UNIVERSIDADE DE SÃO PAULO

ESCOLA DE ENGENHARIA DE SÃO CARLOS

JOÃO PEDRO GOBBI CODOGNOTTO

Projeto e Prototipação de Bracelete Inteligente para Auxílio e Monitoramento de Idosos

São Carlos

2017

JOÃO PEDRO GOBBI CODOGNOTTO

Projeto e Prototipação de Bracelete Inteligente para Auxílio e Monitoramento de Idosos

Monografia apresentada ao Curso de Engenharia de Computação, da Escola de Engenharia de São Carlos da Universidade de São Paulo, como parte dos requisitos para obtenção do título de Engenheiro de Computação.

Orientador: Prof. Dr .Evandro L. L. Rodrigues

São Carlos

2017

Codognotto, João Pedro Gobbi
C669p Projeto e prototipação de bracelete inteligente para auxílio e monitoramento de idosos / João Pedro Gobbi Codognotto; orientador Evandro Luiz Linhares Rodrigues. São Carlos, 2017.

Monografia (Graduação em Engenharia de Computação) -- Escola de Engenharia de São Carlos e Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo, 2017.

1. Monitoramento de Idosos. 2. Wearable. 3. Bluetooth Low Energy (BLE). 4. Android. 5. LightBlue Bean+. I. Título.

FOLHA DE APROVAÇÃO

Nome: João Pedro Gobbi Codognotto

Título: "Projeto e prototipação de bracelete inteligente para auxílio e monitoramento de idosos"

Trabalho de Conclusão de Curso defendido em 29/11/17.

Comissão Julgadora:	**Resultado:**
Prof. Associado Evandro Luís Linhari Rodrigues (Orientador) - SEL/EESC/USP	APROVADO.
Mestre Alex Antonio Affonso Doutorando - SEL/EESC/USP	APROVADO.
Mestre André Luis Martins Doutorando - SEL/EESC/USP	Aprovado.

Coordenador do Curso Interunidades Engenharia de Computação:

Prof. Dr. Maximiliam Luppe

DEDICATÓRIA

Dedico esse trabalho às minhas avós, Dirce e Marisa.

AGRADECIMENTOS

Agradeço a meu Pai, Pedro, e a minha mãe, Marisa, por todo suporte estrutural dado a mim nos meus últimos 22 anos de vida.

Agradeço ao meu irmão, Pedro Henrique, por sempre querer o melhor para mim.

Agradeço meus amigos, por terem feito meu período na universidade mais completo.

E finalmente, agradeço vigorosamente à minha namorada, amiga e parceira, Roberta Costa, por ser uma parte essencial em todas as minhas conquistas recentes.

Sem vocês eu com certeza não teria chegado tão longe, tão rápido.

“It's the questions we can't answer that teach us the most. They teach us how to think. If you give a man an answer, all he gains is a little fact. But give him a question and he'll look for his own answers.”

-Patrick Rothfuss

RESUMO

CODOGNOTTO, J. G. **Projeto e Prototipação de Bracelete Inteligente para Auxílio e Monitoramento de Idosos**. 2017. Monografia (Trabalho de Conclusão de Curso) – Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2017.

Acidentes são uma das maiores causas de morte em idosos e este trabalho foi motivado pela utilização de tecnologias vestíveis na mitigação deste problema. Neste documento é detalhado o desenvolvimento e implementação do projeto de um protótipo de um dispositivo vestível, também conhecido pela expressão da língua inglesa: *wearable*. Este dispositivo é voltado ao público idoso e possibilita uma comunicação quase instantânea com um contato de emergência pré-definido. Outras funcionalidades, como a detecção de queda e a medição de frequência de batimentos cardíacos são implementadas. O projeto utiliza *Bluetooth Low Energy* para a comunicação do dispositivo com um aplicativo para smartphone Android. O desenvolvimento do protótipo foi feito numa placa *LightBlue Bean+,* que conta com um microcontrolador *ATmega* 328p além de um *SoC Bluetooth* responsável por realizar a comunicação. O envio do pedido de emergência é feito por meio de envio de uma mensagem SMS a partir do smartphone previamente conectado no dispositivo. O projeto foi bem-sucedido pois implementou as funcionalidades requeridas, de forma eficiente, que possibilitam a evolução para um produto real vestível para o monitoramento de segurança de idosos.

Palavras-chave: *Wearable*, *LightBlue Bean+*, *Bluetooth Low Energy* (*BLE*), *Android*, Monitoramento de Idosos

ABSTRACT

Accidents are one of the biggest causes of death with the elderly and this work was motivated by the use of wearable technology on the mitigation of this problem. On this document, the development and implementation of a wearable device prototype is detailed. This wearable is meant especially for the elderly public and enables an almost instantaneous communication with a pre-defined emergency contact. Other functionalities, like fall detection and heartbeat rate measuring are also implemented. The project uses Bluetooth Low Energy for the communication between the device and an Android application. The development of the prototype was made on a LightBlue Bean+ development board, that has a microcontroller ATmega 328p, besides an SoC Bluetooth responsible to accomplish communication. The emergency request sending is done through an SMS sent from the smartphone that was previously connected to the device. The project was successful, once it efficiently implemented the required functionalities, which enables its evolution to a final product of a wearable for elderly safety monitoring.

Keywords: *Wearable*, *LightBlue Bean+*, *Bluetooth Low Energy* (*BLE*), *Android, Monitoring System for Seniors.*

SUMÁRIO

1 INTRODUÇÃO

1.1 OBJETIVO

O objetivo deste trabalho é desenvolver um protótipo de um dispositivo *wearable* de auxílio a idosos e um aplicativo *Android* que se comunique com esse dispositivo. O sistema deve ser capaz de notificar um contato pré-definido quando o dispositivo entra em estado de emergência. O dispositivo deve poder entrar em modo de emergência manualmente ou automaticamente, quando é detectada uma queda. Outra funcionalidade que deve ser implementada é a medição de batimentos cardíacos. O dispositivo criado deve ser eficiente energeticamente, possibilitar a mobilidade, além de ser de fácil uso, para aderência do uso pelo seu público alvo: idosos.

O projeto deve ser desenvolvido utilizando materiais que possibilitem uma evolução do protótipo para um futuro projeto de produção em massa. Além disso, há foco no uso de tecnologias que já se encontram presentes no mercado, e na utilização de uma rede de telecomunicação já existente cuja cobertura seja alta.

1.2 MOTIVAÇÃO

Um fato imutável é que o ser humano envelhece e novas limitações físicas e mentais surgem. Tais limitações causam preocupação em famílias e amigos que, contra a vontade dos mais velhos, forçam uma rotina monótona com a necessidade de cuidadores, visitas constantes e asilos. Essas atitudes não são exageradas, uma vez que uma das maiores causas de mortes em idosos é relacionada a acidentes (BROOKS, 2015), que poderiam ter suas consequências mitigadas com um socorro imediato. Uma comunicação amigável, fácil e rápida, entre o idoso e pessoas de confiança, tem o potencial de beneficiar e salvar a vida de muitos idosos.

1.3 ORGANIZAÇÃO DO TRABALHO

Este documento está dividido nas seguintes seções:

1. Introdução, em que o leitor é contextualizado do propósito geral do projeto, e uma visão geral do escopo desde documento é apresentada, assim como uma explanação do estado da Arte para dispositivos similares no momento da composição deste documento.

2. Embasamento teórico, em que alguns conceitos necessários para o entendimento do funcionamento do projeto, bem como sua implementação são explicados e detalhados.

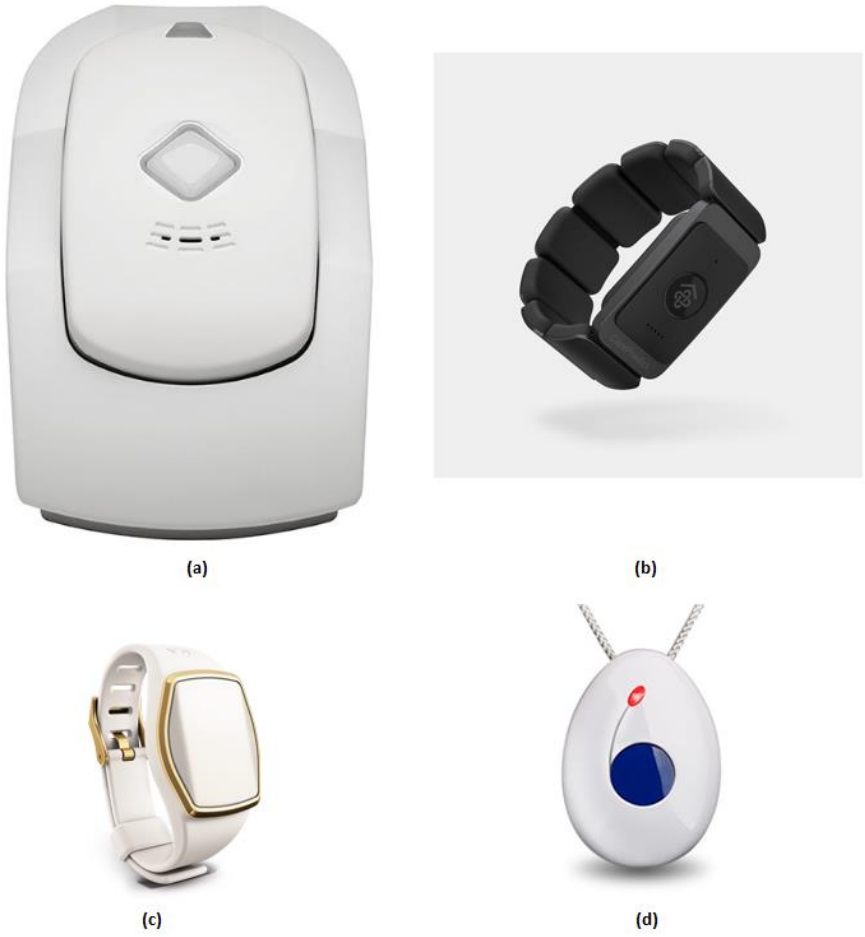
3. Materiais, em que todos os materiais e tecnologias utilizados são relacionados, com uma descrição detalhada dos seus funcionamentos e especificações técnicas.
4. Métodos, em que o desenvolvimento do projeto é finalmente detalhado, com explicações de alguns trechos de códigos, decisões de projeto, e da arquitetura final geral do sistema.
5. Resultados. Nessa seção, o funcionamento do protótipo é demonstrado, com casos de teste que cobrem as funcionalidades implementadas.
6. Conclusão. Na última seção, são sumarizados alguns pontos relevantes observados durante o desenvolvimento deste protótipo e são feitas algumas sugestões de prováveis trabalhos futuros.

1.4 ESTADO DA ARTE

O crescimento do uso de tecnologias vestíveis e da internet das coisas influencia diretamente o crescimento da quantidade de dispositivos para monitoramento de idosos. As soluções atuais contam com o uso de telefonia para a comunicação com uma central de atendimento que, ao receber um pedido de emergência ativado por um simples e grande botão de emergência, entra em contato com o idoso ou familiares. Esses serviços contam com assinaturas mensais, como o serviço americano EverThere, da empresa americana AT&T (AT&T, 2016) e o serviço disponível no Brasil HelpCare (HELPCARE, 2017). O Everthere é

independente, enquanto o HelpCare funciona apenas em residências, por depender de uma base para comunicação por linha de telefone fixo. Também existe no Mercado comunicadores dependentes de *smartphones* para a comunicação e funcionamento como, por exemplo, o GreatCall (GREATCALL, 2017).

Figura 1: (a)Everthere (b)CarePredict (c)GreatCall (d)HelpCare



FONTES: (a) (AT&T, 2016) (b) (CAREPREDICT, 2017) (c) (GREATCALL, 2017) (d) (HELPCARE, 2017)

Não foram encontrados projetos e produtos relevantes que implementavam uma solução livre de assinatura mensal. Além disso, o reconhecimento automático de emergência é limitado nos aparelhos já citados. O mais evoluído neste aspecto é o *CarePredict Tempo* (CAREPREDICT, 2017), que busca padrões e os processa com inteligência artificial.

2 EMBASAMENTO TEÓRICO

2.1 TECNOLOGIAS VESTÍVEIS – *WEARABLE TECHNOLOGY*

Uma tecnologia em expansão e que recebe grandes investimentos, esses dispositivos têm previsão de atingir um mercado de 25 bilhões de dólares, na próxima decada (LAMKIN, 2015). Consiste em uma das maiores vertentes da Internet das Coisas, conceito que conecta elementos comuns do cotidiano, como eletrodomésticos e a computadores à Rede, o que resulta em uma comunicação e um monitoramento constantes.

Os *Wearables,* como são popularmente chamados no Brasil e no mundo, consistem em aparelhos inteligentes, que podem substituir tarefas de computadores e *Smartphone*s, entretanto eles tendem a ser integradas com o usuário, podendo por meio de sensoriamento avançado receber e enviar informações em tempo real (TEHRANI e MICHAEL, 2014).

Segundo Tehrani e Michael (2014), o propósito de um *wearable* é criar uma conexão constante, conveniente, transparente, portátil e quase independente dos usuários com seus diversos eletrônicos. Sendo assim, o dispositivo deve focar no sensoriamento, na comunicação com outros dispositivos, e na sua praticidade para o usuário.

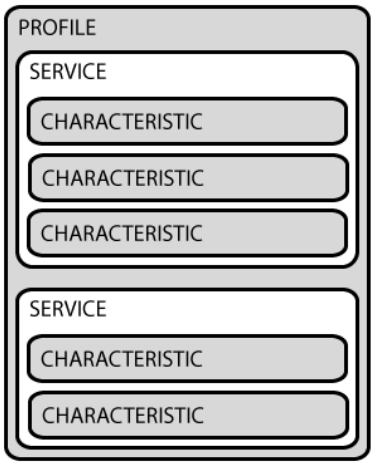
2.2 *BLUETOOTH LOW ENERGY – BLE*

Criado para unificar a comunicação de curta distância entre dispositivos (BLUETOOTH SIG, 2017), o *Bluetooth* consiste em um conjunto de protocolos aplicados em diversas camadas de comunicação (BRAY e STURMAN, 2002) e é muito utilizado para a criação de uma rede PAN (Personal Area Network).

Já sendo quase omnipresente em dispositivos móveis, a tecnologia continua em evolução conforme a necessidade de adaptação e em 2011 foi lançada uma nova vertente da tecnologia: o *Bluetooth Low Energy*, também chamado de *BLE* ou *Bluetooth Smart*. Com o objetivo de ser utilizado amplamente por dispositivos no contexto da Internet das Coisas, o *BLE* facilita sua implementação por constituir Perfil de Acesso Genérico (*GAP*) e Perfil de Atributo Genérico (*GATT)*.

O *Generic Access Profile* (*GAP)* é o perfil no qual o dispositivo *Bluetooth* se baseia para estabelecer e gerenciar conexões e varia, principalmente, conforme a topologia da rede do sistema. O *Generic Attribute Profile* (*GATT)* define o modo pelo qual os dispositivos *BLE* se comunicam. O *GATT* utiliza Serviços (*services*) e Características (Characteristics) para a comunicação, podendo esses ser configurações predefinidas e padronizadas ou configurações personalizadas criadas pelo desenvolvedor do dispositivo. Como pode ser observado na Figura 2, um *serviço* possui diversas *características*, que correspondem aos dados que devem ser sincronizados entre os dispositivos para uma certa aplicação. Um exemplo simples de *GATT* é um *serviço UART*, em que uma das características é o canal RX, que pode ser escrito apenas por outro dispositivo e outra é o canal TX, que deve ser configurado para apenas o dispositivo local ser capaz de escrever (ADAFRUIT, 2014).

Figura 2: Organização do GATT contendo Serviços e, subsequentemente, Características.



FONTE: (ADAFRUIT, 2014)

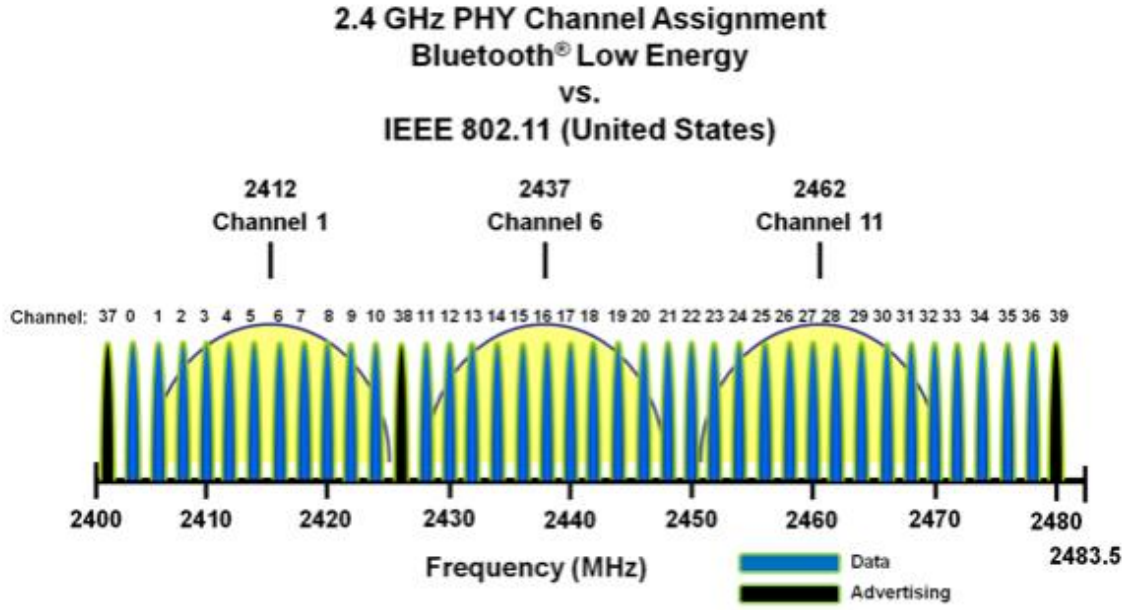
O diferencial mais relevante do *Bluetooth Low Energy* comparado ao *Bluetooth Classic* é que no primeiro a economia de energia está inerente em todas as camadas de comunicação. O *BLE* possui dois tipos de transmissão. A primeira é a de Anúncio (*Advertising*) que consiste na troca de informações para a Descoberta de Dispositivos, Estabelecimento de Conexões e Transmissões *Broadcast*, enquanto a segunda é a de Dados, que cria uma conexão Duplex com uma boa taxa de transmissão.

