Implementação de Algoritmos em Sistemas Embarcados de Baixo Consumo Dotados de Rádio Sem Fio e Análise de Perfil de Consumo Energético

Marcos Felipe Pereira de Assis

Brasília, novembro de 2018

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASÍLIA
Faculdade de Tecnologia

TRABALHO DE CONCLUSÃO DE CURSO

Implementação de Algoritmos em Sistemas Embarcados de Baixo Consumo Dotados de Rádio Sem Fio e Análise de Perfil de Consumo Energético

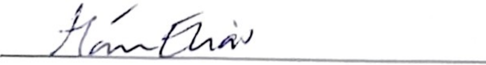
Marcos Felipe Pereira de Assis

Trabalho de Conclusão de Curso submetido ao Departamento de Engenharia Elétrica como requisito parcial para obtenção do grau de Engenheiro de Redes de Comunicação

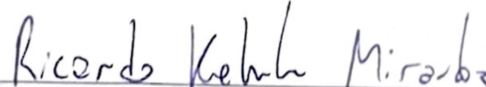
Banca Examinadora

Prof. João Paulo Carvalho Lustosa da Silva, Dr.-Ing, ENE/UnB
Orientador

Prof. Flávio Elias Gomes de Deus, Dr.-Ing, ENE/UnB
Examinador interno



Ricardo Kehrle Miranda, Dr.-Ing, ENE/UnB
Examinador interno



FICHA CATALOGRÁFICA

ASSIS, MARCOS	
Implementação de Algoritmos em Sistemas Embarcados de Baixo Consumo Dotados de Rádio Sem Fio e Análise de Perfil de Consumo Energético [Distrito Federal] 2018.	
xvi, 61 p., 210 x 297 mm (ENE/FT/UnB, Engenheiro, Engenharia Elétrica, 2018).	
Trabalho de Conclusão de Curso - Universidade de Brasília, Faculdade de Tecnologia.	
Departamento de Engenharia Elétrica	
1. Algoritmo	2. Bluetooth
3. Microcontrolador	4. Sinais
I. ENE/FT/UnB	II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

ASSIS, M.F.P (2018). *Implementação de Algoritmos em Sistemas Embarcados de Baixo Consumo Dotados de Rádio Sem Fio e Análise de Perfil de Consumo Energético*. Trabalho de Conclusão de Curso, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, 61 p.

CESSÃO DE DIREITOS

AUTOR: Marcos Felipe Pereira de Assis
TÍTULO: Implementação de Algoritmos em Sistemas Embarcados de Baixo Consumo Dotados de Rádio Sem Fio e Análise de Perfil de Consumo Energético.
GRAU: Engenheiro de Redes de Comunicação ANO: 2018

É concedida à Universidade de Brasília permissão para reproduzir cópias desto Trabalho de Conclusão de Curso e para emprestar ou vender tais cópias somente para propósitos acadêmicos e científicos. Os autores reservam outros direitos de publicação e nenhuma parte desso Trabalho de Conclusão de Curso pode ser reproduzida sem autorização por escrito dos autores.

____

Marcos Felipe Pereira de Assis
Depto. de Engenharia Elétrica (ENE) - FT
Universidade de Brasília (UnB)
Campus Darcy Ribeiro
CEP 70919-970 - Brasília - DF - Brasil

RESUMO

O objetivo deste trabalho é implementar algoritmos de processamento e indicação de sinais voltados para a área médica, que permitam aferiar a saúde do paciente, com base nos sinais vitais de oxigenação sanguínea e batimentos cardíacos, a partir do uso de sensores, e do microcontrolador CC2640R2F. Busca-se também analisar o consumo energético para execução das rotinas, assim como da disponilização das informações via Bluetooth, de modo que seja possível concluir sobre a viabilidade ou não dessa solução no cenário proposto. A decisão baseia-se em atender aos requisitos de consumo energético, em cenários com uma bateria de tamanho reduzido, no viés da microeletrônica, as não somente isso, o sistema proposto visa também requisitos satisfatórios de desempenho e memória, assim será possível monitorar o estado geral do paciente, com agilidade, eficiência, alta disponibilidade e possibilidade de expansão. Por fim, a implementação proposta deve ser de fácil integração com os dispositivos que necessitem de se comunicar com o microcontrolador para adquirir as informações aferidas.

ABSTRACT

The purpose of this work is to implement algorithms of processing and indication of signals facing the medical area, that can measure the health of the patient, based on the vital signal of blood oxygen and heart rate, from the use of sensors and the microcontroller CC2640R2F. It also seeks to analyze the energy consumption to execute the routines, as well as the availability of information using Bluetooth Protocol, so that it is possible to conclude on the feasibility or not of this solution in the proposed scenario. The decision is based on meeting the requirements of energy consumption, in scenarios with a small battery, in the bias of the microelectronics, not only that, the proposed system also aims satisfactory requirements of performance and memory, so it will be possible to monitor the state patient, with agility, efficiency, high availability and possibility of expansion. Finally, the proposed implementation should be easily integrated with the devices that need to communicate with the microcontroller to acquire the measured information.

SUMÁRIO

1 INTRODUÇÃO **1**
1.1 MOTIVAÇÃO 1
1.2 OBJETIVOS 2
1.3 ORGANIZAÇÃO DO TRABALHO 2

2 REFERENCIAL TEÓRICO **3**
2.1 MICROCONTROLADOR - CC2640R2F 3
2.1.1 ARQUITETURA 4
2.1.2 PERIFÉRICOS 5
2.2 *Bluetooth Low Energy* 6
2.2.1 ESPECIFICAÇÕES 6
2.2.2 ARQUITETURA 7
2.3 ALGORITMOS DE ANÁLISE 10
2.3.1 BATIMENTO CARDÍACO 10
2.3.2 OXÍMETRIA DE PULSO 11

3 METODOLOGIA **14**
3.1 IMPLEMENTAÇÃO DO BLUETOOTH 14
3.1.1 INICIALIZAÇÃO 14
3.1.2 TRATAMENTO DE EVENTOS 14
3.1.3 *Callbacks* 16
3.1.4 *Profiles* 18
3.2 *Medical Profile* 19
3.2.1 *Callbacks* 21
3.3 IMPLEMENTAÇÃO DOS ALGORITMOS 21
3.3.1 BLOQUEADOR DC 22
3.3.2 FORMA DIRETA II TRANSPOSTA 22
3.3.3 TRANSFORMADA RÁPIDA DE FOURIER 22
3.3.4 BATIMENTO CARDÍACO - ANÁLISE DE FREQUÊNCIA 23
3.3.5 OXÍMETRIA - RATIO OF RATIOS 24

4 RESULTADOS E DISCUSSÕES **26**
4.1 CÁLCULO DOS BATIMENTOS 26
4.2 CÁLCULO DA OXÍMETRIA 26
4.3 DESEMPENHO 27
4.4 ANÁLISE DE MEMÓRIA 28
4.5 ANÁLISE DE CONSUMO ENERGÉTICO 28

4.5.1 Análise Direta 29
4.5.2 Análise de Estados 30

5 Conclusões 32

REFERÊNCIAS BIBLIOGRÁFICAS 33

Apêndices 35

I Código Fonte 36

LISTA DE FIGURAS

2.1 Arquitetura resumida do microcontrolador CC2640R2F contendo os principais módulos utilizados. 4
2.2 Condição de *start* do I2C. 5
2.3 Condição de *stop* do I2C. 6
2.4 Arquitetura do BLE dividida por módulos presentes na camada de Host e Controle. 7
2.5 Sinal padrão do PPG sem a presença de ruído. 11
2.6 Coeficiente de extinção molar x comprimento de onda para Hb, HbO_2 e H_2O. 12

3.1 Diagrama de Inicialização da Aplicação. 15
3.2 Diagrama de Estados da Aplicação Bluetooth. 17
3.3 Diagrama de Estados do Pareamento. 18
3.4 Características do GAP, em amarelo estão destacados os atributos, os tipos de permissão para cada um, seus valores em hexadecimal e identificadores (UUID). 19
3.5 Características do Device Information Profile. 20
3.6 Características do Medical Profile. 21
3.7 Fase de Preparação do Sinal para Cálculo dos Batimentos. 23
3.8 Fase de Transformação do Sinal para Cálculo dos Batimentos. 24
3.9 Fase de Cálculo dos Batimentos. 24
3.10 Implementação do Algoritmo de Oximetria. 24

4.1 Gráfico Consumo de Corrente no Tempo. 29
4.2 Estados do Microcontrolador, na esquerda, a execução com o Bluetooth ativo, porém com conexão inativa, e na direita o Bluetooth ativo, com a conexão também ativa. 30

LISTA DE TABELAS

3.1 Tabela de Mapeamento SpO_2 para intervalos de valores do R_r.......... 25

4.1 Comparativo de Resultados do Algoritmo de Batimentos.......... 26
4.2 Comparativo de Resultados do Algoritmo de Oxímetria.......... 27
4.3 Comparativo do Tempo de Execução das Rotinas.......... 27
4.4 Tabela de resumo da memória.......... 28
4.5 Consumo de Corrente Módulos CC2640R2F.......... 31

LISTA DE SÍMBOLOS

Siglas

HR	*Heart Rate*
GAP	*Generic Access Profile*
BLE	*Bluetooth Low Energy*
HCI	*Host Controller Interface*
ATT	*Attribute Protocol*
L2CAP	*Logical Link Control and Application*
GATT	*Generic Attribute Profile*
SMP	*Security Management Protocol*
FFT	*Fast Fourier Transform*
IDE	*Integrated Development Environment*
PPG	*Photoplethysmograph*
$\mathrm{F_c}$	*Filter Convergence*
f_s	Taxa de Amostragem
R_r	*Ratio of Ratios*
SpO_2	Nível de Hemoglobinas Oxigenadas no Sangue
LPM	*Low Power Mode*
AM	*Active Mode*
PPD	*Peripheral Power Domain*
SPD	*Serial Power Domain*
RFPD	*RF Power Domain*
TX	*RF Transmitter*
RX	*RF Receiver*
DC	Linha Base do Sinal
AC	Amplitude do Sinal
GPIO	*General Purpose Input/Output*
I2C	Protocolo Serial de Comunicação I2C
UART	*Universal Asynchronous Transmitter/Receiver*
SPI	*Serial Peripheral Interface*
I2C	*Inter-IC Sound*
DMA	*Direct Memory Access*
ADC	*Analog-to-digital Converter*

1 INTRODUÇÃO

O sistema proposto neste trabalho visa monitorar os batimentos cardíacos e o nível de oxigenação sanguínea de um paciente com o uso de métodos não invasivos, e de modo que as informações adquiridas possam ser enviadas via bluetooth, com um consumo energético baixo. Esta introdução está dividida em 2 subseções. A Seção 1.1 estabelece a motivação do trabalho. Os objetivos são enumerados na Seção 1.2.

1.1 MOTIVAÇÃO

A evolução da microeletrônica e o avanço da criação de sistemas embarcados permitiu a transformação das mais diversas áreas. Dentre elas, a área de saúde, mais especificamente, o campo de monitoração dos sinais vitais de um paciente, a partir de tecnologias vestíveis - do inglês, wearable devices - conforme visto em [1]. Esses equipamentos aferem valores para sinais vitais como frequência cardíaca, pressão sanguínea, oxigenação, parâmetros relativos à atividade física e temperatura corporal a partir de métodos não invasivos, o que permite a utilização no dia a dia dos pacientes, e consequentemente, prevenção e alerta quanto aos problemas de saúde que podem ser identificados pela análise desses sinais.

Ainda no viés da evolução, somente adquirir essas informações não é mais suficiente. Também se faz necessário a disponibilização, de forma fácil e rápida desses dados para o paciente, ou até mesmo médicos e familiares que desejam ter acesso as informações de saúde dele. De modo a tornar possível esta demanda, a comunicação via bluetooth fornece um meio de baixo custo, e com flexibilidade de utilização, evidência mostrada em [2], que traz um exemplo onde o bluetooth foi utilizado para comunicação em um dispositivo da categoria de tecnologia vestível.

Métodos não invasivos de monitoração na área médica estão presentes na evolução da tecnologia, como mostrado em [3], [4] e [5], e ultimamente têm ganhado destaque para aplicação de tecnologias vestíveis, pois cada vez mais algoritmos tem surgido para suprir essa necessidade. Isso foi possível devido ao avanço dos microcontroladores, se tornando capazes de processar uma grande quantidade de informações, com efiência, e tamanho cada vez mais reduzido, seguindo uma tendência de miniaturização, principalmente pelo fato dos componentes para viabilização desses sistemas terem se tornado cada vez mais baratos. Nesse cenário, os desafios são implementar algoritmos mais robustos no que tange a desempenho e memória, e que possam aferir os sinais vitais de um paciente, com um consumo energético baixo, aliado a métodos que não provoquem incomodo ao paciente, e de fácil acesso as informações.

A facilidade de acesso aos dados se dá devido a evolução das comunicações sem fio e de sistemas conforme o descrito em [6], principalmente com o surgimento do padrão chamado *Bluetooth*

Low Energy (BLE), que é uma versão do protocolo de comunicação sem fio bluetooth, com baixo gasto de energia, conforme mostrado em [7] e [8]. O BLE é voltado para aplicações que busquem eficiência no consumo energético, que operam com fornecimento de energia limitado, e em geral, com tamanho reduzido. A versão do bluetooth citada é primordial ao se pensar em aplicações da área de saúde, por isso é importante que o dispositivo de monitoração não precise ser recarregado com uma grande frequência, e que também tenha uma alta disponibilidade para que fique continuamente adquirindo dados a respeito do paciente, principalmente se for pensado como um meio de alertar sobre o estado geral do paciente utilizando o dispositivo.

1.2 OBJETIVOS

Os objetivos a serem atingidos são 4 no total, e nesse trabalho dizem respeito as áreas de implementação de algoritmos, comunicação bluetooth e baixo consumo. Eles estão sumarizados a seguir:

- Implementar, em um microcontrolador específico, algoritmos de detecção de batimentos cardíacos, detecção de passos e nível de oxigênio, a partir de dados fornecidos por sensores.
- Analizar o consumo energético desses algoritmos, assim como o espaço em memória gasto, de modo que o microcontrolador possa operar por pelo menos 24 horas, sem que a bateria com capacidade de 150 mAh seja recarregada. Essa bateria foi escolhida devido ao fato de possuir um tamanho da ordem de milímetros, adequando-se a conceito de miniaturização citado em 1.2 e baixo preço.
- Disponibilizar as informações calculadas via Bluetooth, de modo que possa ser lida por outros dispositivos sem fio.
- Validar a viabilidade de uso do microcontrolador pretendido, assim como dos algoritmos utilizados, a partir dos parâmetros de consumo energético.

1.3 ORGANIZAÇÃO DO TRABALHO

Este trabalho está organizado em 5 capítulos, o primeiro deles, é a Introdução, que traz o cenário atual relativo a área no qual esse projeto se enquadra. O segundo, denominado Referencial Teórico, as especificidades do microcontrolador utilizado, assim como do Bluetooth e também dos algoritmos escolhidos para implementação. O terceiro, chamado Metodologia, ilustra o cenário utilizado para adquirir os resultados. O quarto, Resultados e Discussões, apresenta os resultados obtidos, assim como as discussões associadas a eles. E por fim, o capítulo de Conclusão, sumariza a análise dos resultados obtidos, e mostra as possibilidades de trabalhos futuros.

2 REFERENCIAL TEÓRICO

Sistemas que possuem sensores atrelados, e também um processamento e análise dos dados adquiridos, precisam de um concentrador, que irá orquestrar o funcionamento como um todo, assim como processar e disponibilizar as informações para serem lidas quando requisitadas. Seguindo a linha de miniaturização e consumo energético descrita em 1.1, um equipamento muito utilizado para tal aplicação são microcontroladores, devido ao seu tamanho pequeno, e capacidade de processamentos das informações com gasto energético otimizado. Por atender esses requisitos, o microcontrolador de modelo CC2640R2F, fabricado e vendido pela Texas Instruments, foi escolhido.

A escolha se deu, principalmente, devido ao fato de ser compacto do tipo SMD, do inglês, Surface Mounting Device, possuir conexão Bluetooth em sua versão de baixo consumo, periféricos seriais, conversor analógico para digital, gerenciamento otimizado de energia por alternar entre estados de ativação e espera, e fácil programação devido a biblioteca e documentação extensas provida pela própria fabricante.

As especificações, arquitetura e descrição dos periféricos, a respeito do microcontrolador, estão descritos na Seção 2.1. Na Seção 2.2 apresenta-se a explicação sobre o funcionamento do bluetooth de baixo consumo (BLE). Por fim, na Seção 2.3 estão descritos os algoritmos para aferição dos batimentos cardíacos e oximetria implementados.

