UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ENGENHARIA DE SÃO CARLOS

Caynã Simões Ferraz e Natália Moura Aravéquia

Bilateralização e aperfeiçoamento de sistema de transdutores para auxílio à caracterização e avaliação da marcha de tetraplégicos e paraplégicos

São Carlos

2019

Caynã Simões Ferraz e Natália Moura Aravéquia

Bilateralização e aperfeiçoamento de sistema de transdutores para auxílio à caracterização e avaliação da marcha de tetraplégicos e paraplégicos

Monografia apresentada ao curso de Engenharia Elétrica com Ênfase em Eletrônica, da Escola de Engenharia de São Carlos da Universidade de São Paulo, como parte dos requisitos para obtenção do título de Engenheiro Eletricista.

Orientador: Prof. Dr. Alberto Cliquet Jr.

São Carlos

2019

Ficha catalográfica elaborada pela Biblioteca Prof. Dr. Sérgio Rodrigues
Fontes da EESC/USP

Ferraz, Caynã Simões
F381b Bilateralização e aperfeiçoamento de sistema de transdutores para auxílio à caracterização e avaliação da marcha de tetraplégicos e paraplégicos / Caynã Simões Ferraz, Natália Moura Aravéquia; orientador Alberto Cliquet Junior. -- São Carlos, 2019.

Monografia (Graduação em Engenharia Elétrica com Ênfase em Eletrônica) -- Escola de Engenharia de São Carlos da Universidade de São Paulo, 2018.

1. Mapeamento de pressão plantar. 2. Matriz de sensores. 3. Palmilha eletrônica. 4. Medical sensor 3000. 5. Bluetooth. 6. JAVA. I. Aravéquia, Natália Moura. II. Titulo.

Elaborado por Elena Luzia Palloni Gonçalves – CRB 8/4464

FOLHA DE APROVAÇÃO

Nome: Caynã Simões Ferraz

Título: "Bilateralização e aperfeiçoamento de sistema de transdutores para auxílio à caracterização e avaliação da marcha de tetraplégicos e paraplégicos"

Trabalho de Conclusão de Curso defendido e aprovado em 7 / 6 / 2019,

com NOTA 9,0 (nove, zero), pela Comissão Julgadora:

Prof. Titular Alberto Cliquet Júnior - Orientador - SEL/EESC/USP

Dr. Renato Varoto - Pós-doutorado/UNICAMP

Mestre Anelise Ventura Nascimento - Doutoranda - FMRP/EESC/IQSC

Coordenador da CoC-Engenharia Elétrica - EESC/USP:
Prof. Associado Rogério Andrade Flauzino

FOLHA DE APROVAÇÃO

Nome: Natália Moura Aravéquia

Título: "Bilateralização e aperfeiçoamento de sistema de transdutores para auxílio à caracterização e avaliação da marcha de tetraplégicos e paraplégicos"

Trabalho de Conclusão de Curso defendido e aprovado em 7/6/2019,

com NOTA 9,0 (nove, zero), **pela Comissão Julgadora:**

Prof. Titular Alberto Cliquet Júnior - Orientador - SEL/EESC/USP

Dr. Renato Varoto - Pós-doutorado/UNICAMP

Mestre Anelise Ventura Nascimento - Doutoranda - FMRP/EESC/IQSC

Coordenador da CoC-Engenharia Elétrica - EESC/USP:
Prof. Associado Rogério Andrade Flauzino

Este trabalho é dedicado a todas as pessoas
que irão, de alguma forma, fazer bom proveito
do sistema para melhorar a vida de pacientes
tetraplégicos, paraplégicos e outros que possam se beneficiar.

AGRADECIMENTOS

Eu, Caynã, dedico esse trabalho a todas as pessoas que me acompanharam na graduação, em especial meus familiares: Ofélia Ferraz, Itayrê Ferraz e Tauany Ferraz, e a Juliana Pereira, que me deram suporte e me apoiaram em toda a minha jornada. Agradeço, também, a oportunidade de poder trabalhar e desenvolver esse projeto com a Natália Aravéquia, que foi uma amiga leal e esteve comigo nos momentos altos e baixos do trabalho. Por último, mas não menos importante, quero agradecer aos meus amigos: Gabriel Baptista, Gabriel Melo, Gabriel Pascon, Vitor Merli, Cainã Figares, Leonardo Borsato, Renan de Lara, Aurélio Baptista, Filipe Bordon, Nicholas Totti e Lucas Alquati, que me acompanharam na minha graduação e me mantiveram motivado para enfrentar a dura e longa jornada até atingir o objetivo final da graduação: o diploma.

Eu, Natália, agradeço, em primeiro lugar, aos meus pais, Cacilda e José Antonio, por todo apoio e suporte que sempre tive, sem o qual eu não teria conseguido todas as minhas conquistas. Agradeço também à minha irmã Mariana, companheira durante todas as fases de minha vida. Um agradecimento especial ao Paulo Sérgio, que esteve ao meu lado me motivando e me inspirando a sempre seguir em frente. Também agradeço ao Caynã Ferraz pela grande parceria e amizade durante todo o desenvolvimento deste trabalho.

Finalmente, gostaríamos de agradecer ao Professor Dr. Alberto Cliquet Jr. e ao Dr. Renato Varoto por ter nos dado a oportunidade de trabalhar com um projeto tão desafiador e gratificante, o qual esperamos que possa ter um impacto positivo em pesquisas e desenvolvimentos com pacientes tetraplégicos e paraplégicos.

"É possível encontrar a felicidade mesmo nas horas mais sombrias,
basta se lembrar de procurar pela luz."
Alvo Dumbledore

RESUMO

FERRAZ, C. e ARAVÉQUIA, N. **Bilateralização e aperfeiçoamento de sistema de transdutores para auxílio à caracterização e avaliação da marcha de tetraplégicos e paraplégicos**. 2019. 130p. Monografia (Graduação em Engenharia Elétrica com ênfase em Eletrônica) - Escola de Engenharia de São Carlos da Universidade de São Paulo, São Carlos, 2019.

O objetivo deste projeto é a bilateralização de um sistema de palmilha que capta a distribuição de pressão plantar durante a marcha do indivíduo e permite a visualização da mesma na tela do computador (CRITTER, 2015). O sistema já existente possui tais funcionalidades para uma única palmilha, e será modificado para funcionar com duas, permitindo, assim, melhor visualização das pressões dos pés durante a marcha do indivíduo.

Para tanto, a aplicação foi modificada para um sistema com dois módulos ESP32, que são microcontroladores integrados com Bluetooth, sendo um para cada palmilha. Tais módulos se sincronizam através da comunicação master-slave, com uso de um cabo de 2,5m de comprimento, o suficiente para sair de uma placa de aquisição no calcanhar do indivíduo, subir a perna, passar pela cintura e descer a outra perna até encontrar a outra placa, permitindo que o indivíduo caminhe livremente com o sistema em funcionamento.

Os testes demonstraram que a sincronização de dados entre os microcontroladores master e slave estão em pleno funcionamento e já integradas com a aplicação Java, que permite a visualização dos dados de ambas palmilha simultaneamente na tela do computador. Já a placa de aquisição impressa e com componentes soldados, por questões do processo de manufatura disponível ser manual e menos preciso, demonstrou problemas na aquisição dos dados das palmilhas, que são detalhados na discussão dos testes e cuja causa e solução são inferidos.

Desta forma, a comunicação entre as placas de ambas palmilhas, a interface gráfica para visualização dos resultados e o projeto do circuito de aquisição do novo sistema foram finalizados e testados com sucesso. Para se ter o sistema completo e em pleno funcionamento para uso em pacientes, é necessário corrigir os erros de manufatura da placa ou utilizar processos de manufatura mais tecnológicos e precisos para criar uma nova.

Palavras-chave: Mapeamento de pressão plantar. Matriz de sensores. Palmilha eletrônica. Medical Sensor 3000. Bluetooth. Java.

ABSTRACT

FERRAZ, C. e ARAVÉQUIA, N. **Bilateralization and improvement of a transducer system to help in the characterization and evaluation of quadriplegic and paraplegic gait**. 2019. 130p. Monografia (Graduação em Engenharia Elétrica com ênfase em Eletrônica) - Escola de Engenharia de São Carlos da Universidade de São Paulo, São Carlos, 2019.

The goal of this project is the bilateralization of an insole system that captures the distribution of plantar pressure during the gait of an individual and allows the visualization of the same on the computer screen (CRITTER, 2015). The existing system has such functionalities for a single insole, and will be modified to work with two, thus allowing better visualization of the foot pressures during the individual's gait.

To do so, the application was modified to a system with two ESP32 modules, which are microcontrollers integrated with Bluetooth, one for each insole. These modules synchronize through master-slave communication, using a 2.5m long cable, enough to exit an acquisition board in the subject's heel, go up his leg, go through the waist and go down the other leg until it finds the other board, allowing the individual to walk freely with the system in operation.

The tests demonstrated that data synchronization between the master and slave microcontrollers are fully operational and already integrated with the Java application, which allows the data visualization of both insole sensors simultaneously on the computer screen. The acquisition board printed and with welded components, due to problems of the manufacturing process available being manual and less precise, showed problems in the acquisition of the insoles' data, which are detailed in the discussion of the tests and whose cause and solution are inferred.

In this way, the communication between the both insole sensors boards, the graphic interface for visualization of the results and the design of the acquisition circuit of the new system were finalized and tested successfully. In order to have the complete and fully functioning system for patient use, it is necessary to correct the manufacturing errors of the board or to use more technological and accurate manufacturing processes to create a new one.

Keywords: Mapping of plantar pressure. Sensor array. Electronic insole. Medical Sensor 3000. Bluetooth. Java.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

EESC	Escola de Engenharia de São Carlos
USP	Universidade de São Paulo
GPIO	General Purpose Input/Output
UART	Universal Asynchronous Receiver/Transmitter
SPI	Serial Peripheral Interface

LISTA DE SÍMBOLOS

Ω	Ohm
μ	Micro
ρ	Pico

SUMÁRIO

1 INTRODUÇÃO

A paraplegia e a tetraplegia são condições resultantes de uma Lesão da Medula Espinhal (LME), um dos mais graves acometimentos que pode afetar o ser humano. Tal lesão corresponde a qualquer dano às estruturas contidas no canal medular, podendo acarretar em alterações motoras, sensitivas, autonômicas e psicoafetivas (BRASIL, 2013).

Um dos principais aspectos prejudiciais da lesão medular é o indivíduo ficar impossibilitado de caminhar, podendo levar à perda de sua independência funcional. Portanto, tratamentos que visam ajudar na recuperação da marcha de paraplégicos e tetraplégicos são de extrema importância para a reintegração social dos pacientes.

Há diversas alterações que podem ocorrer após a lesão, como a atrofia muscular, que contribui para diversas complicações médicas como fraturas, trombose venosa profunda e diminuição da área média de seção transversal dos músculos localizados abaixo do nível da lesão (CLIQUET et al., 2009). Outra alteração apresentada é a extensa paralisia muscular, responsável pela diminuição dramática da massa óssea, aumentando ainda mais o risco de fraturas (CLIQUET et al., 2006).

Foi comprovado que o tratamento realizado por treinamento de marcha em esteira com estimulação elétrica neuromuscular possibilita aumento da taxa de formação óssea (CLIQUET et al., 2006) e ganho de área de seção transversal de músculos (CLIQUET et al., 2009). A avaliação e caracterização da distribuição de pressão entre a planta do pé do indivíduo e a superfície de apoio são de extrema importância para o controle da marcha.

Diversos dispositivos permitem a análise da distribuição de pressão plantar, e se classificam em dois tipos de sistema: de plataforma e de palmilha de calçado. Sistemas de plataforma são construídos a partir de uma matriz plana e rígida de sensores de pressão, colados no chão para permitir a marcha normal. Podem ser usados tanto para estudos estáticos ou dinâmicos, mas geralmente se restringem a estudos de laboratório e requerem que o pé entre em contato com o centro da plataforma durante a marcha, para garantir uma leitura precisa. Já sistemas de palmilhas permitem o uso com calçados e fora do ambiente de laboratório, e se mostraram ser mais eficientes, flexíveis, de menor custo e permitindo maior mobilidade (RAZAK et al., 2012).

A palmilha de sensores Medical Sensor 3000, produzida pela Tekscan, se destaca por seu material leve e flexível e seus sensores com ótima resolução espacial, requisito importante para avaliação mais precisa da distribuição de pressão da planta do pé.

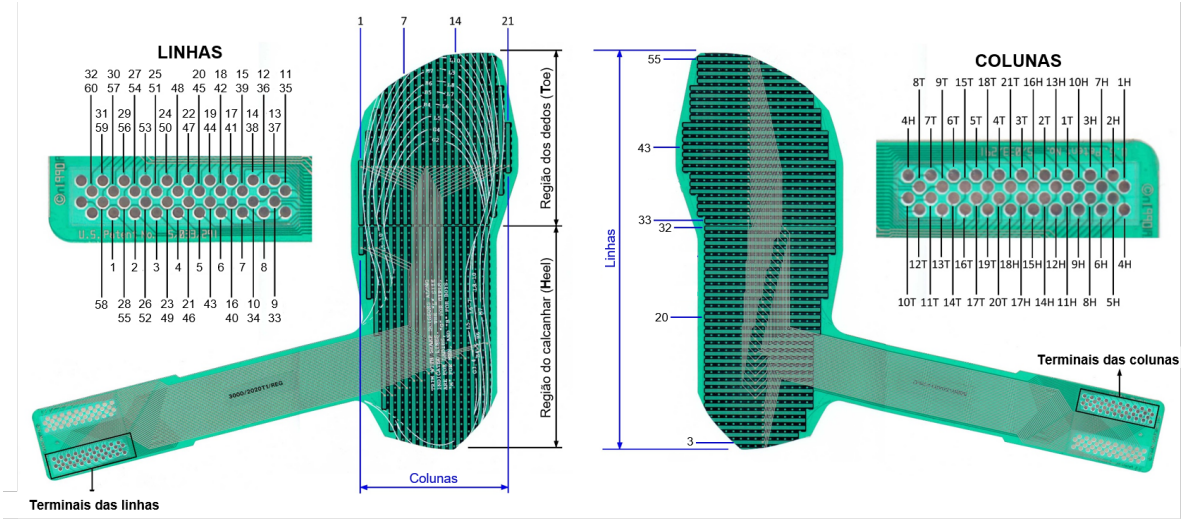
O foco deste trabalho é a bilateralização de um sistema de palmilha que capta a distribuição de pressão plantar e permite a visualização em tempo real da mesma, desenvolvido no trabalho "Sistema de transdutores para dispositivo de auxílio a marcha

de tetraplégicos e paraplégicos"(CRITTER, 2015). Tal sistema é composto pela palmilha Medical Sensor 3000, por um circuito de aquisição de dados e uma aplicação Java que recebe os dados via Bluetooth e os exibe na tela. Este sistema já existente para uma palmilha será modificado para funcionar com duas palmilhas, permitindo a visualização das pressões de ambos os pés e, portanto, melhor controle de toda a marcha do paciente.