Dos 40 canais de 2MHz disponíveis na camada física, 3 são utilizados para a transmissão do tipo Anúncio (*Advertising*) e 37 são utilizados para a transmissão de dados (WARNE, 2017). O padrão das transmissões em ambos é o mesmo, entretanto os 37 canais de dados ficam desligados quando inativos (geralmente a maior parte do tempo), aumentado

significativamente a economia de energia. Portanto, os módulos *BLE* dos dispositivos têm a capacidade de permanecer ociosos ao mesmo tempo em que conseguem ter uma conexão iniciada ou receber dados *Broadcast*.

Utilizando seus diversos canais, a tecnologia também diminui o consumo energético ao necessitar de menos potência para apaziguar efeitos de ruído, que são significativos, uma vez que toda a banda utilizada pelo *Bluetooth* é compartilhada com outras tecnologias, como o WiFi. O *BLE* implementa a *Adaptative Frequency Hopping*, ou Salto Adaptativo de Frequência, em que ao detectar muita interferência em um canal de dado, simplesmente 'salta' para o próximo até encontrar um canal com nível de ruído aceitável. O mesmo não ocorre com os canais de *Advertising*, portanto eles são especialmente selecionados para serem os que menos sofrem de interferência, uma vez que ficam posicionados nas bordas dos canais do WiFi (MICROCHIP TECHNOLOGY, INC., 2017). Este efeito pode ser observado na Figura 3, que compara os canais utilizados por ambas as tecnologias no espectro de frequência.

Figura 3: Espectro de Frequência comparando bandas do BLE com WiFi.



FONTE: (MICROCHIP TECHNOLOGY, INC., 2017)

2.3 SISTEMAS EMBARCADOS: MCUS E SOCS

Projetos de Sistemas Embarcados possuem os requisitos de serem eficientes e confiáveis (MARWEDEL, 2006), conceitos os quais são relativos, pois podem referenciar diferentes tipos de requisitos do sistema. Um sistema confiável que é protegido contra alterações de dados mal-intencionadas é diferente de um sistema confiável protegido contra erros em temperaturas baixas; de modo similar que um sistema eficiente energeticamente não obrigatoriamente será eficiente em seu custo/benefício. Para alcançar os requisitos do sistema que está sendo projetado, os desenvolvedores devem escolher com cuidado as partes que o compõem, sem esquecer ainda de levar em consideração como elas influenciam no tempo de desenvolvimento do produto final. Um sistema embarcado compreende de um conjunto de *Hardware*, a parte física do sistema, e de *Software*, a parte lógica programável que será executada no *Hardware*. O projetista deve, então, ponderar quais partes, tanto de *Hardware* quanto *Software*, serão desenvolvidas internamente e quais serão adquiridas e então integradas ao sistema, considerando que a configuração e integração dessas partes também correspondem ao desenvolvimento do projeto. Quando partes já estão totalmente integradas e funcionais e já foram testadas, o projetista deve abstrair esse conjunto gerado, enxergando-o como uma única parte (ou bloco).

O essencial de qualquer aplicação de um sistema embarcado costuma ser a aquisição de dados e o subsequente processamento destes, que serão feitos por interfaces de Entrada e Saída (*Input/Output* ou *I/O)* e uma ou mais unidades de processamento, respectivamente. As entradas e saídas se conectam a periféricos que são blocos abstraídos, os quais, por meio de sua própria lógica e implementação, interagem com outros sistemas computacionais ou realizam medições de grandezas físicas, possibilitando sensoriamentos e interações com humanos. Com o objetivo de abstrair o essencial para o projetista, foram criados os Microcontroladores (MicroController Unit ou MCU).

Os Microcontroladores encapsulam (ou seja, pré-conectam e abstraem os blocos responsáveis por tarefas) um microprocessador juntamente com blocos de Entrada e Saída e outros blocos essenciais para a realização de processamentos, como cristais osciladores, contadores, temporizadores, memórias (VAHID e GIVAGIS, 2001). Alguns blocos mais complexos e não essenciais, que atualmente são utilizados comumente, também podem ser

integrados, como por exemplo os responsáveis por entradas e saídas seriais (como UART, entre outros).

Andrews (2004) explica que apesar de não possuir uma definição absoluta e formal, o termo SoC (*System on Chip* ou Sistema em Chip) geralmente se refere ao componente que encapsula uma ou mais unidades de processamento, blocos que gerenciam entradas e saídas, blocos de aplicações básicas, além de um ou mais blocos específicos e complexos, que são orientados a aplicações, como por exemplo os que gerenciam Wi-Fi ou *Bluetooth*; Esse nível de abstração gera facilidade na integração e no tempo de desenvolvimento de um sistema embarcado.

Portanto, apesar de nebulosa e não formal, a diferença entre um microcontrolador e um SoC é, resumidamente, que os microcontroladores são mais simples, porém mais gerais, enquanto os SoCs são mais complexos e criados para aplicações mais especificas, como por exemplo o uso de Rede ou aplicações que demandam mais memória do que é tipicamente encontrado em MCUs.

2.3.1 Firmware e Bootloader

Firmware são *software* que controlam *hardware*. Embarcado em microcontroladores ou SoCs, o firmware é eternamente executado com o objetivo de controlar o dispositivo no qual está presente. Um *firmware* pode ser complexo ou simples, independente ou dependente de outros *software* (OSHANA e KRAELING, 2013).

Bootloader é um conjunto de instruções que é executado durante a inicialização de um *hardware*. Esse *software* é responsável por diferentes tarefas, dependendo do *hardware* que é executado. Exemplos de tarefas são: inicialização de registros, inicialização da memória, configurações de *clocks* e, principalmente, a chamada de *firmware* e/ou Sistemas Operacionais (OSHANA e KRAELING, 2013).

2.4 SISTEMA OPERACIONAL *ANDROID*

O *Android* é um sistema operacional com *Kernel Linux* que pode ser executado em diversas plataformas, principalmente em telefones celulares e *tablets*. Fundado em 2003 e posteriormente adquirido pela Google, hoje é o sistema operacional mais utilizado do mundo (ZURIARRAIN, 2017).

Esse sistema operacional *OpenSource* administra os recursos do dispositivo com objetivo de criar uma plataforma responsiva para o usuário; eficiente, principalmente na área energética e na de alocação de recursos; segura e que possibilite uma relativa facilidade de desenvolver aplicações.

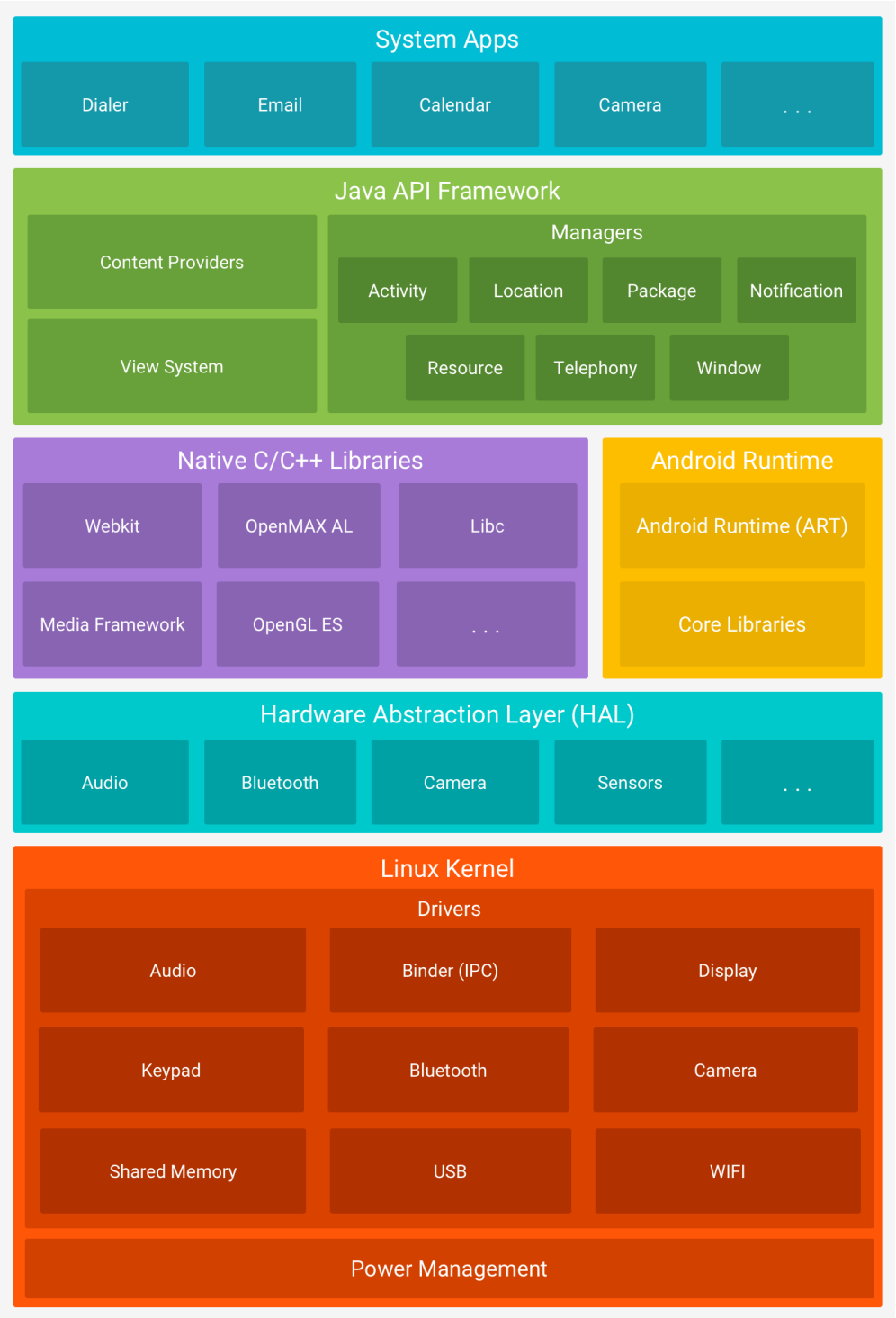
Para promover o desenvolvimento de aplicações, que são escritas principalmente em *Java*, o Google disponibiliza o ambiente de desenvolvimento *Android Studio* e o *Android SDK* (*Software Development Kit* ou *Kit* de desenvolvimento de *Software*). Este pacote de desenvolvimento disponibiliza ferramentas de simulação, ferramentas para *Debbugging*, bibliotecas/*APIs*, além de códigos fonte e ferramentas geradoras de códigos automáticos.

Está disponível aos desenvolvedores também o *Java API Framework*, que consiste, simplificadamente, em um conjunto de *APIs*. Essas *APIs* facilitam o desenvolvimento de aplicativos, uma vez que possibilitam o reuso de alguns componentes do Sistema *Android*, como, por exemplo, o *View System* ou Sistema de Visualização (responsável pelo desenho de estruturas básicas como caixas de texto, tabelas ou botões) e Administradores (*Managers*) de Notificação, Localização, Janela, entre outros, que abstraem a implementação desses recursos na plataforma.

Para atingir tais funcionalidades, esse *framework* se utiliza de uma máquina virtual, chamada *Android Runtime* (*ART)*, e de bibliotecas em C/C++, que devem ser executadas diretamente na máquina real, não na virtual. O *ART* e as bibliotecas em C/C++, por sua vez, fazem uso de códigos específicos para cada periférico do dispositivo. Esses códigos compõem o *HAL* (*Hardware Abstraction Layer* ou Camada de Abstração de *Hardware*), que possibilita que o mesmo código seja compatível com diversos modelos de, por exemplo, câmeras, apenas variando os códigos do *HAL* para seus respectivos *hardware*, o que deve ser feito, naturalmente, pelo fabricante do dispositivo, não das aplicações. A Figura 4, abaixo, ilustra de

forma simples e de fácil legibilidade o relacionamento de todos esses componentes (ANDROID, 2017).

Figura 4: Diagrama que mostra os principais componentes da plataforma Android. Organização com maior nível de abstração conforme mais alto na pilha de blocos exibida.



FONTE: (ANDROID, 2017)

3 MATERIAIS

3.1 PLACA DE PROTOTIPAÇÃO LIGHTBLUE BEAN+

A placa utilizada para o desenvolvimento do projeto foi fabricada pela empresa americana PunchThrough Design. A jovem empresa criada em 2009 é especializada em projetos com *Bluetooth* e têm produtos e projetos sendo utilizados por empresas como Google e organizações como a NASA (PUNCHTHROUGH DESIGN, 2017a).

Figura 5: Página da Internet da desenvolvedora de projetos e produtos com Bluetooth *PunchThrough Design.*

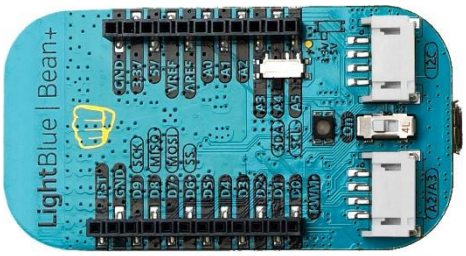
FONTE: Autoria Própria.

Com quase 100mil unidades vendidas, a família de placas de desenvolvimento *LightBlue Bean* tem o objetivo de criar plataformas simples, porém potentes para desenvolvimento de produtos com *Bluetooth Low Energy*. Existem duas versões na família, a mais simples, menor e com baixo custo é a versão *Bean*, que pode ser utilizada no produto final de pequenos projetos pessoais e industriais, e a versão *Bean+*, que é maior, mais robusta e possui mais pinos e conexões. Essa última versão é a utilizada neste projeto. As placas dessa família também contam com bibliotecas e *APIs Android*/*IOS* pré-desenvolvidos que agilizam o tempo de desenvolvimento.

A família de placas tem suporte de "Prototipação para Produção em Massa". Esse é o suporte dado pela empresa PunchThrough ao disponibilizar todos os arquivos e dados

necessários para a transformação do protótipo em um produto final de baixo custo. Para isso, é disponibilizado todo o esquemático da placa com todos os componentes especificados, além do *Bootloader* e *Firmware* necessários, e de suporte pago para consultoria (PUNCH THROUGH DESIGN, 2017b).

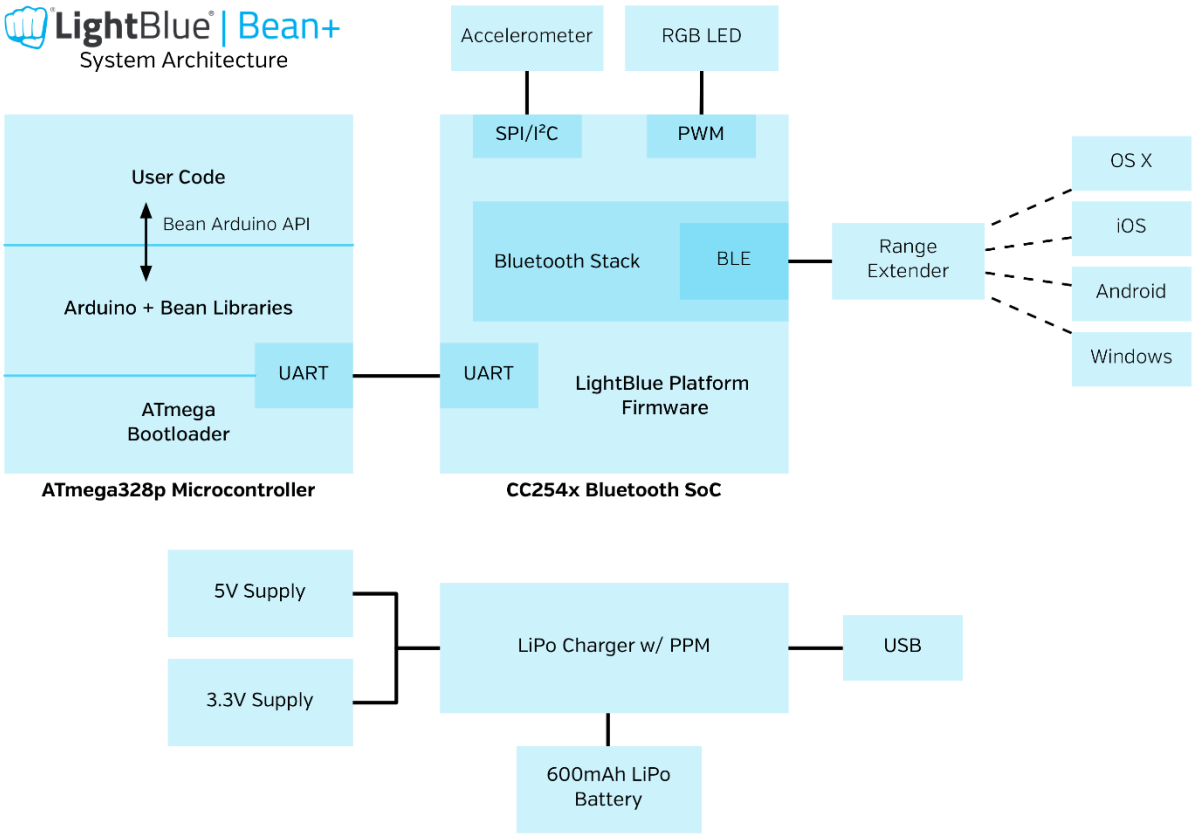
Figura 6: Placa de desenvolvimento LightBlue Bean+



FONTE: *(PUNCHTHROUGH DESIGN, 2017a)*

A arquitetura do sistema, como mostrado na Figura 7, consiste principalmente de um SoC *Bluetooth* CC254x, feito pela Texas Instruments e de um Microcontrolador ATmega328p, produzido pela Atmel. O sistema foca em abstrair detalhes de baixo nível da utilização do *Bluetooth*, utilizando o SoC para a sua administração. Para alcançar essa abstração, a parte programada pelo usuário da placa (na Figura 7 representada como *User Code*) é feita apenas para o ATmega328p e, utilizando bibliotecas especificas, esse MCU se comunica transparentemente com o SoC para utilização do *Bluetooth*. Essa comunicação é feita utilizando UART (*Universal Asynchronous Receiver-Transmitter)*, um barramento simples de dois fios, que por ser assíncrono, necessita pré-configuração pelas duas partes envolvidas.

Figura 7 Arquitetura do Sistema da placa LightBlue Bean+.