2.1 MICROCONTROLADOR - CC2640R2F

O microcontrolador CC2640R2F é vendido pela Texas Instruments, possui um baixo consumo energético, taxado pela fabricante como um "Dispositivo com Consumo Ultra Baixo", com destinação a atender aplicações que necessitem do Bluetooth na versão 4.2 e 5, na faixa de frequência 2,4GHz. Seu processador é do tipo *Acorn Risc Machine* (ARM), e opera com 32 Bits a 48 MHz.

É indicado para aplicações médicas, como a citada ao longo desse trabalho, pois possui modos de gerenciamento de energia que colocam a *Central Process Unit* (CPU) em um estado de consumo energético mínimo, permitindo que pequenas células de energia sejam suficientes para o seu funcionamento por longos períodos. Possui 4 versões, que diferem entre si pelo tamanho e quantidade de pinos. Todas possuem os periféricos de *General Purpose Input/Output* (GPIO), I2C, *Universal Asynchronous Receiver Transmitter* (UART), *Serial Peripheral Interface* (SPI), I2S, Temporizador, *Direct Memory Access* (DMA) e também um conversor *Analog Digital Converter* (ADC) integrado de 12 Bits. Além disso, seu clock opera em 48MHz, explicações mais detalhadas podem ser vistas em [9].

Para programação do Firmware, existe uma biblioteca em C, provida pela própria empresa

fabricante, que auxilia na utilização do microcontrolador para as mais diversas finalidades, além de uma vasta gama de exemplos e documentação, acessados em [10], que auxiliam em seu entendimento. A *Integrated Development Environment* (IDE) para programação e debug indicada é o Code Composer Studio, pois já está integrada com a plataforma de exemplos da Texas Instruments, além de já ter todo o seu ambiente voltado para o uso de microcontroladores, como por exemplo, o plugin de análise do consumo energético, o Energy Tracer.

2.1.1 Arquitetura

O diagrama de blocos indicado na Figura 2.1 resume a arquitetura presente no microcontrolador.

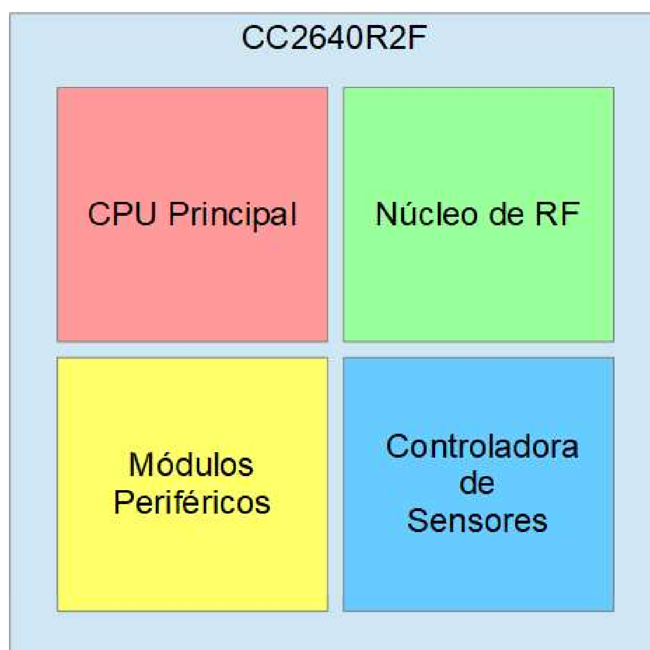


Figura 2.1: Arquitetura resumida do microcontrolador CC2640R2F contendo os principais módulos utilizados.

Como pode ser visto na Figura 2.1, existem 4 módulos principais:

- **CPU Principal:** Responsável pelo processamento do código e execução das rotinas principais.
- **Núcleo de Rádio Frequência (RF):** Gerenciamento do módulo de RF do microcontrolador, tanto *Transmitter* TX quanto *Receiver* (RX), assim como modulação e conversão dos sinais recebidos e enviados.
- **Módulos Periféricos:** Gerencia o funcionamento dos periféricos presentes na placa, o fato de ser independente, não onera no funcionamento da CPU, melhorando o consumo energético.
- **Controladora de Sensores:** Controla, de forma independente da CPU, os sensores como ADC e Comparadores.

Vale ressaltar também o espaço em memória, para que o programa seja executado. O microcontrolador possui 128-KB de Flash, que é onde fica salvo o firmware do microcontrolador. A memória do programa do tipo RAM tem 20KB, podendo ser expandida para 28KB, utilizando-se a memória Cache, como pode ser visto na referência [11].

2.1.2 Periféricos

Como citado anteriormente, o microcontrolador possui um bloco específico para gerenciar o funcionamento dos seus periféricos. Neste trabalho, 3 deles foram utilizados: I2C, ADC e BLE-Stack, que serão explicados na Seções 2.1.2.1, 2.1.2.2 e 2.1.2.3.

2.1.2.1 Barramento Serial - I2C

O protocolo de comunicação serial I2C é baseado no uso de dois sinais, denominados SDA (dados) e SCL (clock). Trabalha na arquitetura mestre-escravo, onde o mestre é responsável por gerar o sinal o do SCL, e também inicia a comunicação com os escravos.

Em um único barramento, é possível colocar até 127 dispositivos, onde cada um deles é identificado por um endereço de 7 bits. Um oitavo bit também é utilizado junto com o endereço, e indica se a operação realizada será de escrita ou leitura.

Para indicar que a comunicação será iniciada, uma condição de *start* é enviada pelo mestre, seu formato é indicado na Figura 2.2. Na condição de start, é preciso que o sinal do SDA esteja no nível alto (1), e transite para o nível baixo (0), com o SCL em nível alto, na sequência, o SCL também deverá transitar para o nível baixo. Por fim, é enviado o endereço do escravo, através do sinal SDA, com o bit indicando se a operação será de escrita ou leitura.

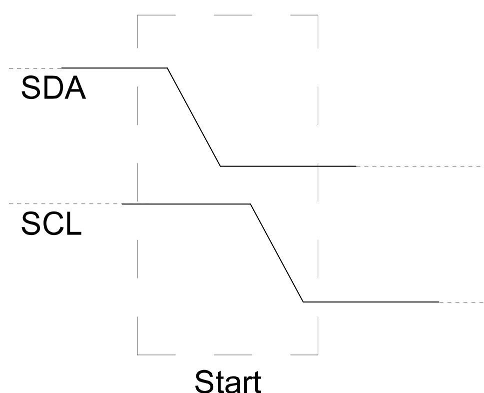


Figura 2.2: Condição de *start* do I2C.

Após a informação ser enviada, é necessário que o mestre mande uma condição de *stop*, com o formato descrito na Figura 2.3. Na condição de stop, é preciso que o sinal do SDA esteja no em nível baixo, e transite para o nível alto, com o SCL em nível baixo, na sequência, o SCL também deverá transitar para o nível alto.

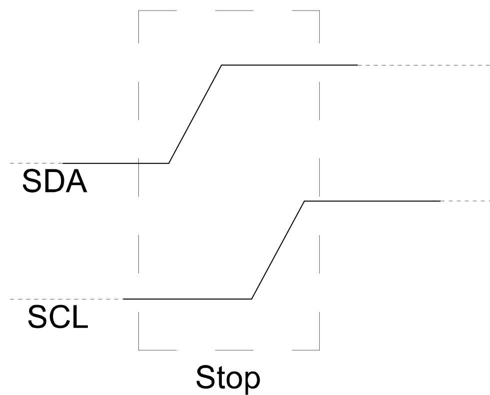


Figura 2.3: Condição de *stop* do I2C.

2.1.2.2 Conversor Analógico para Digital - ADC

O ADC é responsável por converter o sinal analógico em digital, no caso do CC2640R2F, o conversor possui 8 canais, com 12 bits de resolução cada. Seu modo de operação ocorre em etapas, primeiramente, o sinal analógico é discretizado, na sequência quantizado em codificado para o formato digital.

Como o ADC presente no microcontrolador CC2640R2F opera com 12 Bits de resolução, em uma faixa $ADC_{\text{range}} = 4.3$ V. Com isso, possui uma sensibilidade $\Delta V = 1.05$ mV, encontrada a partir da Equação (2.1). Isso significa que valores abaixo disso não são identificados pelo conversor, e assumidos com o valor 0.

$$\Delta V = \frac{ADC_{\text{range}}}{2^{\text{Bits}}}. \tag{2.1}$$

2.1.2.3 ICall - *Bluetooth Low Energy Stack*

O funcionamento do *Bluetooth Low Energy*, BLE, está descrito na Seção 2.2. Porém, o microcontrolador CC2640R2F possui o ICall, um módulo responsável por gerenciar a comunicação entre a camada de aplicação e a pilha de protocolos do BLE.

O ICall é o módulo identifica os eventos que ocorrem durante a comunicação bluetooth, repassa para a aplicação, que então possui as rotinas implementadas para tratamento deles. É o ICall que também gerencia o sincronismo de mensagens, *threads* da aplicação e alocação de memória.

2.2 *BLUETOOTH LOW ENERGY*

2.2.1 Especificações

Bluetooth Low Energy (BLE), ou *Bluetooth Smart*, surgiu da necessidade de dispositivos se comunicarem sem fio, porém com consumo mínimo de energia. Foi introduzido na versão 4.0 do

Bluetooth, e opera principalmente em sistemas microcontrolados. Seu foco se dá em dispositivos que compõem a chamada Internet das Coisas, que vem ganhando cada vez mais destaque na sociedade atual, desde aplicações mais simples, como acionar luzes a partir do celular, e também mais complexas, como a monitoração dos sinais vitais de um paciente no hospital, elucidado em [12].

O BLE opera na faixa frequencial de 2.4GHz, assim como o Wi-Fi. Seu padrão de modulação é o *Gaussian Frequency Shift Keying* (GFSK), a uma taxa máxima de 1 Mbps, e segurança de 128-bit *Advanced Encryption Algorithm* (AES).

2.2.2 Arquitetura

O funcionamento do BLE é dividido em camadas, cada uma com sua função específica, e se relacionando entre si. A Figura 2.4 mostra uma visão dessa arquitetura.

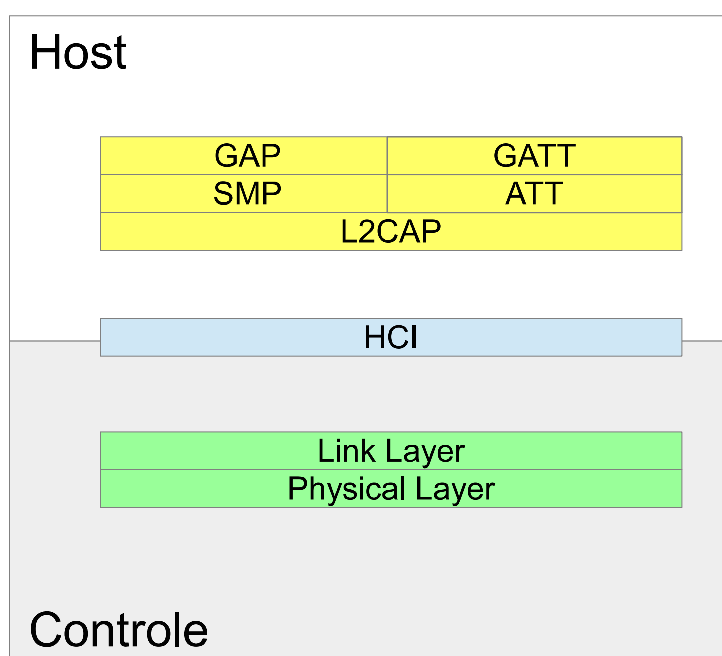


Figura 2.4: Arquitetura do BLE dividida por módulos presentes na camada de Host e Controle.

É possível dividir essas camadas em 2 categorias:

- **Camadas de Host:** GAP, GATT, ATT, SMP e L2CAP.
- **Camadas de Controle:** *Link Layer* e *Physical Layer*.

A interface entre as duas é feita pela camada *Host Controller Interface* (HCI), que pertence a uma categoria independente. A seguir, é mostrada uma descrição mais detalhada de cada uma dessas camadas, com base em [13] e [14].

2.2.2.1 *Generic Access Protocol* (GAP)

O GAP possui como principal função a definição de recomendações e requisitos relativos a modos ou procedimentos de acesso, usados pelos perfis de aplicação. É nessa camada que está contida a informação relativa ao tipo do dispositivo, ou seja, se ela suporta o BLE, o Bluetooth Convencional, ou o modelo híbrido dos dois.

Define 4 modos padrões de operação para os dispositivos:

- *Broadcaster:* O dispositivo fica anunciando seus parâmetros, SSID e preferências, porém não aceita e nem se conecta a nenhum outro.
- *Observer:* O dispositivo somente recebe anúncios advindos de outros dispositivos, porém não aceita, e nem inicia conexões.
- *Peripheral:* O dispositivo, além de anunciar seus parâmetros, também aceita conexões advindas de outros dispositivos.
- *Central:* O dispositivo, além de anunciar seus parâmetros, aceita e inicia conexões com outros dispositivos.

2.2.2.2 *Security Manager Protocol* (SMP)

É a camada responsável pelo gerenciamento de segurança da aplicação, define o modelo de segurança que será utilizado na comunicação entre os dispositivos. Aqui são definidas as diretivas de pareamento, assim como a troca de informações, principalmente no que tange a necessidade ou não de criptografia.

2.2.2.3 *Generic Attribute Profile* (GATT)

Define, usando o ATT, procedimentos e formatos dos serviços e características. Os procedimentos incluem características de leitura, escrita, notificação, indicação e descoberta.

Os dispositivos assumem a arquitetura de cliente-servidor, e é na camada GATT que está contida essa informação. o cliente é quem inicia comandos e requisições para o servidor, e também recebe respostas vindas dele. O servidor irá aceitar comandos e requisições, e enviar respostas para os clientes. O GATT só funciona caso o ATT esteja implementado, pelo fato de o utilizar na comunicação.

2.2.2.4 *Attribute Protocol* (ATT)

O ATT permite que o dispositivo, definido como servidor, exponha seus atributos e respectivos valores para seu cliente.

O ATT define que um atributo tem 4 propriedades principais associadas:

- **Tipo:** Definição a respeito do atributo.
- **Referência:** ID de 16-bit que permite o cliente referenciar o atributo.
- **Valor:** O valor do atributo.
- **Permissões:** As permissões relativas ao atributo, ou seja, se ele pode ser lido, escrito, ou até mesmo, se é necessário autenticação para performar as operações naquele atributo.

2.2.2.5 *Logical Link Control and Adaptation Protocol* (L2CAP)

A função principal do L2CAP é a de multiplexar os dados advindos das 3 camadas superiores (SMP, ATT e Link Layer) em um único sinal, de modo a entregar para a interface de conexão da Link Layer.

2.2.2.6 *Host Controller Interface* (HCI)

É responsável por realizar a comunicação entre o host e as camadas de controle. Possui pacotes de comando, lista de acesso, sincronismo e de eventos. A mais utilizada é a interface UART, mas a norma também especifica como usar a USB para tal fim.

2.2.2.7 *Link Layer*

Essa camada é responsável pelo controle de estados do dispositivo, permitindo que somente um deles fique ativo a cada momento. Ao todo, são 5 os estados existentes:

- *Standby State*;
- *Advertising State*;
- *Scanning State*;
- *Initiating State*;
- *Connection State*.

Dentro da *Link Layer* está contido o endereço do dispotivos, com o tamanho de 48 bits. 40 canais ao todo são utilizados pelo BLE, a informação relativa a qual está sendo utilizado na conexão apresenta-se nessa camada, e variam entre 2402 MHz e 2480 MHz, com 2 MHz de largura de banda.

2.2.2.8 *Physical Layer*

Essa camada é responsável pela transmissão da informação no meio físico. O sinal nesse ponto é modulado e enviado com os parâmetros de modulação e frequências previamente definidos. O

tipo de modulação utilizada é a Gaussian Frequency Shift Keying (GFSK), técnica na qual o sinal modulante varia a frequência de acordo com os valores de entrada. A potência de transmissão mínima é de 0.01 mW (-20 dBm), e a máxima é de 10 mW (+10 dBm), com taxa de bits em 1 Mbps.

2.3 ALGORITMOS DE ANÁLISE

Os algoritmos são responsáveis por adquirir os sinais de frequência cardíaca e oxímetria de um paciente. As Seções 2.3.1 e 2.3.2 mostram os métodos utilizados, que são do tipo não invasivos.

2.3.1 Batimento Cardíaco

A importância da aferição dos batimentos cardíacos da pessoa é devido ao fato de que o paciente pode estar apresentando complicações no caso deles estarem muito altos ou muito baixos. Nessa situação, e seguindo a tendência dos objetivos citados em 1.2, surgiram algoritmos que fazem a medição dos batimentos de forma não invasiva. Dentre eles, é possível utilizar a análise frequencial descrita em [15].

Esse algoritmo baseia-se em estimar os batimentos, com base em um sensor ótico infravermelho, e a transformada de fourier.

