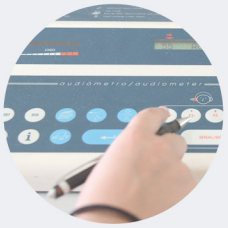
INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA BIOMÉDICA






Desenvolvimento de protótipo de baixo custo para sistema de controlo de sinais vitais

PEDRO MIGUEL PINTO MOREIRA GUIMARÃES
dezembro de 2024

P.PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Desenvolvimento de protótipo de baixo custo para SISTEMA DE CONTROLO DE SINAIS VITAIS

Pedro Miguel Pinto Moreira Guimarães

Dissertação para obtenção do Grau de Mestre em Engenharia Biomédica

Orientador: Prof. Doutor Joaquim Alves

Júri:

Presidente:

Natércia Lima, Professor Adjunto, ISEP

Vogais:

Joaquim Alves, Professor Adjunto, ISEP

Nuno Gueiral, Professor Adjunto, ISEP

Porto, novembro de 2024

Resumo

O objetivo deste projeto é o desenvolvimento de um sistema que permita a centralização, num único dispositivo, do processo de registo de sinais vitais de um indivíduo, especialmente em casos em que exista a necessidade de realizar estas medições de forma regular, como por exemplo em pacientes com histórico de problemas cardíacos, permitindo assim a simplificação de todo o processo de recolha de sinais vitais e consequente registo histórico, com recurso a uma aplicação móvel que controlará todo o processo.

Assim, foi desenvolvido um protótipo baseado em dispositivos médicos de uso pessoal já existentes no mercado, cujo controlo dos mesmos foi centralizado num microcontrolador ESP32. Este sistema permite que os dispositivos sejam operados através de uma aplicação móvel compatível com iOS e Android, proporcionando uma interface que facilita aos utilizadores a operação e centralização do seu historial clínico diretamente nos seus dispositivos móveis.

Este projeto permitiu então obter um protótipo operacional, que cumpre os objetivos propostos, esperando-se igualmente que possa servir de base para o desenvolvimento futuro de um sistema que venha, por exemplo, a ser alvo de utilização em instituições em que esta necessidade se aplique a um grupo pessoas (como um lar de idosos), tendo em vista uma solução que, uma vez implementada, se possa vir a revelar um importante instrumento neste âmbito e cuja sua utilização se possa traduzir em reais mais-valias para os diversos setores de atividade em que a mesma possa ser colocada em utilização.

Palavras-chave: Comunicação sem fios, Sinais Vitais, Sistemas Embebidos

Abstract

The objective of this project is the development of a system that allows the centralization, in a single device, of the recording process of a person's vital signs, especially in situations where the need to perform these measurements occurs regularly, thus allowing that the full process of collecting vital signs can be clear and simple, with the help of a mobile application that will control the entire process.

Thus, a prototype based on medical devices for personal use already available on the market was developed, with its control centralized on an ESP32 microcontroller. This system allows devices to be operated through a mobile app compatible with iOS and Android, providing an interface that makes it easy for users to operate and centralize their medical history directly on their mobile device.

The result is an operational prototype, which meets the proposed objectives, and that could serve as a basis for the future development of a system that can, for example, be used in institutions where this need might apply to a group of people (such as a nursing home), and where, once implemented, it may prove to be an important tool in this area and whose use can translate into real added value to the various sectors of activity in which it might be put to use.

Keywords: Embedded Systems, Vital Signs, Wireless Communication

Índice

1 Introdução 1

1.1 Enquadramento 1

1.2 Objetivos 3

1.3 Recursos necessários 4

2 Bases Teóricas 7

2.1 A plataforma ESP32 7

2.1.1 NodeMCU-32S 8

2.2 Esfigmomanómetro Digital 10

2.3 Oxímetro de Dedo 11

2.4 Tecnologias de Comunicação de Dados 12

2.4.1 I2C 12

2.4.2 BLE 13

2.4.3 Wi-Fi 13

2.5 Tecnologias e Interfaces de Programação 13

2.5.1 Flutter 13

2.5.2 Dart 13

2.5.3 Arduino IDE 14

2.5.4 Android Studio 14

2.5.5 Firebase 14

3 Desenvolvimento 15

3.1 Comunicação ESP32 – Esfigmomanómetro Digital 15

3.1.1 Controlo do Esfigmomanómetro 19

3.2 Comunicação ESP32 – Oxímetro de Dedo *Bluetooth* 20

3.3 Comunicação ESP32 – Aplicação Móvel 22

3.4 Alimentação 23

3.4.1 ESP32 – Modo de Poupança de Bateria 24

3.5 Flutter – Aplicação Móvel 26

3.6 Comunicação Aplicação Móvel - Firebase 27

4 Resultados obtidos 29

4.1 Configuração Final 29

4.2 Experiência do Utilizador 30

4.3 Custo Estimado 37

5 Conclusões 39

5.1 Análise dos Resultados Obtidos 39

5.2 Desenvolvimentos Futuros 39

Referências 41

Anexos 43

Lista de Figuras

Figura 1 – Esfigmomanómetro Digital 2
Figura 2 – Oxímetro de Dedo 2
Figura 3 – Módulo ESP32 3
Figura 4 - Diagrama de Componentes Internos do ESP32 7
Figura 5 - ESP32 com Suporte para Bateria 10
Figura 6 – Exemplo de Comunicação I2C 12
Figura 7 – Interior Esfigmomanómetro Digital Omron M2 15
Figura 8 – Diagrama de Blocos EEPROM Rohm BR24G08-3 16
Figura 9 – Localização EEPROM 16
Figura 10 – Resultado Scan Endereço I2C 17
Figura 11 – Botão *Start* do Esfigmomanómetro 19
Figura 12 – Circuito de Controlo do Botão *Start* do Esfigmomanómetro 19
Figura 13 – Oxímetro de Dedo Bluetooth Jumper JPD-500G 20
Figura 14 – Identificação IDs de Serviço e de Característica Oxímetro de Dedo 20
Figura 15 – Definição IDs de Serviço e de Característica Oxímetro de Dedo 21
Figura 16 – Identificação de Dados de Medições Transmitidos via BLE 21
Figura 17 – Definição Credencias Acesso *Wi-Fi* ESP32 22
Figura 18 – Definição de Strings Dados *Wi-Fi* 22
Figura 19 – Bateria Samsung IN18650-35E 23
Figura 20 – Circuito Controlo Botão e LED Deep Sleep 25
Figura 21 – Código para Upload de Dados para Firebase 27
Figura 22 – Dados Registados no Firebase 28
Figura 23 – Código para Obtenção de Dados do Firebase 28
Figura 24 – Código para Eliminar Dados do Firebase 28
Figura 25 – Diagrama do Sistema 29
Figura 26 – Ecrã de Lançamento da Aplicação 30
Figura 27 – Pedido de Conexão à Rede do ESP32 para Configuração *Wi-Fi* 31
Figura 28 – Página Principal da Aplicação 31
Figura 29 – Lista de Redes *Wi-Fi* Detetadas 32
Figura 30 – Pedido de Introdução da Password da Rede *Wi-Fi* 32
Figura 31 – Pedido de Conexão à Rede *Wi-Fi* Configurada 33
Figura 32 – Menu da Aplicação 34
Figura 33 – Página de Ajuda da Aplicação 34
Figura 34 – Seleção de Dispositivos para Recolha de Dados 35
Figura 35 – Pop-Up com Instruções Rápidas de Utilização 35
Figura 36 – Loading Icon Durante Recolha de Dados 36
Figura 37 – Página de Apresentação de Resultados 36
Figura 38 – Historial de Medições 37
Figura 39 – Configuração ESP32 Slave I2C 43
Figura 40 – Função de Processamento de Dados I2C 44

Figura 41 – Código Client Callback BLE 44
Figura 42 – Código Pesquisa de Servidor BLE 45
Figura 43 – Código de Conexão ao Servidor BLE 45
Figura 44 – Função de Scan Wi-Fi 46
Figura 45 – Função de Leitura da EEPROM 46
Figura 46 – Função de Escrita na EEPROM 47
Figura 47 – Função de Eliminação de Dados da EEPROM 47
Figura 48 – Função de Processamento de Mensagens Recebidas 48
Figura 49 – Função de Execução de Comandos – Parte 1 49
Figura 50 – Função de Execução de Comandos – Parte 2 50
Figura 51 – Função de Gestão do Modo de Poupança de Bateria 51
Figura 52 – Widget de Botões e Switches da Página Principal – Parte 1 52
Figura 53 – Widget de Botões e Switches da Página Principal – Parte 2 53
Figura 54 – Widget de Botões e Switches da Página Principal – Parte 3 54
Figura 55 – Função de Configuração de Switches 54
Figura 56 – Função de Configuração de Botões 55
Figura 57 – Função de Gestão de Estados da Aplicação – Parte 1 55
Figura 58 – Função de Gestão de Estados da Aplicação – Parte 2 56
Figura 59 – Função de Envio de Comandos para ESP32 57
Figura 60 – Função de Verificação da Conexão Wi-Fi 57
Figura 61 – Função para Conexão Wi-Fi ao ESP32 58
Figura 62 – Função de Processamento de Dados 59
Figura 63 – Função de Escolha de Layout da Página de Resultados 59
Figura 64 – Código de Verificação da Ligação ao ESP32 60

Lista de Tabelas

Tabela 1 – Componentes Utilizados 5
Tabela 2 – Definições e Classificações dos Níveis da Pressão Arterial (mmHg) 11
Tabela 3 – Ligações EEPROM-ESP32 17
Tabela 4 – Medições de Teste Realizadas 18
Tabela 5 – Dados Transmitidos via I2C 18
Tabela 6 – IDs de Serviço e de Característica Oxímetro de Dedo 21
Tabela 7 – Modos de Poupança Bateria ESP32 24
Tabela 8 – Análise de Custo dos Componentes 37

Acrónimos e Símbolos

Lista de Acrónimos

BLE	*Bluetooth Low Energy*
BPM	Batidas por Minuto
EEPROM	*Electrically-Erasable Programmable Read-Only Memory*
GPIO	*General Purpose Input/Output*
I2C	*Inter-Integrated Circuit*
ID	Identificador
IDE	*Integrated Development Environment*
IoT	*Internet of Things*
LED	*Light Emitting Diode*
PA	Pressão Arterial
SCL	*Serial Clock*
SDA	*Serial Data*
SpO2	Saturação do Oxigénio no Sangue

1 Introdução

Um estudo realizado em 2018 estimava o valor global do mercado da electromedicina em cerca de 425,5 mil milhões de dólares, sendo previsto o aumento progressivo deste valor até 2025, ano em que o mesmo se cifraria em 612,7 mil milhões de dólares (Fortune Business Insights, 2019). Este número permite-nos comprovar a extrema, e crescente, competitividade deste mercado, não surpreendendo assim que, na procura de tentar responder às mais diversas necessidades, exista também cada vez mais uma enorme diversidade de soluções disponíveis.

Na área do diagnóstico médico em específico, a monitorização regular de sinais vitais é um aspeto crucial no acompanhamento do estado de saúde de um indivíduo, sendo muitas vezes as alterações nestes parâmetros a funcionarem como o primeiro sinal de alarme, identificativo de potenciais problemas de saúde.

Se em ambiente clínico e hospitalar, onde este controlo é fundamental, nomeadamente em situações de internamento, este acompanhamento pode facilmente ser feito com recurso a equipamentos como monitores de sinais vitais, cujo custo é elevado, existem outras situações em que esta necessidade também se pode verificar de forma regular ou pontual, mas em que esse nível de investimento não se justifica. Assim, este processo é normalmente feito com recurso a equipamentos de custo bastante mais reduzido, que são capazes de providenciar leituras suficientemente fiáveis, mas que operam de forma independente uns dos outros, tornando este processo mais moroso, ineficiente e negligenciando a possibilidade de estabelecer um historial clínico do paciente que possa ser útil numa posterior avaliação do estado de saúde do mesmo.

Assim, o desenvolvimento de uma alternativa que possa permitir contornar estes obstáculos, criando uma solução compacta, com o maior nível de portabilidade possível e que permita centralizar todos os dados recolhidos a um custo razoável, poderia revelar-se uma clara mais-valia, com aplicação prática em diversos ambientes.

1.1 Enquadramento

Conforme já referido, em situações em que exista a necessidade de fazer o controlo dos sinais vitais de um indivíduo de forma periódica ou ocasional, é normal que muitas vezes se recorra a equipamentos de uso mais corrente e de valor mais reduzido. Embora a fiabilidade destes equipamentos seja por norma elevada, sendo inclusivamente utilizados por profissionais de saúde no exercício das suas funções, estes destinam-se simplesmente à verificação pontual dos sinais vitais do indivíduo em causa, não permitindo muitas vezes que seja estabelecido um historial de medições do paciente, que permita traçar uma evolução do mesmo ao longo do tempo e assim estabelecer bases de comparação para os valores registados aquando da leitura. Na figura 1 é possível observar um exemplo de um destes equipamentos, no caso um

Esfigmomanómetro Digital de Braço, o qual tem como objetivo primordial a medição da pressão arterial de um indivíduo.

Figura 1 – Esfigmomanómetro Digital (Omron, 2024)

Quando é necessário medir outro tipo de parâmetros, como os valores de SpO2 (saturação do oxigénio no sangue) é comum ver-se em utilização equipamentos designados de Oxímetros de Dedo, como o apresentado na figura 2.

Figura 2 – Oxímetro de Dedo (Jumper, 2024)

Embora a maior parte destes equipamentos possua uma memória que permite registar os valores das medições mais recentemente efetuadas, a verdade é que se torna impossível estabelecer um registo centralizado destas medições. Assim, a capacidade de registo de exames ficará sempre limitada à capacidade do próprio dispositivo, não existindo a comodidade que poderia ser providenciada pela possibilidade de acesso ao historial de medições de forma remota, através de outro dispositivo.

Relativamente ao projeto a desenvolver, a opção tomada em relação à execução do mesmo passa pela utilização de equipamentos e sensores já existentes no mercado e habitualmente utilizados na indústria médica, procedendo, se necessário, à sua adaptação de forma a que se incorporem da melhor forma na solução idealizada. Esta opção, ao invés da utilização de outro

tipo de sensores mais básicos, prende-se com a tentativa de garantir a obtenção de leituras com a maior fiabilidade possível.

A servir de base a todo o projeto estará um módulo ESP32 (Espressif, 2020), apresentado na figura 3 que pelo seu tamanho e custo reduzidos e pelo facto de já ter em si incorporadas as capacidades de *Bluetooth* (Bluetooth Special Interest Group, 2024) e *Wi-Fi* (Wi-Fi Alliance, 2024), o tornam no dispositivo ideal para estabelecer a interface com os diversos componentes do sistema.

Figura 3 – Módulo ESP32 (Espressif, 2020)

No que diz respeito ao desenvolvimento da aplicação móvel, que permita o controlo e gestão de todo o sistema, este será feito com recurso à plataforma Flutter (Flutter, 2024), da Google, a qual, com recurso à linguagem de programação Dart (Google, 2024), possibilitará que o código desenvolvido permita a criação de uma aplicação que seja compatível tanto com dispositivos com sistema operativo Android (Android, 2024) como iOS (Apple Inc., 2024), mantendo assim a sua performance nativa em ambas as plataformas.

Finalmente, e perante a necessidade de recorrer a uma plataforma online na qual possam ser alojados os dados recolhidos, a opção recai no Firebase (Google, 2024), também da Google. Para além de tanto o Firebase como o Flutter possuírem uma grande base de apoio e informação online, o facto de ambos serem distribuídos pela mesma empresa facilita a compatibilização entre a aplicação móvel e a plataforma de *hardware*, facilitando todo o processo de desenvolvimento.