2 PALMILHA

A palmilha utilizada neste trabalho é a Medical Sensor 3000 (Tekscan, Inc., Boston, MA, USA), que pode ser vista na figura 1. A mesma possui algumas delimitações de recorte, possibilitando o recorte de acordo com o tamanho de pé do indivíduo.

Figura 1: Palmilha Medical Sensor 3000



Fonte: Adaptado de Varoto et al. (2017)

Cada palmilha é uma matriz de sensores resistivos, em formato de pé, composta por duas folhas com tinta condutiva. Uma das folhas possui as fitas dispostas horizontalmente, representando as linhas, e a outra possui fitas verticais, representando as colunas. O contato entre uma linha e uma coluna forma um sensor resistivo. Quando um sensor é pressionado, o contato entre as fitas de carbono aumenta, o que reduz a resistividade do mesmo. Assim, quanto maior a pressão aplicada sobre um ponto, menor a resistência.

2.1 Mapeamento da palmilha

A palmilha original possui 60 linhas. A folha das colunas é dividida em duas seções: uma na região dos dedos do pé, e outra do calcanhar. Na palmilha original, a região dos dedos possui 21 colunas, e a do calcanhar possui 18. A região do calcanhar compreende as linhas de 1 a 32, e a dos dedos, de 33 a 60. As linhas e colunas foram enumeradas para a o projeto conforme a figura 1.

Para este projeto, as palmilhas foram recortadas em uma delimitação para um pé um pouco menor. Assim, as palmilhas utilizadas possuem 51 linhas, da 3 até a 53. Quanto

às colunas, foram recortadas da região dos dedos as de número 1, 20 e 21, e da região do calcanhar a 1, 2, 3 e 18, restando, assim, 18 e 14 colunas em cada região, respectivamente.

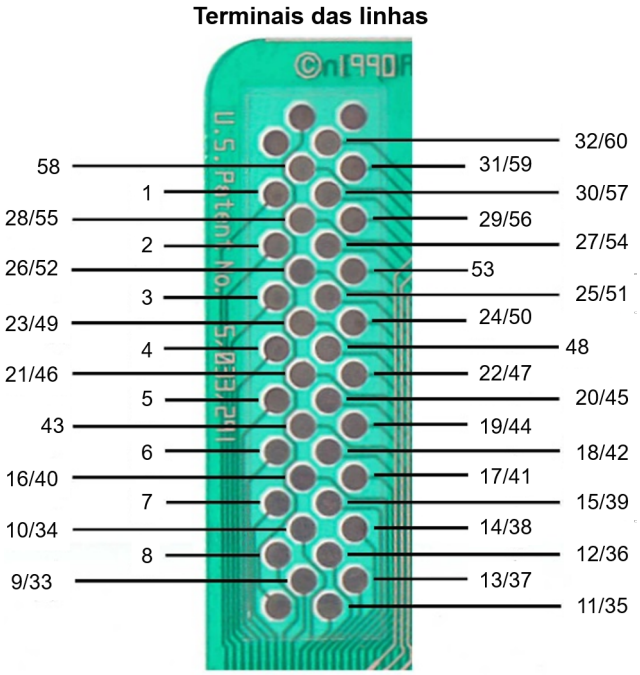
As linhas e colunas da palmilha são acessadas por conectores terminais, exibidos na figura 2.

Figura 2: Terminais de uma palmilha

Fonte: Adaptado de Varoto et al. (2017)

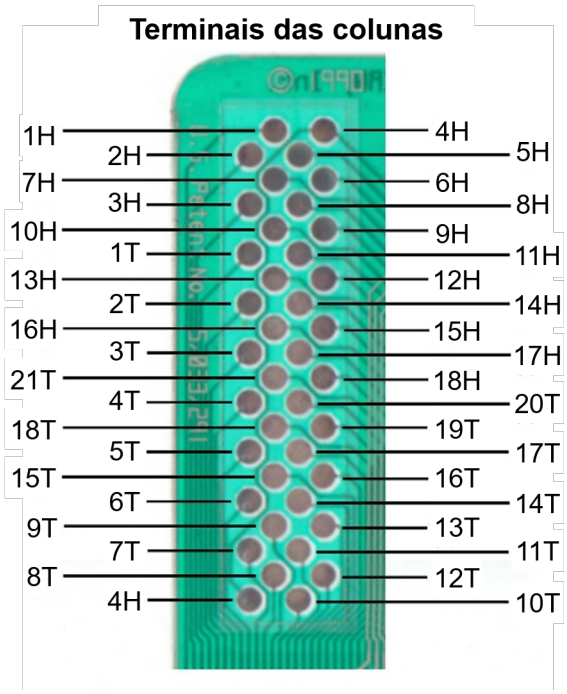
Para o sistema de aquisição de dados, é necessário acessar cada combinação de linhas e colunas através dos terminais para realizar a leitura da resistência de todos os sensores, possibilitando, assim, o mapeamento da pressão de todos os pontos do pé do indivíduo. Para tanto, identificou-se a correspondência de quais terminais acessam quais linhas e colunas, destacada nas figuras 3 e 4.

Figura 3: Identificação dos terminais das linhas



Fonte: Autoria própria

Figura 4: Identificação dos terminais das colunas



Fonte: Autoria própria

3 CIRCUITO DE AQUISIÇÃO

Os sensores da palmilha se relacionam entre linhas e colunas: quando uma linha e uma coluna entram em contato através da pressão aplicada sobre elas, é gerada uma resistência entre os pontos. Cada sensor da palmilha funciona como se cada linha e cada coluna tivessem resistências independentes que, quando conectadas, geram uma resistência para determinada intersecção de linha e coluna.

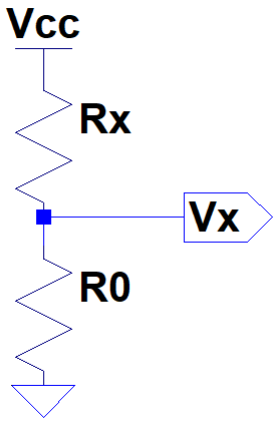
Dessa forma, para se fazer a leitura da resistência de cada ponto da matriz linha x coluna, realizou-se alguns testes na palmilha, conectando-se as saídas de uma determinada linha com uma determinada coluna e utilizou-se um multímetro para medir a resistência daquele cruzamento de sensores. Validou-se, com esse teste, que quando é aplicada uma tensão em um ponto da palmilha, que é o cruzamento de uma linha com uma coluna, há a criação de uma resistência, que para fins explicativos denominaremos de Rx ao longo do capítulo.

Para a aquisição dos valores de cada linha e coluna, faz-se necessária a aquisição dos valores de resistência para cada um dos sensores e, dessa forma, projetou-se um circuito de aquisição, com base no circuito projetado no trabalho "Sistema de transdutores para dispositivo de auxílio a marcha de tetraplégicos e paraplégicos"(CRITTER, 2015). Este circuito é capaz de relacionar a resistência do sensor, Rx, com a tensão medida entre os resistores, conforme a fórmula do divisor de tensão representada pela equação 3.1.

$$Vx = \frac{R0}{R0 + Rx} * Vcc \tag{3.1}$$

O circuito de aquisição projetado está representado na figura 5.

Figura 5: Circuito de aquisição simples



Fonte: Autoria própria

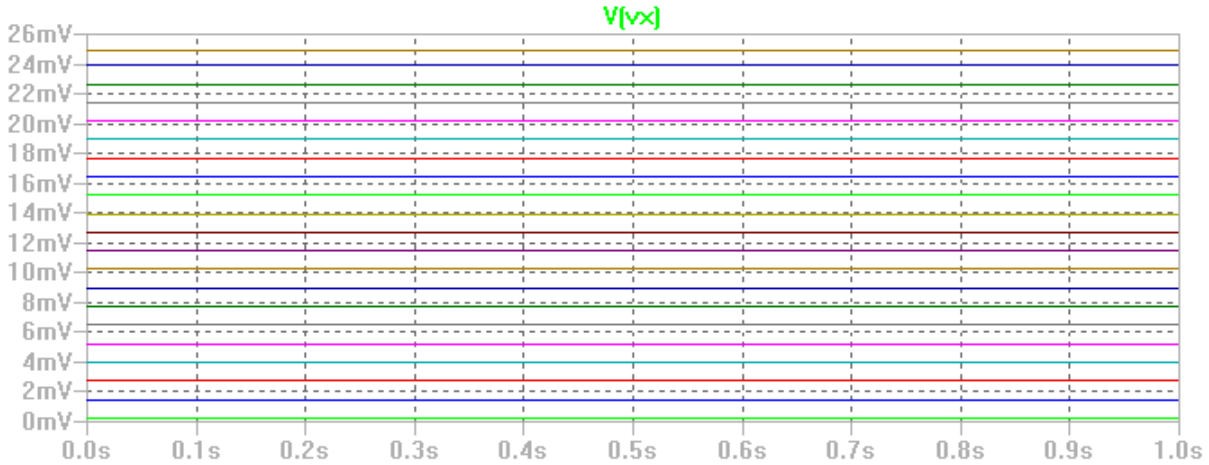
Conforme testes realizados para determinar a variação da resistência e consequente condutância para cada um dos sensores, sabe-se que a resistência de cada sensor varia entre 2 kΩ até aproximadamente 2 MΩ, valor que depende da pressão aplicada sobre os sensores da palmilha (CRITTER, 2015).

Aplicando no circuito uma resistência baixa para R0 é possível concluir pela equação 3.2 que a tensão de saída Vx é proporcional à condutância do sensor (Cx), representado pela resistência Rx, onde cte é uma constante. Assim, pelo circuito montado na figura 5, quando a resistência R0 for muito pequena, tem-se uma relação diretamente proporcional entre a tensão medida (Vx) e a pressão aplicada nos sensores.

$$Vx = \frac{R0}{R0 + Rx} * Vcc \approx \frac{1}{Rx} * R0 * Vcc = cte * Cx \tag{3.2}$$

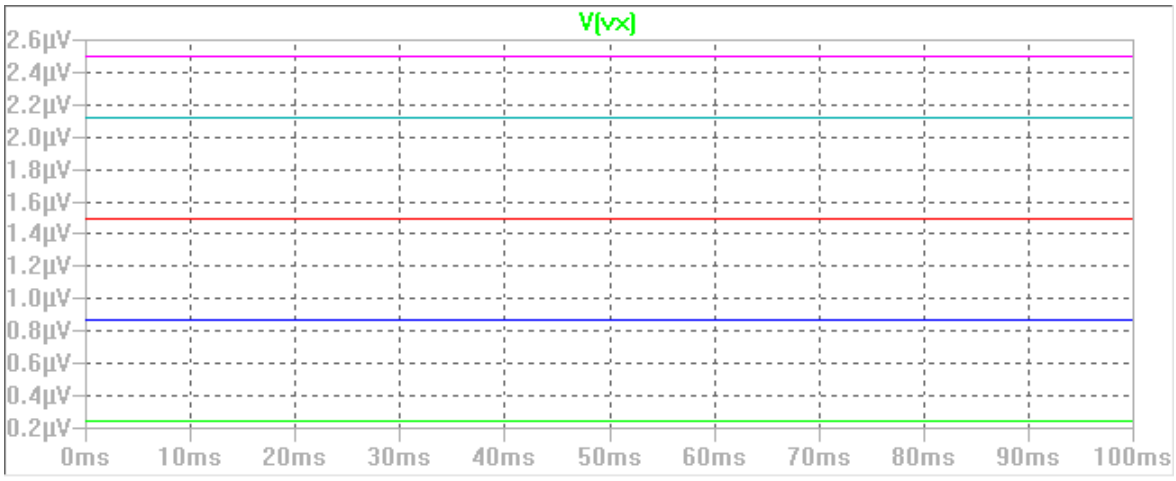
No entanto, para o circuito da figura 5, quando é colocada uma resistência R0 muito pequena, entre 0.1Ω e 10Ω , a tensão de saída do circuito fica muito baixa, conforme mostrado nas figuras 6 e 7, que são simulações no software LTSpice do circuito de aquisição simples, com variação de valores de R0 entre 0.1Ω e 10Ω e para Rx de 2 kΩ e 2 MΩ. Nestas figuras, a linha superior é a que possui R0 com menor valor e a linha inferior é a com R0 assumindo maior valor na variação. Por esse motivo, faz-se necessário a utilização de um circuito amplificador para amplificar a saída desse sinal e fazer com que ele possa ser mais facilmente utilizado para os fins práticos desse trabalho.

Figura 6: Simulação da tensão de saída do circuito de aquisição simples para diversos valores de R0 variando entre 0.1Ω e 10Ω com 0.5Ω de incremento e com Rx de 2 kΩ



Fonte: Autoria própria

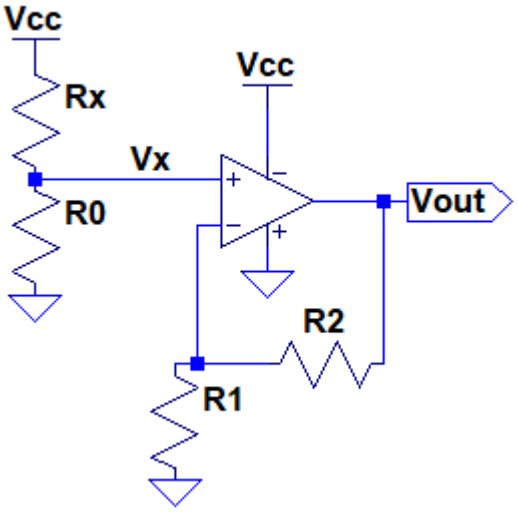
Figura 7: Simulação da tensão de saída do circuito de aquisição simples para diversos valores de R0 variando entre 0.1Ω e 10Ω com 2Ω de incremento e com Rx de 2 MΩ



Fonte: Autoria própria

O circuito amplificador utilizado para fazer a amplificação da tensão de saída é um circuito amplificador não inversor, representado pela figura 8.

Figura 8: Circuito Amplificador Simples



Fonte: Autoria própria

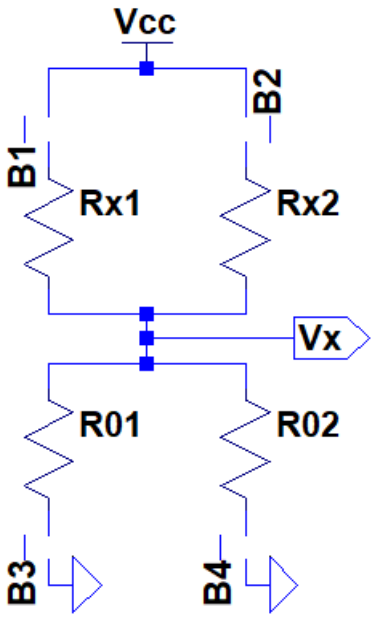
3.1 Seleção das linhas e colunas

Conforme mencionado na seção 2.1, a palmilha possui 60 linhas e no máximo 21 colunas (dependendo da região da mesma). Assim, considera-se necessário criar um circuito

lógico que faça a varredura de 1260 sensores, para se realizar a leitura de cada deles.