Primeiramente, é necessário gerar os dados do sensor ótico, mais precisamente, a quantidade de luz que é refletida após a incidência da luz infravermelha na corrente sanguínea do indivíduo. O sinal gerado, segue o padrão mostrado no gráfico da Figura 2.5, que ilustra um sinal padrão do fotopletismografo, do inglês, photoplethysmograph (PPG).

Em posse dessa curva, calcula-se a transformada discreta de fourier (DFT) do sinal a partir da Equação (2.2).

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{\frac{-j2\pi kn}{N}}, \tag{2.2}$$

Na sequência, é necessário encontrar a frequência dominante, representada por f_{hr}, do espectro obtido pela DFT, que se traduz como sendo a frequência na qual se encontra a componente de maior amplitude da transformada do sinal, como pode ser visto na Equação (2.3), onde N representa o tamanho do espectro.

$$f_{\text{hr}} = \underset{0 \leq k < \frac{N}{2}}{\text{argmax}}(|X(k)|), \tag{2.3}$$

Em posse dessa frequência, o valor dos batimentos, em bpm, é obtido a partir da equação mostrada em (2.4).

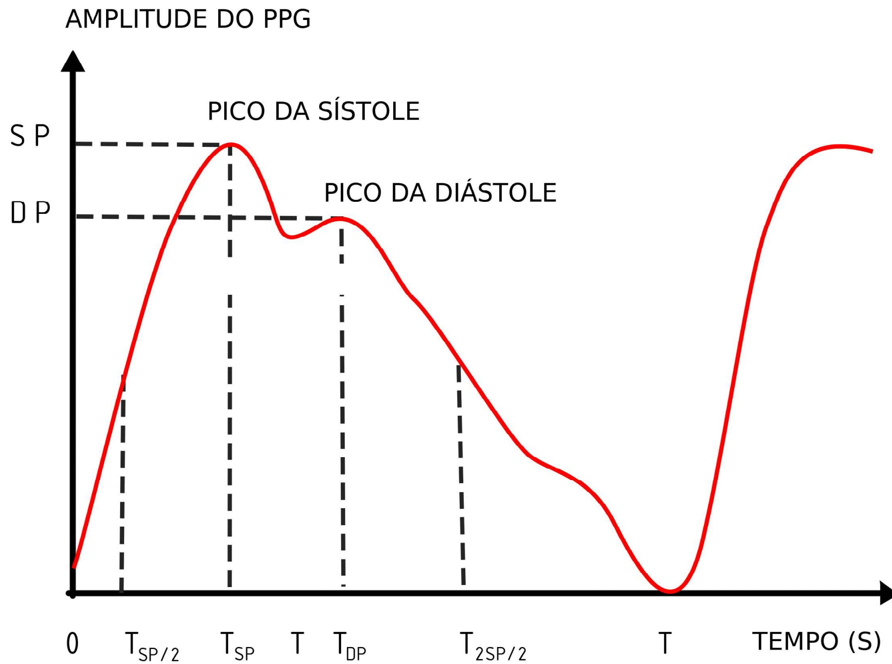


Figura 2.5: Sinal padrão do PPG sem a presença de ruído.

$$\text{BPM} = f_{\text{hr}} \cdot 60. \tag{2.4}$$

2.3.2 Oxímetria de Pulso

O nível de oxigênio pode ser traduzido como a saturação de oxigênio no sangue arterial (SaO_2), que é basicamente a porcentagem de concentração da hemoglobina oxigenada ($[HbO_2]$), com relação ao concentração total de hemoglobina no sangue ($[HbO_2]$ + $[Hb]$), mostrado na Equação (2.5).

$$SaO_2 = \frac{[HbO_2]}{[HbO_2] + [Hb]}, \tag{2.5}$$

A técnica da oxímetria de pulso, descrita em [16] e [17], baseia-se na diferença de absorção da luz entre HbO_2 e Hb por distintos comprimentos de onda. No caso de se usar as luz vermelha e infravermelha, o HbO_2 possui maior absorção na faixa de espectro infravermelho (940nm), e o Hb na faixa do vermelho (640nm), os valores usados para as faixas de luz devem provocar essa inversão para que o método funcione. O gráfico da figura 2.6 ilustra o motivo do escolha.

A incidência dos dois comprimentos sobre o sangue, irá gerar os dados que permitem calcular

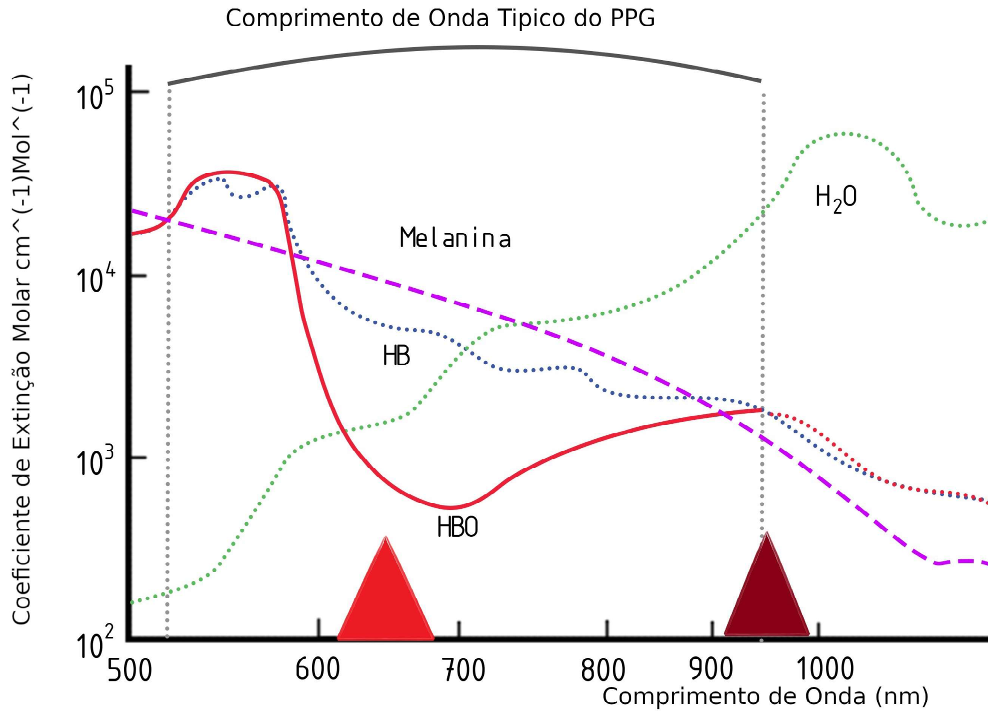


Figura 2.6: Coeficiente de extinção molar x comprimento de onda para Hb, HbO_2 e H_2O.

a relação chamada Ratio of Ratios (R_r), utilizada para encontrar o valor de SaO_2.

Primeiramente, é necessário calcular, para os dois comprimentos de onda, o volume máximo de mudança do sangue durante a sístole, que é sumarizado como a relação entre a amplitude do sinal, denotada como AC, e sua linha base, denotada como DC. Como visto nas Equações (2.6) e (2.7).

$$r_{\text{red}} = \frac{AC_{\text{red}}}{DC_{\text{red}}}, \tag{2.6}$$

$$r_{\text{infrared}} = \frac{AC_{\text{infrared}}}{DC_{\text{infrared}}}, \tag{2.7}$$

Com esses valores em mãos, para encontrar o *Ratio of Ratios*, pode ser utilizada a relação da Equação (2.8).

$$R_r = \frac{r_{\text{red}}}{r_{\text{infrared}}}, \tag{2.8}$$

Uma forma alternativa é utilizar a Equação (2.10) conforme [17], onde I_{ac} é a intensidade da luz somente com o nível AC do sinal presente. Por conveniência, o valor quadrático médio para I foi calculado, pois I_{ac} varia no tempo, com uso da Equação (2.9).

$$I_{\text{rms}} = \sqrt[2]{\frac{1}{N}\sum_{i=1}^{n} I_i^2}, \tag{2.9}$$

onde I_i é i-ésima amostra do sinal analisado.

Em posse desses valores para os dois sinais (Vermelho e Infravermelho), o R_r pode ser contrado a partir da Equação 2.10.

$$\mathrm{R}_r = \frac{\log_{10}(I_{\text{rmsRED}})\lambda_{\text{red}}}{\log_{10}(I_{\text{rmsIR}})\lambda_{\text{ir}}}, \tag{2.10}$$

onde λ_{red} representa o comprimento de onda do sinal vermelho, e λ_{ir} o comprimento de onda do sinal infravermelho.

Independente do método utilizado, a relação entre R e SaO_2 não é linear. A relação desses dois parâmetros é obtida experimentalmente, pois terá a influencia do sensor utilizado. Por esse motivo, é necessário que se faça uma calibração, de forma empírica. Para realizar essa calibração, primeiro é necessário um sensor com precisão médica, e também o sensor ótico.

Primeiro, obtêm-se amostras do sensor ótico e calcula-se o valor de R_r, nesse momento também é preciso capturar o nível de oxigênio com o uso do sensor médico. Repete-se esse passo para uma amostra de pessoas, a número dependerá do quão precisa a aplicação precisa ser, quanto mais pessoas forem utilizadas, melhor será a calibração. Ao fim, será possível traçar uma relação entre SpO_2 e R_r, que será utilizada como referência para o sensor.

3 METODOLOGIA

A fim de atingir os objetivos citados na Seção 1.2, a metodologia utilizada nessa tese foi dividida 5 blocos principais:

- Implementação do Bluetooth;
- Criação do *Medical Profile*;
- Implementação dos Algoritmos;
- Análise de Desempenho e Memória;
- Análise de Consumo Energético.

Todos os itens são relacionados ao microcontrolador CC2640R2F, usado devido ao fato de suas especificações providas vistas em [9], atenderem aos requisitos de energia e comunicação pretendidos para as aplicações citadas na Seção 1.2.

3.1 IMPLEMENTAÇÃO DO BLUETOOTH

A implementação utilizou a versão 4.2 do Bluetooth, que diz respeito ao BLE, devido ao fato de possuir maior compatibilidade com os dispositivos existentes hoje no mercado. Foi dividida em 4 módulos, que podem ser descritos como: Inicialização, Tratamento de Eventos, *Callbacks* e *Profiles*.

3.1.1 Inicialização

A inicialização é responsável pelo ajuste de configurações do Bluetooth, assim como a criação das threads que estarão ativas dentro do microcontrolador. O diagrama do código implementado relativo ao módulo de inicialização pode ser visto na Figura 3.1.

3.1.2 Tratamento de Eventos

A estrutura de eventos é a forma como o código interage com o que acontece no dispositivo Bluetooth, possuem 3 diferentes tipos: Eventos de Aplicação, Eventos de Pilha e Eventos Periódicos. O código implementado faz com que o microcontrolador se mantenha em *standby*, um estado com consumo energético mínimo, aguardando que algum desses eventos ocorram, para então responder adequadamente.

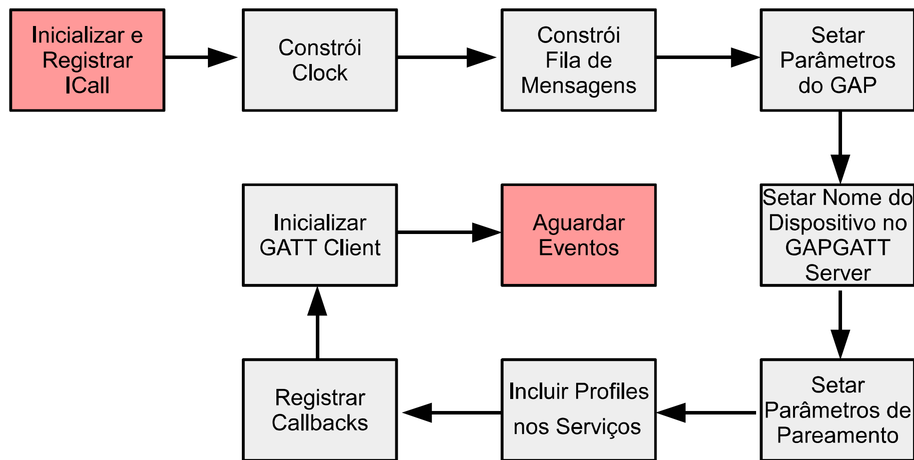


Figura 3.1: Diagrama de Inicialização da Aplicação.

3.1.2.1 Eventos de Aplicação

Os eventos de aplicação são os relativos a interação do microcontrolador com o cliente. Não existem limites para quantos existem, dependerá de cada cenário. Para o estudado, 4 tipos de eventos foram implementados, descritos a seguir:

- *State Change Event:* Ocorre quando o microcontrolador muda seu estado de conexão.
- *Pairing State Event:* Ocorre quando uma requisição de pareamento é iniciada.
- *Passcode Needed Event:* Ocorre quando é necessário que o cliente envie o PIN para realizar alguma operação autenticada.
- *Connection Event:* Ocorre quando uma requisição de conexão é iniciada.

3.1.2.2 Eventos de Pilha

- **Evento de Mensagem GATT:** São eventos advindos de mensagens enviadas pelo protocolo GATT.
- **Evento de Mensagem HCI:** Eventos que surgem a partir partir do funcionamento da camada de HCI.

3.1.2.3 Eventos Periódicos

Os eventos periódicos ocorrem em intervalos de tempos definidos, após uma conexão Bluetooth com o microcontrolador ter ocorrido com sucesso. Para o cenário atual, quando esse evento ocorre, os valores atuais calculados para temperatura, batimentos cardíacos e nível de oxigênio são atualizados no Medical Profile.

3.1.3 *Callbacks*

Callback é uma thread, ou seja, blocos de execução que ocorrem de forma paralela à rotina principal, e são chamados quando determinadas condições são cumpridas. Dentro do código feito, estão sendo utilizado 3 Callbacks: State Change, Passcode e Pair State.

3.1.3.1 *State Change Callback*

É chamado quando o microcontrolador muda de estado, e para cada estado existe uma ação a ser tomada. Ao todo, são 7 estados possíveis:

- *Started:* Indica que o módulo do bluetooth foi inicializado, toma a ação de verificar o endereço atribuído ao microcontrolador, e alterar na característica relativa a isso dentro do Information Profile.
- *Advertising:* O microcontrolador passa a anunciar seu SSID, para o caso do cenário implementado, o nome escolhido foi "SMART BAND".
- *Connected:* Indica que uma conexão foi estabelecida, dá início ao clock que controla a a chamada dos eventos do tipo periódico. Envia também, via interface UART, o endereço do dispositivo ao qual se está conectado.
- *Connected Advertising:* Além de indicar que existe uma conexão estabelecida, continua a anunciar o SSID para que outros dispositivos possam ver.
- *Waiting:* O microcontrolador passa a aguardar novas conexões advindas do mesmo dispositivo ao qual estava conectado anteriormente, com a conexão sendo encerrado.
- *Waiting After Timeout:* O microcontrolador passa a aguardar novas conexões advindas do mesmo dispositivo ao qual estava conectado anteriormente, porém quando o motivo da desconexão é inatividade.
- *Error:* O microcontrolador entrou em um estado de erro devido a alguma problema não mapeado.

O diagrama presente na Figura 3.2 indica como a transição entre os estados pode ocorrer. As setas indicam a ligação entre o estado atual, e para qual a aplicação pode seguir, dependendo de qual evento ocorreu. O estado "Error"pode ocorrer a qualquer momento, por isso nenhuma seta está ligada a ele.

3.1.3.2 *Passcode Callback*

É chamado quando algum dispositivo se conectou ao microcontrolador, e na sequência deseja realizar alguma operação que necessite autenticação através do PIN.