FONTE: (PUNCHTHROUGH DESIGN, 2017c)

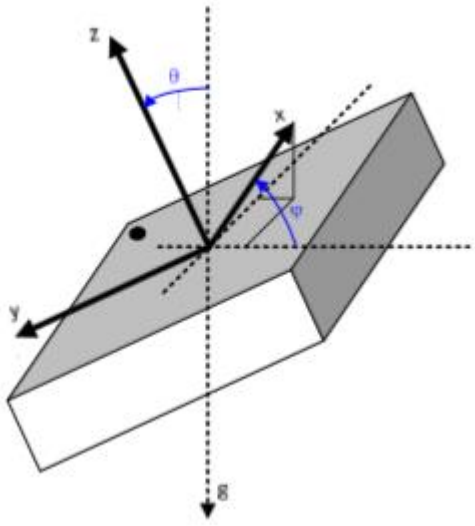
Além da abstração do *Bluetooth*, também é abstraído do desenvolvedor um *LED RGB* e um Acelerômetro. Esses periféricos são conectados diretamente com o SoC *Bluetooth*. A conexão do acelerômetro, implementada com SPI ou I2C, é transparente e, por esse motivo, não deve ser administrada pelo usuário. A conexão do *LED RGB* é feita com uma porta *PWM* (*Pulse Width Modulator*), que significa que o *LED* tem a capacidade de brilhar em diversas intensidades por receber uma entrada digital que, na prática, atua como uma entrada analógica. Essa conexão também é transparente para o usuário, por meio de bibliotecas específicas.

O microcontrolador Atmega 328p é o mesmo utilizado pela conhecida placa Arduino Uno. Arduino é uma plataforma *Open-Source* para desenvolvimento de projetos eletrônicos. Essa plataforma conta com uma grande quantidade de documentação além de diversas ferramentas que ajudam no desenvolvimento de projetos (ARDUINO, 2017). Uma dessas ferramentas é o conjunto de bibliotecas que abstrai diversas funções do microcontrolador

utilizado e essas bibliotecas são compatíveis com a placa *LightBlue Bean+* pela utilização do mesmo microcontrolador além de um *Bootloader* compatível.

O acelerômetro utilizado no *LightBlue Bean+* e, consequentemente, no projeto é o BMA250E, feito pela BOSCH (PUNCHTHROUGH DESIGN, 2017c). Esse é um acelerômetro de 3 eixos, que significa que aceleração é medida nos eixos X, Y e Z, como pode ser observado na Figura 8. A medição é feita utilizando microssistemas Eletromecânicos (MEMS ou *micro electro-mechanical system*), que são sistemas que utilizam grandezas mecânicas e grandezas elétricas. Neste caso, converte aceleração (uma grandeza mecânica) em capacitância (uma grandeza elétrica). Esse sensor de baixo consumo energético possui vários modos de uso e sua saída é digital.

Figura 8: Eixos 'x', 'y' e 'z' mostrados em relação ao eixo da terra 'g'.



FONTE: (BOSCH, 2011)

O BMA250E é altamente configurável e possui modos que detectam padrões ou que geram uma saída conforme a quantidade de [g]s observados. O [g] é uma medida de aceleração e corresponde a uma gravitação da terra, ou 9.6[m/s^2]. A acurácia do sensor é fixa em 10bits, porém sua escala pode ser selecionada entre ±*2g*, ±*4g*, ±*8g* e ±*16g*. Isso significa que, por exemplo, na escala de ±4g os valores entre -4 [g] e + 4 [g] serão mapeados com valores entre -512 e +511, que são os valores limites com sinal que podem ser representados por esse número fixo de bits, calculado utilizando a equação A com o valor total deslocado

para representar também números negativos. O passo da medição e sua precisão será, neste caso, de 0.0078125 [g] e de 0.0068125 [g/bit], podendo ser calculado utilizando a equação B.

$$(2^{-n}) = \textit{número máximo de valores representados por n bits.} \quad (1)$$

$$2 * P/(2^{-n}) = \textit{precisão} \quad (2)$$

Os modos que detectam padrões geram saída ALTA em pinos específicos caso o padrão seja detectado e BAIXO caso contrário, e não foram utilizados no projeto.

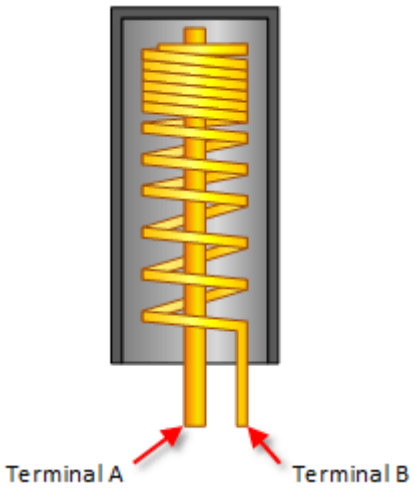
O *LED RGB (Red/Green/Blue* ou Vermelho/Verde/Azul) é um *LED* que é capaz de brilhar em todas as cores compostas por vermelho, verde e azul. Foi utilizado o modelo CL-SF687RGB, feito pela CIEL Light (CIEL LIGHT, 2012) que é composto por 3 *LEDS* nas respectivas cores supracitadas, além de apresentar grande intensidade de brilho e uma superfície quase plana.

O Bean+ também possui uma bateria recarregável de *Lithium-Ion*(*LIPO)*. Essa bateria possui uma carga completa de 600[mAh] e, conforme a completude da carga, fornece de 2.5[V] até 4.2[V] de tensão. O Bean+ é capaz de calcular a porcentagem da carga total carregada a partir desta propriedade, isto é, a propriedade em que a tensão de saída da bateria varia conforme a completude da carga.

3.2 SENSOR DE CHOQUE KY-031

Esse simples sensor detecta vibrações e, consequentemente, choques físicos ao dispositivo. Também é conhecido como *Knock Sensor* por ser utilizado em detecção de Batidas em portas e similares. *Constituído* de um *Knock Switch* e um resistor de *Pull-Up,* a saída é ALTA quando não detectada nenhuma vibração e BAIXA caso contrário. O *Knock Switch* é formado por dois terminais, sendo um em forma de mola, como pode ser visto na Figura 9. A vibração, quando grande o suficiente, causa a conexão de ambos os terminais resultando uma saída em baixo (UDOO NEO, 2017).

Figura 9: Funcionamento do Knock Switch



FONTE: (UDOO NEO, 2017)

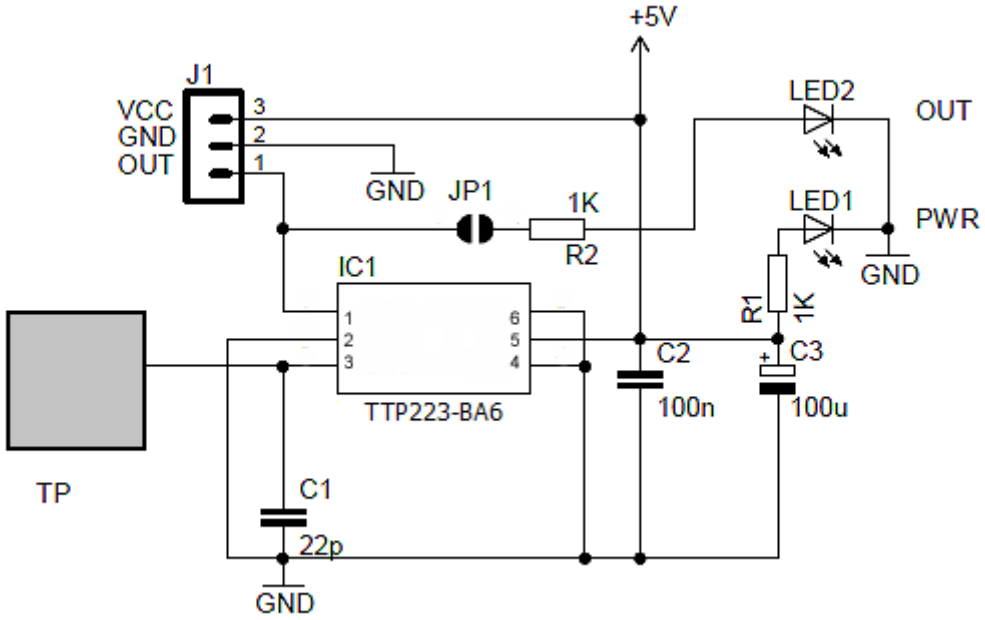
3.3 SENSOR DE TOQUE *TTP223*

O sensor de toque capacitivo tem o objetivo de disponibilizar uma entrada analógica, como a de um botão, porém mais agradável ao usuário e mais durável para o dispositivo, por não possuir partes mecânicas. Sua saída é digital, ALTA quando sentido um toque, BAIXA, caso contrário.

O sensor é constituído por um TTP223, o CI principal, um capacitor C1 de 22[pF], que determina a sensibilidade base da capacitância sentida pelo CI e por outros componentes mais simples, que podem ser vistos na

Figura 10. Esse sensor possui a capacidade de configuração, por meio de suas portas de entrada, entre saída ALTA e BAIXA, além do modo *Toggle* ou Direto. Além disso, possui modo automático de economia de energia quando não detecta toques por 12 segundos e ajuste automático de entrada, que possibilita mais precisão com diferentes capacitâncias de entrada. A capacitância de entrada é gerada por um *SensePad*, que consiste apenas em uma grande área de material condutor que, ao ser pressionada, gera uma capacitância entre o objeto (I.E. o dedo de um operador) e o circuito. (TONTOUCH, 2008)

Figura 10: Esquemático de um TouchSensor



FONTE: (TONTOUCH, 2008)

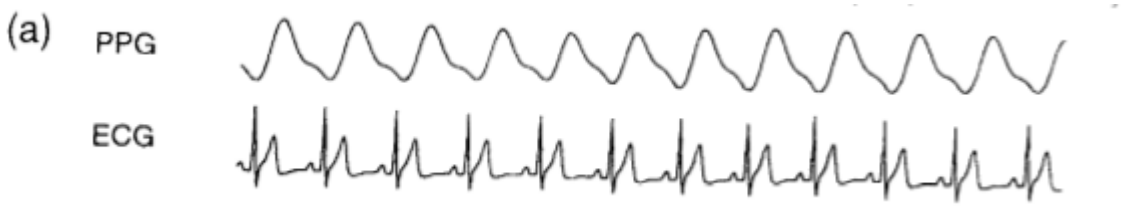
3.4 SENSOR DE BATIMENTOS CARDÍACOS – SEN11574

O Sensor de Batimentos Cardíacos tem o objetivo de analogicamente ler, utilizando luz, a densidade de células de sangue do usuário, formando assim uma curva que se correlaciona com a frequência de batimentos cardíacos.

Essa técnica é chamada de Fotopletismografia (*Photoplethysmography* ou *PPG*) e funciona da seguinte maneira: um emissor de luz e um receptor de luz são posicionados próximos. A luz é direcionada para a pele humana quando em contato e parte desta luz é absorvida enquanto outra parte é refletida. A porção refletida é medida pelo receptor. A quantidade de luz absorvida pela pele, ossos e carne é quase constante enquanto a quantidade absorvida por células do sangue varia devido ao fluxo sanguíneo cardiovascular. Sendo assim, é possível observar picos na absorção de luz que correspondem aos batimentos

cardíacos (NAKAJIMA, TAMURA e H.MIIKE, 1994). A curva formada pode ser observada na Figura 11.

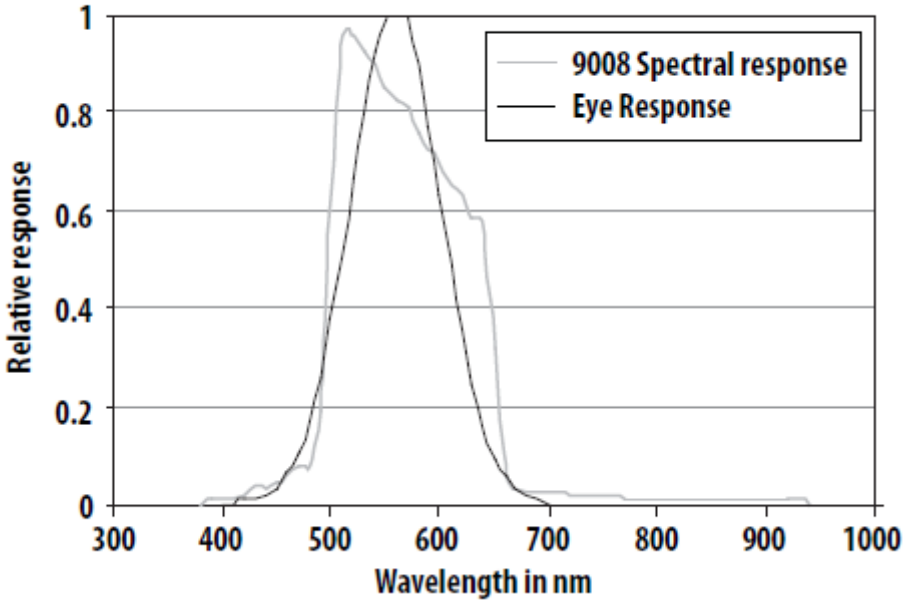
Figura 11: Curva gerada pela técnica de Fotopletismografia.



FONTE: (NAKAJIMA, TAMURA e H.MIIKE, 1994)

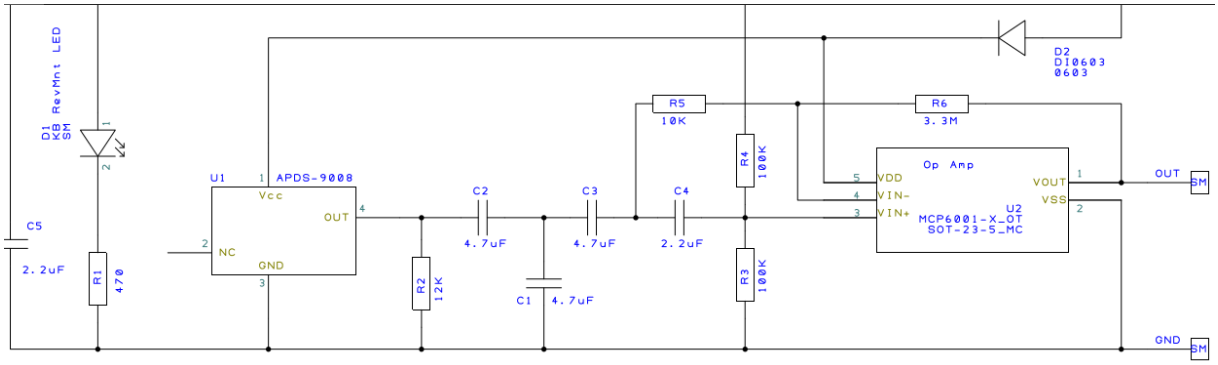
No sensor SEN11574, a emissão de luz é realizada por um forte *LED* de cor verde, pois é utilizado um fotorresistor com o pico de absorção coincidente com comprimentos de onda da luz nessa cor, como é possível observar na curva de absorção do componente, mostrada na Figura 12. A saída do sensor de luz é amplificada, como mostrado no esquemático da Figura 13 ,com o objetivo de facilitar a leitura do sinal. A saída do amplificador corresponde à saída do *SEN11574*.

Figura 12: Curva de Absorção Relativa do Fotoresistor APDS-9008 utilizado pelo sensor de batimento cardíaco.



FONTE: (AVAGO TECHNOLOGIES)

Figura 13: Esquemático do Sensor de Batimentos Cardíacos SENS-11574 utilizado.



FONTE: (JOEL MURPHY, 2017)

4 MÉTODOS

Esse projeto, que tem como objetivo geral uma comunicação rápida e fácil em momentos de emergência, possui duas principais partes. A primeira parte é a do sistema embarcado, isto é, o *hardware* e seu *software* embarcado, que representa o maior volume de desenvolvimento do projeto e será utilizado para a principal interação e controle do usuário. A segunda parte é a do aplicativo *Android*, que é responsável pela comunicação com externos (ou seja, o contato de emergência definido). A comunicação entre o *Wearable* e o Aplicativo *Android* deve ser transparente para o usuário, rápida e eficiente energeticamente. Ambos os software foram desenvolvidos com uma metodologia incremental, com execução de testes funcionais durante cada iteração incremental. Os códigos podem ser encontrados integralmente no *GitHub* do projeto: github.com/jgobbic/TCCWearableGobbi .

4.1 DECISÕES DE PROJETO

Um projeto de um dispositivo vestível (*wearable)* deve considerar o gasto energético, além de, normalmente, uma comunicação prática com *smartphones.* Por esses motivos, foi decidido a utilização do *Bluetooth Low Energy* no projeto, para realizar eficientemente a comunicação com o *smartphone.*

A placa de desenvolvimento *LightBlue Bean+* foi escolhida por dois principais motivos. Primeiramente, ela foi desenvolvida por um grupo especializado em *Bluetooth,* que possui projetos executados para grandes empresas. Além disso, a placa abstrai detalhes trabalhosos do *Bluetooth*, que passa uma maior confiabilidade da comunicação enquanto aumenta a velocidade de desenvolvimento do projeto. A placa, contudo, não seria utilizada se não oferecesse suporte para a transformação do protótipo em produto final. Com detalhes, e esquemáticos, dos circuitos internos da placa, componentes que podem ser encontrados no mercado e *firmware/bootloaders* disponíveis, a criação de um módulo pequeno e de baixo custo compatível com o *software* e *hardware* desenvolvidos neste projeto seria facilitada.

Para a realização da interface do sistema com o usuário, foram escolhidos dois tipos de botões: um físico e um *touch*. O físico, que possui maior resposta tátil, ou seja, é mais

responsivo para o usuário, é utilizado para a ativação do modo de emergência, de forma que o usuário possa ter considerável conforto de que o botão foi de fato pressionado e o pedido de socorro foi enviado. O botão *touch* foi escolhido por ser mais agradável para o toque contínuo, além de ter maior durabilidade de uso constante, e é utilizado para entrar em modo de batimentos cardíacos. A interface de apenas dois botões foi escolhida para facilitar o uso por idosos, que tendem a ter dificuldades na utilização de novas tecnologias.

Foi escolhido desenvolver o aplicativo compatível com o sistema *Android* pois, além de ser o sistema mais utilizado no mundo, é o sistema mais acessível pois existe uma grande variedade de *smartphones* de baixo custo com esse sistema operacional disponíveis.