Para se elaborar o circuito de aquisição e testar se funcionaria, fizeram-se testes em uma protoboard nos quais montou-se o circuito esquematizado na figura 9.

Figura 9: Circuito para teste das lógicas de seleção



Fonte: Autoria própria

Neste circuito, os resistores Rx1 e Rx2 correspondem aos sensores das linhas e os resistores R01 e R02 correspondem aos sensores das colunas. Já os botões B1, B2, B3 e B4 servem como lógica para chavear as seleções. Ou seja, ao pressionar o botão B1, a linha correspondente ao resistor Rx1, ao pressionar B3 a coluna correspondente ao resistor R01 é selecionada e assim por diante. Por este circuito, é possível calcular a tensão de saída em Vx e verificar se o chaveamento dessa forma estaria correto. Para isso, é necessário que se pressione, ao mesmo tempo, um botão referente à linha (B1 ou B2) e um botão referente à coluna (B3 ou B4).

Após a realização dos testes, atribuindo-se valores de Rx1 = 2 kΩ; Rx2 = 3 kΩ; R01 = 1 kΩ e R02 = 2 kΩ, e tendo como valor eficaz do VCC de 3.46V, tirou-se como resultados os dados mostrados na tabela 1. Para o teste que obteve-se como resultado o Vx Prático - 2, realizou-se exatamente com o circuito montado da figura 9, apenas pressionando os botões necessários. Para o teste que obteve-se como resultado o Vx Prático - 1, realizou-se pressionando os botões correspondentes e desconectando os resistores que não seriam utilizados para cada uma das medidas.

Após a análise dos dois resultados notou-se que quando havia a conexão dos

Tabela 1: Resultados do circuito para teste da lógica de seleção

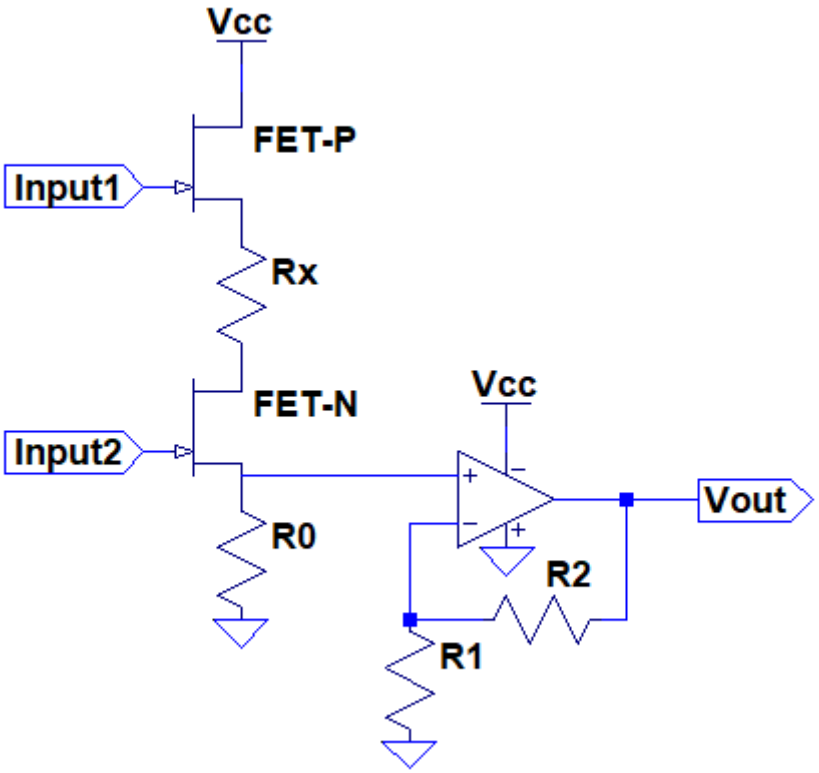
Linha (Rx)	Coluna (R0)	Vx Teórico (V)	Vx Prático (V) - 1	Vx Prático (V) - 2
Rx1	R01	1.1533	1.145	0.96
Rx1	R02	1.73	1.738	1.33
Rx2	R01	0.865	0.86	0.65
Rx2	R02	1.384	1.393	0.9

Fonte: Autoria própria

resistores (teste 2), os resistores que não deveriam ser selecionados interferiam no resultado de Vx, uma vez que corrente continuaria passando por eles independentemente da sequência de acionamento das chaves. Já nos resultados para o teste 1, notou-se que os valores ficaram muito próximos dos teóricos. Dessa forma, viu-se necessária a utilização de, ao invés de botões, chaves eletrônicas (FETs) para que a seleção seja feita apenas dos resistores selecionados e os restantes não interfiram no circuito de análise final.

Com base no circuito projetado no trabalho Sistema de transdutores para dispositivo de auxílio a marcha de tetraplégicos e paraplégicos (CRITTER, 2015) e com o que foi constatado nos testes apresentados na tabela 1, montou-se o circuito de chaveamento da figura 10.

Figura 10: Circuito de aquisição completo



Fonte: Autoria própria

No circuito da figura 10, nota-se dois inputs (input 1 e input 2) que fazem a ativação dos FETs P e FETs N, respectivamente. Quando os dois FETs são ativados, simultaneamente, por meio do microcontrolador, o resistor Rx é selecionado e, então, o sinal a ser lido passa pelo amplificador e permite ser analisado na saída Vout.

O input 1 é responsável por fazer a ativação do FET P, que faz a seleção das linhas da palmilha. Já o input 2 faz a ativação do FET N, que é responsável pela seleção das colunas da palmilha.

3.2 Aplicação prática

Para a adaptação do sistema de uma única palmilha (CRITTER, 2015) para duas palmilhas, a ideia inicial era manter o mesmo circuito de aquisição, com a diferença de que teria que realizar a varredura do dobro de sensores resistivos. Assim, era necessário o uso de um microcontrolador que tivesse mais memória e conversores A/D. Como o sistema original também utiliza um módulo Bluetooth para transmissão dos dados, escolheu-se o módulo ESP32, microcontrolador com maior memória e Bluetooth integrado.

Dessa forma, a ideia inicial do projeto era a de se usar apenas um microcontrolador

ESP32 para fazer o chaveamento de cada um dos FETs P e FETs N, tanto para a palmilha da esquerda quanto para a palmilha da direita, simultaneamente. Assim, projetar-se-ia 8 circuitos de aquisição para cada uma das palmilhas, porém esses 8 circuitos estariam divididos em 2 grupos de 4, onde cada quarteto estaria relacionado ou à parte superior das colunas (Toe) ou à parte inferior (Heel). Ao final, seriam utilizados um total de 8 portas ADC disponíveis no ESP32. Assim, simultaneamente seriam lidas 4 portas ADC relacionadas a palmilha da esquerda e 4 para a da direita. Nesse projeto, cada circuito de aquisição poderia selecionar 4 linhas, totalizando 32 colunas.

Já a seleção dos FETs P (input 2), responsáveis pela seleção das linhas, seria feita através de um circuito composto por 4 demultiplexadores com 16 saídas cada. Cada palmilha estaria relacionada com 2 demultiplexadores. Assim, quando fosse selecionado o primeiro demultiplexador da palmilha da direita, o ESP32 selecionaria, também, o primeiro demultiplexador da palmilha da esquerda ao mesmo tempo, fazendo com que fossem lidos os mesmos dados de linha e coluna tanto para a palmilha da esquerda quanto para a palmilha da direita. Para alternar entre os dois demultiplexadores de cada uma das palmilhas, o ESP32 mandaria sinais no enable de cada um dos demultiplexadores, fazendo com que a seleção das linhas fosse igual para as duas palmilhas.

Para fazer um circuito único de aquisição, pensou-se que o ESP32 e o circuito deveriam ficar na cintura do usuário para que pudesse captar os dados de ambas as palmilhas ao mesmo tempo. No entanto, para fazer a conexão da palmilha até o circuito de aquisição na cintura do paciente, o sinal deveria passar por toda a perna do usuário através de um cabo. Dessa forma, o sinal ainda seria analógico e poderia haver muito ruído e perda de sinal.

Para solucionar isso propôs-se duas soluções. A primeira seria utilizar um Serial Peripheral Interface (SPI) para fazer a conversão de analógico para digital direto dos dados adquiridos da palmilha, para, assim, evitar a perda de dados. Já a outra solução, que foi a escolhida para o decorrer do projeto, é a solução de fazer duas placas de aquisição independentes, cada qual com um microprocessador responsável pelo chaveamento de sua palmilha, onde um microprocessador seria o master e o outro o slave. Os dois microprocessadores fariam o chaveamento de cada uma de suas palmilhas e, então, o slave mandaria as informações já convertidas em digital para o master através de um cabo que percorreria a perna do usuário. Após isso, o master captaria as informações do slave e uniria com os dados lidos da sua própria palmilha para, depois, enviá-las para a interface de análise.

Com a nova abordagem escolhida de fazer um microcontrolador mestre (master) e outro escravo (slave), projetou-se 8 circuitos de aquisição para cada uma das palmilhas, sendo que a saída de cada um dos circuitos entraria em uma das 8 entradas dos conversores A/D no ESP32. Da mesma forma como pensado anteriormente, cada circuito de aquisição

faria o mapeamento de 4 colunas, selecionadas através dos FETs tipo N, totalizando o mapeamento de 32 colunas.

Da mesma forma, os FETs tipo P, para cada uma das palmilhas, seriam selecionados através de dois demultiplexadores que fariam a lógica de seleção das linhas.

3.3 Lógica de seleção das chaves

O chaveamento das colunas é feito pela ativação do FET tipo N. O controle desses FETs são feitos ligando-se diretamente a base deles em 4 portas de saída do microcontrolador, conforme figura 14: portas 27, 14, 12 e 13, responsáveis por ativar os FETs N 1, 2, 3 e 4, respectivamente.

Cada um dos 8 circuitos de aquisição possui 4 FET tipo N que foram mapeados como posição 1, 2, 3 e 4. Como explicado na seção anterior, cada um dos 8 circuitos de aquisição mapeia 4 colunas, e, nessa configuração, quando o microcontrolador envia o sinal de seleção através de suas portas, a posição do FET N ativada para cada um dos 8 circuitos é a mesma, ou seja, quando o microcontrolador selecionar o FET N número 1, ele selecionara todos os FETs N número 1 de cada um dos 8 circuitos de aquisição, fazendo com que seja selecionada mais de uma coluna por vez, apesar de só ser necessário o uso de 4 portas do micro.

Para o chaveamento dos FETs tipo P fez-se o uso de dois demultiplexadores, nos quais cada um deles controla 16 saídas através de 4 entradas. A alternância entre cada um dos demultiplexadores é feita pela alteração do sinal no pino enable. O demultiplexadores quando recebe em seu pino enable nível lógico alto faz com que nenhuma das saídas seja selecionada. Já quando o pino enable esta em nível lógico baixo as saídas funcionam normalmente de acordo com a lógica apresentada na tabela 2, possibilitando a alternância entre os dois demultiplexadores.

Tabela 2: Lógica de seleção das saídas do demultiplexador

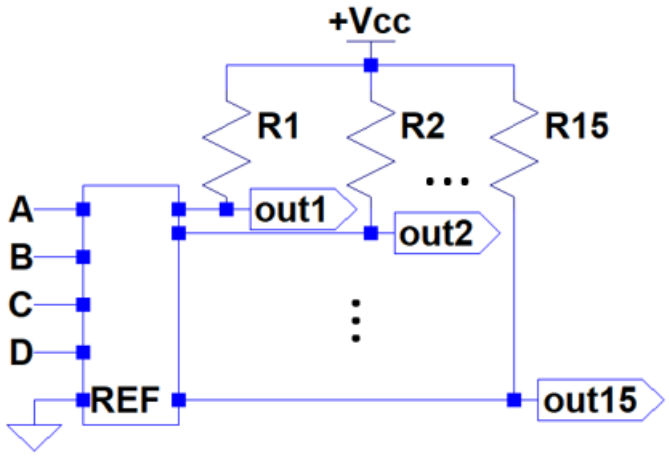
D	C	B	A	Enable	Saída
0	0	0	0	0	X0
0	0	0	1	0	X1
0	0	1	0	0	X2
0	0	1	1	0	X3
0	1	0	0	0	X4
0	1	0	1	0	X5
0	1	1	0	0	X6
0	1	1	1	0	X7
1	0	0	0	0	X8
1	0	0	1	0	X9
1	0	1	0	0	X10
1	0	1	1	0	X11
1	1	0	0	0	X12
1	1	0	1	0	X13
1	1	1	0	0	X14
1	1	1	1	0	X15
X	X	X	X	1	Nenhuma

Fonte: Autoria própria

Os pinos 21, 19, 18, 5 do ESP32 se conectam nos pinos D, C, B e A, respectivamente, do demultiplexador.

Outro fator importante para a lógica de seleção é que os FETs tipo P são ativados quando há tensão nula nas suas bases. Sendo assim, quando o demultiplexador seleciona uma saída, é necessário que essa saída fique em zero e o restante fique em Vcc, para garantir que somente um dos FET tipo P irá ser selecionado. Dessa forma, colocou-se como referência no demultiplexador o terra e acoplou-se resistores em cada uma das saídas do demultiplexador, ligando esses resistores ao Vcc. Assim, ao ser selecionada uma saída, essa saída fica igual a referência do demultiplexador e as restantes ficam em terceiro estado (impedância alta) que, com a logica do circuito da figura 11, faz com que as outras saídas fiquem em estado HIGH. Essa lógica foi a mesma implementada no trabalho "Sistema de transdutores para dispositivo de auxílio a marcha de tetraplégicos e paraplégicos"(CRITTER, 2015), conforme figura 11, extraída do mesmo trabalho.