1.2 Objetivos

Pelos motivos descritos anteriormente, tornam-se claros os problemas e inconvenientes que a utilização dos dispositivos médicos disponíveis no mercado podem trazer. Não só a sua utilização é feita de forma independente como, muitas vezes, os próprios dispositivos nem sequer possibilitam o armazenamento, devidamente identificado, das informações recolhidas. Desta forma, um eventual desejo de estabelecer um historial de registos de um dado indivíduo obrigaria a que estes dados fossem guardados posteriormente de forma manual, o que muitas vezes acaba por nem sequer suceder, seja por falta de disponibilidade ou simplesmente por desleixo.

Surge assim a necessidade de criação de um sistema que possa colmatar estas deficiências e que permita a centralização, num único dispositivo, do processo de registo de sinais vitais de um individuo, tendo especialmente em vista situações em que exista a necessidade de fazer estas medições de forma regular. Pretende-se assim:

• Proceder à interface de um microcontrolador ESP32, o qual será a base de todo o sistema, com alguns equipamentos e sensores médicos já existentes no mercado, designadamente um monitor digital de pressão arterial (esfigmomanómetro digital) e um sensor de SpO2.

• Desenvolver uma aplicação móvel que permita o controlo e gestão do referido sistema.

• Possibilitar o envio dos dados registados para uma plataforma online, de forma a que todo o histórico de medições possa ficar disponível para consulta posterior.

Uma vez concretizados estes objetivos, espera-se que o protótipo desenvolvido possa cumprir com diversos requisitos, nomeadamente:

• Que todo o processo de medição possa ser claro e simples.

• Garantir a realização de medições precisas e fiáveis.

• Que possa ser uma solução de baixo custo, quando comparada com outras opções utilizadas em ambiente hospitalar.

• Garantir a compatibilidade da aplicação desenvolvida tanto com dispositivos Android como iOS.

Espera-se assim que esta solução, uma vez implementada, possa vir a revelar-se um importante instrumento no auxílio ao acompanhamento regular dos sinais vitais de uma ou mais pessoas e a que sua utilização possa traduzir-se em reais mais-valias para os diversos setores de atividade em que o mesmo possa ser colocado em utilização.

1.3 Recursos necessários

Para além dos já referidos ESP32, Esfigmomanómetro Digital e Oxímetro de Dedo são também necessários outros equipamentos e componentes que auxiliarão todo o processo de desenvolvimento deste projeto, tais como multímetro, fontes de tensão e ferro de soldar.

A nível de software será utilizada a IDE do Arduíno (Arduino, 2024) para a implementação de todo o código do ESP32, o Flutter, através do Android Studio (Android Developers, 2021), enquanto plataforma para o desenvolvimento da aplicação móvel compatível tanto com Android como com iOS e o Firebase enquanto plataforma para onde serão enviados e armazenados os diversos dados gerados pelo sistema. A utilização das ferramentas

mencionadas no projeto é justificada pelas suas características específicas que atendem às necessidades técnicas e funcionais do sistema desenvolvido.

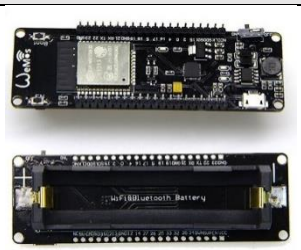
A IDE do Arduíno foi escolhida para a programação do ESP32 devido à sua compatibilidade direta com este microcontrolador, além de oferecer uma interface simples e acessível. Esta ferramenta dispõe de uma vasta biblioteca de recursos e uma comunidade ativa, facilitando o desenvolvimento e otimização do código.

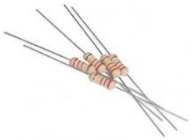
O Flutter foi utilizado como a principal tecnologia para o desenvolvimento da aplicação móvel, pois oferece um *framework* robusto e versátil para criar aplicações nativas tanto para Android quanto para iOS. O uso do Android Studio como ambiente de desenvolvimento integra-se perfeitamente no Flutter, fornecendo um conjunto de ferramentas para teste e *design* de interfaces essenciais para o desenvolvimento de projeto.

O Firebase foi selecionado como a plataforma de *backend* devido à sua capacidade de armazenamento e sincronização de dados em tempo real, uma funcionalidade essencial para garantir que o histórico de medições esteja sempre acessível. Além disso, o Firebase oferece integração nativa com o Flutter, o que simplifica a implementação das funções de envio e consulta de dados.

Em termos de *hardware*, na Tabela 1 é possível observar uma lista dos componentes utilizados na execução deste projeto.

Tabela 1 – Componentes Utilizados

Imagem	Designação	Quantidade
	Módulo ESP32 c/ Suporte para Bateria 18650	1
	Esfigmomanómetro Digital de Braço	1

	Oxímetro de Dedo com *Bluetooth*	1
	Bateria 18650 *Li-ion* 3,7 V 3500 mAh	1
	Botão	1
	LED Verde	1
	Transístor NPN 2N2222	1
	Relé 5 V	1
	Díodo 1N4007	1
	Resistência 220 Ω	1
	Resistência 1k Ω	1
	Resistência 10k Ω	1

2 Bases Teóricas

São aqui abordadas as bases teóricas nas quais as tecnologias utilizadas na elaboração deste projeto se baseiam e que acabaram por servir de suporte à sua escolha para a execução do mesmo.

2.1 A plataforma ESP32

Essa arquitetura permite que possa ser programado de forma independente, sendo um dispositivo amplamente reconhecido por diversas das suas características, como o seu baixo consumo de energia, alto desempenho, amplificador de baixo ruído, robustez, versatilidade e confiabilidade (Oliveira, 2017). Suporta também diversas interfaces de comunicação, tais como os relativamente comuns SPI, UART e I2C, para além de Infravermelho (IR), SDIO, CAN, *Ethernet*, DAC, Sensor de Toque e I2S (Espressif, 2020). Na imagem seguinte é apresentado um diagrama de componentes internos do ESP32.

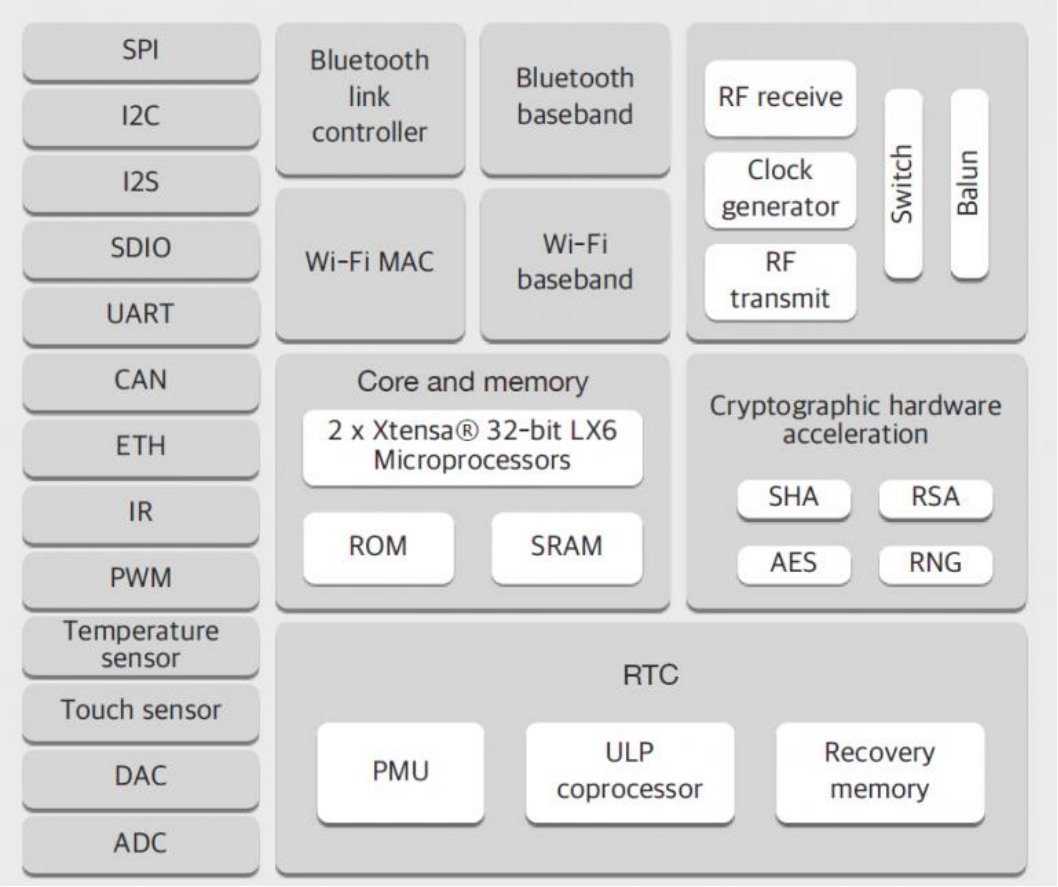


Figura 4 - Diagrama de Componentes Internos do ESP32 *(Espressif, 2020)*

Para além disso, a inclusão de um coprocessador de ultrabaixo consumo, no qual é possível delegar algumas tarefas de simples execução, permite que os processadores principais possam ser colocados em modos de poupança de energia, com vista a uma maior economia a este nível.

Conforme já referido, uma das grandes vantagens do ESP32 é facto de já possuir também *Wi-Fi* e *Bluetooth* v4.2, com suporte para BLE (*Bluetooth Low Energy*), um modo de transmissão de dados de baixo consumo, dispensando-se assim o uso de outros componentes que lhe possam conferir estas potencialidades.

Em termos de plataformas de desenvolvimento, o ESP32 pode ser programado com recurso à sua plataforma própria, desenvolvida pela Espressif. Contudo, uma das suas maiores vantagens é o facto de a sua programação também poder ser efetuada com recurso ao *software* Arduíno IDE, o que o torna incrivelmente versátil e popular.

2.1.1 NodeMCU-32S

Importa referir a existência de uma plataforma baseada no ESP32, amplamente utilizada no desenvolvimento de projetos IoT, o NodeMCU-32S (Ai-Thinker, 2020), o qual conta uma porta micro USB para alimentação e programação e é controlado por um módulo ESP-WROOM-32 (Espressif Systems, 2022). Além do ESP32, o ESP-WROOM-32 possui também um cristal de 40 MHz, memória flash integrada de 4 MB, antena embutida e blindagem contra EMI (Oliveira, 2017).

Abaixo temos as principais características do NodeMCU-32S:

– Baseado no SoC (*System on Chip*) ESP32-D0WDQ6

– Módulo controlador ESP-WROOM-32

– Microprocessador dual core Tensilica Xtensa 32-bit LX6

– *Clock* ajustável de 80 MHz até 240 MHz

– Desempenho de até 600 DMIPS

– ROM de 448 kB

– SRAM de 520 kB

– RTC *Slow* SRAM de 8 kB

– RTC *Fast* SRAM de 8 kB

– Memória flash externa de 32 Mb (4 *megabytes*)

– Opera na faixa de 2,2 V – 3,6 V DC

– Pode ser alimentado com 5 VDC através do conector micro USB

– Opera em nível lógico 3,3 V (não tolerante a 5 V)

– Opera com corrente típica de 80 mA

– Corrente máxima por pino é de 12 mA (recomenda-se usar 6 mA)

– Possui Interfaces de GPIO / Sensores capacitivos / ADC / DAC / LNA pré amplificado / CAN

– Possui 36 GPIOs (algumas versões possuem 30 pinos)

– GPIOs com função PWM / I2C e SPI

– ADC (conversor analógico digital) de 18 canais com resolução de 12 *bits*

– 2 DAC (conversor digital analógico) com resolução de 8 *bits*

– Suporte a redes *Wi-Fi* padrão 802,11 b/g/n

– O *Wi-Fi* opera na faixa de 2,4 a 2,5GHz

– *Wi-Fi* possui opções de segurança WPA / WPA2 / WPA2-Enterprise / WPS

– *Wi-Fi* possui opções de criptografia AES / RSA / ECC / SHA

– Opera nos modos *Station* / SoftAP / SoftAP + *Station*/ P2P

– Antena integrada

– *Bluetooth* v4.2 BR / EDR e BLE (*Bluetooth Low Energy*)

– Opera em temperaturas na faixa de -40º a 85ºC

– Programável via USB, *host* ou *Wi-Fi* (OTA / *Over The Air*)

– Compatível com a IDE do Arduíno

– Compatível com módulos e sensores utilizados no Arduíno

Importa também referir que existem diversas variantes e adaptações destes módulos no mercado. No caso específico deste projeto, e perante a necessidade da existência de uma fonte de energia que pudesse ser responsável pela alimentação do ESP32, a opção passou por fazer uso de um módulo já preparado para a utilização de uma bateria *Li-ion* modelo 18650, o qual é apresentando na figura 5 e que permite que o ESP32 possa ser alimentado de forma independente.

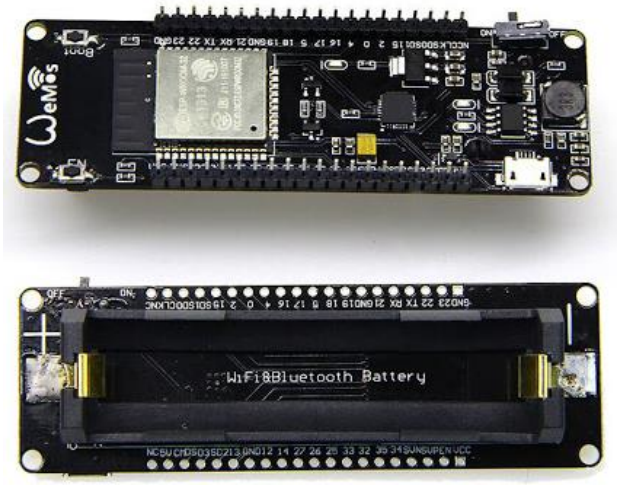


Figura 5 - ESP32 com Suporte para Bateria

2.2 Esfigmomanómetro Digital

No âmbito clínico, um dos parâmetros mais controlados com vista à aferição do estado de saúde de um indivíduo é da tensão arterial. Para tal é feito uso de equipamentos designados de Esfigmomanómetros ou Monitores de Tensão Arterial. Estes equipamentos, nomeadamente nas versões digitais, permitem um controlo simples e rápido de parâmetros como a Pressão Arterial (PA), a força com que o sangue circula pelo interior das artérias no corpo, sendo que a mesma é medida através de duas componentes: a pressão arterial sistólica ou "máxima" e a pressão arterial diastólica ou "mínima". A primeira corresponde ao momento em que o coração contrai, enviando o sangue para todo o corpo. A segunda ocorre quando o coração relaxa para se voltar a encher de sangue (Sociedade Portuguesa de Hipertensão, 2024).

Na tabela abaixo é possível observar valores que podem servir de referência para o controlo destes parâmetros e que auxiliam a estabelecer intervalos indicativos de potenciais problemas de saúde de um indivíduo (Sociedade Portuguesa de Hipertensão, 2024).

Tabela 2 – Definições e Classificações dos Níveis da Pressão Arterial (mmHg) (Sociedade Portuguesa de Hipertensão, 2024)

Definições e classificações dos níveis da pressão arterial medidos no consultório (mmHg)			
CATEGORIA	SISTÓLICA		DIASTÓLICA
Ótima	< 120	e	< 80
Normal	120-129	e/ou	80-84
Normal alta	130-139	e/ou	85-89
Hipertensão grau 1	140-159	e/ou	90-99
Hipertensão grau 2	160-179	e/ou	100-109
Hipertensão grau 3	≥ 180	e/ou	≥ 110
Hipertensão sistólica isolada	≥ 140	e	< 90

2.3 Oxímetro de Dedo

Um outro parâmetro igualmente bastante controlado com vista a uma avaliação clínica de um indivíduo é da saturação de oxigénio no sangue (SpO2), ou seja, a percentagem de oxigénio que se encontra a ser transportada na circulação sanguínea, sendo que, com vista à obtenção destes parâmetros recorre-se a equipamentos designados de Oxímetros.