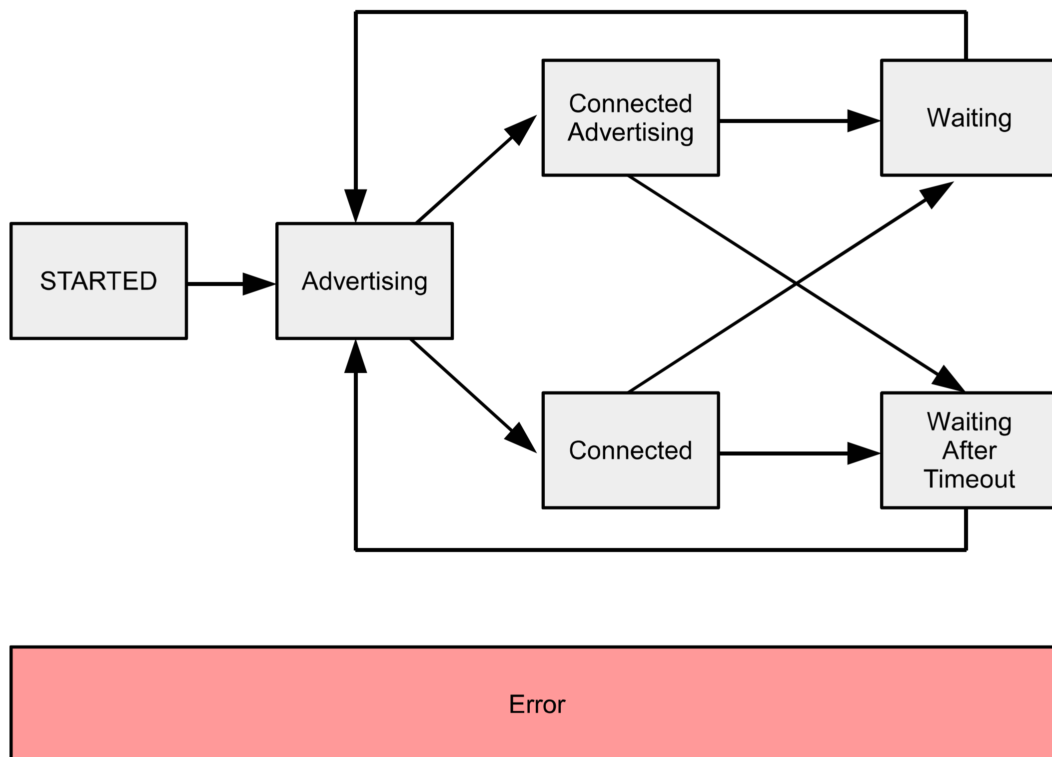


Figura 3.2: Diagrama de Estados da Aplicação Bluetooth.

A rotina irá verificar qual foi o PIN enviado pelo dispositivo, é retornar sucesso ou falha dependendo de estar ou não correto. Seu funcionamento é simples, porém bastante importante para garantir a segurança da aplicação.

3.1.3.3 *Pair State Callback*

É chamado quando algum dispositivo que esteja conectado deseja-se parear com o microcontrolador. Para parear, é necessário enviar o PIN correto setado na aplicação. Dentro desse estado, 4 outros existem, e são tratados pela aplicação:

- *Pairing State Started:* Indica que foi iniciada uma requisição de pareamento.
- *Pairing State Complete:* Indica que o pareamento foi concluído, indicando se foi com falha ou com êxito.
- *Pairing State Bonded:* Indica que uma união foi formado entre o dispositivo e o microcontrolador, ele não precisará mais enviar o PIN para se autenticar durante a sessão ativa.
- *Pairing State Bond Saved:* Indica que uma união foi formado entre o dispositivo e o microcontrolador, ele não precisará mais enviar o PIN para se autenticar durante a sessão ativa, e também nas próximas sessões que iniciar.

O diagrama presente na Figura 3.3 indica como a transição entre os estados pode ocorrer.

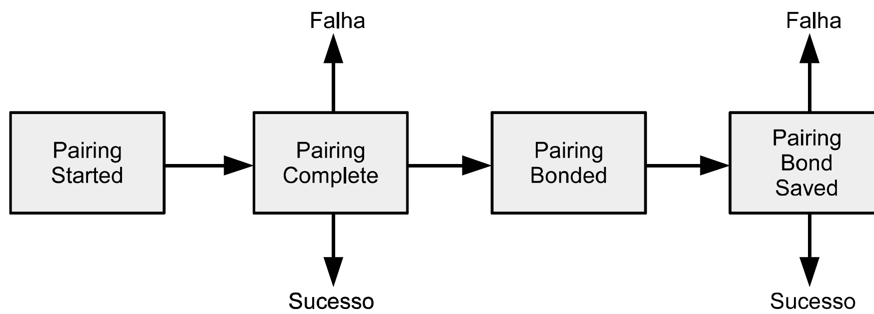


Figura 3.3: Diagrama de Estados do Pareamento.

3.1.4 *Profiles*

A estrutura de profiles diz respeito a como, e quais informações são disponibilizadas para os dispositivos que se conectam ao microncontrolador. Para o cenário proposto, são 3 os existentes: GAP Profile, Device Information Profile e Medical Profile. Os dois primeiros serão explicados a seguir, o último possui uma seção separada, vista em 3.2.

3.1.4.1 GAP *Profile*

Este profile é responsável por anunciar os parâmetros de conexão do dispositivo. Basicamente, traz duas informações principais:

- **SSID:** Nome do módulo de Bluetooth do microcontrolador que é visto quando ele está em estado "Advertising".
- **Preferência de Conexão:** Indica se o dispositivo tem algum tipo de preferência nos parâmetros de conexão.

Com o aplicativo BLE Scanner para dispositivos Android, é possível visualizar as duas informações, como pode ser visto na Figura 3.4. Em vermelho está marcado o nome do dispositivo, e em amarelo as características.

3.1.4.2 *Device Information Profile*

Este profile mostra informações gerais acerca da aplicação bluetooth que está em funcionamento no microcontrolador. Ao todo são 9:

- **System ID**;
- **Model Number**;
- **Serial Number**;
- **Firmware Revision**;

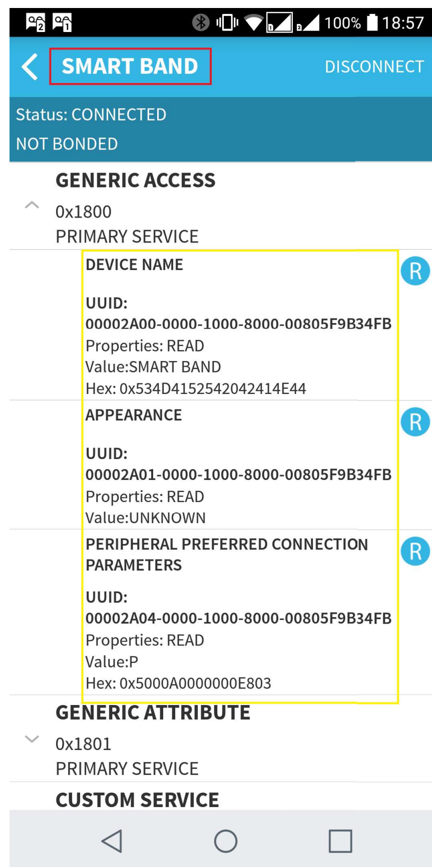


Figura 3.4: Características do GAP, em amarelo estão destacados os atributos, os tipos de permissão para cada um, seus valores em hexadecimal e identificadores (UUID).

- **Hardware Revision**;
- **Software Revision**;
- **Manufacturer Name**;
- **IEEE 11073-20601 Regulatory Certification Data List**;
- **PNP ID**.

A Figura 3.5 mostra parte dos atributos desse profile, marcados em amarelo, e em vermelho, o nome do dispositivo.

3.2 *MEDICAL PROFILE*

O *Medical Profile* foi criado com o intuito de sumarizar todas as informações clínicas de um paciente em um único lugar. Seu objetivo principal é prover esses dados de modo que possam ser utilizados por qualquer aplicação ou dispositivo que deseje fazer uso.

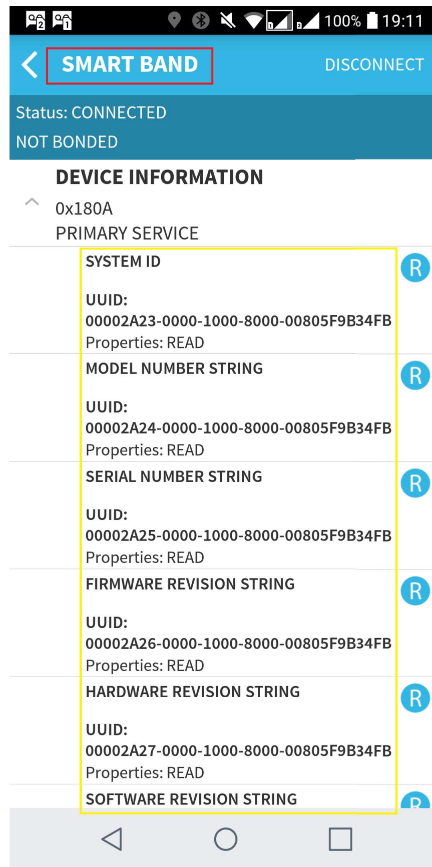


Figura 3.5: Características do Device Information Profile.

Para isso, 6 características foram escolhidas e implementadas, calculadas por sensores atrelados, e algoritmos implementados no microcontrolador:

- *ID:* Identificador do Paciente, escolhido com 1 byte, podendo ser expandido posteriormente.
- *Temperature:* Temperatura Corporal Interna, possui 2 bytes, sendo o primeiro octeto referente a parte inteira da informações, e o segundo octeto relativo ao ponto flutuante.
- *Heart Rate:* Batimentos Cardíacos, escolheu-se somente 1 byte para essa informação, o que permite valores entre 0-256.
- *Oxygen Level:* Nível de Oxigênio, podendo varias de 0-100%, por isso só foi necessário 1 byte.
- *Pressure:* Pressão Diferencial, contém 2 byter, sendo o primeiro octeto relativo a sístole, e o segundo a diástole.
- *Steps Count:* Contagem de passos do paciente, escolhido com 2 bytes, podendo varias de 0-65536.

Todas as características foram escritas com permissão de leitura, não sendo necessário parear com o dispositivo primeiro para que depois se tenha acesso.

A Figura 3.6 mostra as características desse profile, fornecidas pelo aplicativo BLE Scanner para dispositivos Android. Destacado em rosa está o identificador do serviço (Custom Service), em vermelho, o tipo de permissão para cada um dos atributos, em amarelo, o valor atual do atributo em hexadecimal, e em verde, o ID que identifica descrição de atributo (0x2901) junto com o seu valor (ID, Temperatura, Heart Rate, Oxygen Level, Pressure e Steps Count).

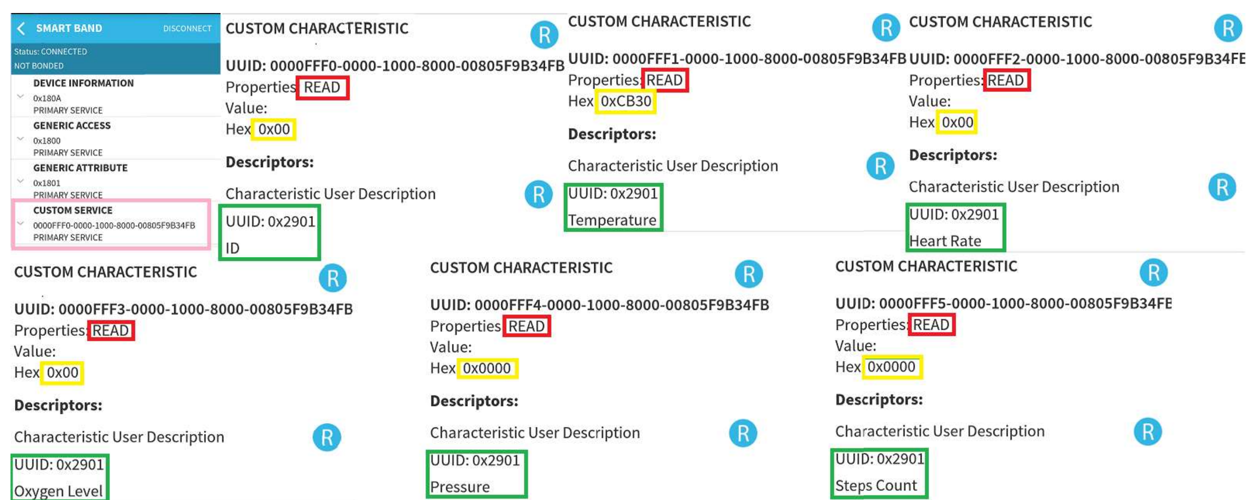


Figura 3.6: Características do Medical Profile.

3.2.1 *Callbacks*

O *Callback* presente e utilizado nesse profile é o readAttributeCallback. Ele é chamado quando o cliente conectado ao microcontrolador, via Bluetooth, requisita leitura em determinado atributo do profile.

Para o caso onde a característica tem somente 1 byte, o valor atual é apresentado diretamente para o cliente. Já no caso dos que tem mais de um byte, o microcontrolador primeiramente copia para o endereço de memória a ser lido o valor atual, para depois mostrar a quem requisitou.

3.3 IMPLEMENTAÇÃO DOS ALGORITMOS

A implementação dos algoritmos foi feita em linguagem C, com a IDE Code Composer Studio em sua versão 8.0, e a SDK versão 2.20 do microcontrolador CC2640R2F, fornecida pela própria Texas Instruments. Para isso, foi necessário implementar funções importantes e complexas para a execução dos algoritmos, dentre elas, filtros e a transformada rápida de fourier, do inglês, fast fourier transform (FFT).

Dentre as técnicas de filtragem, 2 se destacam, a de Bloqueio DC, e a Forma Direta II Transposta. Elas foram escolhidos pois estão implementadas dentra da biblioteca do Matlab, ambiente a partir do qual os códigos foram adaptados. A teoria relativa a implementação dessas duas téc-

nicas estão presentes em 3.3.1 e 3.3.2

3.3.1 Bloqueador DC

É utilizado para remover o componente DC de um sinal, para o caso do processamento digital. É recursivo, e utiliza a Equação (3.1).

$$y(n) = x(n) - x(n-1) + \mathrm{R}y(n-1), \tag{3.1}$$

Sendo x o sinal de entrada, y o sinal de saída e R o fator de bloqueio, geralmente o valor escolhido é entre 0,9 e 1,0.

A função de transferência que representa esse filtro é dada pela equação presente em (3.2).

$$H(z) = \frac{1 - z^{-1}}{1 - \mathrm{R}z^{-1}}. \tag{3.2}$$

3.3.2 Forma Direta II Transposta

É um método bastante utilizado para a filtragem de sinais, seja o filtro do tipo passa-baixa, passa-faixa ou passa-alta. A Equação (3.3) representa seu modelo matemático, e sua função de transferência é a Equação (3.4).

$$a(1)y(n) = b(1)x(n) + ... + b(n_b+1)x(n-n_b) - a(2)y(n-1) - ... - a(n_a+1)y(n-n_a), \tag{3.3}$$

$$Y(z) = \frac{b(1) + b(2)z^{-1} + ... + b(n_b+1)z^{-n_b}}{1 + a(2)z^{-1} + ... + a(n_a+1)z^{-n_a}}, \tag{3.4}$$

Onde an e bn são os coeficientes do filtro escolhido, n_a o número de coeficientes do denominador, n_b o número de coeficientes do numero, x o sinal de entrada e y o sinal de saída.

3.3.3 Transformada Rápida de Fourier

A transformada rápida de fourier é um método computacional para encontrar a transformada discreta de fourier, do inglês Discrete Fourier Transform (DFT), de forma mais rápida. A complexidade computacional da transformada rápida de fourier é do grau $O(nlog(n))$, sendo n o número de amostras, já da transformada discreta de fourier é $O(n^2)$, o que revela uma melhora significante.

O princípio matemático da transformada discreta de fourier é dado pela Equação (3.5). A

transformada rápida usa um princípio algoritmo de meio dos meios para acelerar o processamento que permite encontrar os termos $X(k)$.

$$X(k) = \sum_{n=0}^{N-1} x(n)e^{\frac{-j2\pi kn}{N}}, \tag{3.5}$$

Sendo N o número de amostras, k o k-ésimo termo da transformada, x sinal de entrada e X a transformada de x.

3.3.4 Batimento Cardíaco - Análise de Frequência

A implementação do algoritmo descrito em 2.3.1 pode ser dividida em 3 fases distintas: Preparação, Transformação e Cálculo.

A primeira delas, descrita, em blocos, na Figura 3.7, é responsável pela filtragem do sinal. Primeiramente, ela faz a remoção de artefatos, que elimina as amostras com valor menor que 100, pois são amostras relativas aos ruídos, ou que tem uma discrepância muito grande com relação a amostra anterior. Na sequência, a componente DC do sinal é removida utilizando a técnica descrita em 3.3.1. E por fim, filtra-se o sinal, com o uso de um filtro Butterworth de ordem 6, e a técnica descrita em 3.3.2.

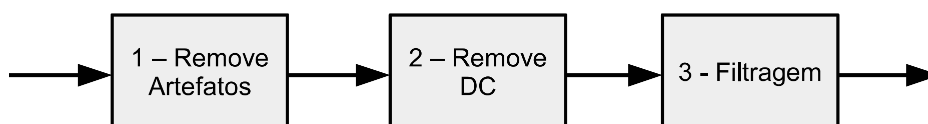


Figura 3.7: Fase de Preparação do Sinal para Cálculo dos Batimentos.

A segunda, apresenta-se na Figura 3.8, é responsável por calcular a FFT do sinal. Ela recebe como entrada, além do sinal filtrado, o valor de convergência do filtro, ele é obtido a partir da fórmula presente na equação (3.6). Na sequência, pega-se amostras do sinal a partir os índices dados por F_c até $F_c + f_s * TempodeJanela$, e acrescenta-se 0's ao final para que se tenha 1024 amostras, pois o algoritmo usado para FFT necessita que as amostras sejam do tipo 2^n. Com isso, calcula-se a FFT do sinal. Para o cenário implementado, f_s é igual a 100 Hz e o $TempodeJanela$ de 10 segundos, pois é o tempo necessário para que o evento periódico seja novamente acionado.

$$F_c = 150\log_{10}(f_s) \tag{3.6}$$

Por último, em posse dos valores de frequência, e da FFT do sinal filtrado, encontra-se o valor dos batimentos por minuto, a partir da teoria descrita em 2.3.1. O diagrama de blocos da Figura 3.9 ilustra graficamente essa fase.