Figura 11: Circuito para lógica de ativação dos FETs tipo P

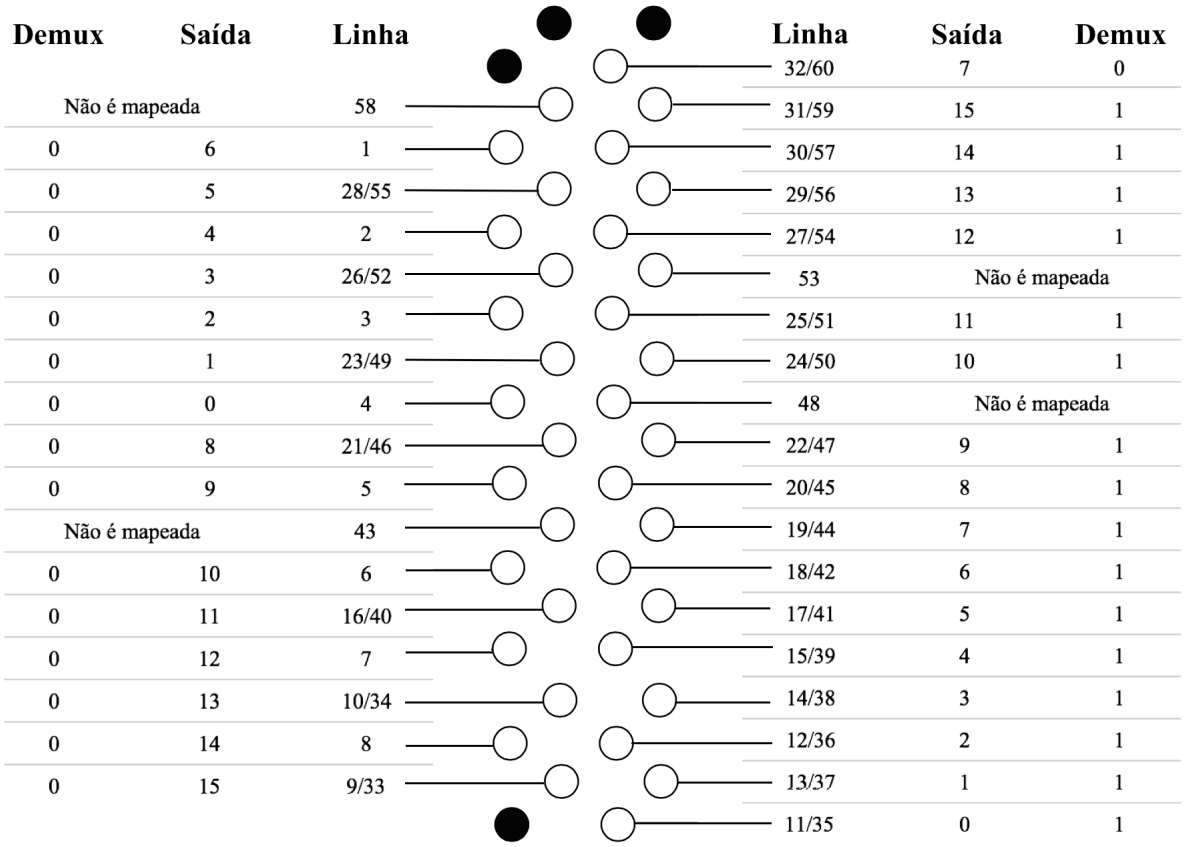


Fonte: Autoria própria

Na figura 11, as saídas out0 a out15 são conectadas nas bases dos FETs tipo P, sendo responsáveis por ativar os FETs e fazer a seleção das linhas, ou seja, na figura 10 as saídas out0 a out15 representam, para cada circuito, o input1.

Enfim, para o desenvolvimento da placa do circuito de aquisição, era necessário definir um mapeamento de conexão dos demultiplexadores nas linhas da palmilha, e dos FETs N nas colunas, conforme os terminais da palmilha (figuras 3 e 4). O mapeamento definido é exibido nas figuras 12 e 13. No mapeamento das linhas, associa-se cada conexão de linha com o pino enable de seu respectivo demultiplexador (0 ou 1), a ser ativado, e uma das 16 saídas do mesmo, a ser lida. No mapeamento das colunas, cada uma das conexões de colunas é associada com seu respectivo fet N, a ser ativado, e a seu conversor AD, a ser lido.

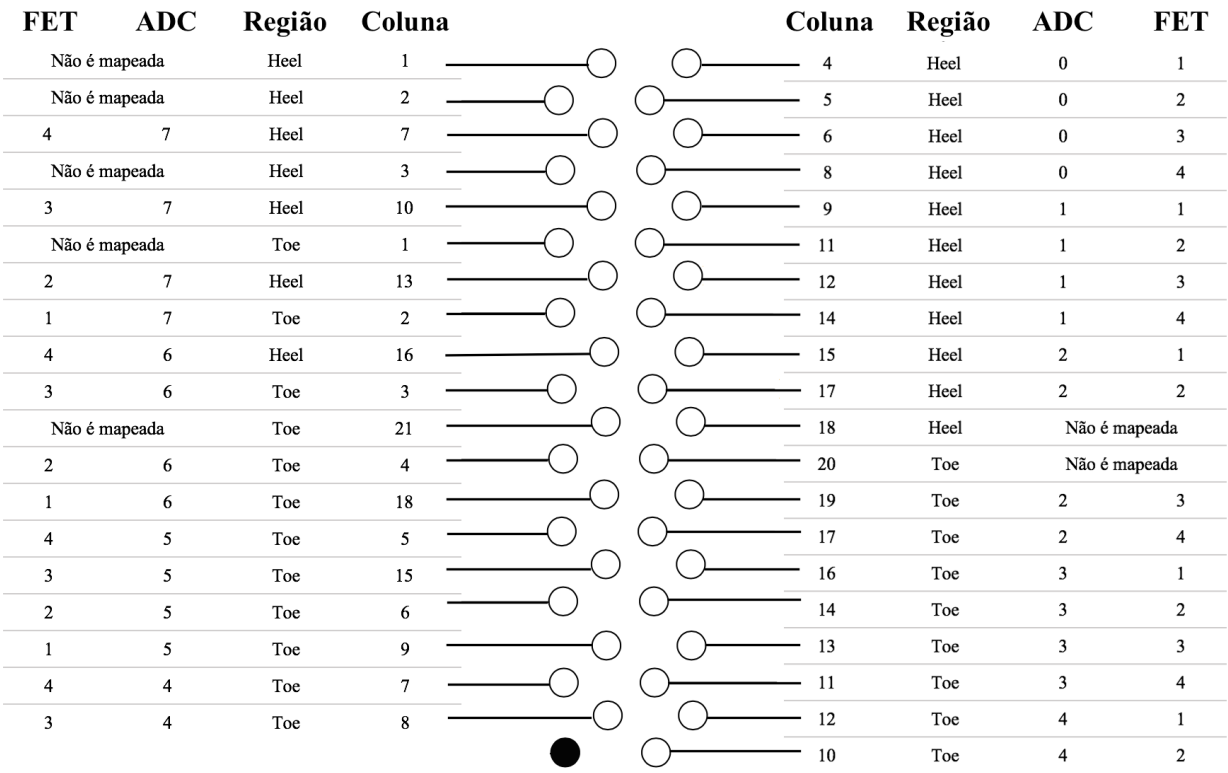
Figura 12: Mapeamento das linhas da palmilha



Demux	Saída	Linha	Linha	Saída	Demux
			32/60	7	0
Não é mapeada		58	31/59	15	1
0	6	1	30/57	14	1
0	5	28/55	29/56	13	1
0	4	2	27/54	12	1
0	3	26/52	53	Não é mapeada	
0	2	3	25/51	11	1
0	1	23/49	24/50	10	1
0	0	4	48	Não é mapeada	
0	8	21/46	22/47	9	1
0	9	5	20/45	8	1
Não é mapeada		43	19/44	7	1
0	10	6	18/42	6	1
0	11	16/40	17/41	5	1
0	12	7	15/39	4	1
0	13	10/34	14/38	3	1
0	14	8	12/36	2	1
0	15	9/33	13/37	1	1
			11/35	0	1

Fonte: Autoria própria

Figura 13: Mapeamento das colunas da palmilha



FET	ADC	Região	Coluna	Coluna	Região	ADC	FET
Não é mapeada		Heel	1	4	Heel	0	1
Não é mapeada		Heel	2	5	Heel	0	2
4	7	Heel	7	6	Heel	0	3
Não é mapeada		Heel	3	8	Heel	0	4
3	7	Heel	10	9	Heel	1	1
Não é mapeada		Toe	1	11	Heel	1	2
2	7	Heel	13	12	Heel	1	3
1	7	Toe	2	14	Heel	1	4
4	6	Heel	16	15	Heel	2	1
3	6	Toe	3	17	Heel	2	2
Não é mapeada		Toe	21	18	Heel	Não é mapeada	
2	6	Toe	4	20	Toe	Não é mapeada	
1	6	Toe	18	19	Toe	2	3
4	5	Toe	5	17	Toe	2	4
3	5	Toe	15	16	Toe	3	1
2	5	Toe	6	14	Toe	3	2
1	5	Toe	9	13	Toe	3	3
4	4	Toe	7	11	Toe	3	4
3	4	Toe	8	12	Toe	4	1
				10	Toe	4	2

Fonte: Autoria própria

3.4 Componentes utilizados

Os componentes escolhidos para serem utilizados no circuito de aquisição são descritos a seguir.

3.4.1 Microcontrolador

Conforme mencionado na seção 3.2, o ESP-WROOM-32 foi o módulo escolhido para a aplicação. Tal módulo apresenta um microcontrolador integrado com Wi-fi e Bluetooth.

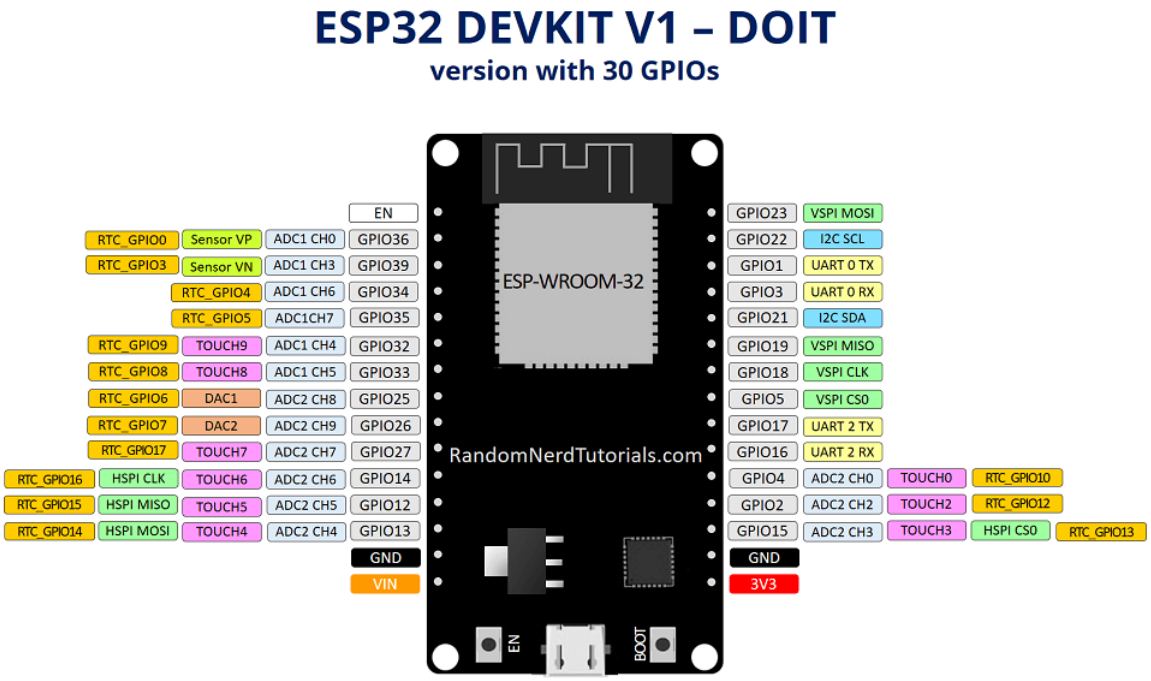
Como o ESP32 já possui módulo Bluetooth integrado, o projeto é simplificado tanto na questão de espaço físico, visto que não seria necessário adicionar um módulo Bluetooth a parte, quanto na parte de programação, pois o mesmo código de captação de dados também pode fazer o envio das informações por Bluetooth.

Para a presente aplicação, em cada palmilha, seria necessário 8 pinos como input para conversores A/D e, como output, 4 pinos para controle dos demultiplexadores, 2 pinos para seus enables, e 4 para controle dos FETs-N. Além disso, para a comunicação entre o

ESP master e o slave, portas Universal Asynchronous Receiver/Transmitter (UART) RX e TX são requisitos. O módulo escolhido deveria ter portas General Purpose Input/Output (GPIO) que atendessem todas estas especificações.

A versão utilizada foi a de 30 pinos, cujos periféricos atendem as especificações acima, englobando conversores A/D e D/A, UART, SPI e suporte a toque capacitivo (touch screen). A pinagem pode ser vista na figura 14.

Figura 14: Pinagem do microcontrolador ESP32



Fonte: Random Nerd Tutorials (2016)

As especificações de memória do ESP32 eram suficientes para a aplicação, e encontram-se na tabela 3.

Tabela 3: Especificações do ESP-WROOM-32

ROM	**RAM**	**FLASH**
448 KB	512 KB	4 MB

Fonte: Espressif Systems ().

O ESP32 possui entrada micro USB para alimentação, permitindo fácil programação e realização de testes. Além disso, outro benefício em se utilizar tal módulo é que o mesmo pode ser programado em diversos ambientes, sendo um deles a Arduino IDE, que permite programação em C, linguagem amplamente utilizada e de simples implementação.

3.4.2 Chave eletrônica

Para a construção do circuito, conforme descrito na seção 3.2, são necessárias dois tipos de FETs: P e N, que servirão como chaves para fazer o chaveamento do circuito. Para tal escolha, baseou-se no circuito já desenvolvido pelo Matheus Critter (CRITTER, 2015), porém também foi levado em consideração a disponibilidade e preço dos componentes.

Sendo assim, o FET tipo P utilizado foi o modelo BSS84P, que tem uma tensão entre Drain e Source de -60V, uma corrente de drain de -0,17A e possui uma resistência entre Drain e Source pequena quando o FET está ativo (8Ω) tendo uma interferência mínima no circuito de aquisição.

Já o FET tipo N utilizado foi um modelo similar ao utilizado pelo Matheus Critter (CRITTER, 2015), porém devido a disponibilidade do componente foi necessário substituí-lo por um similar. Assim, o FET tipo N utilizado no projeto foi o IRLML2505, que possui uma tensão entre dreno e source de 20V, corrente de Drain de até 4.2A e Resistência entre Drain e Source de 0,045Ω, que é uma resistência bem menor do que a utilizada no projeto anterior, garantindo que o FET não interferirá no circuito de aquisição.

3.4.3 Amplificador operacional

A amplificação do sinal fica por conta do amplificador operacional modelo MCP6002 ISN, o mesmo utilizado no circuito de aquisição de uma palmilha (CRITTER, 2015).

Esse amplificador é um dos melhores do mercado, operando em 1MHz, possuindo uma tensão de alimentação entre 1.8V e 6V, além de ter um baixo consumo de corrente, que fica em torno de 100μA. Todas essas características garantem que o amplificador seja válido para o circuito desenvolvido. Além disso, esse amplificador tem uma ótima precisão, facilitando a leitura e evitando perda de dados do sinal a ser amplificado.