De um modo geral, um nível de SpO2 baixo, inferior a 95%, poderá ser indicativo de que os pulmões não estão a funcionar de forma correta e que os mesmos poderão não se estar a revelar capazes de efetuar as trocas gasosas necessárias. Embora um valor inferior ao referido possa não ser obrigatoriamente sinal de um problema grave, já que quadros de gripe ou constipação poderão fazer com que este valor possa baixar ligeiramente sem que exista motivos de preocupação, este tipo de exames pode ser particularmente importante no despiste de doenças pulmonares, cardíacas ou neurológicas, por exemplo, especialmente em situações em que o mesmo possa ser inferior a 90% (Reis, 2021).

Um oxímetro de dedo permite assim fazer esta avaliação de forma rápida e não invasiva, ou seja, evitando a necessidade de fazer recolha de sangue, bastando colocar este dispositivo em contacto com a pele do paciente, geralmente na ponta do dedo do mesmo. Desta forma, e através de um sensor ótico, este dispositivo poderá medir a quantidade de oxigénio no sangue que passa nas artérias por baixo do local onde está a ser feito o exame, indicando o valor lido de forma bastante célere (Reis, 2021).

Alguns destes dispositivos permitem também fazer um controlo de outros parâmetros vitais, como por exemplo a Frequência Cardíaca ou o Índice de Perfusão, valor numérico que indica a intensidade da pulsação no ponto no qual está posicionado o sensor (CPAPS, 2021).

2.4 Tecnologias de Comunicação de Dados

De forma a garantir a comunicação e consequente transmissão de dados entre os vários dispositivos foram utilizadas diversas tecnologias, as quais são neste capítulo apresentadas.

2.4.1 I2C

O protocolo I2C (Inter-Integrated Circuit) é um protocolo de comunicação série implementado apenas com duas linhas (Philips Semiconductors, 2000):

SDA – Serial DAta

SCL – Serial CLock

A existência duma só linha de dados impede que tal como ocorre no protocolo SPI a comunicação seja *full-duplex*, ou seja que se possa realizar em simultâneo o envio e a receção de dados (Mendonça, 2024).

Como a figura seguinte mostra, todos os dispositivos ficam "pendurados" nesse barramento. Não existem quaisquer linhas de seleção pelo que é agora outro o mecanismo de escolha do *slave* com que o master pretende comunicar. Isso é feito na forma dum endereçamento por *software*, ou seja, cada *slave* terá um endereço de 7 *bits* a ser usado quando do estabelecimento da comunicação.

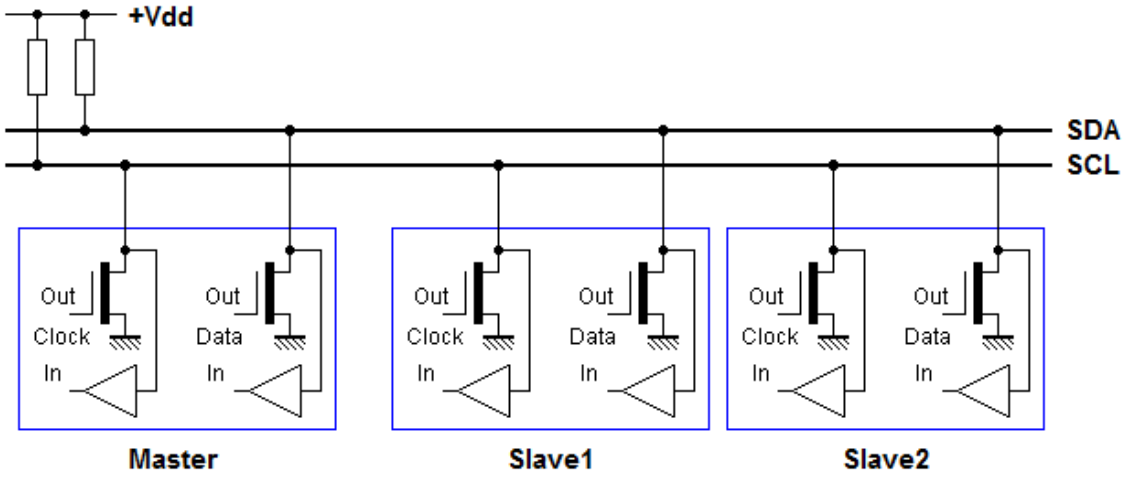


Figura 6 – Exemplo de Comunicação I2C (Mendonça, 2024)

De notar que todas as ligações ao barramento são realizadas por saídas em dreno-aberto. Caso os transístores estejam ao corte as linhas são mantidas a 3,3V (+Vdd) em virtude das resistências de *pull-up*. Quando um dos transístores conduz, a respetiva linha é "puxada" para os 0V (GND). Dessa forma nunca existirão curto-circuitos nas linhas mesmo que ocorram acessos simultâneos (Mendonça, 2024).

2.4.2 BLE

No âmbito deste projeto é importante também entender o conceito de BLE (Random Nerd Tutorials, 2019) já que será esta tecnologia, incorporada no Oxímetro de Dedo a ser utilizado, que nos permitirá efetuar a comunicação com o mesmo com vista à recolha e posterior tratamento dos valores lidos.

Projetado especificamente para facilitar o desenvolvimento de aplicações IoT, BLE (*Bluetooth Low Energy*) é um tipo de *Bluetooth* com um consumo energético bastante inferior ao designado *Bluetooth* "clássico", cerca de 10% quando comparado com este último. Operando igualmente numa frequência de 2,4 GHz, o BLE é apenas ativado quando o mesmo é necessário para efetuar alguma ação, permitindo facilmente conexões de curto e médio alcance (Random Nerd Tutorials, 2019).

2.4.3 Wi-Fi

Wi-Fi (Cisco, 2024) é uma tecnologia de rede sem fio que permite que dispositivos como computadores, dispositivos móveis e outros equipamentos, entre outros, se conectem à *Internet* e possam trocar informações entre si, criando uma rede, através da utilização de ondas de rádio e não requerendo uma conexão física entre o remetente e o recetor.

2.5 Tecnologias e Interfaces de Programação

2.5.1 Flutter

Flutter (Flutter, 2024)é uma *framework* criada pela Google para desenvolver aplicações *cross-platform* com apenas um código-fonte e de forma nativa, utilizando a linguagem Dart e permitindo a criação de interfaces com o utilizador com recurso a um conceito baseado em *Widgets*.

De forma simplificada, podemos dizer que no Flutter tudo são *Widgets*, seja um texto, um botão ou uma imagem, por exemplo, e que é à base destes que todo o processo de desenvolvimento se assenta.

2.5.2 Dart

Criada em 2011, Dart (Google, 2024) é uma linguagem de programação desenvolvida pela Google e utilizada não só para programar aplicações Flutter mas podendo também ser usada de forma isolada para desenvolvimento de *software*, sendo uma linguagem orientada a objetos muito simples de usar e aplicar.

2.5.3 Arduino IDE

Arduíno IDE (do inglês Integrated Development Environment) (Arduino, 2024) é um ambiente de desenvolvimento projetado para possibilitar a programação de placas compatíveis com o mesmo, nomeadamente Arduíno, mas também o ESP32.

2.5.4 Android Studio

O Android Studio é a IDE oficial para o desenvolvimento de aplicações Android. Baseado no IntelliJ IDEA, este *software* oferece diversos tipos de recursos, destacando-se o seu flexível sistema de compilação, baseado em Gradle, um emulador que permite proceder ao teste das aplicações desenvolvidas e à aplicação de mudanças ao código das mesmas sem necessidade de proceder ao seu reinicio, modelos de código e integração com GitHub, diversas *frameworks* ou ainda ferramentas de Lint, que auxiliam na deteção de eventuais problemas de desempenho, usabilidade ou de compatibilidade entre versões, entre outros (Android Developers, 2021).

2.5.5 Firebase

Lançado em 2011, o Firebase é um BaaS (*Backend-as-a-Service*) pertencente à Google e que possibilita diversas funcionalidades úteis para o desenvolvimento de uma aplicação. O Firebase conta um grande conjunto de ferramentas de desenvolvimento. Destas, o *Realtime Database* e o *Cloud Firestore* podem armazenar dados estruturados em documentos e sincronizar os aplicativos correspondentes em milissegundos sempre que ocorre uma transformação de dados. O Firebase suporta o desenvolvimento nas seguintes linguagens: C++, Java, Javascript, Node.js, Objective-C e Swift (Google, 2024).

3 Desenvolvimento

Neste capítulo são exploradas as diversas fases de desenvolvimento do projeto, detalhando-se todos os passos envolvidos na elaboração do mesmo, tanto a nível de *hardware* como de *software*.

3.1 Comunicação ESP32 – Esfigmomanómetro Digital

Conforme já referido, com vista a possibilitar a recolha dos parâmetros de tensão arterial de um indivíduo recorrer-se-á a um esfigmomanómetro digital. Perante a necessidade de escolher um equipamento que pudesse ser incorporado no sistema em desenvolvimento, e após uma análise de mercado, analisando parâmetros como qualidade, fiabilidade e custo, a escolha acabou por recair num Esfigmomanómetro Digital Omron M2 (Figura 1).

Já na figura 7 é possível observar o interior deste dispositivo.

Figura 7 – Interior Esfigmomanómetro Digital Omron M2

No caso dos esfigmomanómetros digitais é recorrente o uso de EEPROMs com vista ao armazenamento de algumas das medições realizadas. No caso do dispositivo selecionado para o desenvolvimento deste projeto, Omron M2, o mesmo faz uso de uma EEPROM de 8 kbit com interface I2C, Rohm BR24G08-3, componente este cujo respetivo diagrama de blocos podemos observar na imagem abaixo (ROHM, 2019).

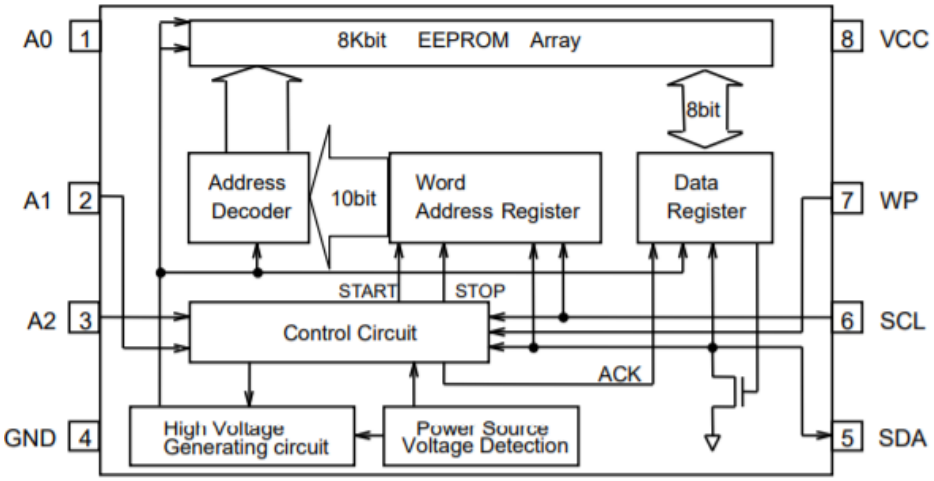


Figura 8 – Diagrama de Blocos EEPROM Rohm BR24G08-3 *(ROHM, 2019)*

A comunicação I2C permite a transmissão de dados entre diversos dispositivos através da utilização de duas linhas de comunicação: *Serial Data* (SDA) e *Serial Clock* (SCL). No caso deste componente, e conforme é possível no diagrama de blocos acima apresentado, estas duas linhas de comunicação encontram-se nos pinos 5 e 6, respetivamente.

Com vista à obtenção dos dados respeitantes às medições efetuadas por este dispositivo é então crucial a localização da EEPROM do mesmo. Na figura 9, abaixo apresentada é possível observar o interior do equipamento, com a já mencionada EEPROM assinalada.

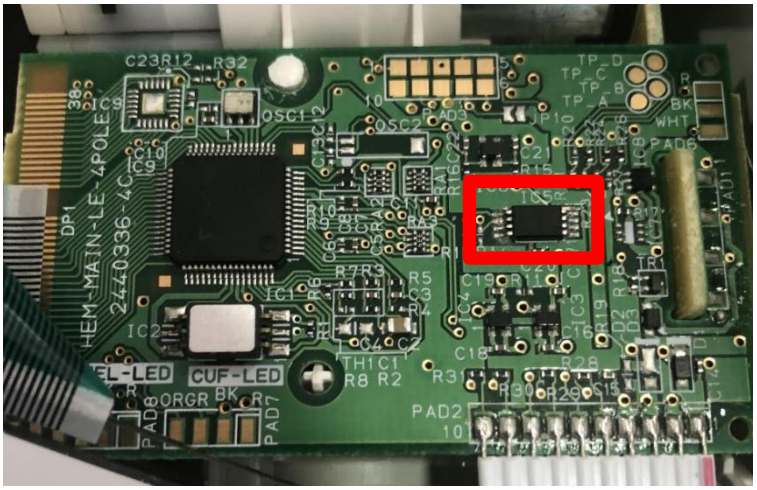


Figura 9 – Localização EEPROM

Localizada a EEPROM, importa então proceder à soldagem de alguns cabos aos pinos 4 (GND), 5 (SDA), 6 (SCL) e 8 (3.3V), conectando de seguida os mesmos ao ESP32 da seguinte forma:

Tabela 3 – Ligações EEPROM-ESP32

EEPROM	**ESP32**
4 – GND	GND
5 – SDA	GPIO 21 (SDA)
6 – SCL	GPIO 22 (SCL)
8 – 3.3V	3.3V

Numa comunicação do tipo I2C cada "escravo" (*slave*) na linha de comunicação possui o seu próprio endereço, um número hexadecimal através do qual é possível estabelecer comunicação com o mesmo. Neste caso, e não sendo essa informação disponibilizada por parte do fabricante do dispositivo, é necessário recorrer a um programa que faça o *scan* de todos os endereços I2C existentes nesta linha de comunicação, *scan* esse cujo resultado é de seguida apresentado.

```
I2C Scanner
Scanning...
I2C device found at address 0x50
I2C device found at address 0x51
I2C device found at address 0x52
I2C device found at address 0x53
done
```

Figura 10 – Resultado Scan Endereço I2C

Uma vez identificados, e correndo um programa que pudesse imprimir todos os dados transmitidos para os endereços em causa, foi possível, após alguns testes, identificar o endereço da EEPROM como sendo 0x50. Desta forma, e através da configuração do próprio ESP32 como *Slave* no referido endereço, conforme observado no código demonstrado na figura 39 dos anexos, será possível receber a informação desejada sempre que a mesma é enviada por parte do microcontrolador responsável pela operação do esfigmomanómetro.

Completado este processo, e tendo o acesso a todos os dados transmitidos no endereço em causa, torna-se então importante proceder a algumas medições de teste que nos permitam estabelecer um padrão na transmissão de dados que nos possibilite a identificação dos valores nos quais estamos interessados, mais especificamente os parâmetros de tensão arterial registados em cada medição e que são apresentados ao utilizador no display do dispositivo. Na tabela seguinte estão apresentados os valores observados em 3 medições realizadas.

Tabela 4 – Medições de Teste Realizadas

Parâmetro	**Medição 1**	**Medição 2**	**Medição 3**
Sistólica	102	109	106
Diastólica	70	71	67
BPM	114	108	112

Já na tabela abaixo é possível observar, em decimal, os primeiros *bytes* transmitidos na linha de comunicação I2C no momento da apresentação das referidas medições de teste no *display* do dispositivo.

Tabela 5 – Dados Transmitidos via I2C

Medição 1	**Medição 2**	**Medição 3**
96	96	96
96	96	96
186	186	186
77	84	81
70	71	67
114	108	112
14	14	14
32	32	32
4	4	4
186	186	186
…	…	…

Conforme é possível observar, e sabendo que valores devemos procurar, é possível detetar um padrão de transmissão de dados que nos indica onde encontrar os valores que procuramos, os quais estão assinalados a verde na tabela anterior. Os valores de Pressão Arterial Diastólica e de BPM são imediatamente identificáveis, enquanto também é possível identificar que, caso adicionemos 25 ao primeiro dos valores assinalados, o mesmo corresponderá igualmente ao valor da Pressão Arterial Sistólica.