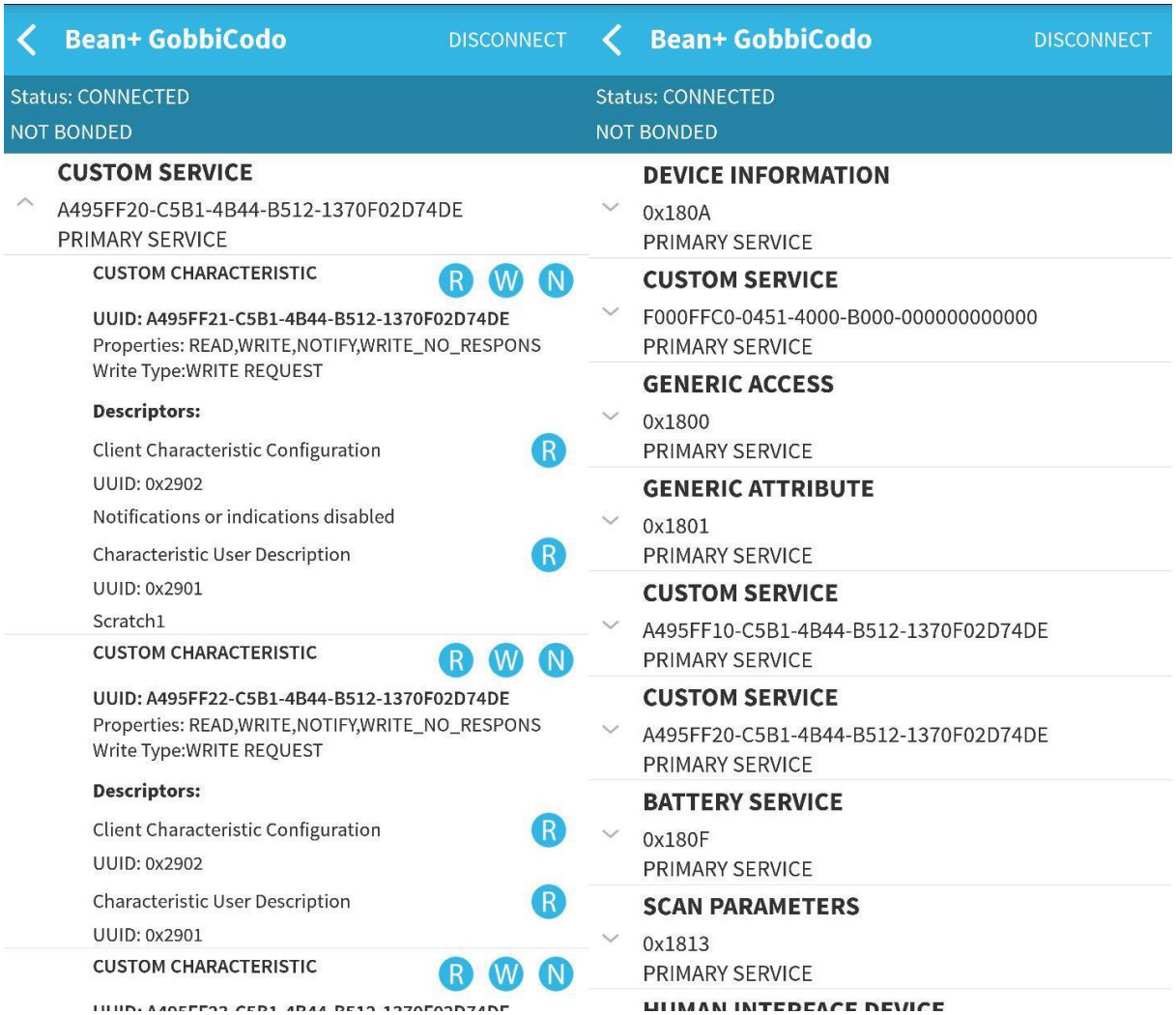
Por último, foi escolhido o envio de mensagens por SMS, ao invés de mensagens por meio de uso da internet, pela maior cobertura do serviço (PRADO, 2017).

4.2 ARQUITETURA GERAL DO SISTEMA

A comunicação entre as duas partes principais é o ponto chave do projeto criado. Essa comunicação, que foi implementada em *BLE,* funciona por meio de *characteristics* implementadas por um *service* genérico, seguindo a estrutura mostrada na Figura 2, do Capítulo 1. Essas *characteristics* são, para o sistema, como dados compartilhados entre os dispositivos conectados, ou seja, na prática, são variáveis compartilhadas entre o dispositivo embarcado e o aplicativo *Android.*

A implementação destas *characteristcs* é feita por meio de um dos 5 *banks* disponibilizados pelo *firmware* do *SoC Bluetooth* do LBB+. Esses bancos são *characteristics* de um *service* genérico. Toda vez que algum dado desse serviço é alterado, por qualquer uma das partes, a outra parte é notificada e é feita a aquisição dos dados. Os *services* e *characteristics* podem ser vistos em detalhes no aplicativo *BLE SCANNER* (BLUEPIXEL TECHNOLOGY LLP, 2017), como mostrado na Figura 14.

Figura 14: Serviços e Caracteristicas do Bean+ mostrados num aplicativo de analise BLE.



FONTE: Autoria Própria.

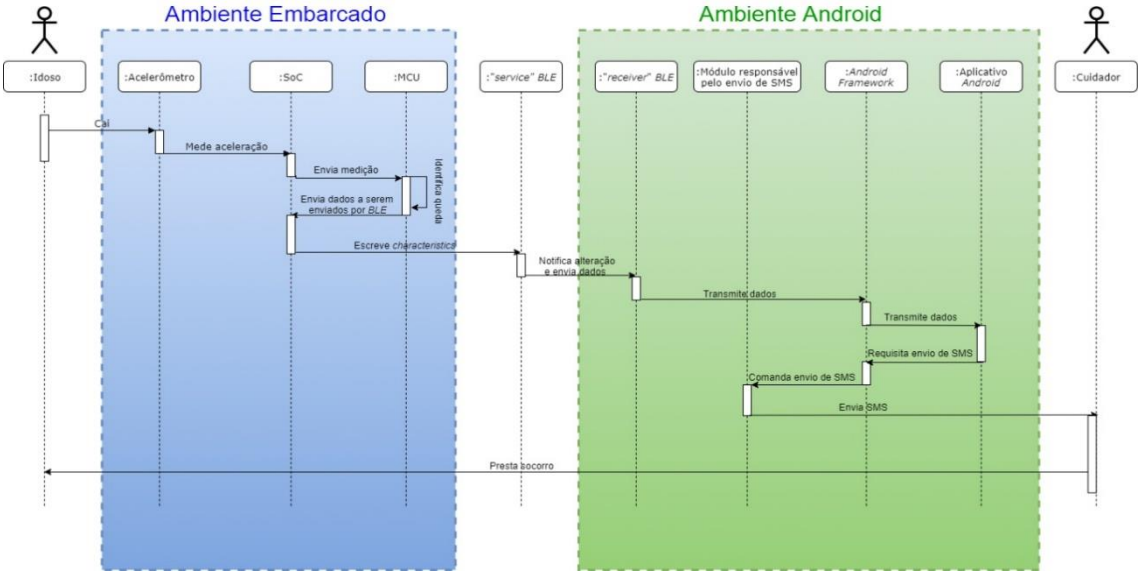
Um diagrama de sequência que representa todo o fluxo de comunicação para o caso de detecção de queda pode ser visto na

Figura *15*.

O sistema foi implementado de maneira modular com o objetivo de facilitar a implementação de novas funcionalidades no futuro. A transmissão de dados entre os dispositivos é feita de maneira que, caso seja desejado a inserção de funcionalidades, é

necessário apenas a utilização de um dos 5 bancos para a transmissão dos dados. Cada banco suporta até 20 bytes de dados e a leitura de cada um é feita separadamente.

Figura 15: diagrama de sequência exibindo o funcionamento no caso de detecção de queda.



FONTE: Autoria Própria.

4.3 IMPLEMENTAÇÃO DO SISTEMA EMBARCADO

Para a implementação do sistema, o primeiro passo é o processo de preparação do ambiente para a gravação do código que será executado no microcontrolador, que pode ser chamado de *firmware* do *MCU*. Esse passo não é trivial, pois, no *LightBlue Bean+* a programação deve ser feita de modo *wireless*, por meio do *Bluetooth Low Energy.* A programação é feita desse modo pois o real responsável pela gravação no microcontrolador é o *SoC Bluetooth* (é importante lembrar a diferença entre ambos, tratada na seção *Sistemas Embarcados: MCUs e SOCs*).

Existem diferentes abordagens para fazer essa programação, e duas delas foram utilizadas no projeto: Uma por meio de um aplicativo *Android*, chamado de *Bean Loader for Android* e outra por meio de uma aplicação de linha de comando em ambiente *Linux*, chamada

de *CLI Loader* (*Client Line Interface Loader*). A abordagem utilizando o ambiente *Android* é mais simples e intuitiva e foi a mais utilizada, uma vez que o acesso a um computador com suporte ao *Bluetooth Low Energy* foi restrito durante a execução do projeto. Sendo mais simples, a robustez e o detalhamento da compilação por meio do aplicativo *Android* são mais baixos, tornando mais difícil o processo de d*ebugging* do código. A programação em um ambiente *Linux* é, portanto, mais completa e facilita o d*ebug,* porém, envolve uma pré-configuração do ambiente mais trabalhosa. Além disso, apenas o *CLI Loader* é capaz de atualizar o f*irmware* do *SoC Bluetooth*. A abordagem com *Linux* foi utilizada quando era necessário maior nível de detalhes para *debuging*.

No ambiente *Linux*, a compilação do código é feita a partir do *Arduino IDE* e a transmissão para a placa de desenvolvimento a partir do *CLI Loader*.

4.3.1 Configuração do ambiente em Linux

A configuração e programação desta placa é diferente do habitual, e por esse motivo, será mostrada em detalhes nesta seção.

Para ser feita a configuração, é necessário ter *Python 2.7* instalado no sistema, além de bibliotecas específicas para a manipulação do *Bluetooth*. A instalação de ambos pode ser feita com a execução dos dois comandos abaixo em um terminal *bash*:

```
$ sudo apt-get install python 2.7.1
$ sudo sudo apt-get install bluetooth bluez libbluetooth-dev libudev-dev
```

Deve-se instalar também o *Node.js*, *framework* que é utilizado pelo *CLI Loader*. Isso pode ser feito executando os comandos abaixo, que fazem d*ownload* do instalador, extraem o arquivo baixado e, por último, instalam o *framework* na pasta adequada.

```
$ curl -O https://nodejs.org/dist/v6.11.5/node-v6.11.5-linux-x64.tar.xz
$ tar -xf node-v6.11.5-linux-x64.tar.xz
$ cp -R node-v6.11.5-linux-x64/* /usr/local/
```

Por último, deve-se instalar o administrador de pacotes *JavaScript* chamado *npm*, e, finalmente, o *CLI Loader*.

```
$ sudo install npm -g
$ sudo npm install -g –unsafe-perm bean-sdk
```

Para a compilação do código, é necessário o *Arduino IDE*, que pode ser baixado, extraído e instalado com os seguintes comandos:

```
$ curl -o http://downloads.arduino.cc/arduino-1.8.5-linux64.tar.xz
$ tar -xf arduino-1.8.5-linux64.tar.xz
$ cd arduino-1.8.5-linux64
$ ./install.sh
```

Com o *Arduino IDE* e o *CLI Loader* instalados, é necessário, então, fazer a instalação das bibliotecas personalizadas do *LBB+* na *IDE*. O comando e uma mensagem de sucesso para toda a execução do processo pode ser visto no código abaixo e na Figura 16 abaixo.

```
$ sudo bean install_bean_arduino_core
```

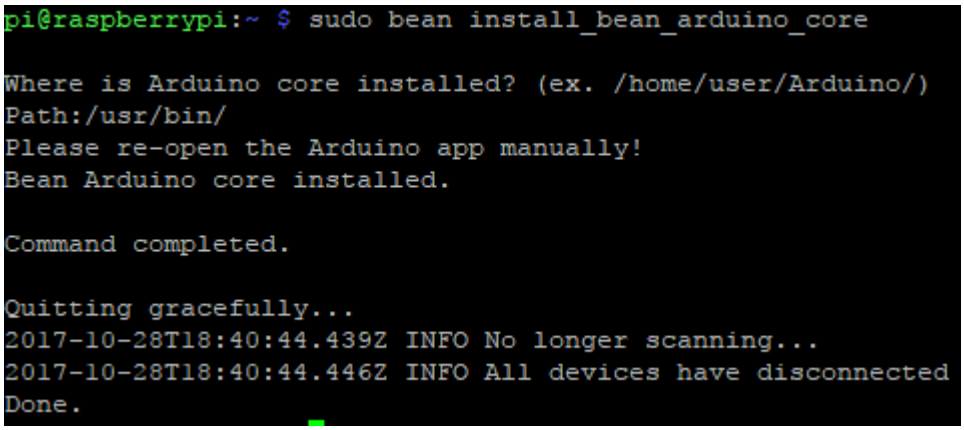


Figura 16: Processo de instalação do CLI Loader e dependências concluído.

FONTE: Autoria Própria.

4.3.2 Configuração no Ambiente Android

No ambiente *Android*, a compilação é feita na Nuvem e apenas o arquivo compilado é automaticamente baixado e transmitido para a placa de desenvolvimento.

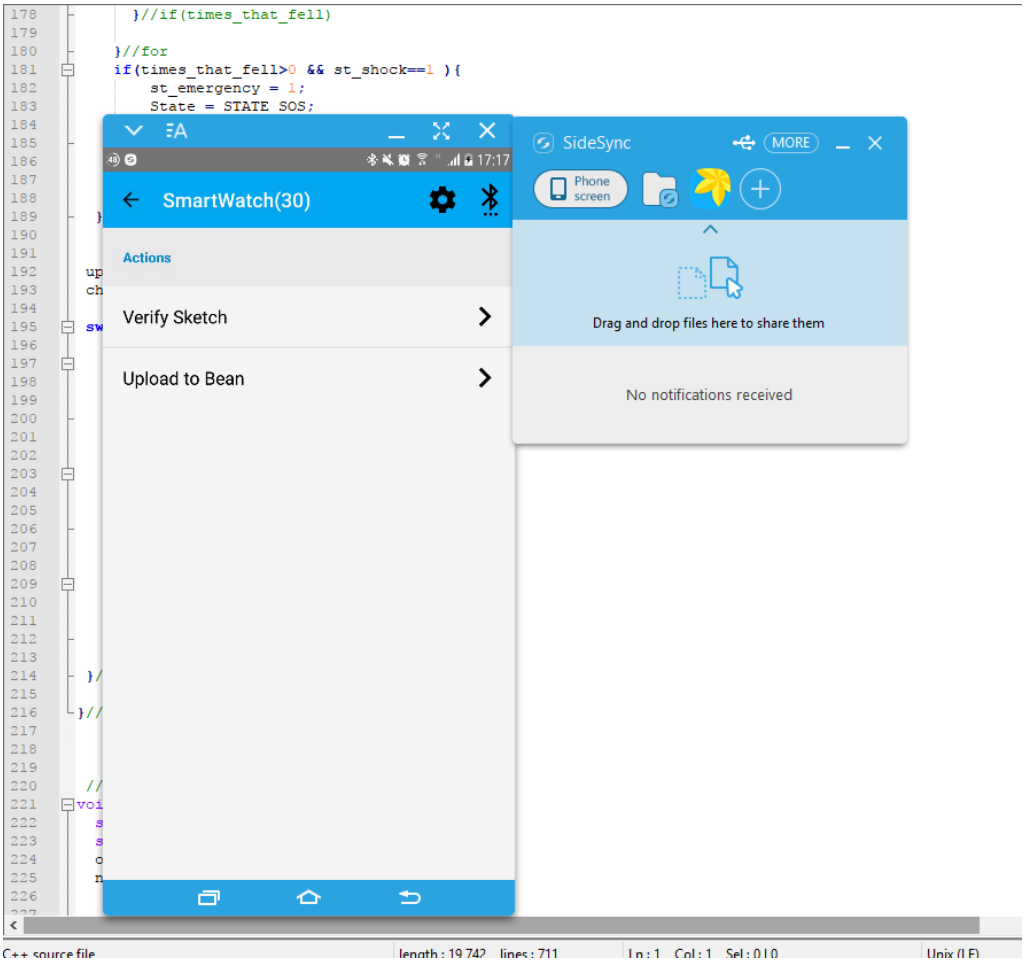
Para a instalação do *Bean Loader for Android,* é apenas necessário o d*ownload* do aplicativo a partir da loja do *Google Play*, que pode ser visto na Figura 17.

Figura 17: Aplicativo Bean Loader *que deve ser descarregado da* PlayStore.

FONTE: Autoria Própria.

Para facilitar a transferência de arquivos entre o computador e o *Smartphone Android,* além da utilização mais ágil do aplicativo, foi instalado no computador em que o projeto estava sendo desenvolvido o programa *SideSync*, da Samsung, que possibilita a transferência de arquivos ao estilo *Drag'n'Drop* (arrastar e soltar), além do controle total do *smartphone* pelo computador.

Figura 18: Software SideSync sendo utilizado para controlar e transferir arquivos para o Smartphone, que aumenta a produtividade utilizando o Bean Loader for Android*.*



FONTE: Autoria Própria.

4.3.3 Atualização do Firmware em ambiente Linux

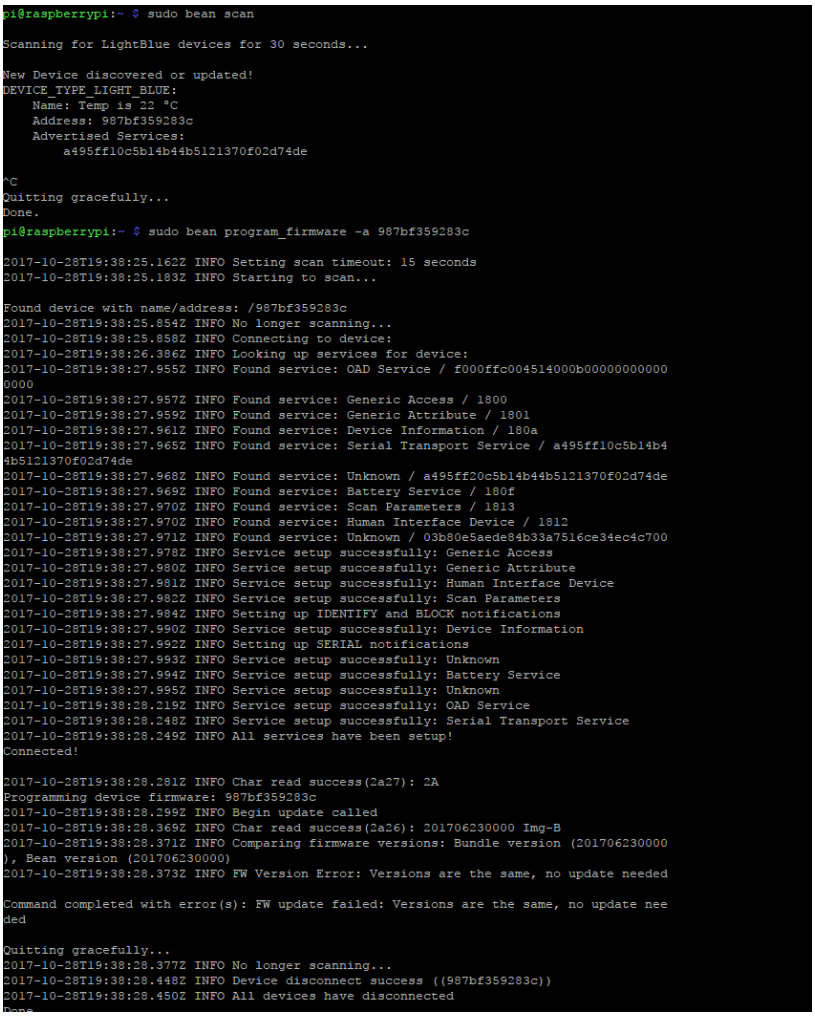
A atualização do *firmware* do *SoC* do *Bean* deve ser feita. Como explicado anteriormente, isso pode ser feito somente pelo *CLI Loader*, pois o aplicativo *Android* ainda não possui suporte para isso. *Esse firmware* é o que administra toda a utilização do *Bluetooth*, além de tratar de tarefas críticas como, por exemplo, a gravação do *software* criado no microcontrolador. Para realizar tal atualização, deve-se encontrar o *Bean+* desejado na lista de dispositivos *BLE* identificados, encontrar o seu endereço *Bluetooth* e posteriormente executar a atualização. Esse endereço é único e fixo para cada dispositivo *Bluetooth* existente. Os comandos que devem ser executados são:

```
$ sudo bean scan
$ sudo bean program_firmware -a 987bf359283c
```

A chave '-a' no comando *bean program_firmware* especifica o endereço do Bean a ser atualizado. O endereço é encontrado com o comando *bean scan*.