3.4.4 Demultiplexador

Conforme dito na seção 3.3, é necessário a utilização de dois demultiplexadores, cada qual com 16 saídas, tornando possível a leitura das 32 colunas. De acordo com a tabela 2, escolheu-se um demultiplexador que respeitasse a lógica mostrada nessa tabela. Assim, o demultiplexador escolhido foi o modelo CD4067BE, que possui 4 portas de entrada e 16 de saída. Além disso, esse demultiplexador permite que a lógica de seleção dos FETs tipo P funcione, uma vez que a saída selecionada do demultiplexador fica em estado LOW, enquanto as saídas não selecionadas ficam com uma corrente de 10ρA, indicando que não há corrente naquela saída, ou seja, uma impedância alta.

3.4.5 Alimentação

A alimentação do ESP32 é feita através de uma bateria. Como o ESP32 permite uma tensão de alimentação entre 6V e 21V, escolheu-se utilizar para alimentá-la uma bateria de 9V. A alimentação é dada pelo pino Vin da figura 14. No entanto, sabe-se que a bateria não provê uma tensão muito linear e por conta disso, acoplou-se capacitores que fazem o controle da sobrecarga e ajustam a equalidade da tensão alimentada no ESP32.

Já o restante do circuito é alimentado pelo 3.3V que o ESP32 gera pelo seu pino 3V3 da figura 14. Essa porta fornece até 800mA, corrente suficiente para o funcionamento do resto do circuito, garantindo que ele opere sem nenhum problema.

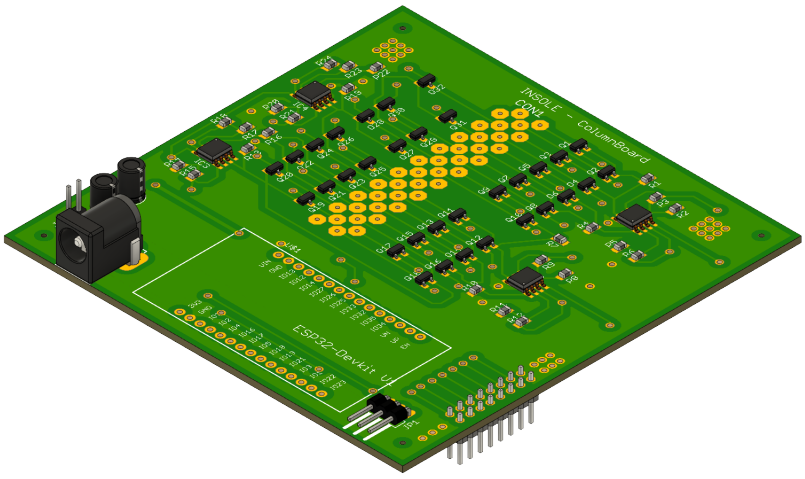
3.5 Placas de aquisição

Após o projeto da placa, relatado nas seções do capitulo 3, manufaturou-se a placa física do circuito de aquisição.

Como os conectores das linhas e das colunas da palmilha ficam em lado opostos, fez-se necessária a fabricação de duas placas, uma superior e uma inferior a fim de que se faça uma pressão entre as duas placas e feche o contato com os conectores, em ambos os lados, da palmilha.

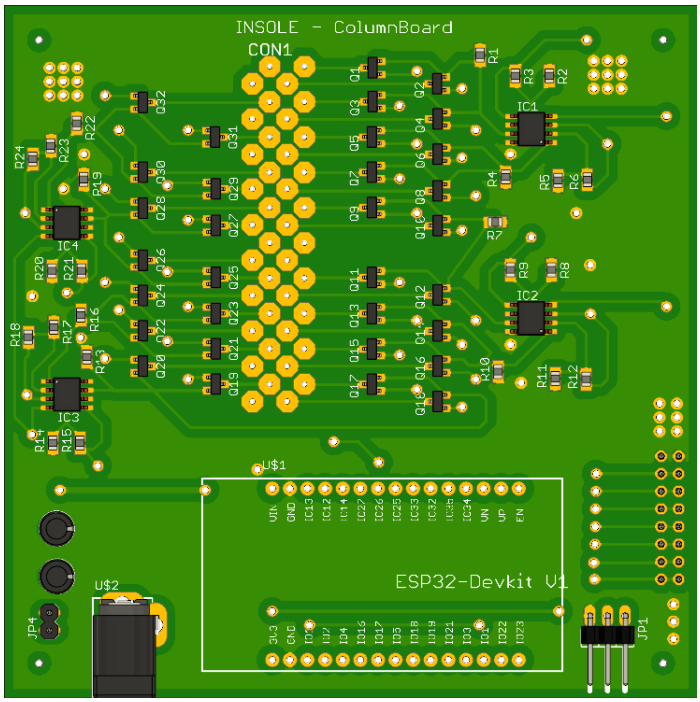
A primeira parte para a fabricação foi a montagem do esquemático da placa no software Eagle. O esquemático em 3D das duas placas estão apresentados nas figuras 15, 16, 17 e 18.

Figura 15: Placa superior de aquisição em 3D - Vista lateral



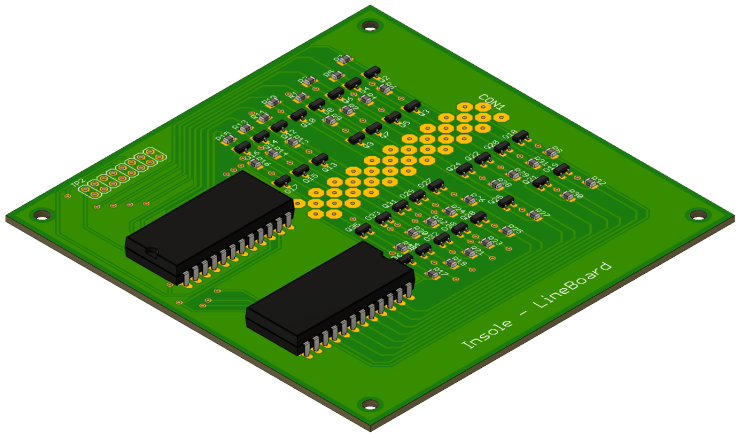
Fonte: Autoria própria

Figura 16: Placa superior de aquisição em 3D - Vista superior



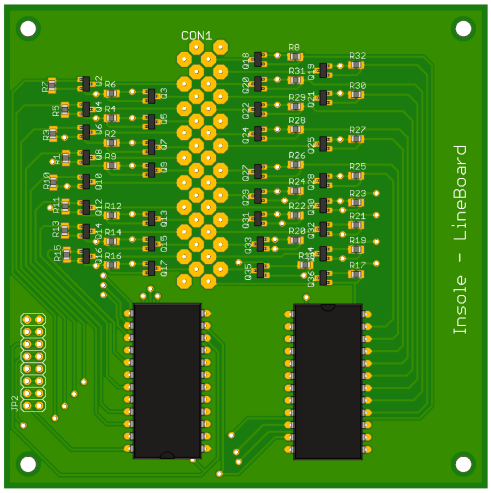
Fonte: Autoria própria

Figura 17: Placa inferior de aquisição em 3D - Vista lateral



Fonte: Autoria própria

Figura 18: Placa inferior de aquisição em 3D - Vista superior



Fonte: Autoria própria

A placa superior contém:

- ESP32
- 4 Amplificadores Operacionais modelo MCP6002ISN
- 32 FET tipo N modelo IRLML2502
- resistores smd
- 2 capacitores
- Conectores para as colunas da palmilha
- Plug para alimentação do ESP
- Conectores para as portas seriais RX-TX
- Conectores para a placa inferior

A placa inferior contém:

- 2 demultiplexadores modelo CD4067BE
- 32 FET tipo P modelo BSS84P
- resistores smd

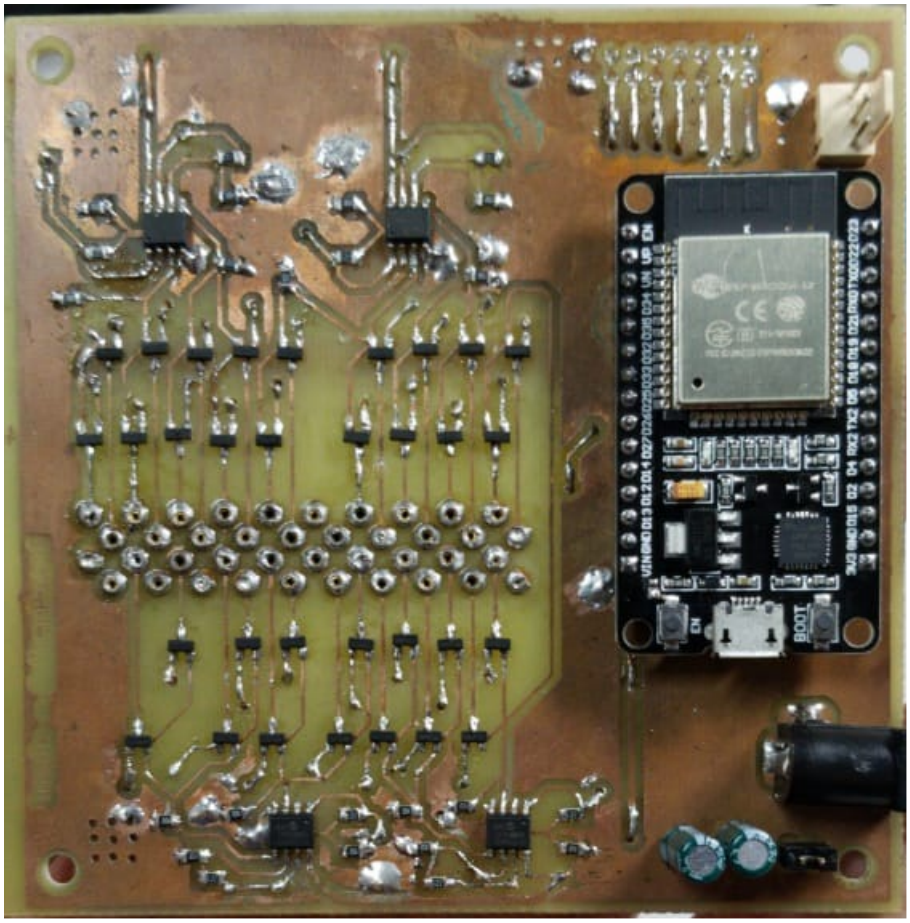
- Conectores para as linhas da palmilha
- Conectores para a placa superior

Após a construção dos esquemáticos, fez-se a placa física. Para tal tarefa utilizou-se os materiais adquiridos e os disponíveis na USP.

As placas foram prensadas e cada componente foi soldado individualmente. A comunicação entre a placa inferior e a superior é feita através de pinos colocados em ambas as placas que se ligam por meio de jumpers. Além disso, nos 4 cantos de cada uma das placas existem furos para que se possa fazer a ligação mecânica entre as duas placas e a palmilha, garantindo estabilidade.

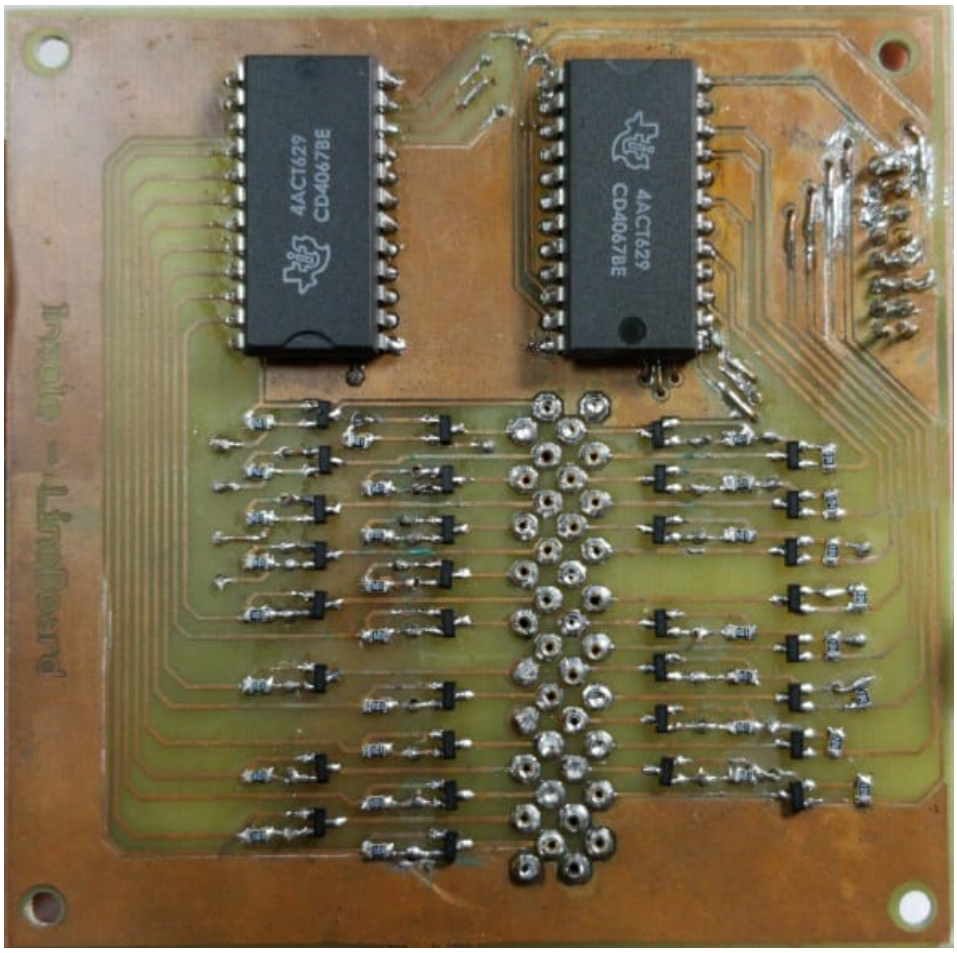
Imagens da placa física estão representadas nas figuras 19 e 20.

Figura 19: Placa de aquisição superior



Fonte: Autoria própria

Figura 20: Placa de aquisição inferior



Fonte: Autoria própria

3.5.1 Custos da placa de aquisição

Os custos de uma única placa de aquisição estão descritos na tabela 4 a seguir.

Tabela 4: Preços dos componentes para uma placa de aquisição

Componente	Descrição	Preço unitário	Unidades	Preço total
ESP-WROOM-32		R$ 37,90	1	R$ 37,90
Papel transfer		R$ 5,45	3	R$ 16,35
Adaptador bateria 9V		R$ 7,50	2	R$ 15,00
Bateria 9V		R$ 19,00	1	R$ 19,00
Conector KK fêmea	2,5mm	R$ 0,53	2	R$ 1,06
Conector KK macho	2,5mm	R$ 0,33	2	R$ 0,66
Terminal KK	2,54mm	R$ 0,20	10	R$ 2,00
Cabo manga	2,5m	R$ 8,63	1	R$ 8,63
MCP6002 I/SN	Amp. op.	R$ 7,50	4	R$ 30,00
BSS84P	MOSFET P	R$ 1,00	32	R$ 32,00
IRLML2502T	MOSFET N	R$ 1,96	32	R$ 62,72
CD4067BE	Demux	R$ 0,50	2	R$ 1,00
Resistor 1K	SMD	R$ 0,10	40	R$ 4,00
Resistor 22K	SMD	R$ 0,10	20	R$ 2,00
Resistor 100K	SMD	R$ 0,10	50	R$ 5,00
100NFX 50V	Capacitor	R$ 0,60	10	R$ 6,00
47UFX 25V	Capacitor	R$ 0,38	4	R$ 1,52
TOTAL		R$ 244,84		

Fonte: Autoria própria

4 CÓDIGOS DOS ESP32S

Conforme mencionado na seção 3.2, a aplicação definida envolve um sistema de microcontroladores master-slave. Desta forma, a programação a ser feita para o ESP32 master e para o slave devem ser diferentes.