Assim, e programando o ESP32 de forma que os 3 valores apresentados após a receção dos valores “96 96 186” possam ser atribuídos a novas variáveis, poderemos então ter acesso aos valores obtidos em cada medição, permitindo que os mesmos possam ser mais tarde enviados para a aplicação móvel, onde serão apresentados. Para tal, foi criada uma função responsável por ler os dados recebidos via I2C e identificar os valores nos quais estamos interessados, atribuindo-os a variáveis e preparando-os para que sejam enviados para a aplicação móvel, a qual é observável na Figura 40 dos anexos deste documento.

3.1.1 Controlo do Esfigmomanómetro

Na imagem abaixo podemos observar o botão responsável pelo início das medições por parte do Esfigmomanómetro Digital.

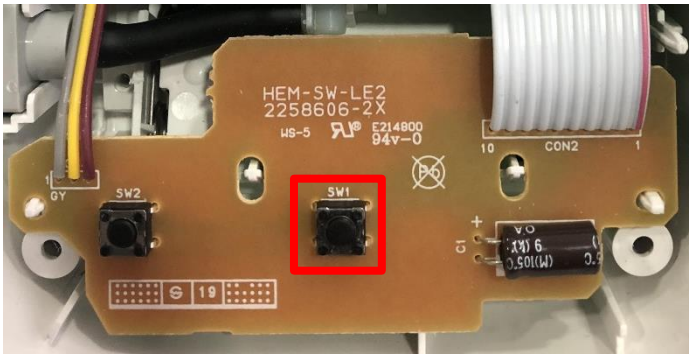


Figura 11 – Botão *Start* do Esfigmomanómetro

De forma a permitir a sua ativação pela aplicação móvel, através da qual será realizado o controlo do sistema, torna-se crucial automatizar este processo. Para tal, e recorrendo a um relé, foi desenhado e implementado o seguinte circuito:

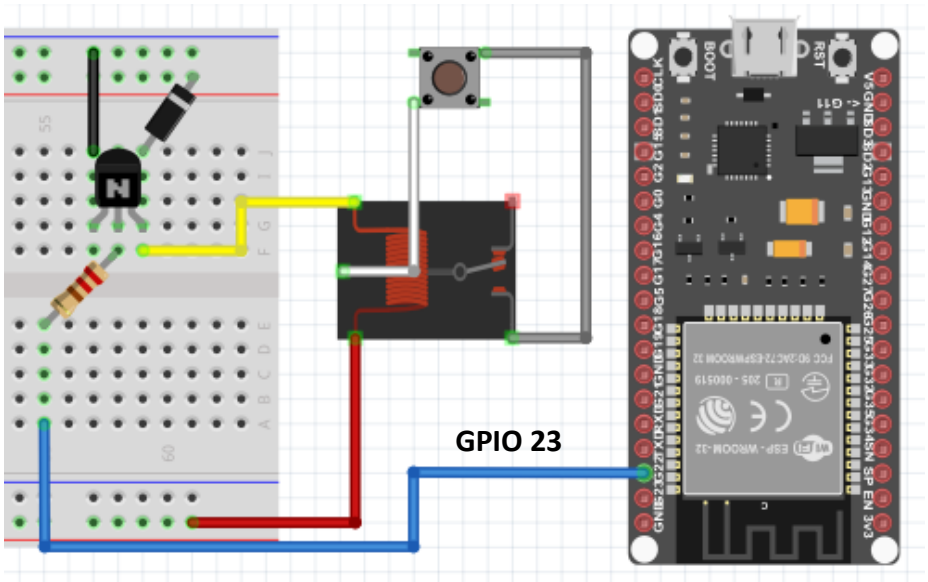


Figura 12 – Circuito de Controlo do Botão *Start* do Esfigmomanómetro

Conforme observado, o controlo deste botão é feito através da saída GPIO 23 do ESP32, permitindo que a mesma seja ativada e a medição iniciada sempre que tal comando seja enviado através da aplicação móvel, evitando que o utilizador tenha de realizar esta ação manualmente.

3.2 Comunicação ESP32 – Oxímetro de Dedo *Bluetooth*

De forma a que seja possível recolher os parâmetros de SpO2, BPM e Índice de Perfusão, surge a necessidade de recorrer a um Oxímetro de Dedo, sendo que neste caso a escolha recaiu num dispositivo Jumper JPD-500G (Jumper, 2024).

Figura 13 – Oxímetro de Dedo *Bluetooth* Jumper JPD-500G

No que diz respeito à comunicação entre o ESP32 e o Oxímetro de Dedo *Bluetooth*, a mesma será realizada através de BLE. Para tal, e de forma a estabelecer a comunicação entre os dois dispositivos, torna-se imperativo identificar quais os identificadores (ID) de Serviço e de Característica do Oxímetro de Dedo em utilização. Desta forma, recorreu-se à aplicação nRF Connect (Nordic Semiconductor, 2024) para fazer o *scan* deste dispositivo e proceder à conexão com o mesmo, com esta aplicação a apresentar-nos os dados pretendidos, conforme demonstrado na imagem abaixo.

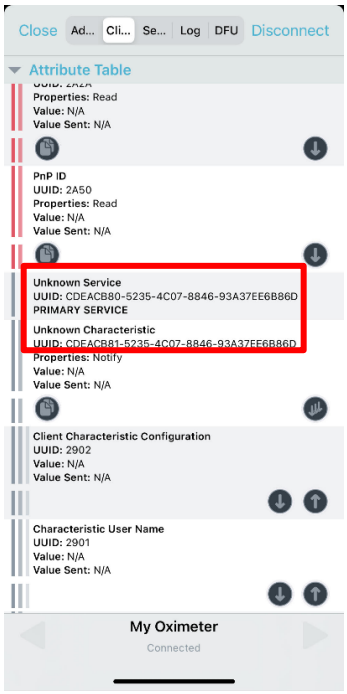


Figura 14 – Identificação IDs de Serviço e de Característica Oxímetro de Dedo

Torna-se assim possível estabelecer os IDs pretendidos.

Tabela 6 – IDs de Serviço e de Característica Oxímetro de Dedo

Tipo ID	**ID**
Serviço	CDEACB80-5235-4C07-8846-93A37EE6B86D
Característica	CDEACB81-5235-4C07-8846-93A37EE6B86D

Fazendo uso desta informação, e com recurso à biblioteca "ESP32 BLE Arduíno" (Kolban, 2017), podemos então configurar o ESP32 para se conectar enquanto cliente ao Oxímetro de Dedo *Bluetooth*. Inicialmente procedemos então à definição dos IDs acima identificados.

```
// The remote service we wish to connect to.
static BLEUUID serviceUUID("CDEACB80-5235-4C07-8846-93A37EE6B86D");
// The characteristic of the remote service we are interested in.
static BLEUUID charUUID("CDEACB81-5235-4C07-8846-93A37EE6B86D");
```

Figura 15 – Definição IDs de Serviço e de Característica Oxímetro de Dedo

A conexão é posteriormente estabelecida, inicialmente através da deteção de quando uma ligação BLE é conectada ou desconectada (código visível na figura 41 dos anexos). Assim que é detetada uma ligação deste tipo é então verificado se estes servidores BLE contêm ou não o serviço requerido e anteriormente definido (Figura 42 dos anexos). Esta pesquisa é então terminada assim que o dispositivo desejado seja encontrado, com o estabelecimento de ligação ao mesmo (Figura 43 dos anexos). Na imagem seguinte é possível observar quais os valores obtidos quando executamos uma função que proceda à impressão de todos os dados transmitidos pelo Oxímetro de Dedo, tal como é possível observar na imagem seguinte, na tentativa de identificar os valores desejados e com os quais iremos trabalhar.

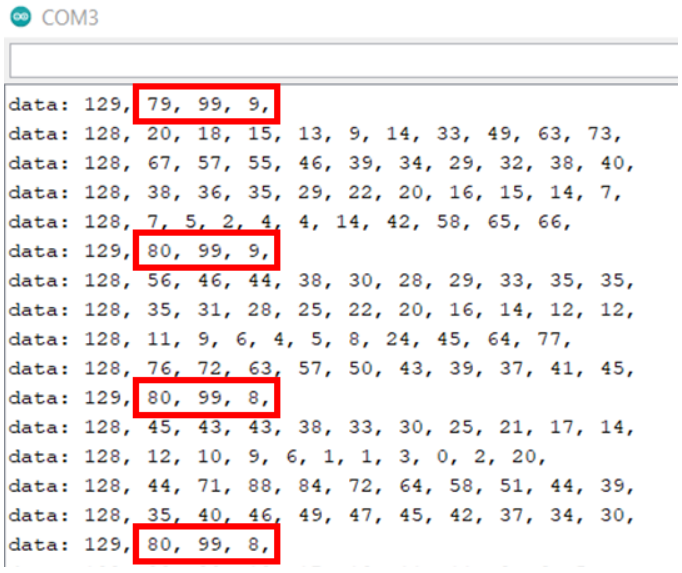


Figura 16 – Identificação de Dados de Medições Transmitidos via BLE

Uma vez mais, e por comparação dos valores acima listados com os apresentados no *display* do Oxímetro de Dedo, é possível rapidamente fazer uma associação entre os mesmos, concluindo-se que após a receção do valor "129" são de seguida apresentados os valores de BPM, SpO2 e Índice de Perfusão, respetivamente, sendo que este último se encontra multiplicado por 10.

Desta forma, e à semelhança do sucedido com o Esfigmomanómetro Digital, é-nos então possível estabelecer as condições que nos permitirão realizar a atribuição dos valores pretendidos às respetivas variáveis. Temos assim os valores lidos pelo sensor de SpO2 devidamente alocados e preparados para que possam ser enviados para a aplicação móvel.

3.3 Comunicação ESP32 – Aplicação Móvel

Conforme já referido, a comunicação entre o ESP32 e a Aplicação Móvel é efetuada com recurso a *Wi-Fi*. Importa então analisar o código desenvolvido de forma a entender melhor a forma como ambos comunicam, o tipo de informações que são trocadas durante a utilização do sistema desenvolvido e como as mesmas são geridas e armazenadas.

Quando inicializado, o ESP32 está configurado para funcionar em modo de *Access Point*, de forma a que, com vista à configuração da rede *Wi-Fi* que deverá ser utilizada pelo sistema, seja possível o acesso ao mesmo por parte do utilizador.

Uma vez feita esta conexão, será então possível que a aplicação móvel possa proceder ao envio de um pedido para o ESP32 para que este último possa realizar um *scan* das redes vizinhas disponíveis, enviando essa informação de volta para a aplicação e permitindo que a mesma seja apresentada ao utilizador. Seguidamente, e após a escolha da rede *Wi-Fi* desejada, é solicitado ao utilizador que possa proceder à introdução da respetiva *password*, sendo essas credenciais enviadas para o ESP32 de forma a que possam ser guardadas na sua EEPROM.

Nas figuras 17 e 18 é possível observar o código que define os dados de ligação ao ESP32 quando o mesmo é iniciado neste modo e também a definição das *strings* que irão ser utilizadas para comunicar com a aplicação móvel desenvolvida.

```
const char* ssid = "HEALTHAPP_SERVER";
const char* password = "12345678";
```

Figura 17 – Definição Credencias Acesso *Wi-Fi* ESP32

```
String _ssid = "";
String _pass = "";

String rcv_ssid = "";
String rcv_pass = "";
```

Figura 18 – Definição de Strings Dados *Wi-Fi*

Para tal, o sistema irá proceder ao scan de todas as redes *Wi-Fi* detetáveis pelo equipamento em que a aplicação se encontra instalada, conforme o código visível na Figura 44 dos anexos deste documento. A partir daqui o utilizador poderá então escolher a rede através da qual deseja estabelecer a comunicação. Por este motivo, neste ponto torna-se também essencial confirmar se se trata da primeira vez em que a aplicação está a ser utilizada no dispositivo em causa ou se, porventura, já terá sido uma feita configuração de rede *Wi-Fi* previamente.

Assim, foram desenvolvidas diversas funções responsáveis por proceder à:

- leitura da EEPROM para deteção de credenciais que possam estar guardadas na mesma (figura 45 dos anexos);

- escrita de novos dados na mesma (figura 46 dos anexos);

- eliminação dos dados existentes e consequente reinício do ESP32, para que seja possível efetuar uma nova configuração (figura 47 dos anexos).

Uma vez estabelecida a comunicação via *Wi-Fi* torna-se então possível gerir o comportamento do ESP32 através de diversos comandos enviados pela aplicação que definem os comandos a executar pelo mesmo, seja em termos de envio de dados para a aplicação móvel ou do início da execução de tarefas por parte do ESP32 como, por exemplo, a ativação do processo de scan de dispositivos BLE, a ativação do relé que inicia o processo de medição do Esfigmomanómetro ou a sinalização de erros caso o tempo de recolha de dados ultrapasse o definido. Os blocos de código maioritariamente responsáveis por estes processos encontram-se disponíveis para consulta nas figuras 48, 49 e 50 dos anexos.

3.4 Alimentação

Conforme já anteriormente abordado, no desenvolvimento deste projeto foi utilizado um módulo ESP32 já preparado para que a sua alimentação possa ser feita com recurso a uma bateria *Li-ion* modelo 18650, sendo que neste caso a opção recaiu numa bateria Samsung IN18650-35E (Samsung, 2015), a qual pode ser observada na seguinte imagem.

Figura 19 – Bateria Samsung IN18650-35E

As baterias de íon-lítio são termicamente estáveis e não estão sujeitas a nenhum efeito de memória sendo, para além disso, caracterizadas por uma densidade de energia particularmente alta. O modelo utilizado na concretização deste projeto, e acima apresentado, reúne as seguintes características:

- Tensão nominal: 3,6 V

- Capacidade: 3350 mAh

- Corrente de carregamento: padrão – 1700 mAh, rápido – 2000 mAh

- Corrente de descarga máxima: contínuo – 8 A, momentâneo – 13 A

- Tensão máxima carga: 4,2 V

- Tensão mínima de corte: 2,65 V

- Dimensões: Ø18,55x65,25 mm

- Peso: 50 g

3.4.1 ESP32 – Modo de Poupança de Bateria

Um aspeto crucial em qualquer sistema é a busca da maior eficiência energética possível. Para tal, revela-se importante procurar garantir que a energia disponível é gerida da melhor forma e que, quando o sistema não está em utilização, o mesmo possa entrar em modo de poupança da mesma.

Tabela 7 – Modos de Poupança Bateria ESP32 (Last Minute Engineers, 2024)

Power mode	Active	Modem-sleep	Light-sleep	Deep-sleep	Hibernation
Sleep pattern	Association sleep pattern			ULP sensor-monitored pattern	-
CPU	ON	ON	PAUSE	OFF	OFF
Wi-Fi/BT baseband and radio	ON	OFF	OFF	OFF	OFF
RTC memory and RTC peripherals	ON	ON	ON	ON	OFF
ULP co-processor	ON	ON	ON	ON/OFF	OFF

Conforme podemos observar na tabela acima, a ativação de qualquer destes modos acaba por impossibilitar completamente a utilização das capacidades de *Bluetooth* e *Wi-Fi* do dispositivo. Assim, e perante a impossibilidade de "acordar" o dispositivo via estas funcionalidades, torna-se imperativo a inclusão de um botão no sistema que permita retirar o dispositivo deste modo,

permitindo a sua utilização e conexão ao mesmo por parte do dispositivo móvel onde se encontra a aplicação.

Este botão, que se encontra ligado à saída GPIO 4 do ESP32 com uma resistência de *pull-up* externo de 1k, permite também efetuar um *reset* ao ESP32 quando pressionado por mais de 3 segundos. Na imagem abaixo é possível observar o seu esquema de ligação, assim como do LED já anteriormente mencionado, representativo do estado do sistema.