A atualização deve ser realizada, e será bem-sucedida quando o *firmware* encontrado no *Bean+* é o mesmo que se tenta instalar, como pode ser visto na Figura 19 abaixo.

Figura 19: Processo de atualização de firmware concluído.



FONTE: Autoria Própria.

4.3.4 Programação do MCU em ambiente Linux

A gravação de código no microcontrolador, como foi dito anteriormente, pode ser feita por ambos os ambientes. Para a realização no ambiente *Linux*, o código deve ser compilado no *Arduino IDE* e transmitido para o *Bean* por meio do *CLI Loader*.

Para o código ser compilado corretamente, compatível com o *Bean+*, deve-se selecionar a placa de desenvolvimento no menu *Tools -> Board -> LightBlue Bean+*, como pode ser visto na Figura 20.

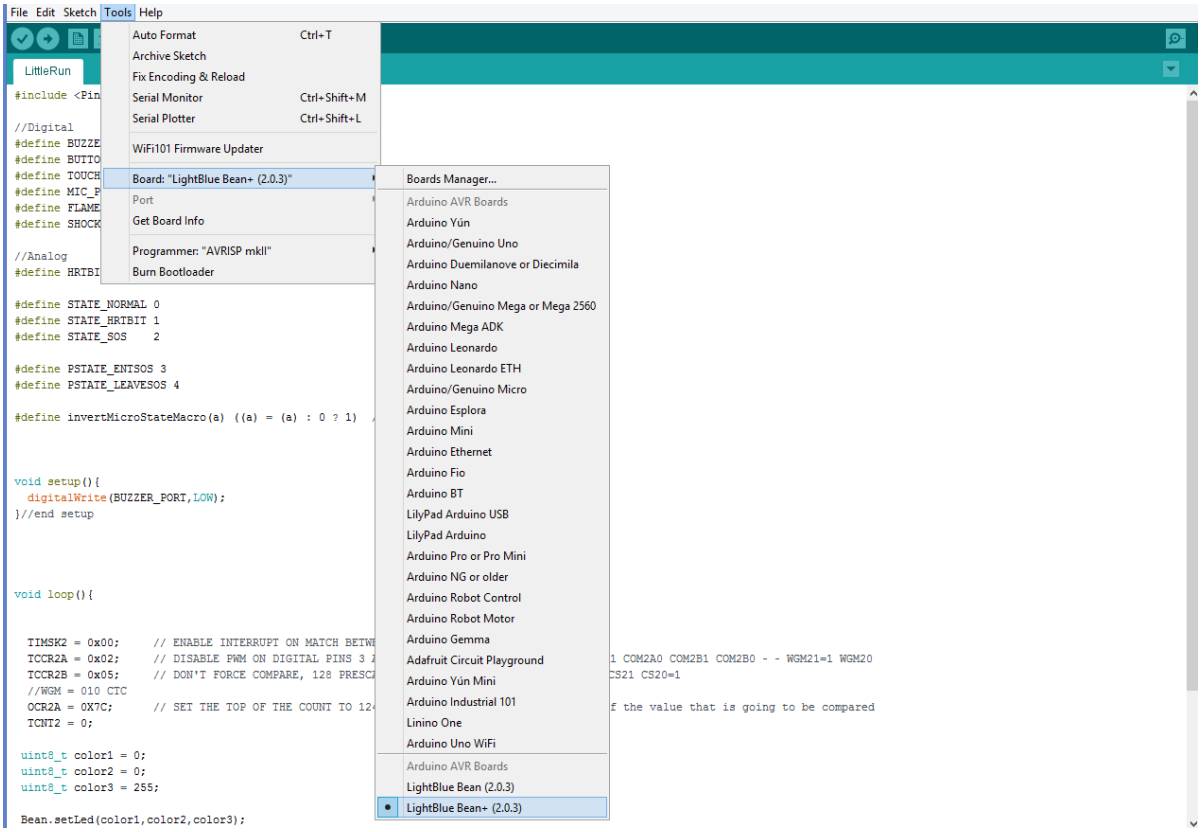


Figura 20: Selecionando a placa de desenvolvimento LightBlue Bean+ *no* Arduino IDE.

FONTE: Autoria Própria.

Após a seleção, deve-se clicar em *verify* e em seguinda *upload*. O botão de u*pload*, ao contrário do que se pode imaginar, não transmite para o *Bean+* o código compilado, apenas o prepara para ser enviado manualmente pelo *CLI Loader.* Os botões são exibidos na Figura 21.

Figura 21: Botões de Verificação e Upload *no Arduino IDE*

FONTE: Autoria Própria.

Após preparado para o envio, deve-se executar os comandos:

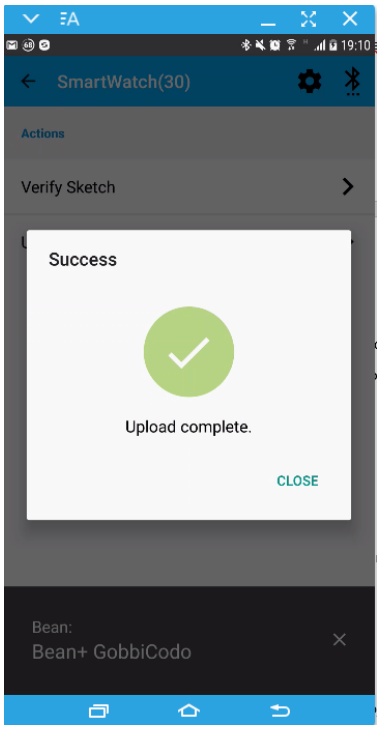
```
$ sudo bean list_compiled_sketches
$ sudo bean program_sketch LittleRun -a 987bf359283c
```

O primeiro comando lista os códigos compilados e prontos para ser enviados, e o segundo efetivamente transmite o *sketch,* isto é, o código selecionado *LittleRun* para o *Bean+* com endereço *Bluetooth 987bf359283c.*

4.3.5 Programação do MCU no ambiente Android

A programação utilizando o *Bean Loader for Android* é um pouco mais simples e direta, feita da seguinte forma. Após enviar o arquivo que contém o código fonte para o *smartphone*, deve-se selecioná-lo no menu "*Choose Sketch*" ("Selecionar *Sketch*"), no aplicativo do *Bean,* e então, após selecionado o *Bean+* alvo, deve-se selecionar "*Upload to Bean*". Uma mensagem apontando o sucesso da operação deve ser exibida, como mostrado na Figura 22.

Figura 22: Mensagem de Upload bem-sucedido no Bean Loader for Android.

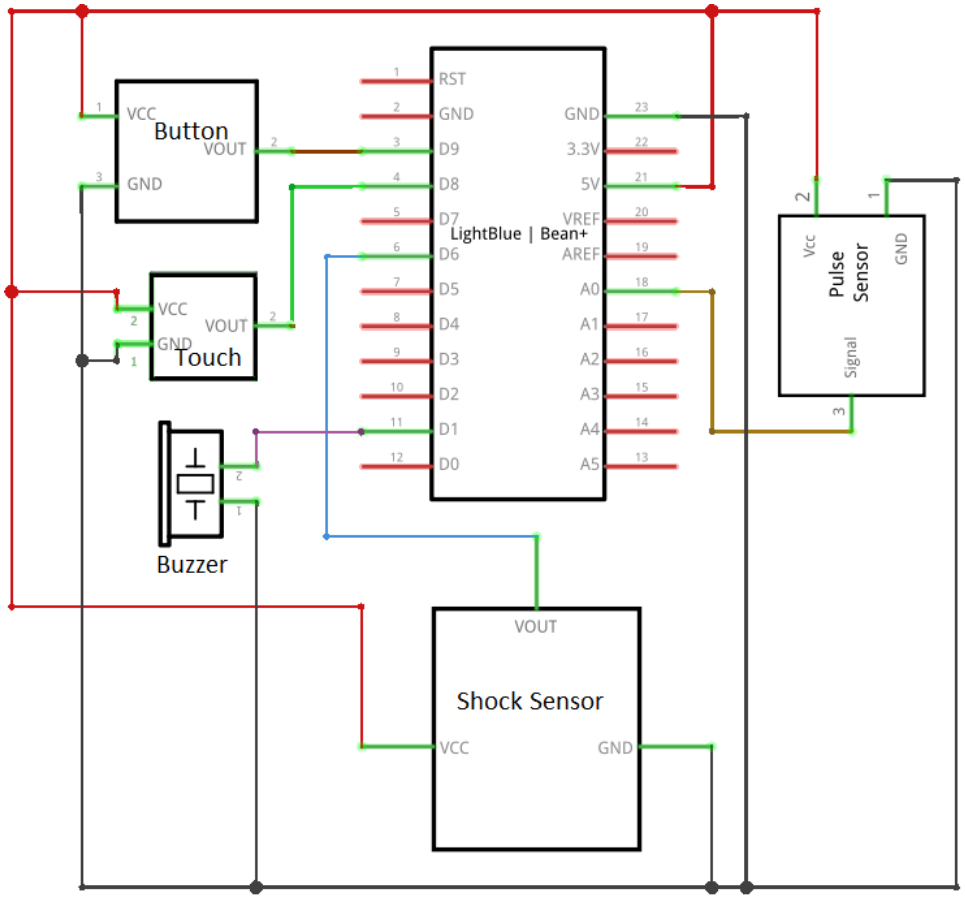


FONTE: Autoria Própria.

4.3.6 Implementação do código embarcado

O conjunto de materiais foi integrado como mostra o esquemático da Figura 23. Como podemos observar, o *Touch Sensor*, o *Botão*, o *Buzzer* e o *Shock Sensor* foram conectados a portas digitais, enquanto o *Sensor de Batimentos Cardíacos* foi conectado a uma porta analógica. Além disso, como foi dito, o *LED RGB* e o *Acelerômetro* estão pré-integrados dentro do *LightBlue Bean+*.

Figura 23: Esquemático do sistema implementado.



FONTE: Autoria Própria.

Pino	**Periférico**
A0	Sensor de batimentos cardíacos
D1	Buzzer
D6	Sensor de choque
D8	Sensor de toque
D9	Botão físico

Tabela 1: Conexões do LightBlue Bean+ com sensores.

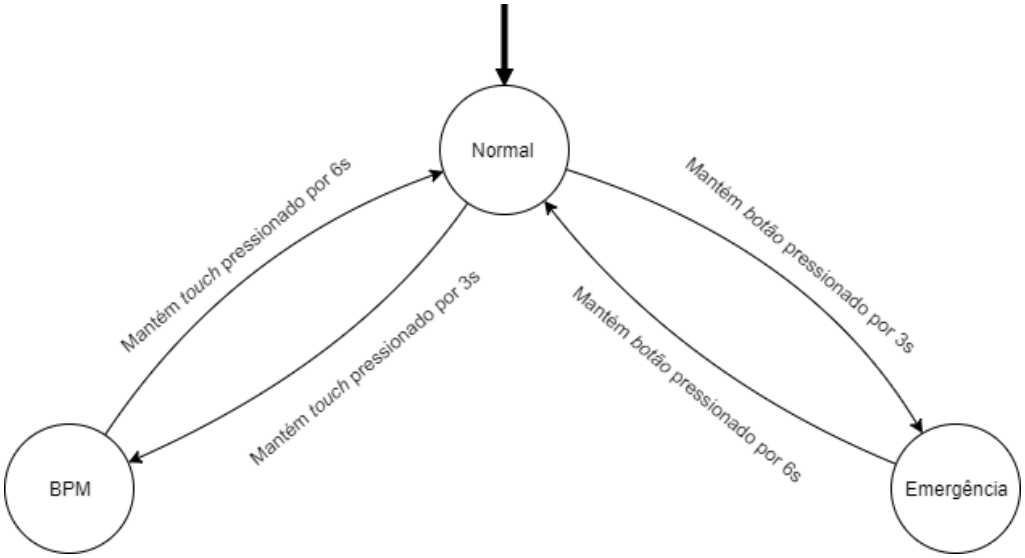
O funcionamento do *software* que é executado pelo microcontrolador é dividido em três partes: *setup(), loop()* e *interrupções.* O *setup()* é executado apenas uma vez, inicialmente no código, e faz configurações iniciais. O *loop()* é executado continuamente, ou seja, recomeça quando sua execução termina (análogo a estrutura *while(True) { }*). As *interrupções* são trechos especiais que são executados quando certos eventos ocorrem. Quando finalizadas, a execução

retorna ao *loop()*. Também existe o *escopo global,* onde são declaradas diretivas de compilação, além da declaração de variáveis globais. As variáveis globais, diferentes das locais, podem ser lidas e modificadas em todas as partes do código.

4.3.7 Funcionamento da máquina de estados que define o funcionamento geral

O funcionamento do *firmware* do microcontrolador é sumarizado por uma máquina de estados que, conforme a leitura feita dos sensores integrados, transaciona entre os estados. A FSM (*Finite State Machine* ou Máquina de Estados Finito) pode ser observada na Figura 24.

Figura 24: Maquina de estados finita que define o funcionamento do sistema embarcado.



FONTE: Autoria Própria.

A implementação da máquina de estados foi feita com a estrutura *switch/case* padrão da linguagem C e é executado continuamente, em todos os ciclos da função *loop()*. Dentro de cada *case (STATE_NORMAL, STATE_SOS e STATE_HRTBIT)* é executada uma função correspondente (*normalState(), sosState()* e *hrtbitState()*) que define as rotinas correspondentes para cada estado. Essas funções serão apresentadas e explicadas posteriormente nesta seção.

```
#define STATE_NORMAL 0
#define STATE_HRTBIT 1
```

```
#define STATE_SOS    2
[…]
void loop(){
      […]
switch(State){
   case STATE_NORMAL:
   {   normalState(); //Execução do Estado Normal
        break; }
   case STATE_SOS:
   {    sosState(); //Execução do Estado de Emergencia
        break; }
   case STATE_HRTBIT:
   {    hrtbitState(); //Execução do Estado de Leitura de Batimentos
        break; }
}
}
```

4.3.8 Estado Normal

O estado *normal*, definido por *STATE_NORMAL* e caracterizado pela rotina *normalState(),* é o estado em que o dispositivo normalmente se encontra. Esse é o estado inicial da máquina, e é o estado sucessor de todos os outros. Nesse estado não é executada nenhuma tarefa especial, apenas alguns procedimentos básicos para cálculo de transição de estados. Esses procedimentos são: accelRead(), updateScratch(), CheckTimers(), checkLongPress() e checkLongPressTouch().

4.3.9 Estado de Emergência

O estado de emergência, definido por *STATE_SOS* e caracterizado pela rotina *sosState(),* é o estado que indica que ocorreu uma emergência. Esse estado é alcançado quando um dos seguintes casos ocorrem: é detectada uma queda do usuário (modo automático) ou é detectado o pressionamento por 3 segundos do botão de emergência (modo manual). Para sair desse estado, deve-se pressionar novamente o botão de emergência, por 6 segundos.

Nesse estado o microcontrolador apenas é responsável por acionar o *buzzer* e piscar a cor azul do *LED RGB,* mantendo a cor vermelha. Essas duas ações são executadas nas linhas de código abaixo:

```
void sosState(){
```

```
  checkLongPress();
  if(quarterSecond%2) {Bean.setLed(255,0,255); } //liga o led azul
  else                {Bean.setLed(255,0,0); }   //desliga led azul
  digitalWrite(LED_PORT,HIGH);                   //aciona o buzzer
      }
```

O *LED* pisca em diferentes cores pois a variável *quarterSecond* é continuamente incrementada, a cada 250[ms] aproximadamente. Isso é feito utilizando interrupções por timer, o funcionamento detalhado será explicado futuramente neste documento na seção Interrupções de Transição de Pino e Aquisição dos Dados dos Sensores. Como um número par é sempre seguido de um número ímpar, e vice-versa, a expressão *quarterSecond%2* sempre resultará intercaladamente em *true* e *false.*

A função *Bean.setLed(r,g,b)* é uma função padrão da biblioteca do *Bean*, assim como todas da classe *Bean*. Essa função faz com que o *MCU* comunique o *SoC* para configurar o *LED RGB* para brilhar nas cores definidas pelos argumentos que, sequencialmente, indicam a intensidade de vermelho, verde e azul. Esses argumentos são do tipo *uint8_t*, que define um valor de 8 bits entre 0 e 255. Quando especificado 0 para os três parâmetros, o *LED* é apagado e esse valor pode ser incrementado até 255, que define o brilho máximo da respectiva cor.

Também pode ser observado na rotina *sosState(),* que apenas a função que checa o pressionamento do botão de emergência é chamada (*checkLongPress*()), ou seja, o botão *touch* será ignorado enquanto o sistema permanecer nesse estado e assim é impossível a transição para o estado de leitura de batimentos cardíacos neste momento. Todas as outras funções básicas são executadas.