Os resultados e discussões a respeito dessa implementação apresentam-se descritos na Seção 4.1.

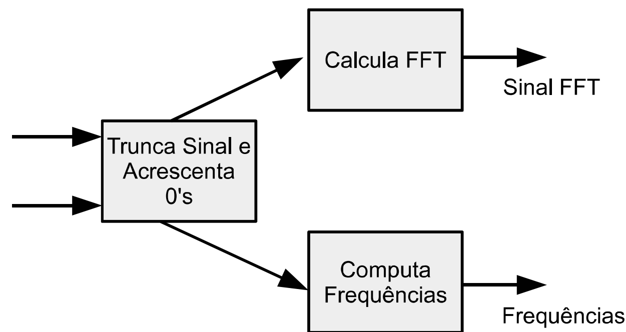


Figura 3.8: Fase de Transformação do Sinal para Cálculo dos Batimentos.

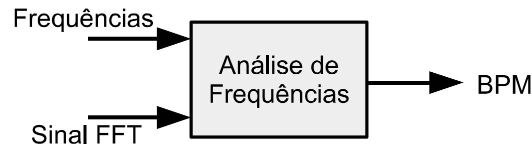


Figura 3.9: Fase de Cálculo dos Batimentos.

3.3.5 Oxímetria - Ratio of Ratios

O algoritmo utilizado para oxímetria está descrito em 2.3.2. Diferente do que encontra os batimentos, é necessário utilizar os dados de sensor óptico infravermelho e também vermelho.

Primeiramente, os dois sinais tem sua componente DC removida, com uso da técnica de DC blocker descrita em 3.3.1. Na sequência, o sinal é filtrado a partir do método mostrado em 3.3.2, e um filtro de Butterworth passa-baixa, de ordem 6, com frequência de corte em 4Hz.

Para os dois sinais filtrados, o valor RMS é calculado, a partir da fórmula presente em (2.9), e por fim, é encontrado o valor de R_r, com a Equação (2.10).

Em posse do valor de R_r, o nível de oxigênio presente no sangue, denotado por SpO_2, é obtido de acordo com a Tabela 3.1, obtido após o método de calibração citado em 3.3.5.1.

O diagrama presente na Figura 3.10 apresenta de forma visual a implementação do algoritmo descrito.

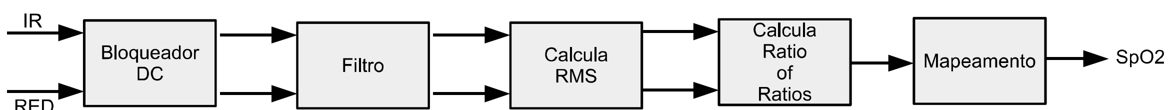


Figura 3.10: Implementação do Algoritmo de Oximetria.

Os resultados e discussões a respeito dessa implementação apresentam-se descritos na Seção 4.2.

3.3.5.1 Calibração do Sensor

O nível de oxigênio, SpO_2, não é obtido diretamente a partir do valor de R_r. Existe uma relação entre os dois, e ela não é linear, como visto em [18]. Desse modo, é necessário definir um modelo matemático, que demonstre uma relação como a vista na Equação (3.7).

$$SpO_2(\%) = f(R_r) \tag{3.7}$$

O método empírico foi escolhido para ser utilizado nesse cenário, em concordância com as as referências [18], e com a metodologia descrita em 2.3.2. Nesse caso, foram feitas uma série de medidas, em 5 pessoas diferentes, relacionando o valor calculado para R_r, e o do SpO_2. Para melhor calibração, um número maior deveria ser escolhido, porém, no que diz respeito ao foco deste trabalho, que está em analisar o desempenho e perfil de consumo, o valor escolhido é suficiente.

Desse modo, a Tabela 3.1 foi definida, pois, para a aplicação pretendida, não é necessário que o SpO_2 apresente valor de ponto flutuante, devido ao fato de que os próprios sensores utilizados por médicos não possuem tal precisão. Valores diferentes dos mostrados na tabela são identificados como 0 pela aplicação.

$SpO_2\%$	R_r
100	[0,891 – 1,0]
99	[0,815 – 0,891[
98	[0,741 – 0,815[
97	[0,666 – 0,741[
96	[0,591 – 0,666[
95	[0,516 – 0,591[
94	[0,441 – 0,516[
93	[0,367 – 0,441[
92	[0,292 – 0,367[
91	[0,215 – 0,292[
90	]0.143 – 0,215[

Tabela 3.1: Tabela de Mapeamento SpO_2 para intervalos de valores do R_r.

4 RESULTADOS E DISCUSSÕES

4.1 CÁLCULO DOS BATIMENTOS

Os batimentos foram calculados em 3 plataformas: Microcontrolador CC2640R2F, compilador GCC e MATLAB. Os dados do microncontrolador foram obtidos a partir da interface serial UART. Desse modo, os resultados, para 5 amostras diferentes advindas do sensor ótico, estão presentes na Tabela 4.1. As 4 primeiras foram feitas com o indivíduo em repouso, a última, após a realização de atividade física intensa. O valor real foi aferido a partir do sensor de pressão e batimentos cardíacos LP200.

Amostra	Valor Real	Microcontrolador	GCC	MATLAB
1	98	99	99	100
2	110	111	111	112
3	85	87	87	90
4	96	99	99	102
5	138	120	120	125

Tabela 4.1: Comparativo de Resultados do Algoritmo de Batimentos.

Os valores do microcontrolador e do compilador GCC são iguais, devido ao foto de utilizarem os mesmos algoritmos, e também as mesmas precisões de ponto flutuante (32 bits). Já o MATLAB tem uma precisão de ponto flututante maior (64 bits), é por conta disso que os resultados são diferentes dos outros.

No que se tange ao cenário da aplicação médica, os 4 primeiros resultados apresentaram um erro de máximo de 3 batimentos, no caso do compilador GCC e do microcontrolador, e de 5 batimentos para o caso do MATLAB. Já no caso da amostra de número 5, as diferenças foram mais expressivas, totalizando 18 batimentos no caso do microntrolador e do compilador GCC, e de 15 batimentos para o MATLAB. Isso mostra que o algoritmo, em conjunto com o sensor MAX30100, não possue precisão confiável para monitorar os batimentos do paciente que tenha realizado atividade física recentemente, porém, no cenário onde ele está em repouso, o margem de erro possui um valor dentro dos padrões, tomando como referência o fato de que o próprio sensor LP200 apresenta um grau de erro em suas medidas.

4.2 CÁLCULO DA OXÍMETRIA

Os valores de R_r e SpO_2 foram calculados em 2 plataformas: Microcontrolador CC2640R2F e compilador GCC. Os dados advindos do microcontrolador foram obtidos com uso da interface serial UART. Os resultados, para 5 amostras diferentes advindas do sensor ótico, estão presentes na Tabela 4.1. O valor real foi medido pelo sensor de oximetria da Elera, amplamente difundido

no mercado.

Amostra	Valor Real (Somente Nível de Oxigênio)	Microcontrolador	GCC
1	97%	0,732 e 97%	0,732 e 97%
2	96%	0,680 e 97%	0,680 e 97%
3	98%	0,733 e 97%	0,733 e 97%
4	99%	0,773 e 98%	0,773 e 98%
5	97%	0,731 e 97%	0,731 e 97%

Tabela 4.2: Comparativo de Resultados do Algoritmo de Oxímetria.

Como pode ser visto, não ouve diferença nos valores calculados pelo microcontrolador e pelo compilador GCC, isso se deve ao fato de suas rotinas estarem iguais, com diferença somente na declaração de variáveis do tipo inteiro. Essa igualdade se deve fato de ambos operarem com a mesma precisão para variáveis do tipo float. Em comparação com o resultado real, ouve diferença em 3 das amostras, isso pode ser explicado pela calibração não ser muito precisa, porém, a diferença de 1% para mais ou para menos está dentro da margem do considerado seguro para a aplicação pretendida, ou seja, não impactaria na análise do estado de saúde do paciente.

4.3 DESEMPENHO

A análise de desempenho é relativa ao tempo de execução dos algoritmos descritos em 3.3.4 e 3.3.5. Para isso, foi feita a medição do tempo dos algoritmos no MATLAB, na linguagem C em um notebook da lenovo com processador Core i7 2,1GHz e 4 GB de RAM, e no microcontrolador CC2640R2F.

O cenário montado utiliza a Launchpad provida pela Texas Instruments, e acoplado em seus pinos relativos ao I2C (SDA e SCL), está o sensor ótico MAX30100, que provê os dados a serem utilizados pelos algoritmos de oxímetria e batimentos cardíacos.

O desempenho é medido com uso de funções que marcam o tempo de início e fim da execução dos algoritmos, e por fim realiza a subtração entre os dois para obter o tempo total de execução. Além das rotinas dos algoritmos, também foram medidos os tempos de execução do cálculo da FFT, do bloqueio dc e da filtragem de sinais.

Função	Microcontrolador	GCC	MATLAB
Bloqueio DC	1,377	0,03148	0,000044
FFT	0,007	0,00017	0,000097
Filtro	0,724	0,01655	0,000042
BPM	2,71	0,06200	0,0012
Oxímetria	0,306	0,00700	0,0000148

Tabela 4.3: Comparativo do Tempo de Execução das Rotinas.

Como pode ser visto, o MATLAB é significativamente mais rápido que as outras duas plataformas, isso se deve ao foto dos seus algoritmos serem otimizados. Somando-se o tempo para

cálculo dos batimentos, e de oxímetria, o microcontrolador levaria aproximadamente 3 segundos para executar os dois algoritmos. Como os dados advindos dos sensores são lidos a cada 10 segundos, de acordo com o tempo escolhido para a rotina periódica, o tempo de 3 segundos é suficiente para execução dos dois algoritmos, antes que uma nova amostra seja requisitada.

4.4 ANÁLISE DE MEMÓRIA

A análise de memória foi feita a partir de ferramentas da própria IDE onde o código foi escrito, e também do arquivo com extensão '.map' que é gerado pelo Code Composer Studio após o firmware ser compilado. Nesse ponto, busca-se verificar se é viável ou não o uso do microcontrolador citado, tendo como base o quanto de memória RAM e Flash o programa utiliza.

Os resultados para análise de memória estão presentes na Tabela 4.4. Os valores em hexadecimal foram extraídos do arquivo '.map'.

Memória	Tamanho (KB)	Usada (KB)	Livre (KB)
Flash	126,976	73,520	53,426
SRAM	17,408	9,423	8,165

Tabela 4.4: Tabela de resumo da memória.

Como pode ser visto, a memória flash tem o tamanho de 126,976 KB, e o programa está utilizando 73,520 KB, o espaço que sobra é portanto 53,456 KB, esse valor representa 42,09% da memória total disponível, uma margem alta para possível expansão da aplicação. No caso da memória RAM, que possui 17,408 KB, está sendo utilizado 9,423 KB, pelas funções, variáveis estáticas e threads, isso faz com que sobre 8,165 KB, a ser utilizado pelas variáveis que surgirem ao longo da execução. Esse valor livre representa 46,9% do total de memória, suficiente para atender o espaço para as variáveis que são declaradas e inicializadas ao longo do programa. Vale salientar também que a memória cache de 8 KB ainda pode ser utilizada para expandir a RAM.

4.5 ANÁLISE DE CONSUMO ENERGÉTICO

A análise de consumo energético possui como objetivo principal validar o uso do microcontrolador CC2640R2F para aplicações que possuam baterias com tamanho reduzido. Em consequência, essas baterias têm uma duração menor quando comparada a de smartphones por exemplo. A análise foi feita com base em [19], pois provê informações a respeito de como deixar a medição o mais limpa possível, em outras palavras, sem que módulos adicionais drenem corrente, haja vista que o microcontrolador está sendo utilizadao a partir da launchpad vendida pela Texas Instruments.

O primeiro cenário, analisado na Seção 4.5.1 de testes utiliza um Arduino UNO para fornecer energia, e medir continuamente, através de sua entrada analógica, e um resistor de *shunt* (1

Ohm), a corrente drenada pelo microcontrolador, na presença da aplicação Bluetooth. O resistor de 1 Ohm foi escolhido, devido ao fato da queda de tensão em seus terminais, que será lida pelo arduino, ser matematicamente igual a corrente. Foi feita a medição no cenário onde a aplicação estivesse em estado de ”Advertising"e também quando um dispositivo qualquer estivesse conectado e requisitando informações.

O segundo cenário, analisado na Seção 4.5.2 utiliza a launchpad, que contém o microcontrolador, ligada ao computador via cabo USB. Dessa forma, é possível utilizar o plugin Energy Tracer, presente na IDE Code Composer Studio, esse plugin permite fazer uma análise de estados do microcontrolador, indicando por quanto tempo cada periférico, e as partem que os compõem, ficou ativo.

Com essas informações, e os dados técnicos de consumo presente em [9] foi possível obter uma estimativa de consumo energético da aplicação.

4.5.1 Análise Direta

A primeira análise, com os dados de consumo advindos do Arduino, gerou o gráfico presente na Figura 4.1. Os valores foram capturados em uma janela de 10 segundos, pois é o tempo total de um ciclo do evento períodico.

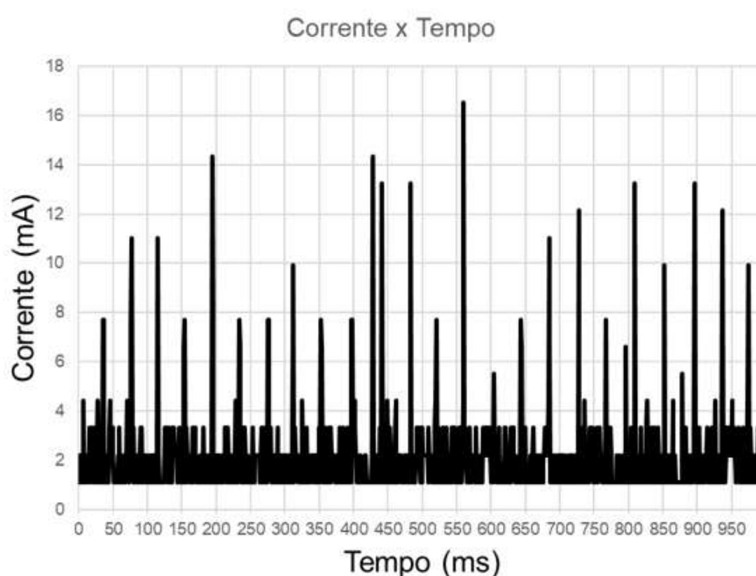


Figura 4.1: Gráfico Consumo de Corrente no Tempo.

A partir dos dados presentes nele, foi extraído o consumo médio de corrente, que resultou no valor de 2.11 mA, com pico máximo de 16.53 mA. Isso mostra que uma bateria de 150 mAh, que é relativamente pequena, teria uma duração de 71.09 horas, que implica em uma vida útil, da bateria, aproximadamente 3 vezes melhor que a pretendida nos objetivos descritos na Seção 1.2, antes de ser recarregada novamente.

4.5.2 Análise de Estados

A segunda análise, que fez uso do Energy Tracer, tem como resultado a porcentagem de tempo em que cada módulo estava ativo em determinado intervalo, conforme mostrado na Figura 4.2. O tempo de análise escolhido foi de 100 segundos, pois é um multiplo de 10 segundos, que é o tempo total de um ciclo do evento periódico, responsável por executar os algoritmos, e também pois o no Energy Tracer valor é dado em porcentagem, dessa forma, 2% representa exatamente 2 segundos no qual o módulo estaria ativo, simplificando os cálculos. À esquerda, têm-se os resultados com o Bluetooth ativo mas sem conexão, e na direita o Bluetooth esteve ativo e um dispositivo conectado requisitando informações.