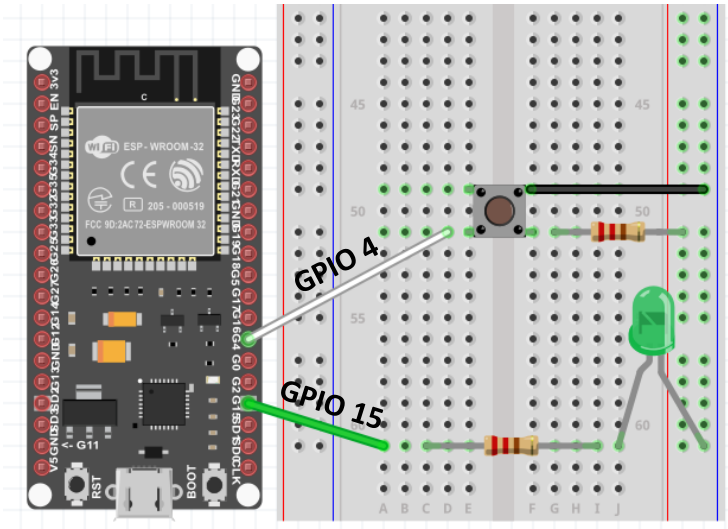


Figura 20 – Circuito Controlo Botão e LED Deep Sleep

No caso do sistema em desenvolvimento o modo de poupança de energia mais adequado acabou por se revelar o de "*Deep Sleep*". Neste modo, o CPU, a maior parte da RAM e todos os periféricos digitais são desligados sendo que as únicas partes do *chip* que se mantêm ligadas são o controlador RTC, os periféricos RTC (incluindo o coprocessador ULP) e as memórias RTC (*Slow* e *Fast*) (Random Nerd Tutorials, 2024).

A parte do código onde foram definidas as condições anteriormente referidas e que é então responsável pela execução deste modo de poupança de bateria pode ser consultada na figura 51 dos anexos.

3.5 Flutter – Aplicação Móvel

Neste capítulo é feita a análise de algumas das partes mais importantes do código desenvolvido no Flutter com vista à criação da aplicação móvel.

Conforme já referido neste documento, no Flutter todo o desenvolvimento é feito à base de *widgets*, blocos de construção fundamentais para a construção interfaces de utilização, que podem ser visíveis ou não ao utilizador. Estes podem ser simples botões, blocos de texto, imagens ou até mesmo *layouts* completos, hierarquicamente organizados de forma a dar corpo à aplicação (Macoratti, 2019). Devido à extensividade do código desenvolvido, neste capítulo serão apenas apresentados de forma geral alguns dos *widgets* mais relevantes que foram desenvolvidos no âmbito desta aplicação, sendo acompanhados de referências para a parte dos anexos onde o respetivo o código desenvolvido se encontra disponível para consulta.

Inicialmente, e como base, é necessário configurar a função responsável por diversos aspetos da página principal da aplicação. Isto inclui aspetos básicos como por exemplo a barra superior e o seu título ou a cor de fundo da página. Esta parte do código pode ser consultada nas figuras 52, 53 e 54 dos anexos. Já a definição das funções responsáveis pelo funcionamento dos diversos botões e *switches* apresentados, os quais acabam por ser a interface gráfica através da qual o utilizador poderá dar comandos ao dispositivo, encontram-se disponíveis para consulta nas figuras 55 e 56 respetivamente.

Um outro aspeto crucial é o da gestão dos estados da aplicação e consequente envio de instruções para o ESP32, dependendo de qual das diversas opções seja selecionada. Para tal foi desenvolvida a função visível nas figuras 57 e 58, a qual define também os *pop-ups* de instruções a serem apresentados antes do início de cada medição, dependendo do tipo de medição que o utilizador optou por realizar.

Dependendo desta escolha, e através da função disponibilizada na figura 59, torna-se necessário indicar ao ESP32 que dispositivos foram selecionados por parte do utilizador no momento em que o mesmo pressiona o botão "*Start"*, ação essa que dá a ordem de início da medição, permitindo que o ESP32 possa proceder à recolha dos dados pretendidos.

Outras funções, tais como a responsável por verificar o estado de conexão de *Wi-Fi*, gerando alertas para o utilizador de forma a que o mesmo possa proceder de acordo com os mesmos, e a função responsável por efetuar a ligação ao ESP32, são consultáveis nas figuras 60 e 61, respetivamente.

Já na figura 62 podemos observar a função responsável por proceder ao processamento de dados, conforme a mensagem que for recebida por parte da aplicação.

Umas das funções desenvolvidas mais relevantes é a responsável por indicar à aplicação qual o *layout* que deve ser gerado para a página de resultados, sendo a opção escolhida na figura 63 referente à obtenção de dados do Oxímetro de Dedo (BLE). Contudo, o código é similar para as restantes opções (Esfigmomanómetro (I2C) ou ambos).

Finalmente, um outro aspeto de destaque, passa por garantir que a aplicação apresenta sempre ao utilizador o verdadeiro estado da conexão. Para tal, e conforme a figura 64, foi desenvolvida uma função responsável por verificar periodicamente se a conexão ao ESP32 se encontra ou não ativa.

3.6 Comunicação Aplicação Móvel - Firebase

No final de cada medição é dada a possibilidade ao utilizador de poder gravar os dados que acaba de recolher, com vista à elaboração de um historial de medições que possa mais tarde ser consultado através da própria aplicação.

Para providenciar esta funcionalidade este sistema faz uso das potencialidades do Firebase, plataforma com a qual a aplicação móvel comunica de forma a que seja possível estabelecer o referido historial.

O Cloud Firestore é um banco de dados NoSQL orientado a documentos. Ao contrário de um banco de dados SQL, não há tabelas nem linhas. Em vez disso, os dados são armazenados em documentos, que são organizados em coleções (Cloud Firestore, 2024).

Cada documento contém um conjunto de pares chave-valor sendo necessário que todos os documentos sejam armazenados em coleções.

```
FirebaseFirestore.instance.collection("History").add({
  "Systolic": double.parse(v1).toInt(),
  "Diastolic": double.parse(v2).toInt(),
  "BPM": double.parse(v3).toInt(),
  "SPO2": double.parse(spo2).toInt(),
  "PI": double.parse(pi),
  "TimeStamp": DateTime.now()
});
```

Figura 21 – Código para *Upload* de Dados para Firebase

"History" especifica a coleção onde desejamos adicionar os dados recolhidos, sendo que a função add() irá adicionar um novo documento na forma de um mapa.

Sendo que todas as variáveis se encontravam no formato de *string*, e de forma a facilitar o seu envio para o Firebase, as mesmas são convertidas para *double* e depois para *int*, à exceção da variável PI que se mantém como *double*, de forma a facilitar o seu envio para o Firebase.

Na imagem seguinte é possível observar os dados registados no próprio Firebase após algumas medições.

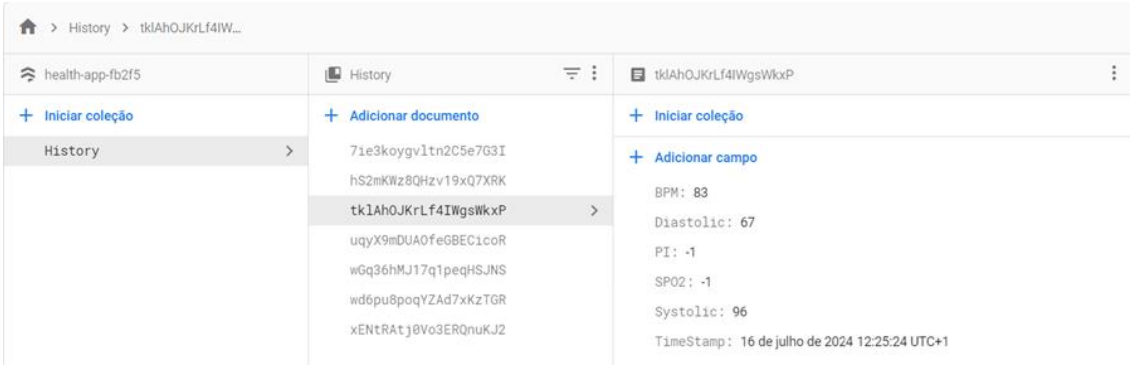


Figura 22 – Dados Registados no Firebase

Já o código de seguida apresentado é utilizado para obter os documentos dentro da coleção "*History*".

```
void _fetchDocs() {
  q = FirebaseFirestore.instance.collection("History").orderBy('TimeStamp');
  q.get().then((docs) {
    for (int i = docs.size - 1; i >= 0; i--) {
      _ls.add(docs.docs[i]);
    }
    _listFetched = true;
    setState(() {});
  });
}
```

Figura 23 – Código para Obtenção de Dados do Firebase

A variável _listFetched é utilizada como uma *flag* para verificar se os dados são obtidos ou não, sendo que os dados apenas serão apresentados ao utilizador após esta *flag* se encontrar no estado verdadeiro.

Quando desejamos eliminar uma das medições anteriormente realizadas, clicando no respetivo botão existente em cada fila da tabela onde as mesmas são apresentadas, o seguinte código será invocado, de forma à proceder à eliminação dos dados em causa da já referida coleção "*History*".

```
_ls[index].reference.delete();
_ls.removeAt(index);
```

Figura 24 – Código para Eliminar Dados do Firebase

4 Resultados obtidos

Depois de desenvolvido o sistema a importa demonstrar os resultados obtidos com o protótipo obtido. Para tal, no decorrer deste capítulo, é apresentada a forma como o utilizador irá interagir com o equipamento e aplicação desde a sua configuração inicial à obtenção das medições pretendidas.

4.1 Configuração Final

Na figura seguinte é possível observar um diagrama do sistema que nos permite melhor entender a sua configuração final, o funcionamento do mesmo e as dependências entre os seus diferentes componentes.

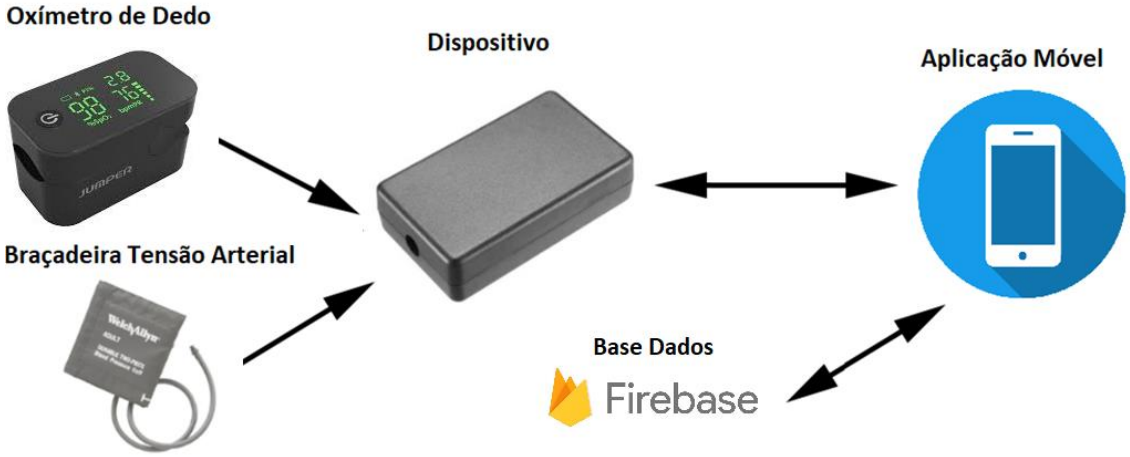


Figura 25 – Diagrama do Sistema

Na caixa apresentada está simbolizada a "central" do sistema desenvolvido, a qual é a base não só do projeto na sua iteração atual, mas que funcionará também como alicerce para desenvolvimentos futuros. Esta caixa simboliza então o encapsulamento de todo o hardware com o qual o paciente não tenha qualquer tipo de contacto o que, no estado atual do sistema, incluiria o ESP32, a sua bateria e todo o mecanismo responsável pela medição da tensão arterial que compõe o esfigmomanómetro utilizado, do qual são removidos todos os componentes que deixem de ser necessários para o seu funcionamento, como o seu *display*, já que os dados recolhidos pelo mesmo passam a ser apresentados na aplicação móvel através da qual todo o seu controlo é realizado.

Podem também ser observados os sensores/acessórios através dos quais se irá proceder à recolha de sinais vitais, níveis, bastando ao utilizador fazer a ligação dos mesmos tanto ao dispositivo como ao paciente, conforme os parâmetros a medir (Tensão Arterial, Pulsação,

SpO2 ou PI). O Firebase, interagindo com a aplicação, possibilitará o armazenamento das informações recolhidas de forma a que estas fiquem disponíveis para posterior consulta e análise de todos os dados recolhidos ao longo de tempo.

4.2 Experiência do Utilizador

Uma vez abordada toda a configuração do sistema importa então apresentar o mesmo numa perspetiva do utilizador.

De forma a que seja possível utilizar o sistema, o utilizador deverá inicialmente garantir que o mesmo se encontra ligado. Para tal, o LED verde existente no sistema deverá estar ligado. Caso contrário o botão existente junto do mesmo deverá ser pressionado de forma a que o dispositivo possa sair do seu modo de poupança de energia e ficar assim disponível para utilização.

Uma vez iniciada a aplicação, o utilizador irá ver uma página de lançamento da mesma, a qual pode ser observada na imagem abaixo e que estará visível durante alguns segundos.

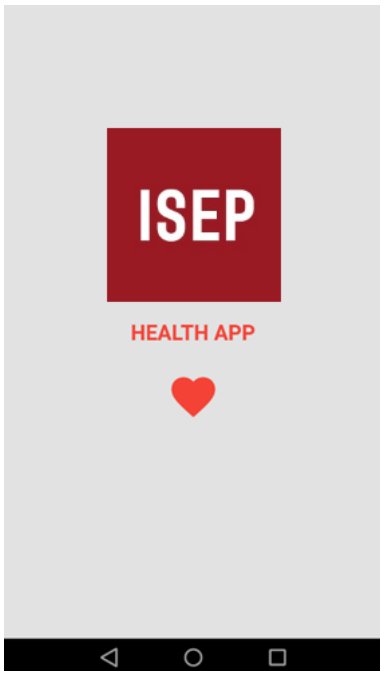


Figura 26 – Ecrã de Lançamento da Aplicação

Seguidamente, o utilizador irá ser encaminhado para a página principal da aplicação. Caso seja a primeira vez que a aplicação é iniciada, e não estando o ESP32 configurado para utilizar a mesma rede *Wi-Fi* a que utilizador está conectado, será apresentado um *pop-up* solicitando que

o dispositivo móvel possa ser ligado ao ESP32, o qual terá sido inicializado em modo de *Access Point*, de forma a proceder a esta configuração.

Figura 27 – Pedido de Conexão à Rede do ESP32 para Configuração *Wi-Fi*

Uma vez conectado ao ESP32 o utilizador poderá regressar à aplicação para proceder à configuração do mesmo para utilizar a rede *Wi-Fi* desejada pelo utilizador. Para tal, deverá selecionar a opção *Wi-Fi* observada no ecrã, permitindo que lhe seja apresentada uma lista com as diversas redes detetadas nesse momento.

Figura 28 – Página Principal da Aplicação

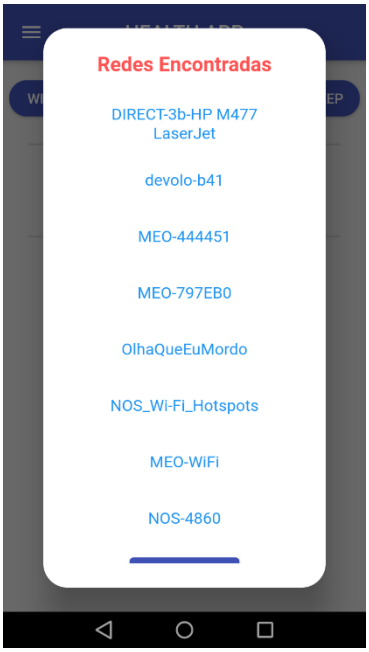


Figura 29 – Lista de Redes *Wi-Fi* Detetadas

Após selecionar a rede desejada, o utilizador deverá proceder à introdução da password da rede *Wi-Fi* em causa e gravar essa informação, o que provocará o reinício do ESP32, já configurado e com pleno acesso à rede *Wi-Fi* desejada.