A partir da leitura do código acima, aparentemente não é executada nenhuma rotina que notificará o aplicativo *Android* do estado de emergência. Isso ocorre porque a notificação é feita durante a transição do estado, e não no interior do estado. Isso é feito atribuindo o valor ‘1’ (*True,* Verdadeiro) para a variavel global “*st_emergency*” sempre que o valor da variável *State* tiver seu valor atribuído para *STATE_SOS.* A variável *st_emergency*, que indica que a *FSM* está no Estado de Emergência, é continuamente enviada para o dispositivo *Android*, mantendo seu valor sempre sincronizado nos dois ambientes, como será demonstrado adiante.

```
[…]
```

```
st_emergency = 1;
State = STATE_SOS;
[…]
```

4.3.10 Estado de medição de batimentos cardíacos

O estado de medição de batimentos cardíacos é definido por *STATE_SOS* e tratado pela rotina *hrtbitState()*. Esse estado é ativado por ação do usuário, e pode ser alcançado de duas maneiras: a primeira mantendo “pressionado” o *touch sensor* por 3 segundos e a segunda por meio do aplicativo *Android*. Para sair desse estado, deve-se “pressionar” o *touch sensor* por 6 segundos.

Esse estado é um estado orientado por interrupções, isto é, a principal lógica implementada no estado se encontra dentro de uma interrupção. Interrupções são rotinas especiais que são chamas quando algum certo evento ocorre. Essa rotina interrompe a execução normal do código, quebrando seu fluxo natural e é executada imediatamente. A interrupção que governa a lógica desse estado é a interrupção *TIMER2_COMPA_vect* que é executada quando um *timer* específico, no caso o *TIMER2* tem seu valor igual a um valor pré-definido, o que explica o termo "*COMPA",* de comparação, no nome da interrupção. Quando o valor do *timer* é igual ao pré-definido, ocorre um *Match*. Normalmente, tenta-se executar o mínimo possível em interrupções, para evitar o mal funcionamento de outras interrupções além de pausas muito longas na execução normal do código (OSHANA e KRAELING, 2013), entretanto, nesta interrupção isso não se trata de um problema, uma vez que no estado de aquisição de batimentos cardíacos não há processamento que demande muito tempo no código principal e, como explicado anteriormente, a *FSM* não permite alterações entre estados sem passar pelo Estado Normal.

Para a configuração da interrupção, foi criada a rotina *hrtBeatInterruptSetup()* que tem seu corpo mostrado no código abaixo e é chamada na rotina *setup()* do *firmware*, definida no início deste capítulo. Essa rotina é responsável pela configuração do *TIMER2* utilizado e ativação da interrupção por comparação deste temporizador.

```
void setup(){
[…]
   hrtBeatInterruptSetup();
[…]
}

[…]

void hrtBeatInterruptSetup(){
        TCCR2A = 0x02;
        TCCR2B = 0x06;
        OCR2A = 0X7C;
        TIMSK2 = 0x02;
        sei();
      }
```

Os termos *TCCR2A, TCCR2B, OCR2A* e *TIMSK2* são registradores de configuração que têm seus detalhes explicados no manual do usuário do microcontrolador *ATmega 328p*. Suas configurações e significados serão sumarizados nas tabelas a seguir. Registradores de configuração deste microcontrolador têm 8bits e a configuração é feita escrevendo em bits específicos. Algumas vezes, um conjunto de bits do mesmo ou de diferentes registradores podem, juntos, fazer parte da configuração do mesmo recurso.

Tabela 2: Configuração do registrador TCCR2A

TCCR2A	**Bit 7**	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
Nome do bit	COM2A1	COM2A0	COM2B1	COM2B0			WGM21	WGM20

Significado	Configura saída em uma porta no evento de *match.* Modo normal, que não gera saída, configurado.		Configura saída em outra porta no evento de *match.* Modo normal, que não gera saída, configurado.		Não utilizado	Não utilizado	Justamente com o WGM22 encontrado no registrador TCCR2B, configura o modo de operação CTC.	
Valor configurado: 0x02	0	0	0	0	0	0	1	0

Tabela 3: Configuração do registrador TCCR2B

TCCR2B	**Bit 7**	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
Nome do bit	FOC2A	FOC2B			WGM22	CS22	CS21	CS20
Significado	Configura saída em uma porta no evento de *match.* Modo normal, que não gera saída, configurado.		Não utilizado	Não utilizado	Configura, juntamente com o WGM21 e WGM20 do registrador TCCR2A o modo de operação CTC.	Define a fonte de *clock* (*ClockSource)*. Configuração de valor '2' define a fonte como o *clk_v/256* onde *clk_y* é o *clock* do microcontrolador de 8MHz resultando numa fonte de *clock* de 31.25kHz para o *timer*.		
Valor configurado: 0x06	0	0	0	0	0	1	1	0

Tabela 4: Configuração da tabela OCR2A

OCR2A	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Nome do bit								
Significado	O valor do OCR2A configura o valor que, quando comparado com o valor atual do contador (regido pelo *clock* configurado com os bits CSxx do TCCR2B) gerará a interrupção. O valor 0x7C hexadecimal configurado corresponde ao valor 124 decimal.							
Valor configurado 0x7C	0	1	1	1	1	1	0	0

Tabela 5: Configuração do TCCR2B

TCCR2B	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Nome do bit						OCIE2B	OCIE2A	TOIE2
Significado	Não utilizado	Não utilizado	Não utilizado	Não utilizado	Não utilizado	Ativa a geração de uma interrupção quando o *match* do registrador OCR2x correspondente ocorre. Apenas configurado a geração para o OCR2A.		Ativa geração de interrupção no overflow. Configurado para não ocorrer.
Valor configurado: 0x02	0	0	0	0	0	0	1	0

Portanto, resumindo o funcionamento gerido pelos registradores: o registrador TCCR2B configura que ocorrerá uma interrupção no *match* com o valor do registrador OCR2A. Além disso, o TCCR2B configura uma fonte de *clock* de 31.25kHz para o incremento do *timer*

e, finalmente, os bits WGM dos registradores TCCR2A e TCCR2B configuram o modo de operação para CTC, que significa *Clear on Time Compare* (limpar na comparação do *timer),* que faz que o valor do *timer* seja resetado para 0 sempre que ocorrer o *match.*

Para calcular o valor correto para o registrador ORC2A, foi utilizada a lógica abaixo.

A rotina de interrupção, é responsável pela leitura da saída do sensor de batimento cardíaco, ou seja, pela amostragem desse sinal analógico. Portanto, para definir uma frequência ideal de amostragem, foi levada em consideração a condição de frequência de amostragem mínima de Nyquist, que é dada pela equação:

$$fa_{minima} = 2*f_{maxsinal}$$

A $f_{maxsinal}$ corresponde à frequência máxima de batimento cardíaco, que é:

$$A\ f_{maxsinal} = 220bpm = 3.67Hz$$

$$Portanto,\ a\ fa_{minima} = 3.67Hz * 2 = \mathbf{7.33Hz}$$

Esse valor corresponde à frequência mínima de interrupções para obter-se uma amostragem livre de serrilhamento. No entando, não é necessario se prender a esse valor, quanto maior a frequência de amostragem, melhor será a digitalização do sinal. Por esse motivo, foi escolhido um valor 34 vezes superior: 250Hz, que não somente atende ao teorema de Nyquist, como vai proporcionar um sinal digital muito mais fiel ao original, sem causar muitos problemas de execução. Além disso, essa frequência de interrupções também será utilizada para o cálculo do pressionamento dos botões por 3[s] ou 6[s], o que será detalhado mais adiante nesta seção. Portanto, o valor correto para o registrador OCR2A é:

$$f_{interrupção} = f_{recebida}/val\_ocr2a$$

$$250 = 31.25k/val\_ocr2a$$

$$\mathbf{val\_ocr2a = 124}$$

$f_{recebida}$ corresponde à frequência de incremento do *timer.*

A rotina de cálculo de batimentos por segundo, que é executada dentro da interrupção, é uma modificação da indicada pelo fabricante do sensor de batimentos

cardíacos utilizado SENS1574 (WORLD FAMOUS ELECTRONICS LLC) e pode ser vista com mais detalhes na referência. Os principais pontos serão abordados a seguir.

Primeiramente, externo à interrupção e dentro da função *hrtbitState()* existe a estrutura:

```
void hrtbitState(){
[...]
            if (QS == true){ QS = false; }
[...]
}
```

Em que *QS* é uma variável que indica que foi detectado um batimento. Essa variável precisa ser limpa continuamente a cada execução do *loop()* e por isso se encontra na rotina do estado, não na interrupção.

Dentro da interrupção, há a condição de apenas executar o código de aquisição de batimentos cardíacos se o estado da *FSM* for o de batimentos cardíacos, como pode ser visto no código da próxima página.

A primeira parte do código busca os pontos mais altos e baixos de uma curva, com o objetivo de reconhecer batimentos cardíacos. Observando a Figura 11, do Capítulo 1, é possível observar que a partir de pontos altos e baixos da curva é possível fazer esse reconhecimento.

```
ISR(TIMER2_COMPA_vect){
[…]

  if(State == STATE_HRTBIT)
  {
    cli();
    Signal = analogRead(HRTBIT_PORT);          //realiza a leitura do sensor
    sampleCounter += 2;
    int N = sampleCounter - lastBeatTime;//calcula o tempo entre as leituras

//  Procura o ponto mais baixo da curva
    if(Signal < thresh && N > (IBI/5)*3){
      if (Signal < T){
        T = Signal;
      }
    }

  //procura o ponto mais alto da curva
    if(Signal > thresh && Signal > P){
      P = Signal;
    }


    if (N > 250){ //Evita ruído de alta frequência
```

```
        //detecta um batimento
    if ( (Signal > thresh) && (Pulse == false) && (N > (IBI/5)*3) ){
      Pulse = true;
      digitalWrite(LED_PORT,HIGH);
      IBI = sampleCounter - lastBeatTime;
      lastBeatTime = sampleCounter;
                  […]

      }
```

Um ciclo cardíaco possui na verdade dois batimentos: o maior é a sístole e o menor a diástole (NAKAJIMA, TAMURA e H.MIIKE, 1994). Para o menor, sístole, não ser confundido com um novo batimento, é adicionada a condição *(N > (IBI/5)*3)* no *if* de detecção de batimentos. Essa condição evita o reconhecimento de um novo batimento cardíaco em um curto espaço de tempo relativo aos batimentos já calculados. De maneira similar, a condição *(N > 250)* evita ruídos de alta frequência, ignorando "batimentos" muito próximos temporalmente (ou seja, funciona como um Filtro Passa Baixa). A variável *IBI* é calculada como o tempo entre os dois últimos batimentos, e *N* representa a diferença entre o tempo "atual" e tempo em que foi detectado o último batimento.

Em seguida, a rotina calcula a frequência do pulso cardíaco, além de recalcular valores limites (*threshold)* para ser reconhecido o batimento. Também, ao reconhecer que o sinal está declinando, apontando a queda da curva *Fotopletismografia*, os dados necessários são resetados para que se inicie um novo reconhecimento de pulsação.

```
                  […]
    //media de tempo dos ultimos 10 batimentos cardiacos.
      for(int i=0; i<=8; i++){
        rate[i] = rate[i+1];
        runningTotal += rate[i];
      }
      rate[9] = IBI;
      runningTotal += rate[9];
      runningTotal /= 10;


          // Tempo de 2 minutos em microsegundos dividido pela media de tempo
     //dos ultimos 10 batimentos da quantidade de batimentos por minuto
      BPM = 120000/runningTotal;
      toScratchBPM = (uint8_t) BPM;
      QS = true;
                    […]
```

Também é possível observar a expressão *toScratchBPM = (uint8_t) BPM em* que a variável *toScratchBPM,* que será utilizada para fazer a transmissão do dado da frequência

cardíaca para o aplicativo *Android* é atribuída com esse valor. Detalhes de como a transmissão é feita serão explanados na seção abaixo Atualização das Bluetooth *Characteristics*.

4.3.11 Interrupções de Transição de Pino e Aquisição dos Dados dos Sensores

A leitura dos dados do sensor de choque, do botão e do sensor de toque é feita utilizando interrupções. Como o ATmega328p tem suporte para apenas 2 interrupções por pinos, foi necessário utilizar a interrupção por porta, chamada *portchange*. Essa interrupção é executada sempre que algum dos pinos que compõem uma porta é alterado.

Para ativar essa interrupção, também chamada *PCINT0_vect*, é necessário configurar o registrador PCICR, selecionando a interrupção da porta0 (também chamada de *PORTB*), e do registrador PCMSK0, que configura quais pinos da porta0 são capazes de engatilhar a interrupção. O funcionamento detalhado dos registradores é mostrado no manual do microcontrolador (ATMEL, 2015). Os pinos pertencentes a porta0 são D9, D8, D7 e D6 do *LightBlue Bean+.*

A configuração desses dois registradores é feita na rotina *setup(),* por meio do código abaixo:

```
[…]
Setup(){
  […]
  PCICR |= (1<<0);
  PCMSK0 |= (0x3C);
  […]
}
[…]
```

Os três periféricos foram conectados a pinos dessa porta, conforme a tabela abaixo, e esses três pinos foram os habilitados para poderem engatilhar a interrupção por porta.

Tabela 6: Pinos detectados pela interrupção de mudança de estado.

Pino	Periférico conectado
D6	Sensor de choque

D8	Sensor de toque
D9	Botão físico

Como a mesma interrupção é executada quando qualquer um dos três pinos troca de estado, é necessário fazer o reconhecimento de qual deles foi alterado por s*oftware*. Para isso, sempre que houver uma interrupção é feita uma comparação entre o estado da porta0 com o estado dessa porta na última execução da interrupção (pré-armazenado em uma variável global *last_PINB*). Naturalmente, o bit que estiver diferente representará o pino que foi alterado e que necessita ser tratado. Essa comparação é feita utilizando a operação binaria ou-exclusivo*,* como pode ser visto no código abaixo.

```
ISR(PCINT0_vect) {
  uint8_t changed_bits;
  changed_bits = PINB ^ last_PINB; //compara os valores antigos com novos
  last_PINB = PINB; //salva o valor para ser comparado na prox. execução
[…]
if (changed_bits & (1 << PINB5)) //BUTTON //se o valor do pino do botão mudou
  {

    readButtom = digitalRead(BUTTON_PORT);

    if(readButtom == 1) //READBUTTON 1 = NOT PRESSING IT
    {
       […]
    }
    else
    {
       [...]
    }
  }
}
```

Essa estrutura é utilizada nos 3 sensores conectados à porta.

4.3.12 Temporizador nos botões

Para realizar a temporização do botão e do sensor de toque, isto é, reconhecer o pressionamento por 3 ou 6 segundos consecutivos, foi aproveitada a interrupção já implementada do *TIMER2* apresentada na seção Estado de medição de batimentos cardíacos.

Esta interrupção, como foi explicado anteriormente, acontece a cada 4ms. Durante a execução dessa interrupção, uma variável chamada *counting4ms* é incrementada, portanto,

essa variável conta a quantidade de ‘4ms’ executados. Quando *counting4ms* chega na contagem de 250, totalizando a contagem de 1 segundo, seu valor é restaurado para 0 e uma nova variável, chamada *oneSecond,* é incrementada. Essa variável expressa a quantidade de segundos passados e ela é a utilizada na temporização dos 3 e 6 segundos.

```
ISR(TIMER2_COMPA_vect){
static uint8_t counting4ms = 0; //variavel estática é compartilhada com todas as
                                //instancias da rotina.
  counting4ms++;
  if(counting4ms==250){
    counting4ms=0;
    oneSecond++; }
[…]
}
```

Após a implementação dessa estrutura, as funções básicas para checagem de botão *checkLongPress* e *checkLongPressTouch* precisam apenas checar a diferença do valor da variável *OneSecond* no instante em que o botão começou a ser pressionado e no instante "atual" de execução do código, levando em consideração a possibilidade de ter ocorrido *overflow* nesse meio tempo. A amostragem do valor no início do pressionamento do botão (ou sensor) é feita na rotina de interrupção, enquanto a comparação e a amostragem do instante corrente são feitas quando *checkLongPress* ou *checkLongPressTouch* são chamadas. É importante entender que a variável *OneSecond* é continuamente atualizada pela interrupção do *timer2,* enquanto a *pressingTouchStartTime* (e correspondentes), é atualizada quando o botão é pressionado. É possível observar essas características no código abaixo.

```
ISR(PCINT0_vect) {
[…]
if (changed_bits & (1 << PINB4))  //TOUCH
  {
    readTouch = digitalRead(TOUCH_PORT);
    if(readTouch == 0){ //READTOUCH 0 = botão não pressionado
      releasingTouchTime = OneSecond;
    }
    else
    {
      updatedTouch = 1;
      handled_3secPushTouch = 0;
      handled_6secPushTouch = 0;
      pressingTouchStartTime = OneSecond;
    }

  }
[…]
}
```

```
void checkLongPressTouch(){
[…]
if( readTouch == 1 ){
     pressingStartTime = pressingTouchStartTime;
     delta_pressing = OneSecond - pressingStartTime;
  }
  […]
  if(delta_pressing>6 && delta_pressing<10  && !handled_3secPushTouch ){
    handled_3secPushTouch = 1;
           […]
  }
[…]
}
```

Foi adicionada a variável *handled_3secPushTouch* (e outras correspondentes para diferentes tempos e para o botão mecânico) como uma *flag* que indica que a devida ação já foi tomada, para evitar que o mesmo código seja executado repetidamente enquanto o *touch/botão* se matem pressionado.