Para a explicação da programação, esta será dividida em etapas, as quais serão detalhadas a seguir, conforme ordem em que ocorrem no código. Os códigos de programação do ESP slave e master encontram-se nos apêndices B e A, respectivamente.

4.1 Declaração de variáveis

Esta etapa está presente tanto no código do master quanto do slave e é igual em ambos.

Primeiramente, são incluídas as bibliotecas a serem utilizadas: "Arduino.h"(no master e slave) e "BluetoothSerial.h"(apenas no master). Aqui, primeiramente ocorre a configuração dos pinos do microcontrolador a serem utilizados, escolhida com base na pinagem do ESP32, exibido na figura 14. Também é setado o pino do LED do ESP32, o qual foi utilizado para testes do código.

Além disso, ocorre a definição da localização das linhas e colunas em vetores, conforme seus mapeamentos das figuras 12 e 13, respectivamente. Para tanto, utiliza-se valores hexadecimais, em que no vetor de colunas, o bit mais significativo é o conversor AD correspondente, e o menos significativo, o FET N. No vetor de linhas, o bit mais significativo corresponde ao demux, e o menos significativo à sua saída correspondente. No vetor de colunas, as colunas da região do calcanhar são armazenadas nas posições pares, e as dos dedos, nas ímpares.

Por fim, as funções a serem criadas no código são declaradas, junto com um caractere verificador.

4.2 Setup

Esta etapa está presente tanto no código do master quanto do slave.

Aqui, inicializa-se os pinos do ESP32 utilizando a função "pinMode()"e as portas seriais a serem utilizadas, as quais são:

- Serial: UART 0, dos pinos 1 e 3, a ser utilizada apenas para efeitos de teste do código
- Serial2: UART 2, dos pinos 16 e 17, a ser utilizada para comunicação entre os dois microcontroladores

- SerialBT: da biblioteca "BluetoothSerial.h", a ser utilizada apenas no master, para envio dos dados via Bluetooth.

Um delay de 3 segundos é inserido, para efeitos de debug do código e maior controle do início da operação dos ESPs.

4.3 Aquisição de dados da palmilha

Esta etapa está presente tanto no código do master quanto do slave e é igual em ambos.

Primeiramente, declara-se o vetor que receberá os dados da palmilha, com 1261 posições, referentes ao start byte e aos dados dos 1260 sensores.

Então, cria-se um loop externo, referente à varredura das colunas, que percorre 42 posições (colunas do calcanhar e dos dedos). Dentro do mesmo, a primeira etapa é uma verificação se a posição atual da varredura das colunas é par. Se for, para tal coluna, faz-se a varredura das linhas 1 até a 32 (região do calcanhar) e, se for ímpar, faz-se a varredura para esta linha das linhas 33 a 60.

Para cada posição da varredura, tem-se uma combinação de linha e coluna. Para cada combinação, primeiramente obtém-se o bit mais significativo da linha atual da varredura, e então seleciona-se o demultiplexador correspondente, ativando seu enable. Depois, obtém-se o bit menos significativo da linha, e ativa-se a saída do demux correspondente.

Em seguida, obtém-se o bit menos significativo da coluna atual da varredura, e ativa-se o FET N correspondente. Depois, obtém-se o bit mais significativo, que indica o conversor A/D a ser lido. Se o valor lido do conversor A/D for maior ou igual a 250, o mesmo será definido como 250. Se a leitura for de uma linha ou coluna não mapeada, o valor a ser inserido no vetor é 0.

Por fim, coloca-se na primeira posição do vetor o start byte. Para o master, este é "255"e para o slave, "254".

4.4 Sincronização entre ESPs

Esta parte do código é diferente para o ESP32 master e o slave. Primeiramente, em ambos códigos, a variável "verificador"é inicializada com o caractere "e". Então, no master, há o envio do caractere "s", que será usado para realização do sincronismo, via Serial2. No slave, o mesmo fica dentro de um loop até ler este caractere "s"da Serial2.

Quando o slave ler, ele também envia o caractere "s"via Serial2 e seta o pino referente ao LED do seu módulo ESP32 em "HIGH", ligando o LED. Então, o master fica dentro de um loop até ler "s"da Serial2 e, quando ler, também liga o seu LED.

Quando estes passos finalizam, os ESPs master e slave estão sincronizados.

4.5 Envio de dados do slave para o master

Nesta parte do código do slave, o mesmo envia seus dados capturados para o master. No código do master, há o recebimento destes.

O slave envia, via Serial2, seu vetor de 1261 posições, uma a uma. Quando enviar todas as posições, seta o pino do seu LED em "LOW", desligando-o. Então, o master entra em um loop em que enquanto houver dados na Serial2, ele recebe os dados vindos por ela em um segundo vetor de dados de 1261 posições. Quando finalizar, também desliga o LED do seu módulo ESP.

4.6 Envio de dados via Bluetooth

Esta etapa está presente apenas no código do ESP master, que realiza o envio dos dados de ambas palmilhas via Bluetooth.

Primeiramente, envia-se o vetor de dados da palmilha do master, posição a posição, incluindo o start byte. Quando finalizado, envia-se o vetor de dados da palmilha do slave, da mesma forma.

5 APLICAÇÃO JAVA

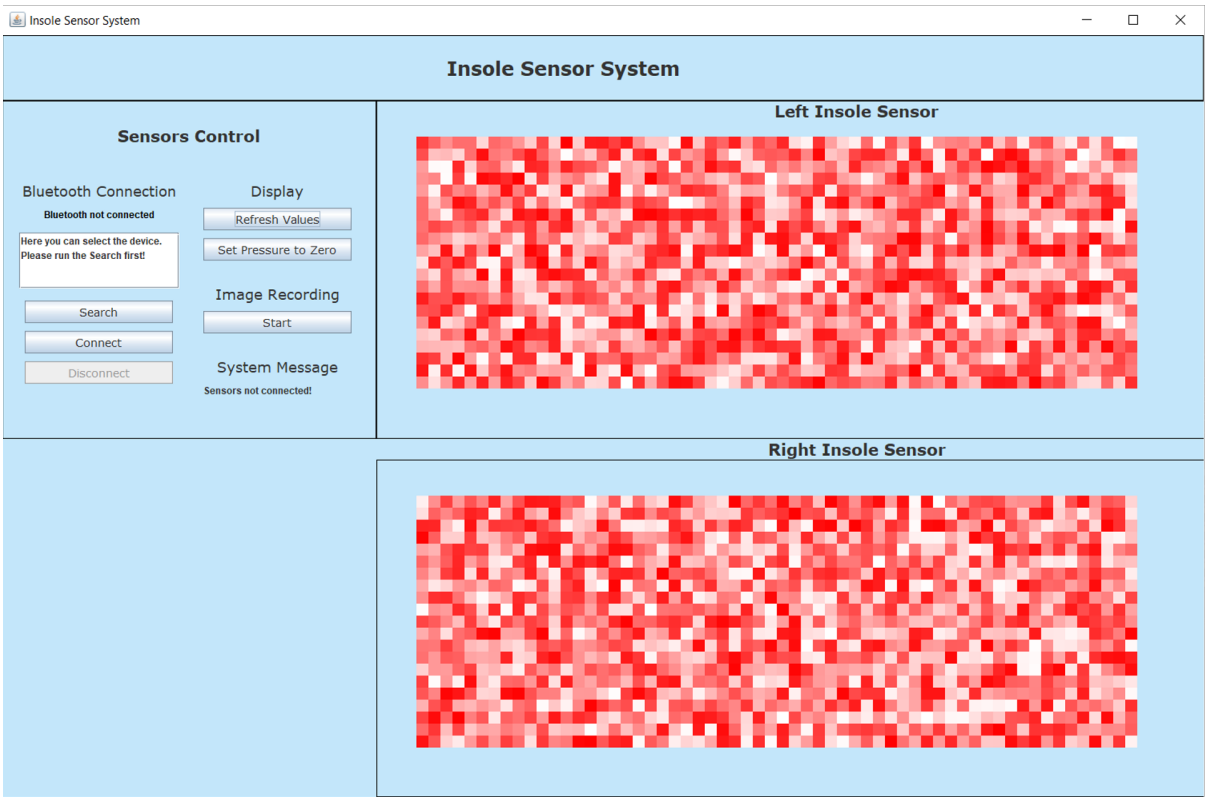
A aplicação Java já existente possuía uma interface gráfica para exibir os valores de pressão medidos em uma única palmilha, com a funcionalidade de busca e conexão do Bluetooth. Era, então, necessário alterar esta aplicação para a funcionalidade com duas palmilhas.

Para a edição dos códigos Java, foi utilizada a Eclipse IDE, que permite também a execução dos programas. Os códigos da aplicação encontram-se nos apêndices C e D.

Como um único dispositivo Bluetooth faz o envio de todos os dados, deixou-se apenas uma seção de busca e conexão do mesmo. Para os objetivos deste projeto, o código da aplicação Java foi alterado para se exibir na mesma interface 2 "telas", uma para os valores de pressão de cada palmilha.

A nova interface gráfica da aplicação está exibida na figura 21.

Figura 21: Interface gráfica da aplicação Java



Fonte: Autoria própria

Para diferenciar os dados recebidos de cada palmilha e saber em qual das telas exibi-los, alterou-se a parte de recepção dos dados via Bluetooth. Sabe-se que o ESP32

master realiza o envio de 2 vetores de 1261 posições, sendo que a primeira delas é o start byte. Assim, adicionou-se uma etapa no código Java em que caso o byte lido seja 255, significa que o vetor a ser recebido vem do master e, portanto, seus dados de pressão da sua palmilha serão exibidos na tela superior. Caso o byte seja 254, o vetor em questão vem do slave e será exibido na tela inferior.

6 TESTES E RESULTADOS

6.1 Códigos dos ESP32s e Aplicação Java

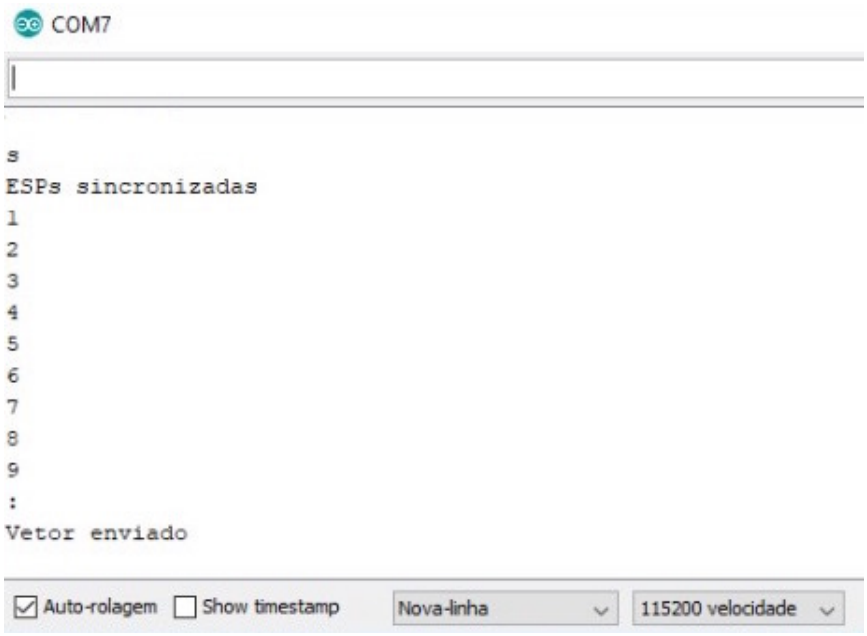
Os testes iniciais dos códigos do ESP foram feitos utilizando o monitor serial, ou seja, usando o comando "Serial.print()"para imprimir na tela do monitor.

Para testar a sincronização entre o ESP master e slave, alterou-se os códigos da seguinte maneira:

- Sincronização: Quando o master envia o caractere "s"via Serial2, imprime-se "s"na tela via Serial. Quando houver a leitura deste caractere pelo slave e depois que o mesmo enviar "s"via Serial2, também imprime-se "s"na tela.
- Envio de dados pelo slave: cria-se um vetor do tipo "char", de 10 posições, com os valores de 49 até 58. Envia-se posição a posição do vetor via Serial2 e imprime-as na tela usando o comando "Serial.print()". Desta forma, como o vetor era do tipo "char", deve-se obter na tela os valores "char", conforme tabela ASCII, ou seja: "1, 2, 3, 4, 5, 6, 7, 8, 9, :". Ao término do envio de todas posições do vetor, imprime-se na tela a frase "Vetor enviado".
- Recebimento de dados pelo master: cria-se um vetor do tipo "int"de 10 posições, chamado "var2", que recebe os valores vindos do slave via Serial2, posição a posição. Imprime-se cada uma na tela com o comando "Serial.print(var2[count], DEC)". Dessa forma, como o vetor é do tipo "int"e imprime-o com o parâmetro "DEC", deve-se obter os números de 49 a 58 na tela. Ao final do recebimento de todo o vetor, imprime-se na tela a frase "Vetor recebido".

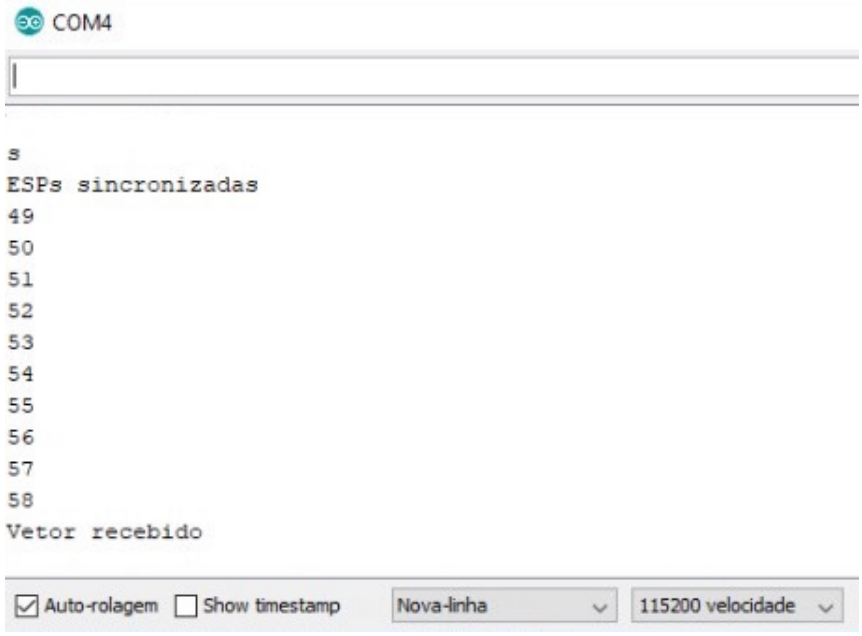
Conectou-se o slave no monitor serial COM7 e o master no COM4. As imagens do monitor serial do slave e do master, respectivamente, podem ser vistas nas figuras 22 e 23, respectivamente. Analisando-as, nota-se que o resultado foi conforme o esperado e o teste foi bem sucedido, validando-se a parte do código de sincronização entre os ESPs.