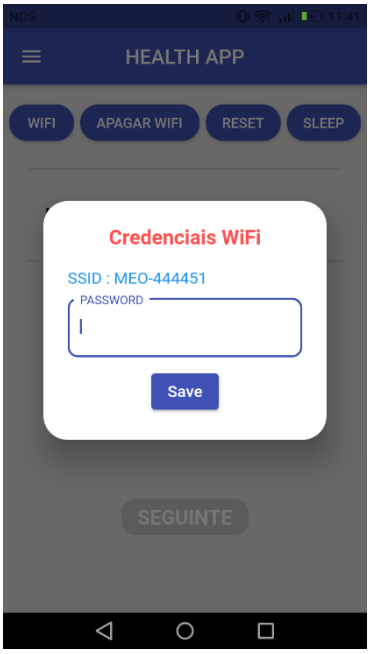


Figura 30 – Pedido de Introdução da Password da Rede *Wi-Fi*

Desta forma, da próxima vez que o utilizador iniciar a aplicação ser-lhe à pedido que conecte o dispositivo móvel à rede *Wi-Fi* para a qual o ESP32 foi configurado.

Figura 31 – Pedido de Conexão à Rede *Wi-Fi* Configurada

Uma vez realizada esta operação, então todo o processo de configuração estará terminado e todo o sistema poderá ser utilizado sem qualquer problema.

Na página principal da aplicação, e conforme observado anteriormente na figura 28, para além da já referida opção de configuração *Wi-Fi* existem várias outras funcionalidades. Uma delas é de apagar a configuração *Wi-Fi* existente, o que permitirá que o ESP32 possa assim voltar a ser iniciado em modo de *Access Point*. Para além disso é também visível uma opção de "*Reset*" que permite reinicializar o ESP32 em caso de necessidade e também um botão de "*Sleep*", que permite colocar o ESP32 em modo de poupança de bateria, conforme já anteriormente abordado.

Do lado esquerdo desta página é possível também aceder a um menu. Neste menu, para além de ser possível aceder à página que inclui o historial de medições realizadas, página essa que será abordada mais à frente, é também possível consultar uma página de ajuda (figura 33), onde estão descritos todos os passos e cuidados que o utilizador deve ter para que seja possível utilizar o sistema com sucesso.

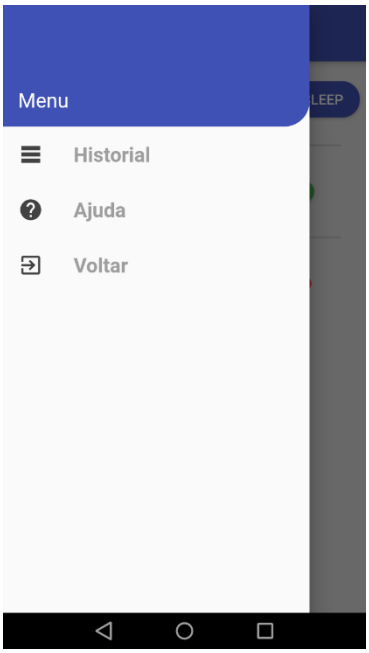


Figura 32 – Menu da Aplicação

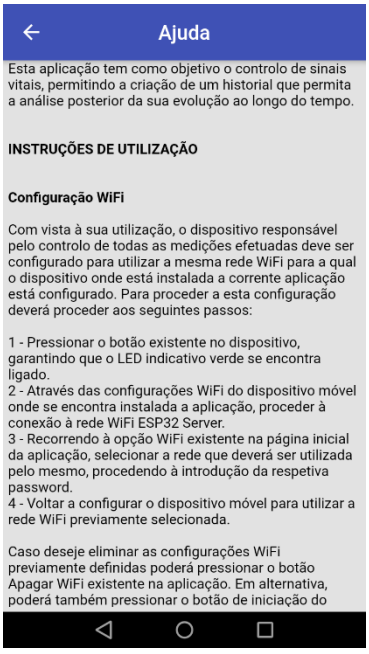


Figura 33 – Página de Ajuda da Aplicação

Para realizar uma medição, o utilizador deverá, primeiramente, garantir que se encontra conectado ao ESP32, selecionando o botão correspondente e garantindo que o mesmo se mantém verde. Uma vez confirmada esta ligação, o utilizador deverá então selecionar quais os dispositivos que serão utilizados para efetuar uma medição e pressionar o botão “Seguinte”.

Figura 34 – Seleção de Dispositivos para Recolha de Dados

Dependendo da escolha efetuada (Esfigmomanómetro, Oxímetro ou ambos) irá ser apresentado ao utilizador um *pop-up* com breves informações, na procura de garantir que todos os dispositivos pretendidos se encontram corretamente ligados.

Figura 35 – Pop-Up com Instruções Rápidas de Utilização

O utilizador deverá então pressionar o botão "*Start*" e aguardar a apresentação de resultados.

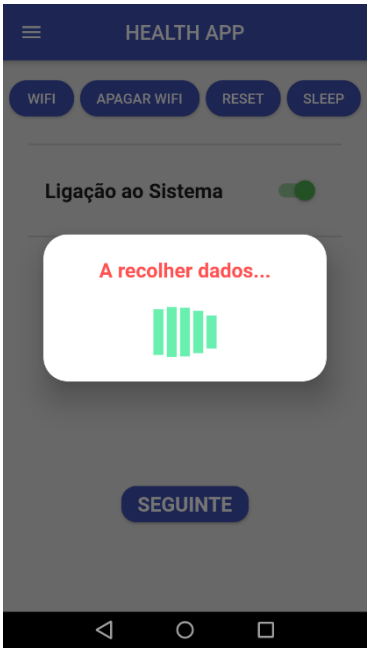


Figura 36 – Loading Icon Durante Recolha de Dados

Uma vez recebidos os dados, são apresentados na aplicação.

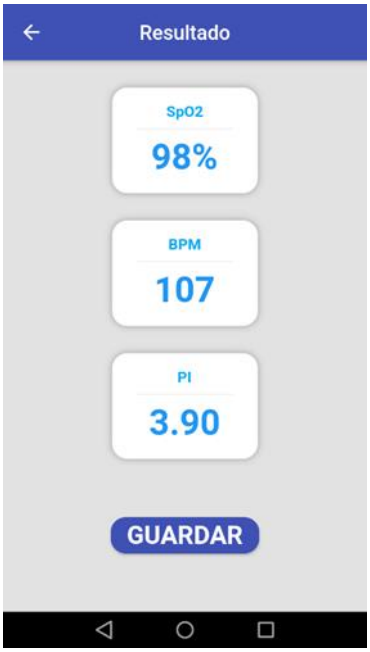


Figura 37 – Página de Apresentação de Resultados

Conforme é possível observar, é oferecida ao utilizador a possibilidade de, pressionando o botão “Guardar”, adicionar os dados recolhidos ao historial de medições. Neste caso, o utilizador será então encaminhado para a página onde se encontra todo o historial de medições efetuado.

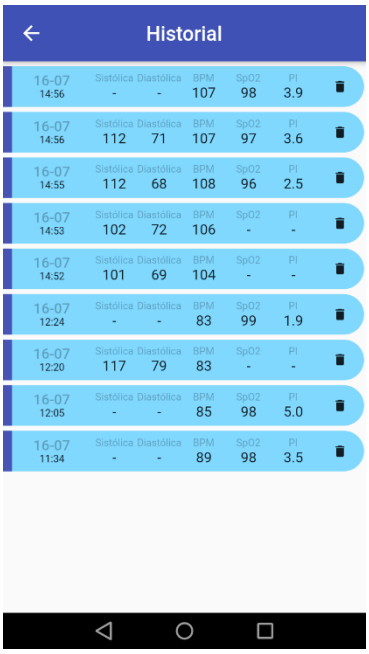


Figura 38 – Historial de Medições

Conforme é possível observar, é igualmente dada ao utilizador a opção de remover medições do seu historial, caso as mesmas já não se revelem necessárias ou possam ter sido gravadas por equívoco.

4.3 Custo Estimado

Na tabela abaixo é possível observar uma análise de custos dos componentes mais relevantes utilizados na elaboração deste sistema.

Tabela 8 – Análise de Custo dos Componentes

Imagem	Designação	Preço
	Módulo ESP32 c/ Suporte para Bateria 18650	16,95 €

	Esfigmomanómetro Digital Braço Omron M2	32,99 €
	Oxímetro de Dedo com *Bluetooth* Jumper JPD-500G	32,99 €
	Bateria 18650 *Li-ion* 3.7V 3500mAh	7,99 €
Custo Total		**90,92 €**

Refira-se que os dados indicados têm como origem valores apresentados por lojas nacionais e europeias à data da elaboração deste documento e que os mesmos já incluem outros custos, como portes de envio.

Excluindo eventuais reduções de custos que poderiam derivar da produção em massa deste sistema, as quais seriam difíceis de quantificar, podemos ainda assim referir que o valor de custo total apresentado poderia ser facilmente reduzido na ordem dos 25 % caso a opção passe por recorrer a outros fornecedores, nomeadamente no mercado chinês, ainda que mantendo a utilização de dispositivos das exatas marcas e modelos apresentados.

Esta redução ser ainda mais drástica e atingir os 35 % caso optássemos por, por exemplo, recorrer a um esfigmomanómetro digital de uma marca menos conceituada.

O custo dos restantes componentes utilizados neste projeto foi excluído devido ao facto de o custo dos mesmos ser negligenciável quando comparados com o custo total do projeto.

5 Conclusões

Uma vez terminado o projeto importa analisar os resultados obtidos, comparando-os com os objetivos inicialmente traçados, assim como projetar eventuais desenvolvimentos futuros.

5.1 Análise dos Resultados Obtidos

O projeto desenvolvido alcançou com sucesso os objetivos propostos, demonstrando a viabilidade de um sistema centralizado para o registo e monitorização de sinais vitais. A integração do microcontrolador ESP32 com dispositivos médicos como o monitor digital de pressão arterial e o sensor de SpO2 foi realizada de forma eficaz, comprovando a capacidade do sistema de captar medições precisas e fiáveis.

Além disso, a aplicação móvel desenvolvida revelou-se intuitiva, permitindo não apenas o controlo do sistema, mas também a visualização e gestão dos dados de forma acessível e prática, sendo que a possibilidade de armazenar o resultado das medições realizadas adiciona uma dimensão importante ao projeto, ao assegurar que o historial do paciente pode ser armazenado e consultado futuramente.

A solução proposta destaca-se ainda pela compatibilidade da aplicação com dispositivos Android e iOS, ampliando desta forma o seu potencial de acessibilidade e utilização. Este protótipo cumpre, assim, os requisitos inicialmente estabelecidos e apresenta-se como uma ferramenta útil em contextos onde a monitorização regular de sinais vitais é essencial.

O projeto, além de atender às necessidades técnicas e funcionais, também abre espaço para futuras expansões, como a integração com outros sensores e a incorporação de análises preditivas, reforçando o seu potencial como uma ferramenta útil em diversos cenários de saúde.

5.2 Desenvolvimentos Futuros

Numa visão de futuro, é fácil de imaginar eventuais desenvolvimentos de que este sistema poderia ser alvo posteriormente.

Numa primeira linha, um dos desenvolvimentos mais óbvios passaria por proceder à interface do sistema atual com mais equipamentos e sensores, que pudessem possibilitar o controlo de outro tipo de parâmetros, tais como um termómetro que permita proceder ao controlo da temperatura corporal de um indivíduo, medidores de glicemia, com foco no controlo dos níveis deste parâmetro no sangue, tendo em vista pacientes diabéticos, ou até mesmo um dos módulos de ECG existentes no mercado, compatíveis com ESP32, que permitam realizar

eletrocardiogramas simples. Uma adição deste tipo, contudo, já seria orientada para outro tipo de situações, já que exames deste tipo necessitariam de pessoal especializado que pudesse posteriormente proceder à sua análise e interpretação.

Visto que o ESP32 tem capacidades de comunicação tanto *Bluetooth* como *Wi-Fi*, ficam então abertas várias possibilidades para que futuramente se possa vir a estabelecer interface de outros dispositivos com este sistema, expandindo as capacidades do mesmo.

Outra melhoria óbvia seria a de permitir a criação de perfis de pacientes na aplicação móvel, de forma a que as medições recolhidas possam ser associadas a um dado indivíduo, uma vez as mesmas sejam concluída. Isto permitiria o aumento da abrangência de utilização do sistema, possibilitando que o mesmo seja utilizado em situações em que exista necessidade de fazer este controlo de forma regular a um grupo de pessoas. Um outro desenvolvimento que poderia decorrer deste último poderia passar pelo desenvolvimento de uma plataforma *web*, através da qual toda esta informação e o historial dos diversos pacientes pudesse ser consultado sem necessidade de acesso ao dispositivo móvel utilizado para gerir o sistema, permitindo assim o acesso de outras pessoas, com diferentes níveis de acesso dependentes das credenciais atribuídas às mesmas. Isto permitiria que, por exemplo, todo o historial de medições pudesse ser posteriormente ser alvo de consulta remota por parte de um médico ou outra pessoa responsável.

Futuramente, a própria aplicação poderia ser também preparada para possibilitar a definição de alarmes, através dos quais o responsável pelo sistema possa receber alertas imediatos caso os valores lidos ultrapassem certos parâmetros e assim lhe seja possível proceder a uma resposta mais célere a essa situação.