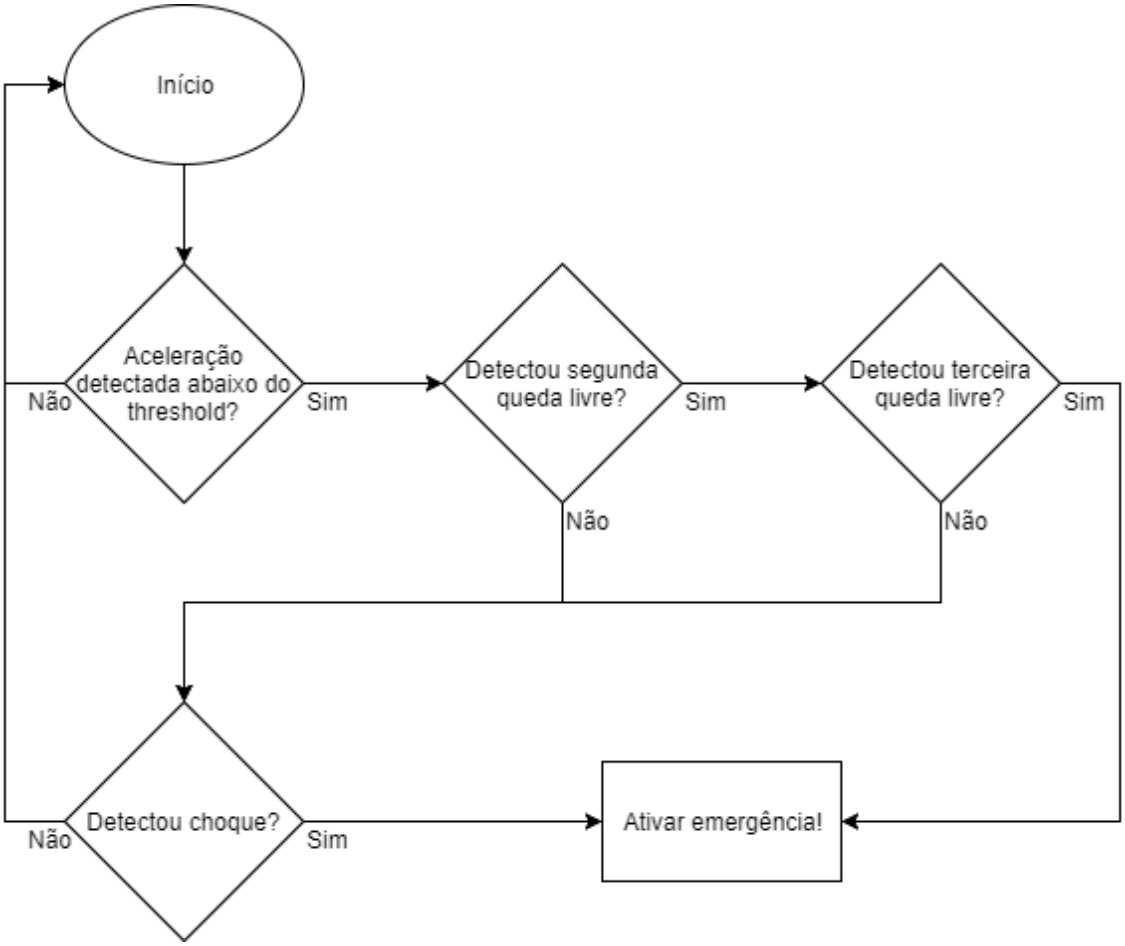
4.3.13 Função de Detecção de Queda

A checagem do acelerômetro é feita em todos os ciclos de execução da rotina *loop()* por meio da rotina *accelRead()*. Essa rotina reconhece momentos de queda livre do dispositivo, que podem ou não significar uma queda do usuário. Ela acusa que houve de fato uma queda do usuário de duas diferentes maneiras: a primeira é com diversas medidas consecutivas que apontam a queda livre do dispositivo; a segunda é medindo uma queda livre do dispositivo e um impacto.

Essas medidas são feitas por meio da rotina incluída na biblioteca do LightBlue Bean+ *Bean.getAcceleration()*, que retorna as acelerações medidas pelo acelerômetro e as armazena em uma estrutura (*struct*) que possui um campo para cada eixo cartesiano: x, y e z. Quando a soma dessas três medidas é menor que um certo valor de limite, obtido empiricamentedefinido no código por *FALLINGSIZE_threshold*, significa que houve um momento de queda livre do dispositivo.

Quando é detectado um momento de queda livre, são realizadas mais leituras do acelerômetro para verificar se essa detecção se tratou de somente um movimento brusco ou se de fato o usuário caiu, caso seja detectado um mínimo de leituras consecutivas que apontam queda livre, definido no código como *FALLINGTIMES_threshold*, houve uma queda do usuário e as rotinas correspondentes de tratamento são chamadas.

Figura 25: Fluxo de execução da rotina de detecção de queda.



FONTE: Autoria Própria.

Caso esse mínimo não tenha sido atingido, mas pelo menos um momento de queda livre tiver sido detectado, ainda é possível que tenha havido uma queda do usuário. Para esses casos, é verificado se houve impacto do dispositivo. O impacto é detectado pelo *shock sensor*, que tem seu valor copiado para a variável global *st_shock* na rotina de interrupção.

Ambas as condições, assim como a leitura de dados correspondente podem ser observadas no código a seguir:

```
void accelRead(){
  dvcAccel = Bean.getAcceleration();
  magnetude = abs(dvcAccel.xAxis) + abs(dvcAccel.yAxis) + abs(dvcAccel.zAxis);

  if(magnetude<FALLINGSIZE_threshold){ //detectado queda livre
    times_that_fell++;
    for(uint8_t i = 0; i<=(FALLINGTIMES_threshold-1) ; i++){
      dvcAccel = Bean.getAcceleration(); //novas leituras
      magnetude = abs(dvcAccel.xAxis) + abs(dvcAccel.yAxis) +
abs(dvcAccel.zAxis);

      if(magnetude<FALLINGSIZE_threshold) times_that_fell++;
```

```
            //primeira condição para detecção de queda
      if(times_that_fell==FALLINGTIMES_threshold){
        st_emergency = 1;
        State = STATE_SOS;                  ///Entra em modo emergência
        Bean.setLed(255,0,0);
      }
    }

          //segunda condição para detecção de quedas
    if(times_that_fell>0 && st_shock==1 ){
        st_emergency = 1;
        State = STATE_SOS;              //Entra em modo de emergência
        Bean.setLed(255,0,0);
      }
    times_that_fell = 0;
  }
}
```

4.3.14 Atualização das Bluetooth *Characteristics*

A verificação da necessidade de envio de dados por *Bluetooth* é feita em todos os ciclos da rotina *loop()* no interior da rotina *updateScratch()*, que é responsável por administrar os dados que estão sendo compartilhados. Caso eles tenham sido alterados, devem ser atualizados e enviados.

Isso é feito mantendo sempre uma cópia do último dado enviado e comparando com o atual estado dos dados. Se não forem iguais, os dados atualizados devem ser enviados para o aplicativo *Android*, o que é feito por meio da rotina disponível da biblioteca do *LBB+*,como demonstrado no código a seguir:

```
void updateScratch()
{
[…]
 if(oldScratchBuffer[0] != toScratchBuffer[0] || oldScratchBuffer[1] !=
toScratchBuffer[1])
  {
    Bean.setScratchData(2,toScratchBuffer,2);
  }
[…]
}
```

A função *Bean.setScratchData(uint8_t bank, uint8_t* data, uint8_t size)* escreve em um dos 5 bancos (*banks*) que define as *Bluetooth characteristics* do *Bean+* para ser lido pelo aplicativo *Android*. O parâmetro *bank* indica em qual banco o dado será escrito, o parâmetro *data* contém os dados a serem escritos e *size* é a quantidade de bytes a ser escrita. O dado enviado é o vetor *toScratchBuffer*, que possui dois bytes: o primeiro contendo os dados de queda, de emergência e de alguns sensores concatenados e o segundo contendo a última medição de batimentos cardíacos.

Tabela 7: Organização dos dados a serem enviados para as BLE Characteristics

Byte/Bit	7	6	5	4	3	2	1	0
1-Sinais	Emergency		HearthBeat					
2-hrtbit	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0

A estrutura que concatena as *flags* em um byte e a estrutura que concatena o bit de batimentos cardíacos e os que contêm as *flags* é mostrada no código abaixo.

```
void updateScratch()
{
[…]
toScratchByteWrite = ((readShock & 0x01)<<1 | (st_fell & 0x01)<<2 |
                          (st_hrtbit & 0x01)<<5 | (st_emergency & 0x01)<<7);

[…]
  toScratchBuffer[0] = toScratchByteWrite;
  toScratchBuffer[1] = toScratchBPM;
[…]

      }
```

Além de ser enviar dados, essa rotina também é responsável por tratar o recebimento de dados. Isso é feito lendo o banco de memórias e checando os campos que devem ser escritos pelo aplicativo *Android*. A leitura dos dados é feita utilizando uma função simples, que retorna uma estrutura de vários *bytes* enfileirados. A leitura e, sequencialmente o tratamento do dado é exibido no código abaixo.

```
ScratchData toScratchRead;
[…]
toScratchRead = Bean.readScratchData(2); //ler do banco 2
scratchByte = toScratchRead.data[0];
[…]
if(scratchByte & 0x4){
      activateBuzzer2sec();
```

```
      Bean.setLedBlue(255);
    }else
    {
      Bean.setLedBlue(0);
    }
```

Neste código é feito o tratamento da informação que ativa o modo de leitura de batimentos cardíacos a partir do aplicativo *Android*.

4.4 IMPLEMENTAÇÃO *ANDROID*

A aplicação *Android* foi desenvolvida utilizando o *Android Studio* e testada em um aparelho Samsung Galaxy s8 rodando *Android* 6.0. O desenvolvimento da aplicação teve foco em funcionalidades e por esse motivo não serão expostos detalhes do desenvolvimento da interface de usuário.

Primeiramente, é necessária a compreensão superficial de como um projeto *Android* é organizado. Existem arquivos *.java* que determinam as lógicas implementadas pela aplicação. Existem arquivos *.xml* que determinam como serão desenhados os elementos de uma aplicação. Existem arquivos *.gradle* que determinam *scripts* que regem a compilação do código e, finalmente, existe o arquivo *AndroidManifest.xml*, que determina configurações básicas para a execução do aplicativo.

Os arquivos *.xml* cosméticos são encontrados no GitHub do projeto, e sua confecção foi feita com o uso dos recursos visuais do *Android Studio*.

O arquivo *build.gradle*, que possui diretrizes de compilação, necessitou ser alterado para a importação da biblioteca do *Bean SDK*, que é a API criada pela *PunchThrough* para a utilização do *Bean.* A linha abaixo foi adicionada:

```
dependencies {
[…]
    compile 'com.punchthrough.bean.sdk:sdk:2.1.1'
[…]
}
```

Também é necessária a edição do arquivo *AndroidManifest.xml*, pois este arquivo controla as permissões do Sistema que o aplicativo pode ter quando executado. As permissões que necessitam ser adicionadas são as de leitura do estado do telefone, de recebimento e

envio de mensagens SMS, de Bluetooth, além de permissão de localização, necessária para a utilização do *Bean SDK*. As linhas abaixo são as que devem ser adicionadas

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.gobbi.tccapp">
    <uses-permission android:name="android.permission.READ_PHONE_STATE" />
    <uses-permission android:name="android.permission.SEND_SMS" />
    <uses-permission android:name="android.permission.RECEIVE_SMS" />
    <uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
    <uses-permission android:name="android.permission.BLUETOOTH" />

    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
</manifest>
```

Essas linhas indicam que o aplicativo pode requisitar essas permissões, porém ainda é necessária a aprovação futura do usuário quando o aplicativo é executado pela primeira vez.

Para compreender a implementação do aplicativo, é necessário compreender o conceito de c*allback. Callbacks* são funções utilizadas por funções assíncronas, ou seja, que não têm seu resultado no momento do fim de sua execução. Ao invés disso, é necessário esperar algum processamento (ou resposta) que chegará após um tempo indeterminado. Quando o resultado estiver disponível, a função de c*allback* é chamada.

O arquivo *MainActivity.java* define a lógica que o aplicativo executará quando for aberto. Nesse arquivo ainda é necessário a importação de bibliotecas pois o arquivo *.gradle* apenas indica quais bibliotecas serão disponíveis para a importação, ao invés de realmente importá-las.

A rotina *onCreate* define a execução que será feita quando o aplicativo é aberto, e é análogo à um *main()* da programação em C. Nesta rotina é feito, primeiramente, a requisição das permissões necessárias para o usuário, por meio de rotinas como a mostrada abaixo.

```
ActivityCompat.requestPermissions(this,
                      New String[] {Manifest.permission.ACCESS_FINE_LOCATION},
                      8);
```

Para fazer a busca de dispositivos, é necessária a declaração de uma variável do tipo *BeanDiscoveryListener* definido pela *BeanSDK.* Essa variável define uma classe que administra a busca por novos dispositivos *LightBlue Bean+*. Nesta classe, existem funções de *callback,* que serão assincronamente chamadas quando certos eventos ocorrem. Exemplos dessas *callbacks* são: *OnDiscoveryComplete()* e *OnBeanDiscovered().* Para definir, portanto, o que deve

ser executado quando um novo *Bean* é descoberto, deve-se sobrescrever a função *onBeanDiscovered()*, como foi feito no código abaixo.

```
final BeanDiscoveryListener listener = new BeanDiscoveryListener() {
            @Override
            public void onBeanDiscovered(Bean bean, int rssi) {
                beans.add(bean);}
                    […]
               }
```

O código acima é executado sempre que um *Bean* é encontrado e adiciona o dispositivo para uma lista de dispositivos encontrados chamada de *bean*.

Para começar a busca por dispositivos, é necessário executar uma função que notifica a classe *BeanDiscoveryListener* como é feito abaixo.

```
 button1.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                BeanManager.getInstance().startDiscovery(listener);
            }
        });
```

A administração de um dispositivo conectado é feita por uma classe similar, chamada *BeanListener*. Essa classe tem o funcionamento ditado também sobrescrevendo funções de *callback*. As mais importantes são: *onConnected() e onScratchValueChanged().*

A primeira serve apenas para exibir mensagens com informações sobre o dispositivo e é mostrada a seguir:

```
final BeanListener beanListener = new BeanListener() {

    @Override
    public void onConnected() {
        currentBean = beans.get(0);
        bean.readDeviceInfo(new Callback<DeviceInfo>() {
            @Override
            public void onResult(final DeviceInfo deviceInfo) {
                runOnUiThread(new Runnable() {
                    @Override
                    public void run() {
                        mainConsole.addInfo("Connected to device!");
                        mainConsole.addInfo(deviceInfo.hardwareVersion());
                        mainConsole.addInfo(deviceInfo.firmwareVersion());
                        mainConsole.addInfo(deviceInfo.softwareVersion());

                    }
                });

            }
        });
[...]
    }
```

A sintaxe confusa apresentada pela função ocorre devido a particularidades da linguagem. Basicamente, estão sendo definidos objetos durante a própria declaração dos mesmos.

A função *onScratchValueChanged()* é chave para o sistema. Ela é a rotina que é automaticamente chamada quando o valor das *charachteristcs* é alterado pelo *wearable*. Nela, é feito o caminho inverso mostrado na seção Atualização das Bluetooth *Characteristics*. Os dados chegam, portanto, em bytes que devem ter seus dados extraídos. Depois da extração, deve ser feita uma comparação com os dados antigos, para averiguar quais sofreram alteração. Esse processo é exibido no código a seguir:

```
final BeanListener beanListener = new BeanListener() {
    […]
    @Override
    public void onScratchValueChanged(ScratchBank bank, byte[] value) {
        int temp;

        scratchData[0] = value[0];
        scratchData[1] = value[1];

        temp = value[1] & 0xFF; //transforma signed em unsigned

         […]
        int st_emergency = (scratchData[0] & 1<<7) >> 7;
         […]
        int st_emergencyOld = (scratchDataOld[0] & 1<<7) >> 7;
         […]

        if(st_emergency == 1 && st_emergencyOld == 0){ //entrou em emergencia
            […]
            String msg = name + ", preciso de ajuda! Me ligue";
             […]
            sendSMS(phoneNumber,msg);
        }else if(st_emergency == 0 && st_emergencyOld == 1){ //saiu de
emergencia
            […]
            String msg = name + ", ja estou bem.";
             […]
            sendSMS(phoneNumber,msg);
        }else{
            //nothing is done
        }

        scratchDataOld[0] = scratchData[0]; //mantem copias antigas
        scratchDataOld[1] = scratchData[1];

    }
```

Finalmente, é feito o envio da mensagem de texto. O envio e recebimento da mensagem de texto é administrado pelo mesmo objeto em todos os aplicativos *Android*. Por isso, não se deve declarar um novo administrador de mensagem de texto e sim obter o administrador padrão. Isso é feito com a chamada *SmsMenager sms = Sms.Manager.getDefault()*. Deve-se então enviar um pedido para esse administrador com a intenção de enviar a mensagem, com número e corpo definido. Esse procedimento pode ser visto no código da rotina abaixo:

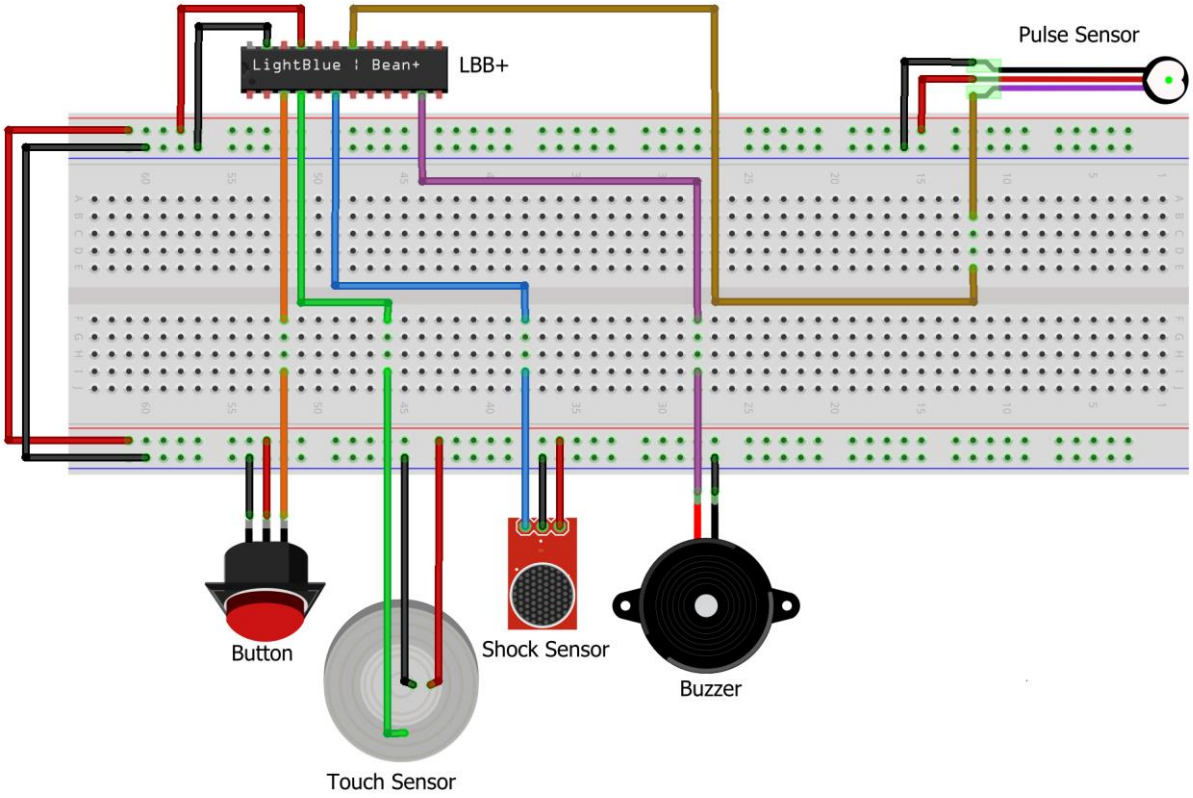
```
    private void sendSMS(String phoneNumber, String message) {