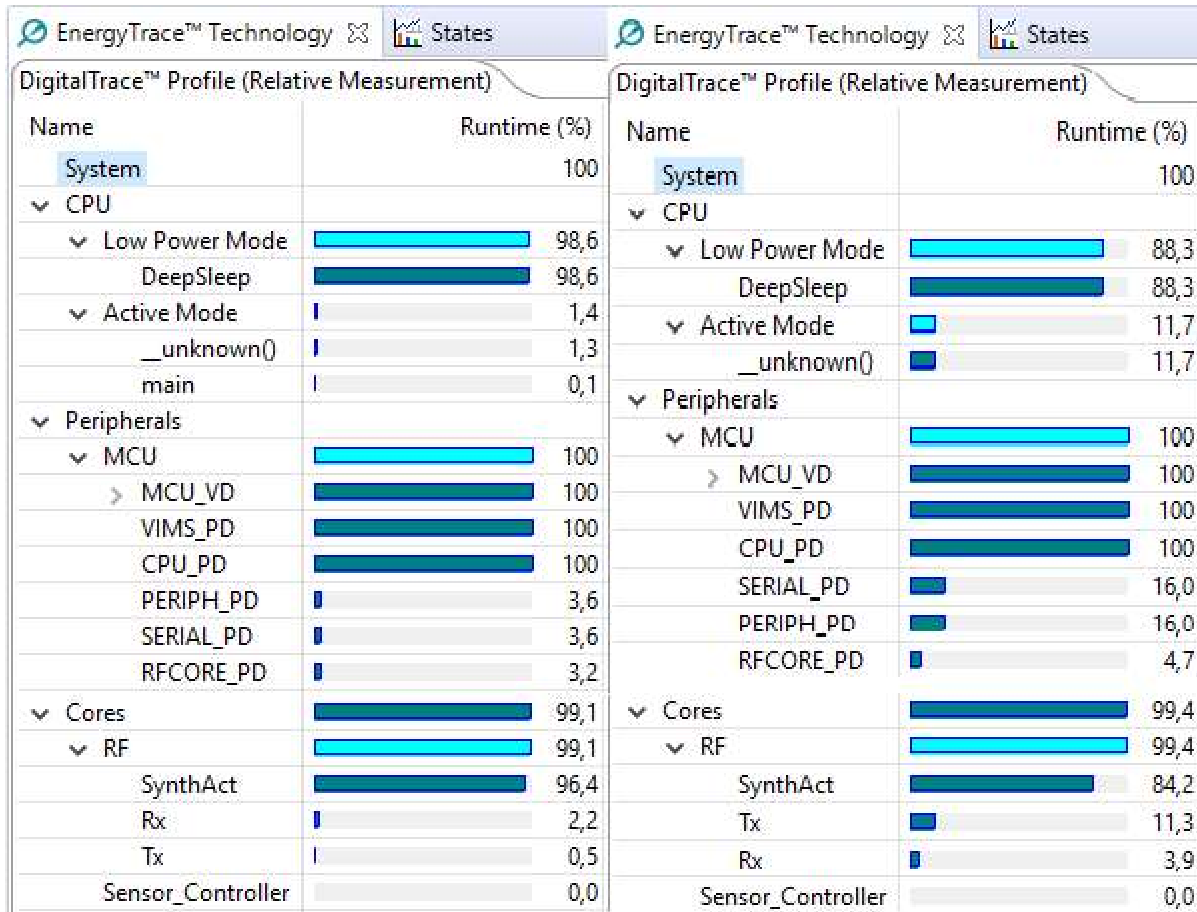


Figura 4.2: Estados do Microcontrolador, na esquerda, a execução com o Bluetooth ativo, porém com conexão inativa, e na direita o Bluetooth ativo, com a conexão também ativa.

O consumo total de energia em 100 segundos para cada um dos cenários pode ser sumarizado a partir do modelo presente na equação 4.1.

$$C_{\text{total}} = C_{\text{CPU}} + C_{\text{PH}} + C_{\text{Cores}}, \tag{4.1}$$

Sendo C_{CPU}, C_{PH} e C_{Cores}, que representam o consumo de corrente do respectivo módulo, obtidos a partir das Equações 4.2, 4.3 e 4.4.

$$C_{\text{CPU}} = \frac{LPM_{\text{corrente}} \cdot LPM_{\text{tempo}} + AM_{\text{corrente}} \cdot AM_{\text{tempo}}}{100}, \tag{4.2}$$

$$C_{\text{PH}} = \frac{PPD_{\text{corrente}} \cdot PPD_{\text{tempo}} + SPD_{\text{corrente}} \cdot SPD_{\text{tempo}} + RFPD_{\text{corrente}} \cdot RFPD_{\text{tempo}}}{100}, \tag{4.3}$$

$$C_{\text{Cores}} = \frac{TX_{\text{corrente}} \cdot TX_{\text{tempo}} + RX_{\text{corrente}} \cdot RX_{\text{tempo}}}{100}, \tag{4.4}$$

A Tabela 4.5, extraída a partir de [9], mostra o consumo de corrente para cada um dos módulos.

Módulo	Consumo (mA)
LPM	0.003
AM	1.45 + 31μA/MHz
PPD	0.02
SPD	0.02
RFPD	0.237
TX	6.1
RX	5.9

Tabela 4.5: Consumo de Corrente Módulos CC2640R2F.

Os demais módulos não possuem um cosumo significante de corrente, ou não são mapeados pelo Energy Tracer, como é o caso do ADC, e por isso não estão sendo considerados no equacionamento.

Dessa forma, o consumo de corrente médio obtido, para o primeiro caso de não haver conexão estabelecida no bluetooth, foi de 0.21 mA, e para o segundo caso, onde havia conexão, foi de 1.282 mA.

No pior dos casos, que é com o Bluetooth ativo, uma bateria de 150 mAh duraria aproximadamente 117 horas, divergindo do valor obtido em 4.5.1. Isso se deve ao fato de que o energy tracer não consegue medir o consumo do ADC, que esteve ativo durante toda a aplicação, para converter os resultados oriundos de um sensor de temperatura, que, segundo [9], consome 0.75 mA quando ativo.

É possível portanto, assumir para o segundo caso, um consumo energético em torno de $C_{\text{total}} = 2,032\text{mA}$, o que deixa próximo os valores obtidos nos dois tipos de análise. A duração da bateria de aproximadamente 73,82 horas, medida de acordo com o método da análise de estados, representa uma duração aproximadamente 3 vezes maior que a dos objetivos descritos em 1.2.

5 CONCLUSÕES

O estudo aqui presente visou contribuir para o meio acadêmico no que tange as possibilidades de implementação de algoritmos em sistemas embarcados, principalmente com foco no baixo consumo energético, com uma clara preocupação na miniaturização dos componentes eletrônicos, principalmente microcontroladores e sensores.

Os resultados presentes nesse trabalho mostraram como se comportou, no que tange ao desempenho e consumo energético, o microcontrolador CC2640R2F na execução de rotinas e algoritmos complexos, com base nas características esperadas pela aplicação a qual ele se destina. É possível verificar que ele atende os objetivos citados em 1.2, e também, vale deixar claro que o diferencial é seu consumo de energia baixo, aliado a um bom desempenho, e memória que atende os requisitos previamente propostos.

Vale deixar exposto o fato de que mais algoritmos podem ser implementados, como por exemplo a detecção de passos, detecção de quedas e pressão sanguínea, que não foram estudados nesse trabalho, mas que podem sim ser incorporados, desde que os métodos utilizados sigam a premissa inicial de serem não invasivos, de modo que não gerem incomodo ou desconforto ao paciente.

A comparação de métodos mais eficientes para os cálculos de oximetria e batimentos cardíacos também é válida, diferentes algoritmos não foram abordados nesse trabalho devido ao fato do foco principal ser ligado a análise de consumo e desempenho do microcontrolador CC2640R2F na execução de algoritmos com alta complexidade computacional.

É importante salientar a flexibilidade fornecida pelo bluetooth na aplicação implementada, que, por ser um protocolo altamente difundido nos dispositivos eletrônicos, permite uma facilidade na comunicação, sem grande demanda de adaptação, desde que se usem as especificações da norma presente em [13].

Os algoritmos para cálculo do nível de oxigênio e batimentos cardíacos também apresentaram bons resultados, mesmo que tenham divergido dos valores reais, essa diferença é baixa, quando levado em conta os requisitos de precisão da aplicação, com exceção do valor para o batimento cardíaco obtido após a realização de atividade física. Parte dessas divergências são explicadas pelos ruído presente, que mesmo após filtragem, se mantêm, devido ao fato dos cabos e terminais de conexão introduzirem uma interferência. Devido a isso, destaca-se a possibilidade de usar o microcontrolador em uma PCB, com os sensores ligados diretamente, isso diminui o ruído presente na aplicação, e gera resultados mais robustos e precisos.

Em resumo, é possível perceber que o avanço na tecnologia dos microcontroladores surgiu para expandir as aplicações no que tange a área de internet das coisas, principalmente quando ligada a outras áreas, como por exemplo a medicina. Com o passar do tempo, essas relações tendem a aumentar, e se solidificar cada vez mais como sendo uma ferramenta importante no que se diz respeito a vida cotidiana das pessoas que compõem a sociedade atual e futura.

REFERÊNCIAS BIBLIOGRÁFICAS

1 MAJUMDER, S.; MONDAL, T.; DEEN, M. J. *Wearable Sensors for Remote Health Monitoring*. 2017.

2 GARGIULO, G.; BIFULCO, P.; CESARELLI, M.; JIN, C.; MCEWAN, A.; SCHAIK, A. V. Wearable dry sensor with bluetooth connection for use in remote patient monitoring systems. *Global Telehealth*, 2010.

3 STRANGMAN, G.; BOAS, D. A.; SUTON, J. P. Non-invasive neuroimaging using near-infrared light. *Biological Psychiatry*, v. 52, 2002.

4 FOLKE, M.; CERNERUD, L.; EKSTROM, L.; HOK, B. Critical review of non-invasive respiratory monitoring in medical care. *Medical and Biological Engineering and Computing*, v. 41, 2003.

5 FORTIN, J.; HABENBACHER, W.; HELLER, A.; HACKER, A.; GRULLENBERGER, R.; INNERHOFER, J.; PASSATH, H.; WAGNER, C.; HAITCHI, G.; FLOTZINGER, D.; PACHER, R.; WACH, P. Non-invasive beat-to-beat cardiac output monitoring by an improved method of transthoracic bioimpedance measurement. *Computers in Biology and Medicine*, v. 36, 2006.

6 PRETTZ, J. B.; COSTA, J. P. C. L. da; ALVIM, J. R.; MIRANDA, R. K. Efficient and low cost mimo communication architecture for smartbands applied to postoperative patient care. 2017.

7 *Power consumption analysis of Bluetooth Low Energy, ZigBee and ANT sensor nodes in a cyclic sleep scenario*.

8 *How low energy is bluetooth low energy? Comparative measurements with ZigBee/802.15.4*.

9 TEXAS INSTRUMENTS. *CC13X0, CC26X0 SimpleLink Wireless MCU - Technical Reference Manual*. [S.l.], 2015.

10 INSTRUMENTS, T. *Drive Library SDK CC2640R2F*. 2017. Disponível em: <http://software-dl.ti.com/simplelink/esd/simplelink_cc2640r2_sdk/1.50.00.58/exports/docs/driverlib_cc13xx_cc26xx/cc26x0/driverlib/index.html>.

11 HERNES, M. *Increase RAM Size on the CC2640R2F Bluetooth low energy Wireless MCU*. [S.l.], 2017.

12 FEKI, M. A.; KAWSAR, F.; BOUSSARD, M.; TRAPPENIERS, L. The internet of things: The next technological revolution. *Computer*, v. 46, 2013.

13 BLUETOOTH SIG PROPRIETARY. *BLUETOOTH SPECIFICATION Version 4.2 [Vol 0]*. [S.l.], 2014.

14 GOMEZ, C.; OLLER, J.; PARADELLS, J. Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology. *Sensors*, 2012.

15 G.P.M, P. *Development of a Healthcare Wearable Platform Using a Photoplethysmography Sensor*. 2018.

16 LOPEZ, S. *Pulse Oximeter Fundamentals and Design*. [S.l.], 2012.

17 NITZAN, M.; ROMEM, A.; KOPPEL, R. Pulse oximetry: Fundamentals and technology update. *Medical Device: Evidence and Research*, 2014.

18 STUBAN, N.; MASATSUGU, N. Non-invasive calibration method for pulse oximeters. *Periodica Polytechnica*, 2008.

19 LEE, J. L. C.; HERNES, M. *Measuring Bluetooth Low Energy Power Consumption*. [S.l.], 2017.

APÊNDICES

I CÓDIGO FONTE

```c
/*
 * algorithms.c
 *
 * Author: Marcos Assis
 *
 */

#include <stdint.h>
#include <stddef.h>
#include <stdlib.h>

#include "matlab_aux.h"

uint8_t fft_HR(float_t *signal, uint16_t signal_size, float_t fs, float_t
    time_window);
uint8_t oximetry(float_t *signal_ir, float_t *signal_red, uint16_t signal_size,
    float_t fs, float_t time_window);

uint8_t fft_HR(float_t *signal, uint16_t signal_size, float_t fs, float_t
    time_window)
{
    uint8_t bpm_estimate = 0;

    double_t butter_b[] = {0.0250E-04, 0.1498E-04, 0.3746E-04, 0.4994E-04, 0.3746E
        -04, 0.1498E-04, 0.0250E-04};
    double_t butter_a[] = {1.0, -5.02944, 10.60704, -11.99932, 7.67547, -2.63106,
        0.37745};

    uint16_t signal_noart_size;
    float_t *signal_noart;
    signal_noart = remove_artifacts(signal, signal_size, &signal_noart_size);
    free(signal_noart);

    uint16_t signal_nodc_size = signal_noart_size;
    float_t signal_nodc[signal_nodc_size];
    dc_blocker(signal_nodc, signal_noart, 0.95, signal_noart_size);

    uint16_t signal_filt_size = signal_nodc_size;
    float_t signal_filt[signal_filt_size];
    matlab_filter(signal_filt, butter_b, butter_a, signal_nodc, 0.0,
        signal_nodc_size, BUTTER_ORDER+1);

    uint16_t filter_convergence = calc_filter_convergence(fs);
    uint16_t signal_zeros_size = time_window*fs + 1000;
    float_t signal_zeros[signal_zeros_size];
```

```
40      get_subvector(signal_zeros, signal_filt, filter_convergence,
            filter_convergence + (time_window*fs));
41
42      uint16_t signal_fft_size = 1024;
43      Complex_Number signal_fft[signal_fft_size];
44      uint16_t signal_cplx_size = signal_fft_size;
45      Complex_Number signal_cplx[signal_cplx_size];
46      for(int i = 0; i < signal_fft_size; i++)
47      {
48          Complex_Number z;
49          z.real = signal_zeros[i];
50          z.imag = 0.0;
51          signal_fft[i] = z;
52      }
53      matlab_fft(signal_fft, signal_cplx, signal_fft_size);
54
55      float_t freqs[signal_fft_size];
56      matlab_linspace(freqs, 0, fs, signal_fft_size);
57      bpm_estimate = bpm_fft(signal_fft, freqs, signal_fft_size);
58
59      return bpm_estimate;
60  }
61
62  uint8_t oximetry(float_t *signal_ir, float_t *signal_red, uint16_t signal_size,
        float_t fs, float_t time_window)
63  {
64      double_t butter_b[] = {0.0250E-04, 0.1498E-04, 0.3746E-04, 0.4994E-04, 0.3746E
            -04, 0.1498E-04, 0.0250E-04};
65      double_t butter_a[] = {1.0, -5.02944, 10.60704, -11.99932, 7.67547, -2.63106,
            0.37745};
66
67      float_t signal_ir_nodc[signal_size];
68      dc_blocker(signal_ir_nodc, signal_ir, 0.95, signal_size);
69
70      float_t signal_ir_filt[signal_size];
71      matlab_filter(signal_ir_filt, butter_b, butter_a, signal_ir, 0.0, signal_size,
             BUTTER_ORDER+1);
72
73      float_t signal_red_nodc[signal_size];
74      dc_blocker(signal_red_nodc, signal_red, 0.95, signal_size);
75
76      float_t signal_red_filt[signal_size];
77      matlab_filter(signal_red_filt, butter_b, butter_a, signal_red, 0.0,
            signal_size, BUTTER_ORDER+1);
78
79      float_t ir_rms, red_rms, r_factor;
80      uint8_t oxygen_level;
81
82      ir_rms = rms_signal(signal_ir_filt, signal_size);
83      red_rms = rms_signal(signal_red_filt, signal_size);
84      r_factor = get_r_factor(ir_rms, red_rms, 940.0, 640.0);
85      oxygen_level = map_r_oxygen(r_factor);
```

```
86
87     return oxygen_level;
88 }
```

```
1  /*
2   * matlab_aux.h
3   *
4   * Author: Marcos Assis
5   */
6
7  #ifndef APPLICATION_MATLAB_AUX_H_
8  #define APPLICATION_MATLAB_AUX_H_
9
10 //Includes
11 #include <math.h>
12 #include <stdint.h>
13 #include <stddef.h>
14 #include <stdlib.h>
15
16 //Definitions
17 #define BUTTER_ORDER 6
18
19 // Structs
20 typedef struct Complex_Number{
21     double_t real;
22     double_t imag;
23 } Complex_Number;
24
25 typedef struct Signal{
26     float_t data;
27     uint16_t size;
28 } Signal;
29
30 //Functions
31 uint8_t bpm_fft(Complex_Number *signal, float_t *freqs, uint16_t size);
32 uint8_t map_r_oxygen(float_t r_factor);
33 uint16_t calc_filter_convergence(float_t sample_rate);
34 float_t get_r_factor(float_t ir_rms, float_t red_rms, float_t ir_wavelength,
       float_t red_wavelength);
35 float_t rms_signal(float_t *signal, uint16_t signal_size);
36 float_t matlab_mean(float_t *array, uint16_t size);
37 float_t * remove_artifacts(float_t *signal, uint16_t size, uint16_t *new_size);
38 void matlab_findpeaks(uint16_t *peaks, float_t *signal, uint16_t size, uint16_t *
       output_size);
39 void get_subvector(float_t *subvector, float_t *vector, uint16_t begin_index,
       uint16_t end_index);
40 void matlab_xcorr(float_t *signal_xcorr, float_t *signal, uint16_t size);
41 void matlab_linspace(float_t *out, float_t init, float_t end, uint16_t points);
42 void matlab_fft(Complex_Number *signal_fft, Complex_Number *signal, uint16_t n);
43 void dc_blocker(float_t *nodc_signal, float_t *signal, float_t dc_blocker_ratio,
```

```c
    uint16_t size);
void matlab_filter(float_t *y, double_t *b, double_t *a, float_t *signal, double_t
     zi, uint16_t size_signal, uint16_t size_a);
void matlab_diff(float_t *y, float_t *x, uint16_t size);
void matlab_logical(uint16_t *y, float_t *x, float_t *signal, uint16_t size,
    uint16_t *valid_size);
void logical_cut_signal(float_t *valid_signal, uint16_t *logical_vector, float_t *
    x, uint16_t size, uint16_t new_size);

#endif /* APPLICATION_MATLAB_AUX_H_ */
```

```c
/*
 * matlab_aux.c
 *
 * Author: Marcos Assis
 */

#include "matlab_aux.h"

uint8_t bpm_fft(Complex_Number *signal, float_t *freqs, uint16_t size)
{
    double_t interval[size];
    interval[0] = 0;
    uint16_t i;
    uint8_t bpm;
    double_t max_value = 0.0;
    uint16_t max_value_index = 0;

    for(i = 1; i < size; i++)
    {
        interval[i] = sqrt((signal[i].real*signal[i].real) + (signal[i].imag*
            signal[i].imag));
    }

    max_value = interval[0];

    for(i = 0; i < size/2; i++)
    {
        if(interval[i] > max_value)
        {
            max_value = interval[i];
            max_value_index = i;
        }
    }

    bpm = freqs[max_value_index]*60;

    return bpm;
}