Referências

Ai-Thinker. (2020). *NodeMCU-32S specification*. Obtido de Ai-Thinker: https://docs.ai-thinker.com/_media/nodemcu32-s_specification_v1.3.pdf

Android. (2024). *What is Android*. Obtido de Android: https://www.android.com/what-is-android/

Android. (2024). *What is Android*. Obtido de Android: https://www.android.com/what-is-android/

Android Developers. (17 de Maio de 2021). *Conheça o Android Studio*. Obtido de Android Developers: https://developer.android.com/studio/intro?hl=pt-br

Apple Inc. (2024). *iOS*. Obtido de Apple: https://www.apple.com/ios/

Arduino. (2024). *Arduino Integrated Development Environment (IDE)*. Obtido de Arduino: https://www.arduino.cc/en/software

Bluetooth Special Interest Group. (2024). *Bluetooth Technology Overview*. Obtido de Bluetooth: https://www.bluetooth.com/learn-about-bluetooth/tech-overview/

Cisco. (2024). *O que é Wi-Fi?* Obtido de Cisco: https://www.cisco.com/c/pt_br/products/wireless/what-is-wifi.html

Cloud Firestore. (2024). Obtido de Firebase: https://firebase.google.com/docs/firestore

Codemagic. (18 de Janeiro de 2019). *What is Flutter? Benefits and limitations*. Obtido de Codemagic: https://blog.codemagic.io/what-is-flutter-benefits-and-limitations/

CPAPS. (2021). *Oxímetro de pulso: como fazer a leitura?* Obtido de CPAPS: https://www.cpaps.com.br/blog/oximetro-de-pulso-como-fazer-a-leitura/

Curvello, A. (27 de Fevereiro de 2018). *ESP32 – Um grande aliado para o Maker IoT*. Obtido de FILIPEFLOP: https://www.filipeflop.com/blog/esp32-um-grande-aliado-para-o-maker-iot/

Espressif. (2020). *ESP32 Series Datasheet.* Obtido de https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf

Espressif Systems. (2022). *Espressif Systems*. Obtido de ESP32-WROOM-32 datasheet: https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32_datasheet_en.pdf

Flutter. (2024). *Flutter documentation*. Obtido de Flutter: https://flutter.dev/

Fortune Business Insights. (2019). *Medical Devices Market*. Obtido de Fortune Business Insights: https://www.fortunebusinessinsights.com/industry-reports/medical-devices-market-100085

Google. (2024). *Dart programming language*. Obtido de Dart: https://dart.dev/overview

Google. (2024). *Firebase Documentation*. Obtido de Firebase: https://firebase.google.com/docs

Jumper. (2024). *JPD-500G*. Obtido de Jumper: https://www.jumper-medical.com/en/pros_d.aspx?CateId=328&sCateId=335&pid=405

Kolban, N. (18 de 12 de 2017). *ESP32 BLE Arduino*. Obtido de Arduino Docs: https://docs.arduino.cc/libraries/esp32-ble-arduino/

Last Minute Engineers. (2024). *Insight Into ESP32 Sleep Modes & Their Power Consumption*. Obtido de Last Minute Engineers: https://lastminuteengineers.com/esp32-sleep-modes-power-consumption/

Macoratti, J. C. (Junho de 2019). *Flutter - Apresentando Widgets*. Obtido de Macoratti: http://www.macoratti.net/19/06/flut_widgt1.htm

Mendonça, H. S. (2024). *SPI e I2C*. Obtido de https://paginas.fe.up.pt/~hsm/misc/old/comp/spi-e-i2c/

Nordic Semiconductor. (2024). *nRF Connect for Desktop*. Obtido de Nordic Semiconductor: https://www.nordicsemi.com/Products/Development-tools/nRF-Connect-for-Desktop

Oliveira, E. (2017). *Conhecendo o NodeMCU-32S ESP32*. Obtido de MasterWalker: https://blogmasterwalkershop.com.br/embarcados/esp32/conhecendo-o-nodemcu-32s-esp32

Omron. (2024). *M2*. Obtido de Omron: https://www.omron-healthcare.pt/produtos/m2

Philips Semiconductors. (2000). *The I2C-bus specification and user manual (Version 2.1)*. Obtido de Philips Semiconductors: https://www.nxp.com/docs/en/user-guide/UM10204.pdf

Pinheiro, F. (2020). *Flutter: O que são widgets e qual sua importância*. Obtido de TreinaWeb: https://www.treinaweb.com.br/blog/flutter-o-que-sao-widgets-e-qual-sua-importancia

Pinto, P. (6 de Fevereiro de 2021). *Firebase: A plataforma que procura para as suas apps Web e Mobile*. Obtido de Pplware: https://pplware.sapo.pt/tutoriais/firebase-a-plataforma-que-procura-para-as-suas-apps-web-e-mobile/

Random Nerd Tutorials. (16 de Maio de 2019). *Getting Started with ESP32 Bluetooth Low Energy (BLE) on Arduino IDE*. Obtido de Random Nerd Tutorials: https://randomnerdtutorials.com/esp32-bluetooth-low-energy-ble-arduino-ide/

Random Nerd Tutorials. (8 de Outubro de 2024). *ESP32 Deep Sleep with Arduino IDE and Wake Up Sources*. Obtido de Random Nerd Tutorials: https://randomnerdtutorials.com/esp32-deep-sleep-arduino-ide-wake-up-sources/

Reis, M. (Julho de 2021). *Oximetria: como usar o oxímetro e valores normais de saturação*. Obtido de Tua Saúde: https://www.tuasaude.com/oximetria/

ROHM. (11 de Junho de 2019). *I2C BUS EEPROM (2-Wire) BR24G08-3 Datasheet.* Obtido de https://datasheetspdf.com/pdf-file/1077711/Rohm/BR24G08-3/1

Samsung. (9 de Julho de 2015). Obtido de https://www.orbtronic.com/content/samsung-35e-datasheet-inr18650-35e.pdf

Silva, E. (6 de Maio de 2021). *Firebase: o que é e quando usar no desenvolvimento mobile?* Obtido de GeekHunter: https://blog.geekhunter.com.br/firebase-o-que-e-e-quando-usar-no-desenvolvimento-mobile/

Sociedade Brasileira de Pneumologia e Tisiologia. (2023). *Oximetria de Pulso*. Obtido de Sociedade Brasileira de Pneumologia e Tisiologia: https://sbpt.org.br/portal/publico-geral/doencas/oximetria-de-pulso/

Sociedade Portuguesa de Hipertensão. (2024). *Conheça Melhor a Hipertensão Arterial*. Obtido de Sociedade Portuguesa de Hipertensão: https://sphta.org.pt/hipertensao-arterial-hta-o-que-e-3/

Wi-Fi Alliance. (2024). *Discover Wi-Fi*. Obtido de Wi-Fi Alliance: https://www.wi-fi.org/discover-wi-fi

Anexos

```
bool slave_begin(int sda, int scl, int address){
    i2c_config_t config_slave;
    config_slave.sda_io_num = gpio_num_t(sda);
    config_slave.sda_pullup_en = GPIO_PULLUP_ENABLE;
    config_slave.scl_io_num = gpio_num_t(scl);
    config_slave.scl_pullup_en = GPIO_PULLUP_ENABLE;
    config_slave.mode = I2C_MODE_SLAVE;
    config_slave.slave.addr_10bit_en = 0;
    config_slave.slave.slave_addr = address & 0x7F;
    esp_err_t res = i2c_param_config(portNum, &config_slave);

    if (res != ESP_OK) {
        Serial.println("invalid I2C parameters");
        return false;
    }

    res = i2c_driver_install(
            portNum,
            config_slave.mode,
            2 * 100,  // rx buffer length
            2 * 100,  // tx buffer length
            0);

    if (res != ESP_OK) {
        log_e("failed to install I2C driver");
    }
    Serial.println("I2C Slave Started");
    return res == ESP_OK;
}
```

Figura 39 – Configuração ESP32 Slave I2C

```
bool data_update(){
    uint8_t inputBuffer[100] = {0};
    int16_t inputLen = 0;
    int i=0;
    inputLen = i2c_slave_read_buffer(portNum, inputBuffer, 100, 10);
    if (inputLen <= 0) {
        return false;
    }
    for(i=0; i<inputLen;i++) {
      Serial.print(inputBuffer[i], DEC);
      Serial.print(" ");
    }
    Serial.println();
    i2c_beginning_data = true;
    no_i2c_data = millis();
    if(inputBuffer[0] == 96 && inputBuffer[1] == 96) {
      no_valid_data = millis();
      sys = int(inputBuffer[3]) + 25;
      dia = int(inputBuffer[4]);
      bpmi2c = int(inputBuffer[5]);
      I2CResponseData = "Var:" + String(sys) + "," + String(dia) + "," + String(bpmi2c);
      Serial.println(I2CResponseData);
      sendI2CResponse = true;
      return true;
    }
    else if(inputBuffer[0] == 12 && inputBuffer[1] == 12){
      I2CResponseData = "NO_I2C_DATA";
      Serial.println("I2C Device Error Received");
      sendI2CResponse = true;
      return true;
    }
    else{
      return false;
    }
}
```

Figura 40 – Função de Processamento de Dados I2C

```
class MyClientCallback : public BLEClientCallbacks {
    void onConnect(BLEClient* pclient) {
    }

    void onDisconnect(BLEClient* pclient) {
      connected = false;
      Serial.println("onDisconnect");
    }
};
```

Figura 41 – Código Client Callback BLE

```
class MyAdvertisedDeviceCallbacks: public BLEAdvertisedDeviceCallbacks {
    void onResult(BLEAdvertisedDevice advertisedDevice) {
      if (advertisedDevice.haveServiceUUID() && advertisedDevice.isAdvertisingService(serviceUUID)) {
        Serial.print("Device found: ");
        Serial.println(advertisedDevice.getName().c_str());
        BLEDevice::getScan()->stop();
        myDevice = new BLEAdvertisedDevice(advertisedDevice);
        doConnect = true;
        doScan = true;
      }
    }
};
```

Figura 42 – Código Pesquisa de Servidor BLE

```
bool connectToServer() {
  Serial.print("Forming a connection to ");
  Serial.println(myDevice->getAddress().toString().c_str());

  pClient  = BLEDevice::createClient();
  Serial.println(" - Created client");

  pClient->setClientCallbacks(new MyClientCallback());

  if(!(pClient->connect(myDevice))){
    requestData = 1;
    return false;
  }

  Serial.println(" - Connected to server");

  BLERemoteService* pRemoteService = pClient->getService(serviceUUID);
  if (pRemoteService == nullptr) {
    Serial.print("Failed to find our service UUID: ");
    Serial.println(serviceUUID.toString().c_str());
    pClient->disconnect();
    return false;
  }
  Serial.println(" - Found our service");

  pRemoteCharacteristic = pRemoteService->getCharacteristic(charUUID);

  if (pRemoteCharacteristic == nullptr) {
    Serial.print("Failed to find our characteristic UUID: ");
    Serial.println(charUUID.toString().c_str());
    pClient->disconnect();
    return false;
  }
  else{
    Serial.println(" - Found our characteristic TX");
  }

  if (pRemoteCharacteristic->canNotify()){
    Serial.println(" - characteristic TX Can Notify");
    pRemoteCharacteristic->registerForNotify(notifyCallback);
  }

  connected = true;
  return true;
}
```

Figura 43 – Código de Conexão ao Servidor BLE

```
void scan_wifi(){
  int n = WiFi.scanNetworks();
  sendResponse = false;
  ResponseData = "";
  if (n == 0) {
    ResponseData = "SCAN:NULL";
  } else {
    ResponseData = "SCAN:";
    for (int i = 0; i < n; ++i) {
      ResponseData += WiFi.SSID(i);
      if(i < n-1){
        ResponseData += ',';
      }
      delay(10);
    }
  }
  sendResponse = true;
}
```

Figura 44 – Função de Scan Wi-Fi

```
bool read_ee(){
  int ei = 0;
  _ssid = "";
  _pass = "";
  char tmp = char(EEPROM.read(ei));
  while(tmp != NULL){
    _ssid = _ssid + tmp;
    ei++;
    tmp = char(EEPROM.read(ei));
    if(tmp == NULL) break;
  }
  ei++;
  ei++;
  tmp = char(EEPROM.read(ei));
  while(tmp != NULL){
    _pass = _pass + tmp;
    ei++;
    tmp = char(EEPROM.read(ei));
    if(tmp == NULL) break;
    if(ei > 500) return false;
  }

  if(_ssid.length() > 0 && _pass.length() > 0){
    Serial.print("SSID : ");
    Serial.println(_ssid);
    Serial.print("PASSWORD : ");
    Serial.println(_pass);
    return true;
  }
  else{
    return false;
  }
}
```

Figura 45 – Função de Leitura da EEPROM

```
bool write_ee(String newssid, String newpass){
  int si = 0, sl = 0, pi = 0, pl = 0;
  sl = newssid.length() + 1;
  pl = newpass.length() + 1;
  if(sl > 0 && pl > 0){
    while (si < sl){
      EEPROM.write(si, byte(newssid[si]));
      si++;
    }
    EEPROM.write(si, NULL);
    si++;
    while (pi < pl){
      EEPROM.write((si+pi), byte(newpass[pi]));
      pi++;
    }
    EEPROM.write((si+pi), NULL);
    EEPROM.commit();
    return true;
  }
  else{
    return false;
  }
}
```

Figura 46 – Função de Escrita na EEPROM

```
void erase_ee(){
  int ei = 0;
  char tmp = char(EEPROM.read(ei));
  while(tmp != NULL){
    EEPROM.write(ei, NULL);
    ei++;
    tmp = char(EEPROM.read(ei));
    if(tmp == NULL) break;
  }
  ei++;
  ei++;
  tmp = char(EEPROM.read(ei));
  while(tmp != NULL){
    EEPROM.write(ei, NULL);
    ei++;
    tmp = char(EEPROM.read(ei));
    if(tmp == NULL) break;
  }
  EEPROM.commit();
  Serial.println("Restarting in 3 Seconds");
  delay(3000);
  ESP.restart();
}
```

Figura 47 – Função de Eliminação de Dados da EEPROM

```
inline void msg_handler(){
  if(msg.length() > 0){
    no_data = millis();
    int ai = msg.indexOf('&');
    msg = msg.substring(0,ai);
    if(msg.equals("BLE")){
      requestData = 1;
    }
    else if(msg.equals("I2C")){
      requestData = 2;
    }
    else if(msg.equals("SLEEP")){
      requestData = 3;
    }
    else if(msg.equals("BOTH")){
      requestData = 4;
    }
    else if(msg.indexOf("SSID:") >= 0){
      rcv_ssid = "";
      rcv_pass = "";
      rcv_ssid = msg.substring(5);
    }
    else if(msg.indexOf("SCAN") >= 0){
      scan_wifi();
    }
    else if(msg.indexOf("PASS:") >= 0){
      rcv_pass = msg.substring(5);
      if(rcv_ssid.length() > 0 && rcv_pass.length() > 0){
        Serial.print("SSID : ");
        Serial.println(rcv_ssid);
        Serial.print("PASSWORD : ");
        Serial.println(rcv_pass);
        delay(500);
        if(write_ee(rcv_ssid, rcv_pass)) {
          client.print("Saved");
          delay(500);
          client.stop();
          Serial.println("Client disconnected");
          delay(500);
          Serial.println("Restarting in 5 Seconds");
          delay(3000);
          ESP.restart();
        }
        else client.print("Not Saved");
      }
      else client.print("Not Saved");
    }
    else if(msg.equals("ERASE")){
      erase_ee();
    }
    else if(msg.equals("RESET")){
      pClient->disconnect();
      Serial.println("Restarting in 3 Seconds");
      delay(3000);
      ESP.restart();
    }
    else{
      client.print("OK");
    }
    Serial.println(msg);
    msg = "";
  }
}
```

Figura 48 – Função de Processamento de Mensagens Recebidas

```
inline void result_data_processing(){
  if(sendResponse){
        client.print(ResponseData);
        sendResponse = false;
        ResponseData = "";
      }
      if(sendBLEResponse){
        client.print(BLEResponseData);
        sendBLEResponse = false;
        BLEResponseData = "";
      }
      if(sendI2CResponse){
        client.print(I2CResponseData);
        sendI2CResponse = false;
        i2c_data_update = false;
        i2c_beginning_data = false;
        I2CResponseData = "";
      }
      if(requestData == 1){
        if(!connected){
          BLEDevice::init("");
          BLEScan* pBLEScan = BLEDevice::getScan();
          pBLEScan->setAdvertisedDeviceCallbacks(new MyAdvertisedDeviceCallbacks());
          pBLEScan->setInterval(1349);
          pBLEScan->setWindow(449);
          pBLEScan->setActiveScan(true);
          pBLEScan->start(15, false);
        }
        ble_data = true;
        requestData = 0;
      }
      else if(requestData == 2){
        i2c_data_update = true;
        digitalWrite(I2C_BUTTON, HIGH);
        delay(I2C_BUTTON_DELAY);
        digitalWrite(I2C_BUTTON, LOW);
        delay(I2C_BUTTON_DELAY);
        no_i2c_data = millis();
        no_valid_data = millis();
        i2c_beginning_data = false;
        requestData = 0;
      }
      else if(requestData == 3){
        requestData = 0;
        if(connected) pClient->disconnect();
        if(client.connected()) client.stop();
        esp_sleep_enable_ext0_wakeup(GPIO_NUM_4,0);
        Serial.println("Going to sleep now");
        digitalWrite(SLEEP_LED, LOW);
        digitalWrite(I2C_BUTTON, LOW);
        delay(2000);
        esp_deep_sleep_start();
      }
      else if(requestData == 4){
        i2c_data_update = true;
        digitalWrite(I2C_BUTTON, HIGH);
        delay(I2C_BUTTON_DELAY);
        digitalWrite(I2C_BUTTON, LOW);
        delay(I2C_BUTTON_DELAY);
        no_i2c_data = millis();
        no_valid_data = millis();
        i2c_beginning_data = false;
        if(!connected){
          BLEDevice::init("");
          BLEScan* pBLEScan = BLEDevice::getScan();
          pBLEScan->setAdvertisedDeviceCallbacks(new MyAdvertisedDeviceCallbacks());
          pBLEScan->setInterval(1349);
          pBLEScan->setWindow(449);
          pBLEScan->setActiveScan(true);
```