Figura 22: Monitor serial do slave no teste de sincronização



Fonte: Autoria própria

Figura 23: Monitor serial do master no teste de sincronização

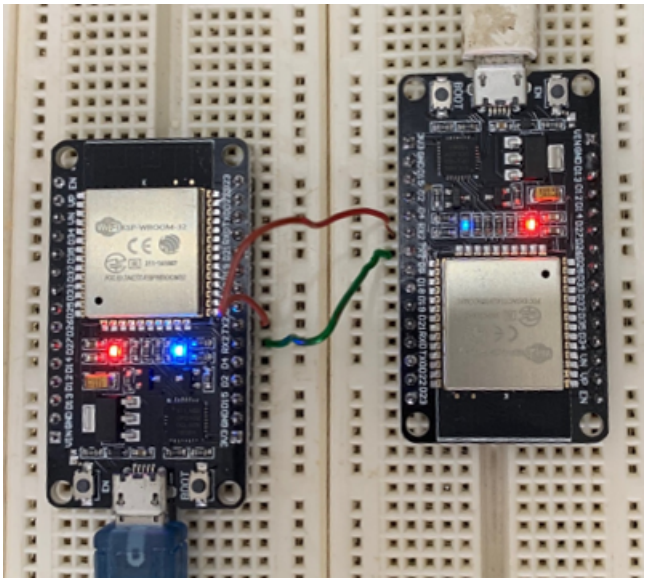


Fonte: Autoria própria

Na figura 24, pode-se visualizar os dois módulos ESPs alimentados por cabos micro USB e conectados entre si por dois fios, em que o pino TX de um ESP conecta-se no RX

do outro, e vice-versa. O LED azul de ambos módulos ESP32 estão ligados no momento da foto, pois, conforme especificado nos códigos, os LEDs piscam quando os módulos estão sincronizados. A frequência em que os LEDs piscavam era muito alta, tornando quase imperceptível a mudança de estados a olho nu, o que significa que a frequência de transmissão de dados entre as ESPs era muito alta.

Figura 24: Módulos ESPs em sincronismo



Fonte: Autoria própria

Posteriormente, realizou-se testes com os ESPs na interface da aplicação Java, permitindo visualizar tanto se o recebimento dos dados dos ESPs estava correto, quanto se a aplicação Java estava em funcionamento.

Para um primeiro teste, alterou-se os códigos dos ESPs para que enviassem um vetor de dados aleatório, com a função "random()", de 1261 posições, em que a primeira é o start byte.

Primeiramente, alimenta-se o módulo slave e depois o master. Então, após 3 segundos, que é o tempo de debug inserido no código, os módulos sincronizam e seus LEDs azuis começam a piscar.

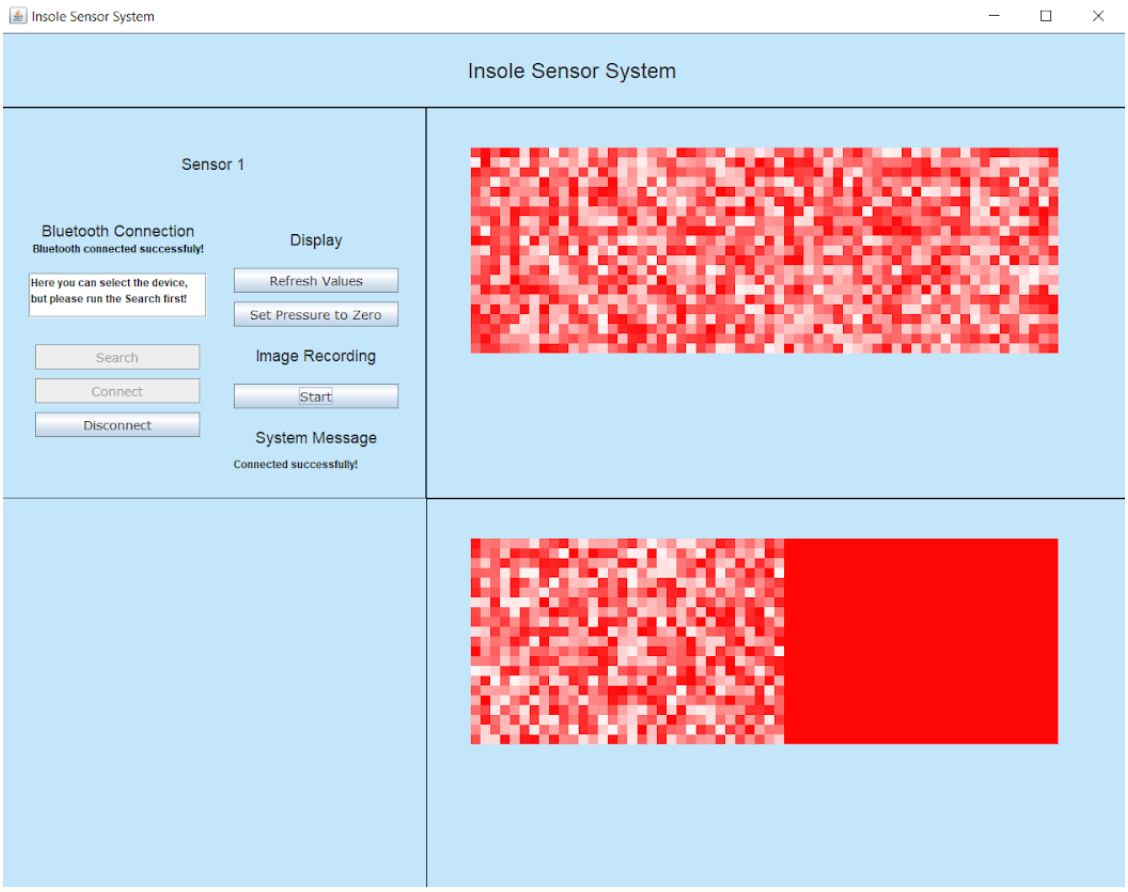
Então, apertou-se o botão "Search"da interface Java, o qual faz com que a busca por dispositivos Bluetooth se inicie. Após aproximadamente 10 segundos, os nomes dos dispositivos encontrados são exibidos na painel de rolagem abaixo de "Bluetooth Connection", incluindo o "ESP32". Então, clica-se no nome "ESP32"e, em seguida, no botão "Connect". Após poucos segundos, é exibida a mensagem "Bluetooth connected successfully"e os dados do master e slave são exibidos nas telas superior e inferior, respectivamente.

Observou-se que assim que o Bluetooth é conectado, a frequência com que os LEDs piscam é reduzida.

Como ambos módulos estavam enviando vetores aleatórios, as telas exibidas na interface continham pixels em tons de vermelho aleatórios, assim como na figura 21. Estes valores se alteravam, devido à função "random()", a cada transferência de dados, ou seja, na frequência de sincronismo. Tal frequência corresponde à frequência em que os LEDs azuis dos ESPs piscam, e foi estimada entre 3,6 e 3,8 Hz.

Realizou-se um novo teste, em que o vetor de dados do slave recebia o valor fixo de 248 (vermelho mais forte) para os pontos em conexão com as linhas de 33 a 60, correspondente à região dos dedos. O restante do vetor recebia valores aleatórios, assim como todo o vetor do master. O resultado obtido encontra-se na figura 25.

Figura 25: Interface Java para teste em que vetor do slave recebe valor fixo nas linhas de 33 a 60



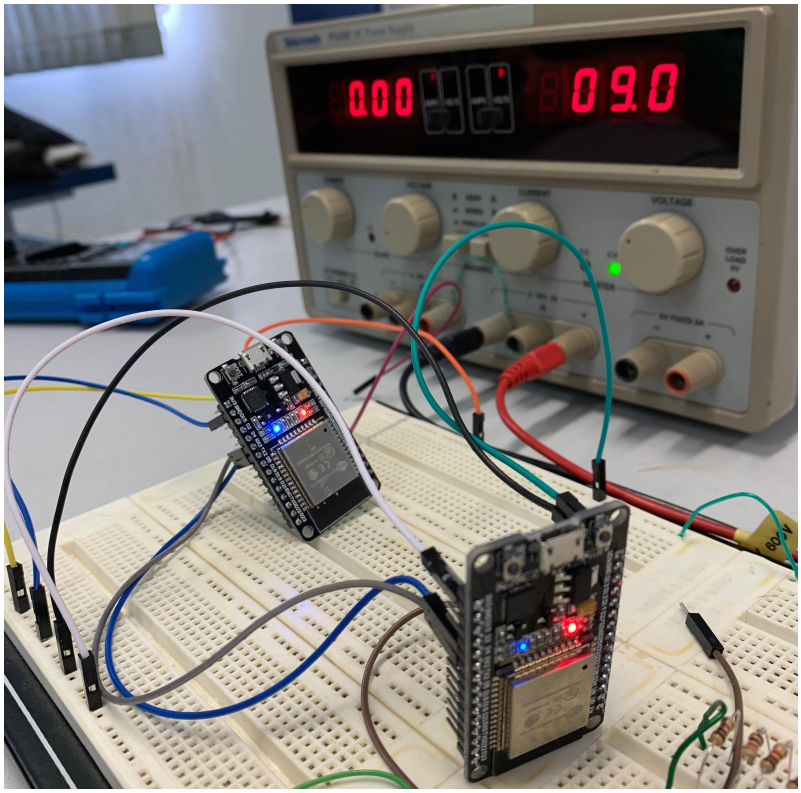
Fonte: Autoria própria

Além disso, realizou-se um teste alimentando os ESPs com uma fonte de alimentação de 9V, ao invés de alimentá-los com o cabo USB. Para isto, além de conectar as portas RX e TX dos módulos (RX de um no TX do outro, e vice-versa), os pinos GND (Ground) foram

aterrados. Observou-se, também, que os pinos de Enable não precisam ser conectados a nenhuma tensão e que, caso sejam conectados a tensões altas (acima de 3V), pode ocorrer um curto-circuito deste pino com o regulador de 3.3V do ESP, impossibilitando o uso do módulo. Assim, os pinos EN foram deixados soltos.

O teste foi realizado com sucesso, comprovando que o sincronismo e a transmissão e recepção de dados dos microcontroladores funcionariam quando os mesmos estivessem soldados nas placas de aquisição, as quais seriam alimentadas por uma bateria de 9V. A figura 26 mostra os ESPs em sincronismo, no momento em que seus LEDs azuis estão acesos.

Figura 26: Teste do código dos ESPs quando alimentados por fonte de 9V



Fonte: Autoria própria

Por fim, outro teste realizado com os ESPs foi utilizando um cabo manga de 2,5m de comprimento para ligar suas portas RX e TX, ao invés de jumpers pequenos. Utilizou-se o cabo para realizar os mesmos testes acima, com sucesso. Desta forma, demonstrou-se que o sincronismo e transmissão de dados entre os ESPs funcionariam quando cada módulo estivesse em uma placa, presa em cada perna do indivíduo, com o cabo de 2,5m conectando os módulos entre si.

6.2 Placa de aquisição

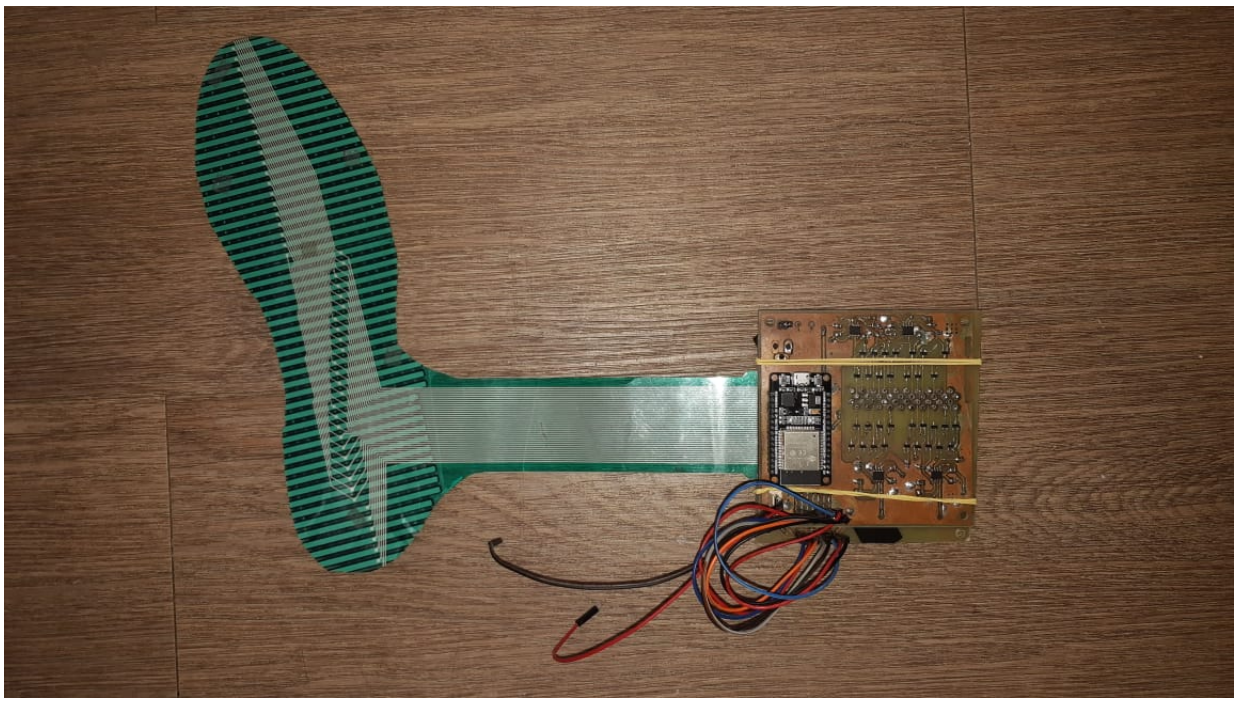
A estratégia adotada para a verificação do funcionamento da placa de aquisição foi a de manufaturar somente uma das placas (pé direito ou esquerdo) e testá-la primeiramente, para, após a validação dos resultados iniciais, manufaturar a segunda placa.