```

```
39  float_t matlab_mean(float_t *array, uint16_t size)
40  {
41      float_t mean;
42      float_t sum = 0;
43      uint16_t i;
44  
45      for(i = 0; i < size; i++)
46      {
47          sum += array[i];
48      }
49  
50      mean = sum/size;
51  
52      return mean;
53  }
54  
55  float_t rms_signal(float_t *signal, uint16_t signal_size)
56  {
57      uint16_t i;
58      float_t pow_sample;
59      float_t sum = 0.0;
60      float_t rms_sample;
61  
62      for(i = 0; i < signal_size; i++)
63      {
64          pow_sample = signal[i]*signal[i];
65          sum += pow_sample;
66      }
67  
68      rms_sample = sqrt((sum/signal_size));
69      return rms_sample;
70  }
71  
72  float_t get_r_factor(float_t ir_rms, float_t red_rms, float_t ir_wavelength,
        float_t red_wavelength)
73  {
74      float_t r_factor;
75  
76      r_factor = (log10(red_rms)*red_wavelength)/(log10(ir_rms)*ir_wavelength);
77  
78      return r_factor;
79  }
80  
81  uint8_t map_r_oxygen(float_t r_factor)
82  {
83      if(r_factor >= 0.143 && r_factor < 0.215)
84      {
85          return 90;
86      }
87      else if(r_factor >= 0.215 && r_factor < 0.292)
88      {
89          return 91;
```

```
90      }
91      else if(r_factor >= 0.292 && r_factor < 0.367)
92      {
93          return 92;
94      }
95      else if(r_factor >= 0.367 && r_factor < 0.441)
96      {
97          return 93;
98      }
99      else if(r_factor >= 0.441 && r_factor < 0.516)
100     {
101         return 94;
102     }
103     else if(r_factor >= 0.516 && r_factor < 0.591)
104     {
105         return 95;
106     }
107     else if(r_factor >= 0.591 && r_factor < 0.666)
108     {
109         return 96;
110     }
111     else if(r_factor >= 0.666 && r_factor < 0.741)
112     {
113         return 97;
114     }
115     else if(r_factor >= 0.741 && r_factor < 0.815)
116     {
117         return 98;
118     }
119     else if(r_factor >= 0.815 && r_factor < 0.891)
120     {
121         return 99;
122     }
123     else if(r_factor >= 0.891 && r_factor < 1.0)
124     {
125         return 100;
126     }
127     else
128     {
129         return 0;
130     }
131 }
132
133 void matlab_findpeaks(uint16_t *peaks, float_t *signal, uint16_t size, uint16_t *
        output_size)
134 {
135     uint16_t i, j;
136     uint16_t new_size = 0;
137
138     for(i = 1; i < (size-1); i++)
139     {
140         if((signal[i] > signal[i-1]) && (signal[i] > signal[i+1]))
```

```
141             {
142                 new_size++;
143             }
144         }
145 
146         j = 0;
147 
148         for(i = 1; i < (size-1); i++)
149         {
150             if((signal[i] > signal[i-1]) && (signal[i] > signal[i+1]))
151             {
152                 peaks[j] = signal[i];
153                 j++;
154             }
155         }
156 
157         *output_size = j;
158     }
159 
160     void get_subvector(float_t *subvector, float_t *vector, uint16_t begin_index,
            uint16_t end_index)
161     {
162         uint16_t i;
163 
164         for(i = begin_index; i <= end_index + 1000; i++)
165         {
166             if(i > end_index)
167             {
168                 subvector[i - begin_index] = 0.0;
169             }
170             else
171             {
172                 subvector[i - begin_index] = vector[i];
173             }
174 
175         }
176     }
177 
178     void matlab_xcorr(float_t *signal_xcorr, float_t *signal, uint16_t size)
179     {
180         uint16_t n, m;
181 
182         for(n = 0; n < size; n++)
183         {
184             for(m = 0; m < size; m++)
185             {
186                 signal_xcorr[n] += signal[m]*signal[m+n];
187             }
188         }
189     }
190 
191     void matlab_linspace(float_t *out, float_t init, float_t end, uint16_t points)
```

```
192 {
193     float_t space = (end - init)/points;
194     uint16_t i;
195
196     for(i = 0; i < points; i++)
197     {
198         out[i] = i*space;
199     }
200 }
201
202 void matlab_fft(Complex_Number *v, Complex_Number *tmp, uint16_t n)
203 {
204     if(n>1)
205     {           /* otherwise, do nothing and return */
206         uint16_t k,m;    Complex_Number z, w, *vo, *ve;
207         ve = tmp; vo = tmp+n/2;
208         for(k=0; k<n/2; k++)
209         {
210             ve[k] = v[2*k];
211             vo[k] = v[2*k+1];
212         }
213         matlab_fft(ve,v,n/2);     /* FFT on even-indexed elements of v[] */
214         matlab_fft(vo,v,n/2);     /* FFT on odd-indexed elements of v[] */
215         for(m=0; m<n/2; m++)
216         {
217             w.real = cos(2*M_PI*m/(double_t)n);
218             w.imag = -sin(2*M_PI*m/(double_t)n);
219             z.real = w.real*vo[m].real - w.imag*vo[m].imag; /* real(w*vo[m]) */
220             z.imag = w.real*vo[m].imag + w.imag*vo[m].real; /* imag(w*vo[m]) */
221             v[  m  ].real = ve[m].real + z.real;
222             v[  m  ].imag = ve[m].imag + z.imag;
223             v[m+n/2].real = ve[m].real - z.real;
224             v[m+n/2].imag = ve[m].imag - z.imag;
225         }
226     }
227 }
228
229
230 uint16_t calc_filter_convergence(float_t sample_rate)
231 {
232     uint16_t filter_convergence;
233     sample_rate = 150*log10(sample_rate);
234
235     if(round(sample_rate) <= sample_rate)
236     {
237         filter_convergence = round(sample_rate);
238         return filter_convergence;
239     }
240     else
241     {
242        filter_convergence = round(sample_rate) - 1;
243        return filter_convergence;
```

```
244         }
245     }
246 
247     void dc_blocker(float_t *nodc_signal, float_t *signal, float_t dc_blocker_ratio,
        uint16_t size)
248     {
249         double_t b[] = {1.0, -1.0};
250         double_t a[] = {1.0, (-1.0)*dc_blocker_ratio};
251         double_t zi = signal[0];
252 
253         matlab_filter(nodc_signal, b, a, signal, zi, size, 2);
254     }
255 
256     void matlab_filter(float_t *y, double_t *b, double_t *a, float_t *signal, double_t
         zi, uint16_t size_signal, uint16_t size_a)
257     {
258         uint16_t i, j;
259         double_t Ym;
260         Ym = zi;
261         for(i = 0; i < size_signal; i++)
262         {
263             if(i >= size_a)
264             {
265                 Ym = 0.0;
266                 for(j = 0; j < size_a; j++)
267                 {
268                     Ym += b[j]*signal[(i)-j];
269                 }
270                 for(j = 1; j < size_a; j++)
271                 {
272                     Ym -= a[j]*y[(i)-j];
273                 }
274             }
275             y[i] = Ym;
276         }
277     }
278 
279     float_t * remove_artifacts(float_t *signal, uint16_t size, uint16_t *new_size)
280     {
281         float_t diff_vector[size];
282         uint16_t logical_vector[size];
283         uint16_t valid_size;
284 
285         matlab_diff(diff_vector, signal, size);
286         matlab_logical(logical_vector, diff_vector, signal, size, &valid_size);
287 
288         float_t *noart_signal = (float_t *) malloc(sizeof(float_t)*valid_size);
289         logical_cut_signal(noart_signal, logical_vector, signal, size, valid_size);
290         *new_size = valid_size;
291 
292         return noart_signal;
293     }
```

```
294
295 void matlab_diff(float_t *y, float_t *x, uint16_t size)
296 {
297     uint16_t i;
298
299     y[0] = 0;
300
301     for(i = 1; i < size; i++)
302     {
303         y[i] = x[i] - x[i-1];
304     }
305 }
306
307 void matlab_logical(uint16_t *y, float_t *x, float_t *signal, uint16_t size,
        uint16_t *valid_size)
308 {
309     uint16_t i;
310     uint16_t new_size = 0;
311
312     for(i = 0; i < size; i++)
313     {
314         if((fabs(x[i]) > 100.0) || (fabs(signal[i]) < 100.0))
315         {
316             y[i] = 0;
317         }
318         else
319         {
320             new_size++;
321             y[i] = 1;
322         }
323     }
324
325     *valid_size = new_size;
326 }
327
328 void logical_cut_signal(float_t *valid_signal, uint16_t *logical_vector, float_t *
        x, uint16_t size, uint16_t new_size)
329 {
330     uint16_t i;
331     uint16_t j = 0;
332
333     for(i = 0; i < size; i++)
334     {
335         if(logical_vector[i])
336         {
337             valid_signal[j] = x[i];
338             j++;
339         }
340     }
341 }
```

```c
/*
 * i2c.c
 *
 * Author: Marcos Assis
 */

#include <stdint.h>
#include <stddef.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>

/* Driver Header files */
#include <ti/drivers/I2C.h>

/* Board Header file */
#include "Board.h"


uint16_t get_i2c_value_bpm(float_t *buffer, uint16_t slave_address, uint8_t bytes)
    ;
uint16_t get_i2c_value_oxygen(float_t *buffer_ir, float_t *buffer_red, uint16_t
    slave_address, uint8_t bytes);


uint16_t get_i2c_value_oxygen(float_t *buffer_ir, float_t *buffer_red, uint16_t
    slave_address, uint8_t bytes)
{
    uint8_t          tx_buf[1];
    uint8_t          rx_buf[4];
    I2C_Handle       i2c;
    I2C_Params       i2cParams;
    I2C_Transaction i2cTransaction;
    uint16_t red_led, ir_led, size, i;

    I2C_init();

    /* Create I2C for usage */
    I2C_Params_init(&i2cParams);
    i2cParams.bitRate = I2C_400kHz;
    i2c = I2C_open(Board_I2C0, &i2cParams);
    if (i2c == NULL) {
        while (1);
    }

    i2cTransaction.slaveAddress = slave_address;
    i2cTransaction.writeBuf = tx_buf;
    i2cTransaction.writeCount = 0;
    i2cTransaction.readBuf = rx_buf;
    i2cTransaction.readCount = 4;