Figura 49 – Função de Execução de Comandos – Parte 1

```
            pBLEScan->start(15, false);
          }
          ble_data = true;
          requestData = 0;
        }
        else if(requestData == 0){
          if(!i2c_data_update){
            flush_i2c();
            delay(50);
          }
        }
        if(i2c_data_update == true){
          if(data_update()){
            i2c_data_update = false;
            delay(2000);
            digitalWrite(I2C_BUTTON, HIGH);
            delay(I2C_BUTTON_DELAY);
            digitalWrite(I2C_BUTTON, LOW);
            delay(I2C_BUTTON_DELAY);
          }
          if(!i2c_beginning_data && millis() - no_i2c_data > 15000){
            i2c_data_update = false;
            client.print("NO_I2C_DATA");
            Serial.println("NO_I2C_DATA");
          }
          else if(i2c_beginning_data && millis() - no_i2c_data > 60000){
            i2c_data_update = false;
            client.print("NO_I2C_DATA");
            Serial.println("NO_VALID_DATA");
          }
          if(millis() - no_valid_data > 60000){
            i2c_data_update = false;
            client.print("NO_I2C_DATA");
            Serial.println("NO_VALID_DATA");
          }
        }
}
```

Figura 50 – Função de Execução de Comandos – Parte 2

```
inline void sleep_handler(){
  if(digitalRead(SLEEP_BUTTON) == LOW){
    btn_pressed = millis();
    while(digitalRead(SLEEP_BUTTON) == LOW) delay(100);
    btn_released = millis();
    if(btn_released - btn_pressed > 3000){
      erase_ee();
    }
    else{
      if(connected) pClient->disconnect();
      if(client.connected()) client.stop();
      esp_sleep_enable_ext0_wakeup(GPIO_NUM_4,0);
      Serial.println("Going to sleep now");
      digitalWrite(SLEEP_LED, LOW);
      digitalWrite(I2C_BUTTON, LOW);
      delay(2000);
      esp_deep_sleep_start();
    }
  }
  if(millis() - no_data > 15000){
    if(client.connected()) client.stop();
  }
  if(millis() - no_data > 300000){
    if(connected) pClient->disconnect();
    if(client.connected()) client.stop();
    esp_sleep_enable_ext0_wakeup(GPIO_NUM_4,0);
    Serial.println("Going to sleep now");
    digitalWrite(SLEEP_LED, LOW);
    digitalWrite(I2C_BUTTON, LOW);
    delay(2000);
    esp_deep_sleep_start();
  }
}
```

Figura 51 – Função de Gestão do Modo de Poupança de Bateria

```dart
return Scaffold(
  appBar: AppBar(
    centerTitle: true,
    title: Text("HEALTH APP"),
  ),  // AppBar
  drawer: DrawBarMenu(),
  backgroundColor: const Color(0xEFF1F1F1),
  resizeToAvoidBottomInset: false,
  body: Column(
    children: [
      Container(
        height: MediaQuery.of(context).size.height / 8,
        alignment: Alignment.center,
        child: Row(
          mainAxisAlignment: MainAxisAlignment.spaceEvenly,
          children: [
            _myElevatedButtons(_wifi_pressed, "WIFI"),
            _myElevatedButtons(_erase_wifi_pressed, "APAGAR WIFI"),
            _myElevatedButtons(_reset_pressed, "RESET"),
            _myElevatedButtons(_sleep_pressed, "SLEEP"),
          ],
        ),  // Row
      ),  // Container
      Divider(
        thickness: 1.5,
        endIndent: 25,
        indent: 25,
      ),  // Divider
      _myLabelSwitch(
        label: "Ligação ao Sistema",
        mySwitch: Switch(
```

Figura 52 – Widget de Botões e Switches da Página Principal – Parte 1

```
        value: switchState,
        onChanged: (value) {
          switchState = value;
          if (switchState)
            connect();
          else
            disconnect();
          setState(() {});
        },
        activeColor: Colors.green,
        inactiveThumbColor: Colors.red,
        inactiveTrackColor: Colors.redAccent,
      ),  // Switch
    ),
    Divider(
      thickness: 1.5,
      endIndent: 25,
      indent: 25,
    ),  // Divider
    _myLabelSwitch(
      label: "Esfigmomanómetro",
      mySwitch: Switch(
        value: i2c_switchState,
        onChanged: _i2c_changed,
        activeColor: _i2c_active,
        inactiveThumbColor: _i2c_inactive,
        inactiveTrackColor: _i2c_track_color,
      ),  // Switch
    ),
    _myLabelSwitch(
      label: "Oxímetro",
      mySwitch: Switch(
        value: ble_switchState,
        onChanged: _ble_changed,
        activeColor: _ble_active,
        inactiveThumbColor: _ble_inactive,
        inactiveTrackColor: _ble_track_color,
      ),  // Switch
    ),
```

Figura 53 – Widget de Botões e Switches da Página Principal – Parte 2

```
Expanded(
  child: Container(
    alignment: Alignment.center,
    child: ElevatedButton(
      child: Text(
        "SEGUINTE",
        style: TextStyle(
          fontSize: 20,
          fontWeight: FontWeight.w700,
          color: Colors.white,
        ),  // TextStyle
      ),  // Text
      onPressed: _next_pressed,
      style: ButtonStyle(
        shape: MaterialStateProperty.all<RoundedRectangleBorder>(
          RoundedRectangleBorder(
            borderRadius: BorderRadius.circular(15),
          ),  // RoundedRectangleBorder
        ),
      ),  // ButtonStyle
    ),  // ElevatedButton
  ),  // Container
),  // Expanded
```

Figura 54 – Widget de Botões e Switches da Página Principal – Parte 3

```
Widget _myLabelSwitch({
  String label,
  Switch mySwitch,
}) {
  return Container(
    height: MediaQuery.of(context).size.height / 8,
    alignment: Alignment.center,
    child: Row(
      mainAxisAlignment: MainAxisAlignment.spaceEvenly,
      children: [
        Text(
          label,
          style: TextStyle(fontSize: 20, fontWeight: FontWeight.bold),
        ),  // Text
        mySwitch,
      ],
    ),  // Row
  );  // Container
}
```

Figura 55 – Função de Configuração de Switches

```
Widget _myElevatedButtons(
  Function myOnTap,
  String title,
) {
  return ElevatedButton(
    onPressed: myOnTap,
    child: Text(title.toString()),
    style: ButtonStyle(
      shape: MaterialStateProperty.all<RoundedRectangleBorder>(
        RoundedRectangleBorder(
          borderRadius: BorderRadius.circular(17.5),
        ),  // RoundedRectangleBorder
      ),
    ),  // ButtonStyle
  );  // ElevatedButton
}
```

Figura 56 – Função de Configuração de Botões

```
      if (switchState) {
        setState(() {
          _wifi_pressed = () {
            senddata("SCAN&");
          };
          _erase_wifi_pressed = () {
            addStringToSF("SSID", "");
            addStringToSF("PASS", "");
            senddata("ERASE&");
            disconnect();
          };
          _reset_pressed = () {
            senddata("RESET&");
            disconnect();
          };
          _sleep_pressed = () {
            senddata("SLEEP&");
            disconnect();
          };
          _i2c_changed = (value) {
            setState(() {
              i2c_switchState = value;
            });
          };
          _ble_changed = (value) {
            setState(() {
              ble_switchState = value;
            });
          };
```

Figura 57 – Função de Gestão de Estados da Aplicação – Parte 1

```
    if (i2c_switchState || ble_switchState) {
      _next_pressed = () async {
        if (!i2c_switchState && ble_switchState) {
          instruction_dialog(ble_title, ble_instruction, _start_pressed);
        } else if (i2c_switchState && !ble_switchState) {
          instruction_dialog(i2c_title, i2c_instruction, _start_pressed);
        } else if (i2c_switchState && ble_switchState) {
          instruction_dialog(both_title, both_instruction, _start_pressed);
        }
      };
    } else {
      _next_pressed = null;
    }
    _i2c_active = Colors.green;
    _i2c_inactive = Colors.red;
    _i2c_track_color = Colors.redAccent;
    _ble_active = Colors.green;
    _ble_inactive = Colors.red;
    _ble_track_color = Colors.redAccent;
    if (!i2c_switchState && ble_switchState) {
      current_state = "BLE";
    } else if (i2c_switchState && !ble_switchState) {
      current_state = "I2C";
    } else if (i2c_switchState && ble_switchState) {
      current_state = "BOTH";
    }
  });
} else {
  setState(() {
    _wifi_pressed = null;
    _erase_wifi_pressed = null;
    _reset_pressed = null;
    _sleep_pressed = null;
    _i2c_changed = null;
    _ble_changed = null;
    _next_pressed = null;
    _i2c_active = _inactive;
    _i2c_inactive = _inactive;
    _i2c_track_color = Colors.grey;
    _ble_active = _inactive;
    _ble_inactive = _inactive;
    _ble_track_color = Colors.grey;
  });
}
```

Figura 58 – Função de Gestão de Estados da Aplicação – Parte 2

```
_start_pressed = () async {
  if (!i2c_switchState && ble_switchState) {
    senddata("BLE&");
    Navigator.of(context).pop();
  } else if (i2c_switchState && !ble_switchState) {
    senddata("I2C&");
    Navigator.of(context).pop();
  } else if (i2c_switchState && ble_switchState) {
    senddata("BOTH&");
    Navigator.of(context).pop();
  }
  if (await gettingDataDialog() == null) {
    _collecting_data_popup = false;
  }
};
```

Figura 59 – Função de Envio de Comandos para ESP32

```
check_wifi() async {
  String ssid_stored = await getStringValuesSF("SSID");
  if (ssid_stored == null) {
    ssid_stored = "";
  }
  await WiFiForIoTPlugin.isEnabled().then((val) {
    _isEnabled = val;
  });
  await WiFiForIoTPlugin.isConnected().then((val) {
    _isConnected = val;
  });
  if (_isEnabled && _isConnected) {
    await WiFiForIoTPlugin.getSSID().then((value) => _ssid = value);
    if (_ssid.isNotEmpty) {
      show_snackbar("Ligado a " + _ssid);
      if (_ssid == "HEALTHAPP_SERVER") {
        connect();
      } else if (ssid_stored.isEmpty && _ssid != ssid_stored) {
        NoWifiAlertDialog("Não conectado ao Sistema", "Conecte-se a: \nSSID: HEALTHAPP_SERVER\n Password: 12345678");
      } else if (ssid_stored.isNotEmpty && _ssid != ssid_stored) {
        NoWifiAlertDialog("Não conectado a " + ssid_stored, "Conecte-se a: \nSSID: " + ssid_stored + "\npara prosseguir");
      } else if (ssid_stored.isNotEmpty && _ssid == ssid_stored) {
        connect();
      }
    }
  } else if (_isEnabled && !_isConnected) {
    if (ssid_stored.isEmpty) {
      NoWifiAlertDialog("Não conectado ao Sistema", "Conecte-se a: \nSSID: HEALTHAPP_SERVER\n Password: 12345678");
    } else {
      NoWifiAlertDialog("Não conectado a " + ssid_stored, "Conecte-se a: \nSSID: " + ssid_stored + "\npara prosseguir");
    }
  } else if (!_isEnabled && !_isConnected) {
    if (ssid_stored.isEmpty) {
      NoWifiAlertDialog("WiFi desligado", "Ligue o WiFi e conecte-se a: \nSSID: HEALTHAPP_SERVER\n Password: 12345678");
    } else {
      NoWifiAlertDialog("WiFi desligado", "Ligue o WiFi e conecte-se a: \nSSID: " + ssid_stored + "\npara prosseguir");
    }
  }
}
```

Figura 60 – Função de Verificação da Conexão Wi-Fi

```dart
void connect() async {
  try {
    if (_ssid == "HEALTHAPP_SERVER") {
      socket = await Socket.connect('192.168.4.1', 8080);
      _connection_state = true;
      senddata("SCAN&");
    } else {
      String ip = await search();
      if (ip != null) {
        print("IP : " + ip);
        socket = await Socket.connect(ip, 8080);
        _connection_state = true;
      } else {
        _connection_state = false;
      }
    }
  } catch (e) {
    print(e);
    _connection_state = false;
  }
  if (_connection_state) {
    socket.listen((event) {
      handle_msg(event);
    }, onDone: () {
      handle_onDone();
    }, onError: (e) {
      print(e);
    });
    print('Ligado');
    setState(() {
      switchState = true;
    });
  } else {
    print('Erro: Não Ligado');
    Future.delayed(const Duration(milliseconds: 1000), () {
      setState(() {
        switchState = false;
      });
    });  // Future.delayed
  }
}
```

Figura 61 – Função para Conexão Wi-Fi ao ESP32

```
void handle_msg(Uint8List event) async {
  String text = utf8.decode(event);
  print(text);
  setState(() {});
  if (text.startsWith("OK") && _send_refresh) {
    _acknowledged = true;
  }
  if (text.startsWith("Não guardado")) {
    show_snackbar("Falha ao guardar");
    Future.delayed(const Duration(milliseconds: 2000), () {
      wifiCredentialsDialog(selected_wifi);
    });  // Future.delayed
  } else if (text.startsWith("Guardado")) {
    print("Guardado");
    await disconnect();
    Future.delayed(const Duration(milliseconds: 2000), () async {
      String ssid_stored = await getStringValuesSF("SSID");
      NoWifiAlertDialog("WiFi está desligado", "Ligue o WiFi e conecte-se a: \nSSID: " + ssid_stored + "\npara prosseguir");
    });  // Future.delayed
  } else if (text.startsWith("Var")) {
    var ci = text.indexOf(',');
    var ci2 = text.indexOf(',', ci + 1);
    v1 = text.substring(4, ci);
    v2 = text.substring(ci + 1, ci2);
    v3 = text.substring(ci2 + 1);
  } else if (text.startsWith("SPO")) {
    var ci = text.indexOf(',');
    var ci2 = text.indexOf(',', ci + 1);
    spo2 = text.substring(4, ci);
    bpm = text.substring(ci + 1, ci2);
    pi = text.substring(ci2 + 1);
  } else if (text.startsWith("SCAN:")) {
    String scan_data = text.substring(5);
    print(scan_data);
    if (scan_data == "NULL") {
      wifiScanDialog(null);
    } else {
      selected_wifi = await wifiScanDialog(scan_data);
      if (selected_wifi != null) wifiCredentialsDialog(selected_wifi);
    }
  } else if (text.startsWith("NO_I2C_DATA")) {
    if (_collecting_data_popup) Navigator.of(context).pop();
    show_snackbar("Sem Dados... Tente Novamente");
  }
```

Figura 62 – Função de Processamento de Dados

```
if (current_state == "BLE") {
  if (spo2.isNotEmpty && bpm.isNotEmpty && pi.isNotEmpty) {
    if (_collecting_data_popup) Navigator.of(context).pop();
    Navigator.push(
        context,
        PageTransition(
            type: PageTransitionType.rightToLeft,
            child: ResultPageBLE(spo2, bpm, pi),
            duration: Duration(milliseconds: 300)));  // PageTransition
    spo2 = "";
    bpm = "";
    pi = "";
  }
```

Figura 63 – Função de Escolha de Layout da Página de Resultados

```
periodic_task = Timer.periodic(Duration(seconds: 5), (timer) {
  if (_socket_refreshed) {
    _socket_refreshed = false;
    if (_send_refresh && _acknowledged) {
      _send_refresh = false;
      _acknowledged = false;
    } else if (_send_refresh && !_acknowledged) {
      _send_refresh = false;
      _acknowledged = false;
      setState(() {
        _connection_state = false;
        switchState = false;
        socket.close();
        show_snackbar("Disconnected");
      });
    }
  } else {
    if (_connection_state) {
      _send_refresh = true;
      _acknowledged = false;
      senddata("refresh");
    }
  }
});  // Timer.periodic
```

Figura 64 – Código de Verificação da Ligação ao ESP32