        SmsManager sms = SmsManager.getDefault();
        PendingIntent sentPI;
        String SENT = "SMS_SENT";
        sentPI = PendingIntent.getBroadcast(this, 0,new Intent(SENT), 0);
        sms.sendTextMessage(phoneNumber, null, message, sentPI, null);
    }
```

5 RESULTADOS

Nesta seção, serão demostrados os resultados obtidos neste projeto. Os diferentes recursos do funcionamento do protótipo serão exibidos sucintamente a seguir. Uma apresentação do protótipo criado é mostrada na Figura 26.

Figura 26: Ilustração da montagem final do projeto.



FONTE: Autoria Própria.

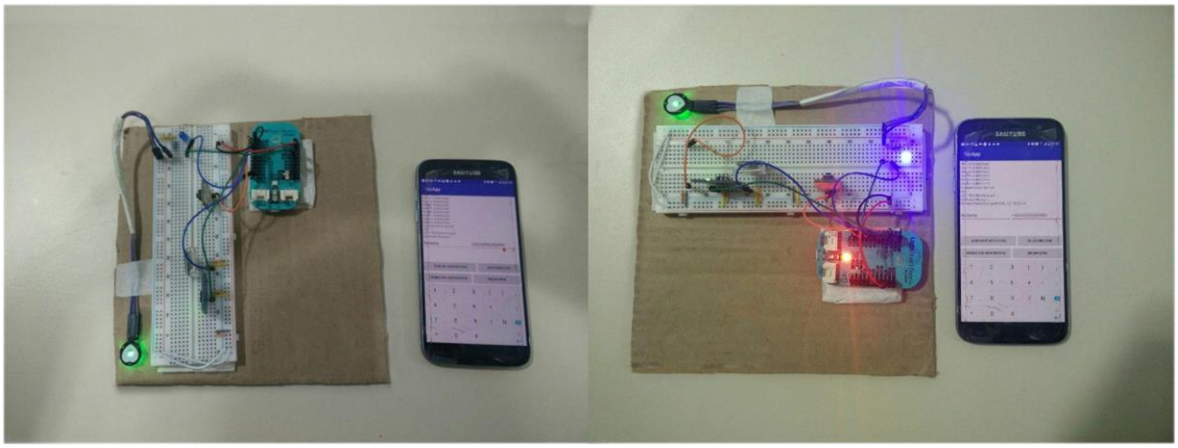
Foi determinada a autonomia energética de 18 horas. Para uma bateria de 600mAh, isso determina o gasto energético de 33.3[mAh] por hora. Esse valor é aceitável para um protótipo criado em *protoboard*, entretanto deve ser melhor em um produto final.

5.1 ATIVAÇÃO DO MODO DE EMERGÊNCIA POR QUEDA

A ativação da emergência por queda é funcional. O sistema detectou a queda, notificou por *BLE* para o aplicativo instalado no *smartphone*, que enviou automaticamente uma mensagem para um contato configurado.

As **Error! Reference source not found.** mostram o estado do dispositivo antes e depois da queda, e o aplicativo conectando ao dispositivo e reconhecendo a mudança de estado.

Figura 27: Teste da ativação por queda.



FONTE: Autoria Própria.

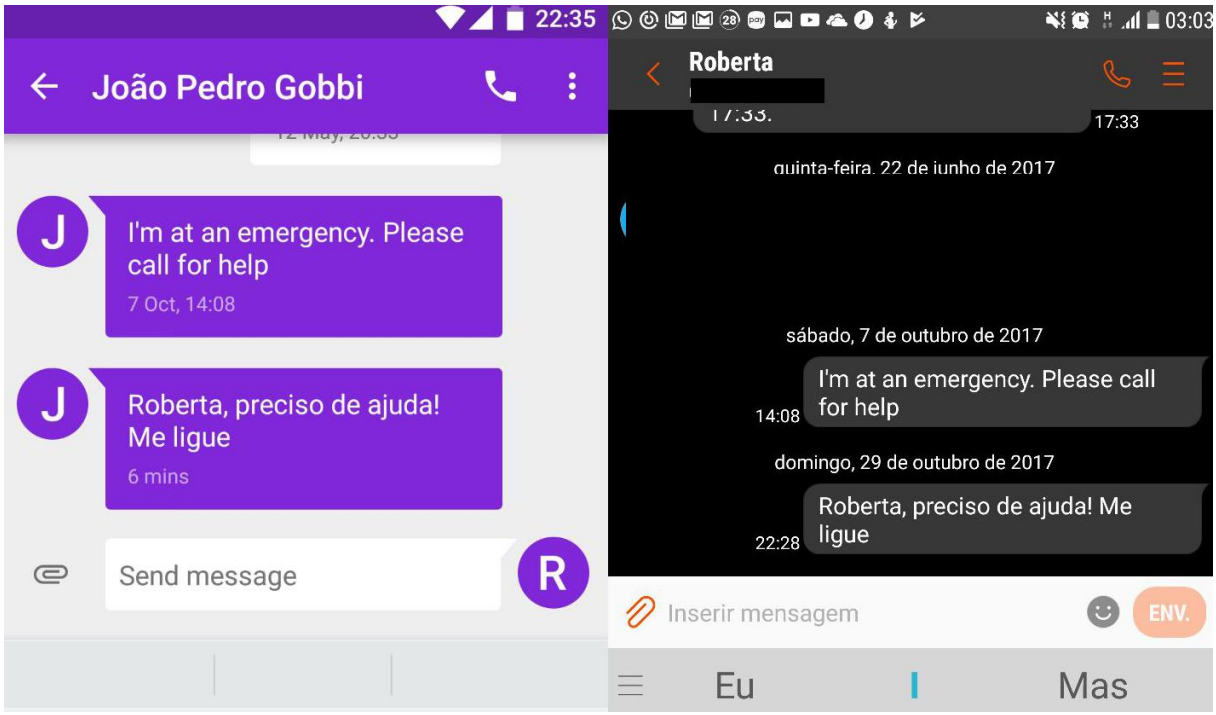
Para a validação do modo de emergência por queda, foram realizados testes em diversas circunstancias exibidos na tabela abaixo.

Tabela 8: Testes de detecção de queda.

Altura	Superfície	Resultado
10cm	Macia	Não Detectou Queda
30cm	Dura	Detectou Queda
60cm	Dura	Detectou Queda
120cm	Macia	Detectou Queda
170cm	Dura	Detectou Queda
170cm	Macia	Detectou Queda

A Figura 28 todo mostra a mensagem que foi enviada para o contato selecionado na configuração do aplicativo.

Figura 28: Captura de tela do smartphone que recebeu a mensagem SMS, a esquerda, e do que enviou o mensagem SMS, a direita.



FONTE: Autoria Própria.

5.2 ATIVAÇÃO DO MODO DE EMERGÊNCIA MANUAL

A ativação da emergência por meio do do botão é funcional. O sistema detectou o pressionamento do botão pelo tempo necessário e ativou o modo de emergência. Em seguida, o modo de emergência foi desativado de modo análogo. As mensagens de emergência e da saída do modo de emergência foram enviadas.

5.3 MEDIÇÃO DE BATIMENTOS CARDÍACOS

A medição de batimentos cardíacos é funcional. O sistema entra no estado de medição, mede corretamente e envia os dados para o *smartphone* para a exibição na tela. Na Figura 29, é possível observar as medições de batimento cardíaco sendo feitas e atualizadas em tempo

real. Nota-se que inicialmente há um período de adaptação, onde são reconhecidos valores equivocados (como 19, 51, 38, entre outros), porém seguidamente, os valores se estabilizam em uma medição correta (com valores em torno de 70).

Figura 29: Captura de tela do aplicativo desenvolvido recebendo dados de batimento cardíacos, que após um certo tempo de adaptação reconhece os valores corretos.

FONTE: Autoria Própria.

6 CONCLUSÃO

O projeto criado apresentou funcionamento adequado como esperado, além de alcançar os objetivos desejados.

Os conceitos intrínsecos de programação de sistemas embarcados foram extensamente utilizados neste projeto. A compreensão de boas práticas para a confecção do código foi gradual durante a execução do projeto, sendo necessária diversas vezes a alteração de códigos já funcionais, com o objetivo de melhorar robustez e qualidade.

O *Bluetooth Low Energy*, assim como o próprio *Bluetooth* clássico, é uma tecnologia utilizada diariamente, mas cujos conceitos ainda são pouco conhecidos por desenvolvedores. A implementação da comunicação *Bluetooth* foi surpreendentemente simples, sendo o maior desafio entender como utilizar o *BLE* de maneira eficiente e eficaz no projeto.

O *Android* foi outra tecnologia totalmente nova para o desenvolvedor deste projeto. A linguagem de programação tem seu maior desafio na compreensão dos diversos conceitos utilizados para escrever poucas linhas de código. Além desses conceitos da própria linguagem, existem muitos conceitos de programação orientada a objetos utilizados pela linguagem, o que causou mais uma dificuldade para o desenvolvedor.

Finalmente, o *LightBlue Bean+* é uma ótima ideia de placa de prototipação para *wearables* que utilizam *BLE*. Dito isso, o *Bean+* se mostrou decepcionante. Mesmo sendo um projeto ativo para a *PunchThrough,* foi observado abandonado, não recebendo atualizações básicas há um tempo considerável. Atualizações essas que seriam essenciais, pois grande parte das funcionalidades não se mostravam robustas, falhando inesperadamente. Além disso, na documentação, que à primeira vista parecia promissora, faltavam dados essenciais que apenas são necessários quando é preciso *debugar* códigos. Sendo assim, o desenvolvedor do projeto necessitou praticar diversas vezes “engenharia reversa” para obter informações úteis. Além disso, grande parte das funcionalidades, diversas vezes básicas, da família *Bean* foi implementada apenas para *IOS*, tendo a implementação para *Android* adiada indefinidamente. A comunidade quase nula da placa corroborou para os problemas causados pela falta de robustez e documentação.

Apesar dos problemas com a placa de desenvolvimento, tanto o planejamento quanto a execução do projeto foram um sucesso. Foram compreendidas, planejadas e executadas as partes necessárias para a criação de um protótipo funcional, que se assemelha consideravelmente a soluções existentes atualmente no mercado.

6.1 TRABALHOS FUTUROS

A execução deste projeto foi feita de forma modular para melhor receber melhorias em trabalhos futuros. Sendo assim, próximos passos para a melhoria do protótipo são:

- Aprimoramento do algoritmo de detecção de queda, pois ainda é um campo em que diversas pesquisas estão sendo feitas;
- Implementação de mais bio-sensores no *wearable*;
- Implementação de sensores capazes de reconhecer um ambiente perigoso para o idoso, detectando incêndios ou incidências fortes de raio UV;
- Recebimento e tratamento de mensagens SMS que realizam ações especificas no *wearable*;
- Envio e recebimento de mensagens de voz por meio do dispositivo *wearable*;
- Implementação e envio de geolocalização junto com a mensagem de emergência;
- Melhoria do aplicativo, melhorando sua robustez e criando uma interface gráficas agradável ao usuário;
- Criação de uma pulseira que capacite a utilização no pulso;
- Envio de informações do nível de bateria para o aplicativo Android;
- Transformar o projeto do protótipo em um projeto de produção em massa

REFERÊNCIAS

ADAFRUIT. Introduction to Bluetooth Low Energy, 2014. Disponivel em: <https://learn.adafruit.com/introduction-to-bluetooth-low-energy>. Acesso em: 15 out. 2017.

ANDREWS, J. R. **Co-verification of Hardware and Software for ARM SoC Design**. [S.l.]: Newnes, 2004.

ANDROID. Android Reference. **Android for Developers**, 2017. Disponivel em: <https://developer.android.com/index.html>. Acesso em: 30 nov. 2017.

ARDUINO. **Arduino**, 2017. Disponivel em: <https://www.arduino.cc/>. Acesso em: 23 nov. 2017.

AT&T. EverThere. **AT&T**, 2016. Disponivel em: <https://www.att.com/gen/press-room?pid=25140&cdvn=news&newsarticleid=37328>. Acesso em: 28 nov. 2017.

ATMEL. ATMEL 8BIT 328p USER MANUAL, 2015.

AVAGO TECHNOLOGIES. APDS-9008 Miniature Surface-Mount Ambient Light Photo Sensor.

BLUEPIXEL TECHNOLOGY LLP. BLE Scanner Playstore Page. **Play Store**, 2017. Disponivel em: <https://play.google.com/store/apps/details?id=com.macdom.ble.blescanner&hl=pt_BR>. Acesso em: 29 out. 2017.

BLUETOOTH SIG, 2017. Disponivel em: <https://www.bluetooth.com/what-is-bluetooth-technology/bluetooth-origin>. Acesso em: 16 set. 2017.

BOSCH. BMA Digital, triaxial acceleration sensor Data Sheet, 2011. Disponivel em: <http://www1.futureelectronics.com/doc/BOSCH/BMA250-0273141121.pdf>.

BRAY, J.; STURMAN, C. F. Bluetooth: Unifying the Telecommunications and Computing Industries, 2002. Disponivel em: <http://www.informit.com/articles/article.aspx?p=27591>. Acesso em: 16 set. 2017.

BROOKS, M. Falls Cause Most Accidental Deaths in Elderly Americans. **Medscape**, 2015. Disponivel em: <https://www.medscape.com/viewarticle/844322>. Acesso em: 27 nov. 2017.

CAREPREDICT, 2017. Disponivel em: <https://www.carepredict.com/assisted-living-memory-care/>. Acesso em: 28 out. 2017.

CIEL LIGHT. CL-SF687 DataSheet, 2012. Disponivel em: <http://www.ciellight.com/pdf/smdled/3528full/CL-SF687RGB.pdf>. Acesso em: 25 nov. 2017.

GREATCALL. GreatCall, 2017. Disponivel em: <https://www.greatcall.com/devices/lively-wearable-senior-activity-tracker?kbid=62750>. Acesso em: 28 out. 2017.

HELPCARE. HelpCare, 2017. Disponivel em: <http://www.helpcarebrasil.com.br/monitoramento-de-idosos>. Acesso em: 28 out. 2017.

IMAGE. **Wikipedia:** The Free Encyclopedia. Disponivel em: <http://commons.wikimedia.org/wiki/File:Fitbit_Charge_HR.jpg>. Acesso em: 16 set. 2017.

JOEL MURPHY. Pulse Sensor Amplified. **TAPR Open Hardware**, 2017. Disponivel em: <http://tapr.org/OHL>. Acesso em: 30 out. 2017.

LAMKIN, P., 2015. Disponivel em: <https://www.forbes.com/sites/paullamkin/2015/10/29/wearable-tech-market-to-treble-in-next-five-years/#1bde62b82c77>. Acesso em: 16 set. 2017.

MARWEDEL, P. **Embedded System Design**. [S.l.]: [s.n.], 2006.

MICROCHIP TECHNOLOGY, INC. Bluetooth Low Energy Channels, 2017. Disponivel em: <http://microchipdeveloper.com/wireless:ble-link-layer-channels#toc1>. Acesso em: 16 set. 2107.

NAKAJIMA, K.; TAMURA, T.; H.MIIKE. Monitoring of heart and respiratory rates by photoplethysmography using digital filtering technique, 1994.

OPEN HANDSET ALLIANCE. Platform Architecture. **Android for Developers**, 2017. Disponivel em: <https://developer.android.com/guide/platform/index.html>. Acesso em: 26 nov. 2017.

OSHANA, R.; KRAELING, M. **Software Engineering for Embedded Systems**. [S.l.]: [s.n.], 2013.

PRADO, J. Qual operadora tem a melhor cobertura do brasil? **tecnoblog**, 2017. Disponivel em: <https://tecnoblog.net/211938/qual-operadora-melhor-cobertura-sinal-4g-3g-2g/>. Acesso em: 04 dez. 2017.

PUNCH THROUGH DESIGN. Prototype to Production. **Bean**, 2017b. Disponivel em: <https://punchthrough.com/bean/docs/guides/everything-else/proto-to-prod/>. Acesso em: 22 out. 2017.

PUNCHTHROUGH DESIGN. PunchThrough About. **PunchThrough**, 2017a. Disponivel em: <https://punchthrough.com/>. Acesso em: 22 out. 2017.

PUNCHTHROUGH DESIGN. Accelerometers. **Bean**, 2017c. Disponivel em: <https://punchthrough.com/bean/docs/guides/features/accelerometer/>.

PUNCHTHROUGH DESIGN. Technical Specs. **Bean**, 2017c. Disponivel em: <https://punchthrough.com/bean/docs/guides/getting-started/tech-specs/>. Acesso em: 22 out. 2017.

TEHRANI, K.; MICHAEL, A., 2014. Disponivel em: <http://www.wearabledevices.com/what-is-a-wearable-device/>. Acesso em: 16 set. 2017.

TONTOUCH. 1 KEY TOUCH PAD DETECTOR IC - TTP223-BA6, 2008. Disponivel em: <https://radiokot.ru/konkursCatDay2014/53/01.pdf>. Acesso em: 2017.

UDOO NEO. KY-031 Knock Sensor. **UDOO Neo Documentation**, 2017. Disponivel em: <https://www.udoo.org/docs-neo/Cookbook_Arduino_M4/Knock_sensor.html>. Acesso em: 2017 out. 15.

VAHID, F.; GIVAGIS, T. **Embedded System Deisgn:** A Unified Hardware/Software Approach. [S.l.]: [s.n.], 2001.

WARNE, W. Bluetooth Low Energy - It starts with Advertising. **Bluetooth**, 2017. Disponivel em: <https://blog.bluetooth.com/bluetooth-low-energy-it-starts-with-advertising>. Acesso em: 16 set. 2017.

WORLD FAMOUS ELECTRONICS LLC. Pulse Sensor GitHub. **Pulse Sensor**. Disponivel em: <https://github.com/WorldFamousElectronics>. Acesso em: 29 nov. 2017.

ZURIARRAIN, J. Android já é o sistema operacional mais usado do mundo. **El Pais Brasil**, 2017. Disponivel em: <https://brasil.elpais.com/brasil/2017/04/04/tecnologia/1491296467_396232.html>. Acesso em: 26 out. 2017.