```

```
49      size = 10*100;
50      buffer_ir = (float_t *) malloc(size*sizeof(float_t));
51      buffer_red = (float_t *) malloc(size*sizeof(float_t));
52
53      for (i = 0; i < size; i++)
54      {
55          if (I2C_transfer(i2c, &i2cTransaction))
56          {
57              ir_led = (rx_buf[0] << 8) | rx_buf[1];
58              buffer_ir[i] = ir_led;
59              red_led = (rx_buf[2] << 8) | rx_buf[3];
60              buffer_red[i] = red_led;
61          }
62          else
63          {
64              buffer_ir[i] = 0;
65              buffer_red[i] = 0;
66          }
67          /* Sleep for 9950 usecond, because MAX30100 have fs = 100 Hz */
68          usleep(9950);
69      }
70
71      /* Deinitialized I2C */
72      I2C_close(i2c);
73
74      return size;
75  }
76
77  uint16_t get_i2c_value_bpm(float_t *buffer, uint16_t slave_address, uint8_t bytes)
78  {
79      uint8_t         tx_buf[1];
80      uint8_t         rx_buf[4];
81      I2C_Handle      i2c;
82      I2C_Params      i2cParams;
83      I2C_Transaction i2cTransaction;
84      uint16_t ir_led, size, i;
85
86      I2C_init();
87
88
89      /* Create I2C for usage */
90      I2C_Params_init(&i2cParams);
91      i2cParams.bitRate = I2C_400kHz;
92      i2c = I2C_open(Board_I2C_TMP, &i2cParams);
93      if (i2c == NULL) {
94          while (1);
95      }
96
97      tx_buf[0] = 0x05;
98      i2cTransaction.slaveAddress = slave_address;
99      i2cTransaction.writeBuf = tx_buf;
100     i2cTransaction.writeCount = 1;
```

```c
101     i2cTransaction.readBuf = NULL;
102     i2cTransaction.readCount = 0;
103
104     i2cTransaction.slaveAddress = slave_address;
105     i2cTransaction.writeBuf = tx_buf;
106     i2cTransaction.writeCount = 0;
107     i2cTransaction.readBuf = rx_buf;
108     i2cTransaction.readCount = 4;
109
110     size = 10*100;
111     buffer = (float_t *) malloc(size*sizeof(float_t));
112
113
114     for (i = 0; i < size; i++)
115     {
116         if (I2C_transfer(i2c, &i2cTransaction))
117         {
118             ir_led = (rx_buf[0] << 8) | rx_buf[1];
119             buffer[i] = ir_led;
120         }
121         else
122         {
123             buffer[i] = 0;
124         }
125         /* Sleep for 10000 usecond */
126         usleep(9950);
127     }
128
129     /* Deinitialized I2C */
130     I2C_close(i2c);
131
132     return size;
133 }
```

```c
1  /*
2   * analog.c
3   *
4   *      Author: Marcos Assis
5   */
6  #include <stdint.h>
7  #include <stddef.h>
8
9  #include <ti/drivers/ADC.h>
10 #include <ti/drivers/PIN.h>
11
12 #include "Board.h"
13
14 uint32_t get_ADC_analog_temperature(uint8_t channel);
15
16 uint32_t get_ADC_analog_temperature(uint8_t channel)
```

```
17  {
18      ADC_Handle adc;
19      ADC_Params params;
20      uint16_t adc_value;
21      uint32_t adc_microvolt;
22      int_fast16_t res;
23
24      ADC_init();
25
26      ADC_Params_init(&params);
27      if(channel == 0)
28      {
29          adc = ADC_open(Board_ADC0, &params);
30      }
31      else
32      {
33          adc = ADC_open(Board_ADC1, &params);
34      }
35
36
37      if (adc == NULL)
38      {
39          // ADC_open() failed
40          ADC_close(adc);
41          return NULL;
42      }
43
44      res = ADC_convert(adc, &adc_value);
45      if(res == ADC_STATUS_SUCCESS)
46      {
47          adc_microvolt = ADC_convertRawToMicroVolts(adc, adc_value);
48          ADC_close(adc);
49          return adc_microvolt;
50      }
51
52      // ADC_convert() failed
53      ADC_close(adc);
54      return NULL;
55  }
```

```
1   /*
2    * medical_profile.h
3    *
4    * Author: Marcos Assis
5    */
6
7   #ifndef MEDICALPROFILE_H
8   #define MEDICALPROFILE_H
9
10  #ifdef __cplusplus
```

```
11 extern "C"
12 {
13 #endif
14
15
16 // Profile Parameters
17 #define MEDICALPROFILE_ID                 0  // R uint8 - Patient Identificator
18 #define MEDICALPROFILE_TEMPERATURE        1  // R uint16 - Body Internal
       Temperature
19 #define MEDICALPROFILE_HR                 2  // R uint8 - Heart Rate
20 #define MEDICALPROFILE_OXYGEN             3  // R uint8 - Oxygen Level
21 #define MEDICALPROFILE_PRESSURE           4  // R uint16 - Pressure
22 #define MEDICALPROFILE_STEPSCOUNT         5  // R uint16 - Steps Count
23 #define MEDICALPROFILE_ALERTS             6  // R uint8 - Alerts
24
25
26 // Medical Profile Service UUID
27 #define MEDICALPROFILE_SERV_UUID          0xFFF0
28
29 // Key Pressed UUID
30 #define MEDICALPROFILE_ID_UUID            0xFFF0
31 #define MEDICALPROFILE_TEMPERATURE_UUID   0xFFF1
32 #define MEDICALPROFILE_HR_UUID            0xFFF2
33 #define MEDICALPROFILE_OXYGEN_UUID        0xFFF3
34 #define MEDICALPROFILE_PRESSURE_UUID      0xFFF4
35 #define MEDICALPROFILE_STEPSCOUNT_UUID    0xFFF5
36 #define MEDICALPROFILE_ALERTS_UUID        0xFFF6
37
38 // Simple Keys Profile Services bit fields
39 #define MEDICALPROFILE_SERVICE            0x00000001
40
41 #define MEDICALPROFILE_TEMPERATURELEN     2
42 #define MEDICALPROFILE_PRESSURELEN        2
43 #define MEDICALPROFILE_STEPSCOUNTLEN      2
44
45 // Callback when a characteristic value has changed
46 typedef void (*medicalProfileChange_t)( uint8 paramID );
47
48 typedef struct
49 {
50   medicalProfileChange_t        pfnMedicalProfileChange;  // Called when
       characteristic value changes
51 } medicalProfileCBs_t;
52
53 extern bStatus_t MedicalProfile_AddService( uint32 services );
54
55 extern bStatus_t MedicalProfile_RegisterAppCBs( medicalProfileCBs_t *appCallbacks
     );
56
57 extern bStatus_t MedicalProfile_SetParameter(uint8 param, uint8 len, void *value);
58
59 extern bStatus_t MedicalProfile_GetParameter(uint8 param, void *value);
```

```
60
61
62 /*********************************************************************
63 *********************************************************************/
64
65 #ifdef __cplusplus
66 }
67 #endif
68
69 #endif
```

```
1  /*
2   * medical_profile.h
3   *
4   * Author: Marcos Assis
5   */
6
7  #include <string.h>
8  #include <icall.h>
9  #include "util.h"
10
11 #include "icall_ble_api.h"
12
13 #include "medical_profile.h"
14
15 #define SERVAPP_NUM_ATTR_SUPPORTED        25
16
17 CONST uint8 medicalProfileServUUID[ATT_BT_UUID_SIZE] =
18 {
19   LO_UINT16(MEDICALPROFILE_SERV_UUID), HI_UINT16(MEDICALPROFILE_SERV_UUID)
20 };
21
22 // Characteristic 1 UUID: 0xFFF1
23 CONST uint8 medicalProfilechar1UUID[ATT_BT_UUID_SIZE] =
24 {
25   LO_UINT16(MEDICALPROFILE_ID_UUID), HI_UINT16(MEDICALPROFILE_ID_UUID)
26 };
27
28 // Characteristic 2 UUID: 0xFFF2
29 CONST uint8 medicalProfilechar2UUID[ATT_BT_UUID_SIZE] =
30 {
31   LO_UINT16(MEDICALPROFILE_TEMPERATURE_UUID), HI_UINT16(
       MEDICALPROFILE_TEMPERATURE_UUID)
32 };
33
34 // Characteristic 3 UUID: 0xFFF3
35 CONST uint8 medicalProfilechar3UUID[ATT_BT_UUID_SIZE] =
36 {
37  LO_UINT16(MEDICALPROFILE_HR_UUID), HI_UINT16(MEDICALPROFILE_HR_UUID)
38 };
```

```
39
40 // Characteristic 4 UUID: 0xFFF4
41 CONST uint8 medicalProfilechar4UUID[ATT_BT_UUID_SIZE] =
42 {
43   LO_UINT16(MEDICALPROFILE_OXYGEN_UUID), HI_UINT16(MEDICALPROFILE_OXYGEN_UUID)
44 };
45
46 // Characteristic 5 UUID: 0xFFF5
47 CONST uint8 medicalProfilechar5UUID[ATT_BT_UUID_SIZE] =
48 {
49  LO_UINT16(MEDICALPROFILE_PRESSURE_UUID), HI_UINT16(MEDICALPROFILE_PRESSURE_UUID)
50 };
51
52 // Characteristic 5 UUID: 0xFFF5
53 CONST uint8 medicalProfilechar6UUID[ATT_BT_UUID_SIZE] =
54 {
55  LO_UINT16(MEDICALPROFILE_STEPSCOUNT_UUID), HI_UINT16(
       MEDICALPROFILE_STEPSCOUNT_UUID)
56 };
57
58 // Characteristic 6 UUID: 0xFFF6
59 CONST uint8 medicalProfilechar7UUID[ATT_BT_UUID_SIZE] =
60 {
61  LO_UINT16(MEDICALPROFILE_ALERTS_UUID), HI_UINT16(MEDICALPROFILE_ALERTS_UUID)
62 };
63
64
65 extern void* memcpy(void *dest, const void *src, size_t len);
66
67 static medicalProfileCBs_t *medicalProfile_AppCBs = NULL;
68
69
70 // Simple Profile Service attribute
71 static CONST gattAttrType_t medicalProfileService = { ATT_BT_UUID_SIZE,
     medicalProfileServUUID };
72
73
74 // Simple Profile Characteristic 1 Properties
75 static uint8 medicalProfileIDProps = GATT_PROP_READ;
76
77 // Characteristic 1 Value
78 static uint8 medicalProfileID = 0;
79
80 // Simple Profile Characteristic 1 User Description
81 static uint8 medicalProfileIDUserDesp[17] = "ID";
82
83
84 // Simple Profile Characteristic 2 Properties
85 static uint8 medicalProfileTEMPERATUREProps = GATT_PROP_READ;
86
87 // Characteristic 2 Value
88 static uint8 medicalProfileTEMPERATURE[MEDICALPROFILE_TEMPERATURELEN] = {0, 0};
```

```
89
90 // Simple Profile Characteristic 2 User Description
91 static uint8 medicalProfileTEMPERATUREUserDesp[17] = "Temperature";
92
93 // Simple Profile Characteristic 3 Properties
94 static uint8 medicalProfileHRProps = GATT_PROP_READ;
95
96 // Characteristic 3 Value
97 static uint8 medicalProfileHR = 0;
98
99 // Simple Profile Characteristic 3 User Description
100 static uint8 medicalProfileHRUserDesp[17] = "Heart Rate";
101
102 // Simple Profile Characteristic 4 Properties
103 static uint8 medicalProfileOXYGENProps = GATT_PROP_READ;
104
105 // Characteristic 4 Value
106 static uint8 medicalProfileOXYGEN = 0;
107
108 // Simple Profile Characteristic 4 User Description
109 static uint8 medicalProfileOXYGENUserDesp[17] = "Oxygen Level";
110
111 // Simple Profile Characteristic 5 Properties
112 static uint8 medicalProfilePRESSUREProps = GATT_PROP_READ;
113
114 // Characteristic 5 Value
115 static uint8 medicalProfilePRESSURE[MEDICALPROFILE_PRESSURELEN] = {0, 0};
116
117 // Simple Profile Characteristic 5 User Description
118 static uint8 medicalProfilePRESSUREUserDesp[17] = "Pressure";
119
120 // Simple Profile Characteristic 5 Properties
121 static uint8 medicalProfileSTEPSCOUNTProps = GATT_PROP_READ;
122
123 // Characteristic 5 Value
124 static uint8 medicalProfileSTEPSCOUNT[MEDICALPROFILE_STEPSCOUNTLEN] = {0, 0};
125
126 // Simple Profile Characteristic 5 User Description
127 static uint8 medicalProfileSTEPSCOUNTUserDesp[17] = "Steps Count";
128
129 // Simple Profile Characteristic 6 Properties
130 static uint8 medicalProfileALERTSProps = GATT_PROP_READ;
131
132 // Characteristic 6 Value
133 static uint8 medicalProfileALERTS = 0;
134
135 // Simple Profile Characteristic 6 User Description
136 static uint8 medicalProfileALERTSUserDesp[17] = "Alerts";
137
138 // Attributes table
139 static gattAttribute_t medicalProfileAttrTbl[SERVAPP_NUM_ATTR_SUPPORTED] =
140 {
```

```
141   // Simple Profile Service
142   {
143     { ATT_BT_UUID_SIZE, primaryServiceUUID }, /* type */
144     GATT_PERMIT_READ,                         /* permissions */
145     0,                                        /* handle */
146     (uint8 *)&medicalProfileService            /* pValue */
147   },
148
149     // Characteristic 1 Declaration
150     {
151       { ATT_BT_UUID_SIZE, characterUUID },
152       GATT_PERMIT_READ,
153       0,
154       &medicalProfileIDProps
155     },
156
157       // Characteristic Value 1
158       {
159         { ATT_BT_UUID_SIZE, medicalProfilechar1UUID },
160         GATT_PERMIT_READ,
161         0,
162         &medicalProfileID
163       },
164
165       // Characteristic 1 User Description
166       {
167         { ATT_BT_UUID_SIZE, charUserDescUUID },
168         GATT_PERMIT_READ,
169         0,
170         medicalProfileIDUserDesp
171       },
172
173     // Characteristic 2 Declaration
174     {
175       { ATT_BT_UUID_SIZE, characterUUID },
176       GATT_PERMIT_READ,
177       0,
178       &medicalProfileTEMPERATUREProps
179     },
180
181       // Characteristic Value 2
182       {
183         { ATT_BT_UUID_SIZE, medicalProfilechar2UUID },
184         GATT_PERMIT_READ,
185         0,
186         medicalProfileTEMPERATURE
187       },
188
189       // Characteristic 2 User Description
190       {
191         { ATT_BT_UUID_SIZE, charUserDescUUID },
192         GATT_PERMIT_READ,
```

```
193         0,
194         medicalProfileTEMPERATUREUserDesp
195       },
196       // Characteristic 2 Declaration
197           {
198             { ATT_BT_UUID_SIZE, characterUUID },
199             GATT_PERMIT_READ,
200             0,
201             &medicalProfileHRProps
202           },
203
204             // Characteristic Value 2
205             {
206               { ATT_BT_UUID_SIZE, medicalProfilechar3UUID },
207               GATT_PERMIT_READ,
208               0,
209               &medicalProfileHR
210             },
211
212             // Characteristic 2 User Description
213             {
214               { ATT_BT_UUID_SIZE, charUserDescUUID },
215               GATT_PERMIT_READ,
216               0,
217               medicalProfileHRUserDesp
218             },
219
220     // Characteristic 3 Declaration
221     {
222       { ATT_BT_UUID_SIZE, characterUUID },
223       GATT_PERMIT_READ,
224       0,
225       &medicalProfileOXYGENProps
226     },
227
228       // Characteristic Value 3
229       {
230         { ATT_BT_UUID_SIZE, medicalProfilechar4UUID },
231         GATT_PERMIT_READ,
232         0,
233         &medicalProfileOXYGEN
234       },
235
236       // Characteristic 3 User Description
237       {
238         { ATT_BT_UUID_SIZE, charUserDescUUID },
239         GATT_PERMIT_READ,
240         0,
241         medicalProfileOXYGENUserDesp
242       },
243
244     // Characteristic 4 Declaration
```

```
245     {
246       { ATT_BT_UUID_SIZE, characterUUID },
247       GATT_PERMIT_READ,
248       0,
249       &medicalProfilePRESSUREProps
250     },
251
252       // Characteristic Value 4
253       {
254         { ATT_BT_UUID_SIZE, medicalProfilechar5UUID },
255         GATT_PERMIT_READ,
256         0,
257         medicalProfilePRESSURE
258       },
259       // Characteristic 4 User Description
260       {
261         { ATT_BT_UUID_SIZE, charUserDescUUID },
262         GATT_PERMIT_READ,
263         0,
264         medicalProfilePRESSUREUserDesp
265       },
266       // Characteristic 4 Declaration
267           {
268             { ATT_BT_UUID_SIZE, characterUUID },
269             GATT_PERMIT_READ,
270             0,
271             &medicalProfileSTEPSCOUNTProps
272           },
273
274             // Characteristic Value 4
275             {
276               { ATT_BT_UUID_SIZE, medicalProfilechar6UUID },
277               GATT_PERMIT_READ,
278               0,
279               medicalProfileSTEPSCOUNT
280             },
281             // Characteristic 4 User Description
282             {
283               { ATT_BT_UUID_SIZE, charUserDescUUID },
284               GATT_PERMIT_READ,
285               0,
286               medicalProfileSTEPSCOUNTUserDesp
287             },
288       // Characteristic 4 Declaration
289           {
290             { ATT_BT_UUID_SIZE, characterUUID },
291             GATT_PERMIT_READ,
292             0,
293             &medicalProfileALERTSProps
294           },
295
296             // Characteristic Value 4
```

```
297             {
298               { ATT_BT_UUID_SIZE, medicalProfilechar5UUID },
299               GATT_PERMIT_READ,
300               0,
301               &medicalProfileALERTS
302             },
303             // Characteristic 4 User Description
304             {
305               { ATT_BT_UUID_SIZE, charUserDescUUID },
306               GATT_PERMIT_READ,
307               0,
308               medicalProfileALERTSUserDesp
309             },
310 };
311
312 static bStatus_t medicalProfile_ReadAttrCB(uint8_t connHandle,
313                                           gattAttribute_t *pAttr,
314                                           uint8_t *pValue, uint8_t *pLen,
315                                           uint8_t offset, uint8_t maxLen,
316                                           uint8_t method);
317
318 // Profile Callbacks
319 CONST gattServiceCBs_t medicalProfileCBs =
320 {
321   medicalProfile_ReadAttrCB,  // Read callback function pointer
322   medicalProfile_WriteAttrCB, // Write callback function pointer
323   NULL                       // Authorization callback function pointer
324 };
325
326 bStatus_t MedicalProfile_AddService( uint32 services)
327 {
328   uint8 status;
329
330   if ( services & MEDICALPROFILE_SERVICE )
331   {
332     // Register GATT attribute list and CBs with GATT Server App
333     status = GATTServApp_RegisterService( medicalProfileAttrTbl,
334                                           GATT_NUM_ATTRS( medicalProfileAttrTbl ),
335                                           GATT_MAX_ENCRYPT_KEY_SIZE,
336                                           &medicalProfileCBs );
337   }
338   else
339   {
340     status = SUCCESS;
341   }
342
343   return ( status );
344 }
345
346 bStatus_t MedicalProfile_RegisterAppCBs( medicalProfileCBs_t *appCallbacks )
347 {
348   if ( appCallbacks )
```

```
349    {
350      medicalProfile_AppCBs = appCallbacks;
351
352      return ( SUCCESS );
353    }
354    else
355    {
356      return ( bleAlreadyInRequestedMode );
357    }
358  }
359
360  bStatus_t MedicalProfile_SetParameter( uint8 param, uint8 len, void *value)
361  {
362    bStatus_t ret = SUCCESS;
363    switch ( param )
364    {
365      case MEDICALPROFILE_ID:
366        if ( len == sizeof(uint8) )
367        {
368            medicalProfileID = *((uint8*)value);
369        }
370        else
371        {
372          ret = bleInvalidRange;
373        }
374        break;
375
376      case MEDICALPROFILE_TEMPERATURE:
377        if (len == MEDICALPROFILE_TEMPERATURELEN)
378        {
379            memcpy(medicalProfileTEMPERATURE, value, len);
380        }
381        else
382        {
383          ret = bleInvalidRange;
384        }
385        break;
386
387      case MEDICALPROFILE_PRESSURE:
388        if (len == MEDICALPROFILE_PRESSURELEN)
389        {
390            memcpy(medicalProfilePRESSURE, value, len);
391        }
392        else
393        {
394          ret = bleInvalidRange;
395        }
396        break;
397
398      case MEDICALPROFILE_STEPSCOUNT:
399            if (len == MEDICALPROFILE_STEPSCOUNTLEN)
400            {
```