(show: false), //
Desabilitar pontos no gráfico
                      ),
                    ],
                  ),
                ),
              ),
            ),
            const SizedBox(height: 20),

            // Exibir mensagem se o sensor não detectar o dedo
            if (dataState.sensorWarning)
              const Text(
                'Por favor, coloque o dedo sobre o sensor.',
                style: TextStyle(
                  fontSize: 20,
                  color: Colors.red,
                  fontWeight: FontWeight.bold,
                ),
              ),
            const SizedBox(height: 20),
            Text(
              'BPM: ${dataState.currentBPM.toStringAsFixed(2)}',
              style: const TextStyle(
                fontSize: 26,
                fontWeight: FontWeight.bold,
                color: Colors.teal,
              ),
            ),
            const SizedBox(height: 10),
            Text(
              'SpO2: ${dataState.currentSpO2.toStringAsFixed(2)}%',
              style: const TextStyle(
                fontSize: 26,
                fontWeight: FontWeight.bold,
                color: Colors.teal,
              ),
            ),
```

```
            ],
          ),
        ),
      );
    }
}
```