Após os testes no código do ESP32 e a manufatura da placa, programou-se o microcontrolador para que ele funcionasse de acordo e realizou-se todos os testes de conectividade da placa necessários. Contatou-se que estava tudo corretamente ligado.

Para o teste, alimentou-se o ESP através de seu cabo micro USB, para facilitar caso fosse necessário programá-lo e também garantir o funcionamento com uma tensão adequada.

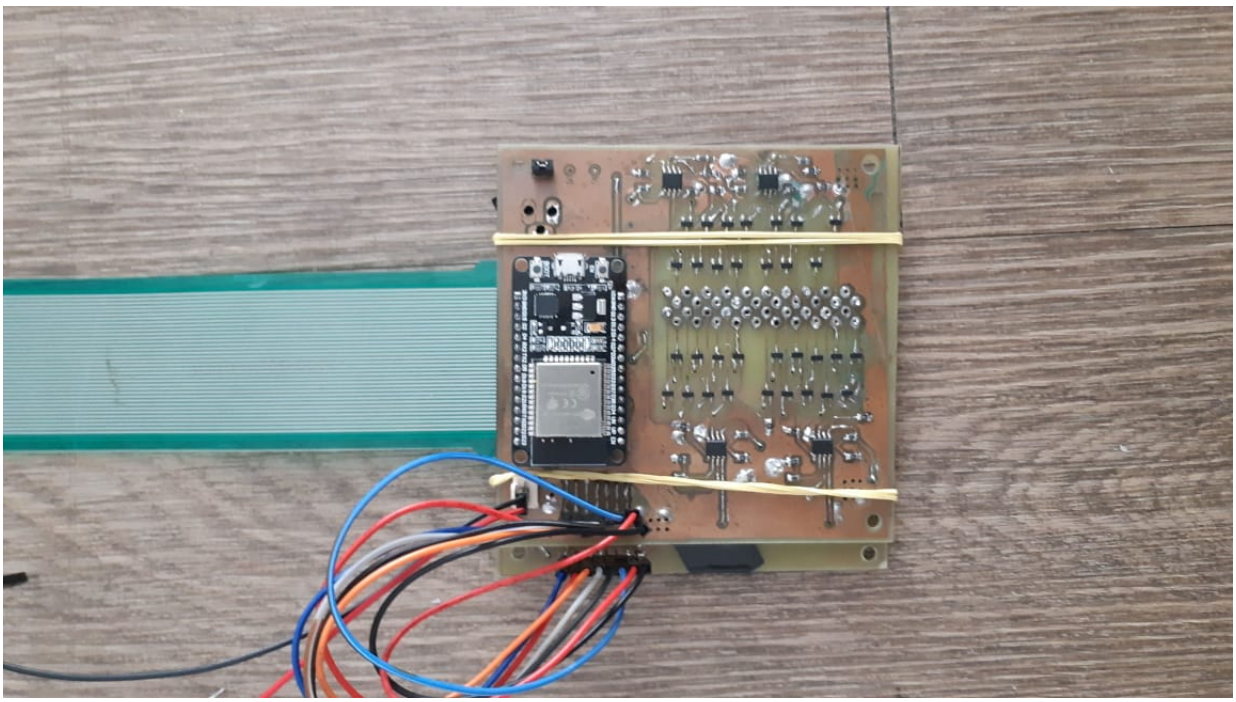
Ao manufaturar a placa, houve um deslocamento dos conectores que fazem contato com a palmilha e, dessa forma, a placa superior não ficou completamente alinhada com a placa inferior. Isso pode ter ocorrido devido a algum erro de distanciamento no esquemático ou à geração do arquivo .pdf para impressão da placa, que pode acarretar em um deslocamento dos componentes. Para tentar corrigir isso, pensou-se em criar um adaptador em uma impressora 3D, porém descartou-se essa possibilidade. A estratégia, então, foi deslocar uma das placas e tentar fazer o alinhamento dos contatos à olho nu e, após isso, envolveu-se as placas com elásticos para fixar os contatos, já que os 4 furos nos cantos das placas não poderiam ser utilizados devido ao desalinhamento. A placa final já conectada pelos jumpers, com os elásticos de sustentação e acoplada a palmilha está representada na figura

Figura 27: Placa de aquisição acoplada à palmilha



Fonte: Autoria própria

Figura 28: Placa de aquisição acoplada à palmilha



Fonte: Autoria própria

Após fazer o alinhamento, ligou-se os dois ESPs através do cabo micro-USB e verificou-se todas as portas e ligações para verificar se as tensões estavam chegando corretamente e aguardou que os microcontroladores sincronizassem. No entanto a sincronização não se realizou.

Como todos os testes de sincronização antes da manufatura da placa funcionaram, percebeu-se que alguma coisa tinha acontecido com o ESP e, então, foi feito o debug do código parte por parte. Ao realizar o debug do código notou-se que o problema estava com a comunicação Bluetooth, ou seja, quando o código chegava na parte da comunicação Bluetooth ele parava de funcionar e dava um erro inesperado. Ao analisar as portas seriais percebeu-se que a IDE retornava o erro da figura 29. Ao pesquisar a respeito do erro encontrou-se que era um erro já reportado por diversos usuários do ESP32 e estava ligado a memória FLASH do ESP.

Figura 29: Erro da memória Flash

```
rst:0xc (SW_CPU_RESET),boot:0x13
(SPI_FAST_FLASH_BOOT)
```

Fonte: Autoria própria

Esse erro, segundo diversos relatos em fóruns é um erro que acontece com frequência mas que ninguém até hoje descobriu o porquê dele. No entanto, haviam algumas possibilidades para solucionar esse problema.

A primeira coisa a se fazer era retirar o ESP32 da placa de aquisição e verificar se todas as suas portas estavam funcionando corretamente e se não havia nenhum curto no microcontrolador, que poderia ter sido causado no momento da solda, já que é feita em alta temperatura. Após a constatação de que o microcontrolador estava em perfeitas condições tentou-se a segunda solução que era fazer o upload do código alterando o *partition scheme* no momento da programação. Porém essa solução não funcionou.

Seguindo novamente os relatos de fóruns, realizou-se a limpeza da memória flash do ESP32 através do comando "erase_flash". Este comando solucionou o problema, fazendo com que o ESP32 voltasse a funcionar normalmente.

Soldou-se então, novamente, o ESP32 na placa e após isso, ligou-se os dois ESPs e foi constatado que elas sincronizaram perfeitamente.

Ao fazer a conexão do ESP32 com o código Java, notou-se que quando a palmilha era pressionada, os dados que apareciam na tela estavam com uma tensão muito baixa, aparecendo com um tom de vermelho fraco, quase branco, além de que quase não havia alteração.

Supôs-se, então, que havia algo de errado com a manufatura da placa, como: tamanho das trilhas, algum curto circuito imperceptível, solda não muito bem aplicada e diversos outros fatores que uma manufatura de placa manual poderia ocasionar.

Para validar que o código do ESP32 estava funcionando, fez-se a aplicação de uma tensão de 3.3V nas entradas dos conversores A/D do ESP32. Com essa aplicação de tensão, obteve-se as figuras 30, 31, 32, 33, 34 e 35.

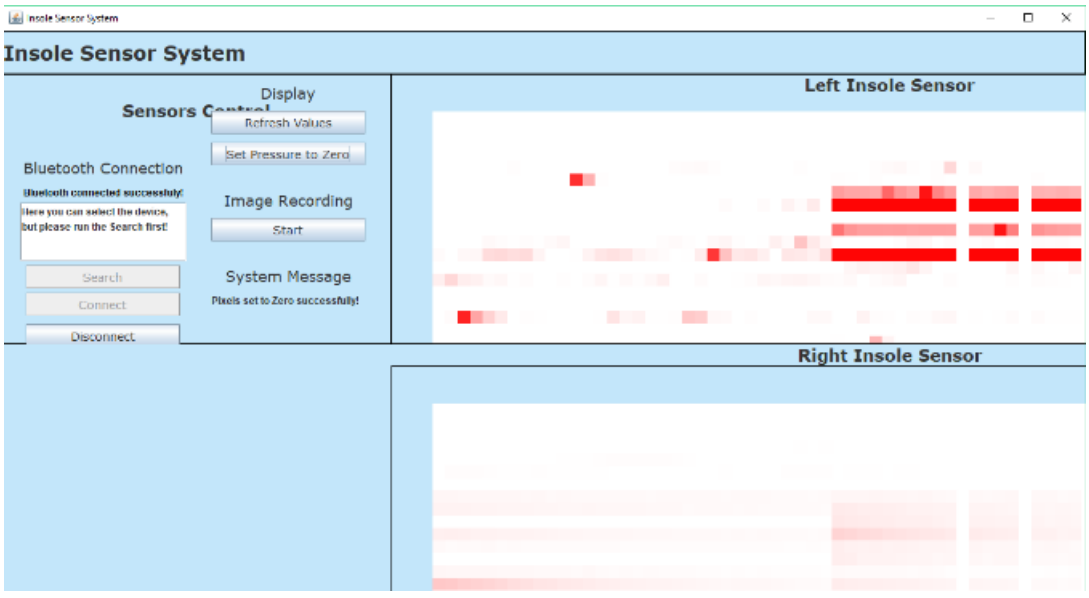
A seguir, será exemplificada a aplicação realizada com a figura 30, obtida colocando-se tensão de 3.3V no pino 33 do ESP32. Tal pino corresponde ao sexto conversor AD utilizado, conforme definido no vetor de ADCs no código do microcontrolador. Assim, conforme mapeamento da figura 13, tal conversor lê os dados das colunas 5, 6, 9 e 15 da região dos dedos da palmilha. Assim, a figura obtida comprova que a leitura do ADC correspondia ao esperado, já que na aplicação Java visualizou-se tons escuros de vermelho (tensão alta, 3.3V) na posição das colunas correspondentes.

Para os outros pinos ADCs, também obteve-se resultados satisfatórios. Com essa constatação, percebeu-se que o circuito estava enviando um sinal muito baixo para os pinos A/D, pois apenas ao colocar-se diretamente a tensão de 3.3V nos pinos, obteve-se os sinais desejados e sua visualização na aplicação Java.

Figura 30: Imagens de aquisição da palmilha com 3.3V no pino 33 do terminal A/D - em verde, mapeamento em relação à palmilha

Fonte: Autoria própria

Figura 31: Imagens de aquisição da palmilha com 3.3V no pino 32 do terminal A/D



Fonte: Autoria própria

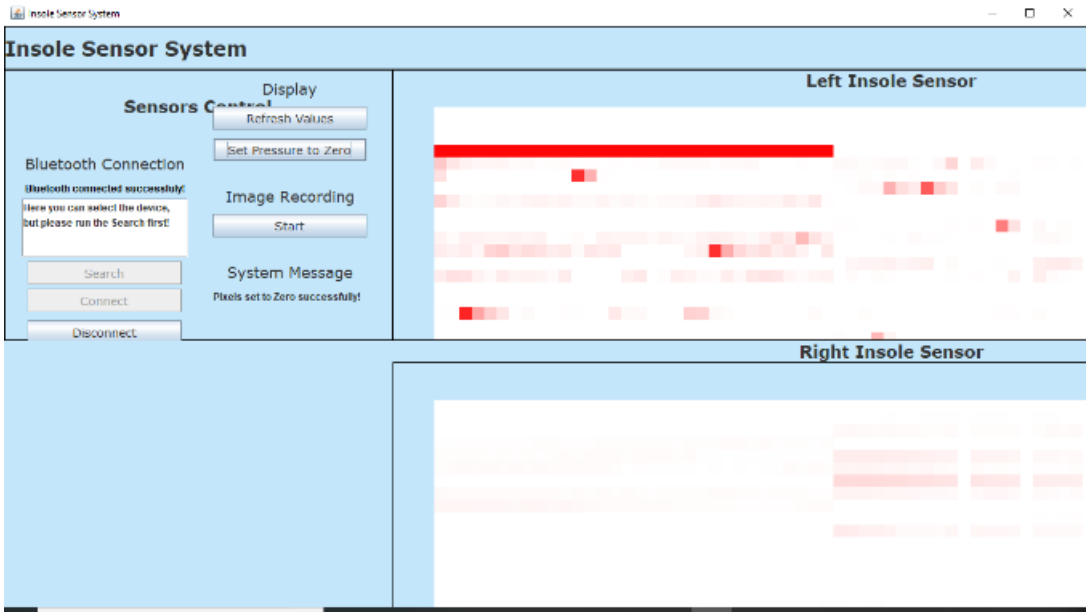
Figura 32: Imagens de aquisição da palmilha com 3.3V no pino 34 do terminal A/D

Fonte: Autoria própria

Figura 33: Imagens de aquisição da palmilha com 3.3V no pino 35 do terminal A/D

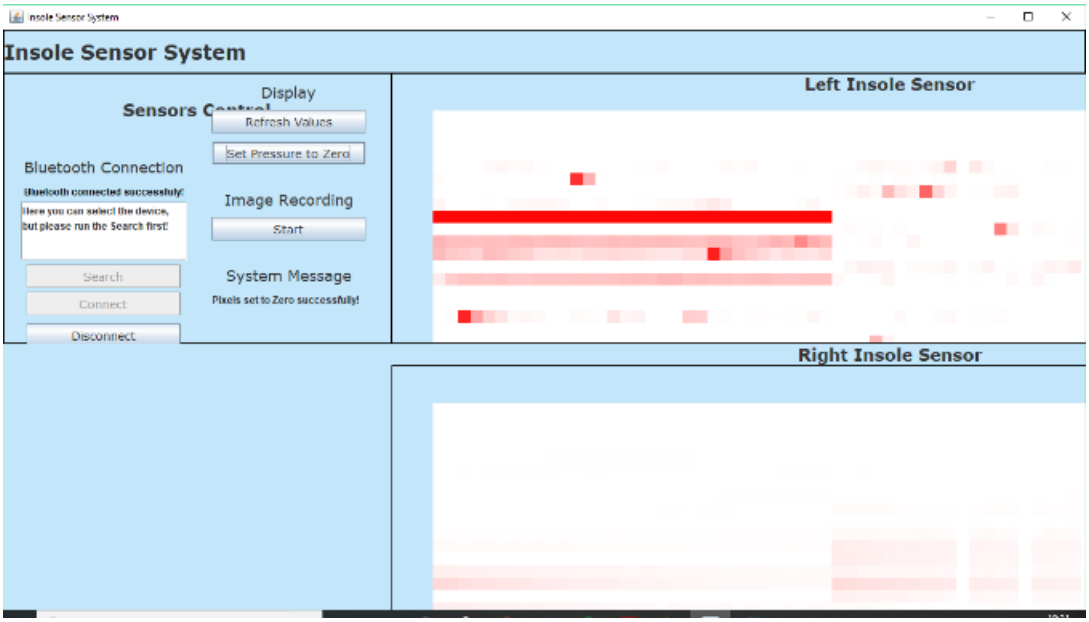
Fonte: Autoria própria

Figura 34: Imagens de aquisição da palmilha com 3.3V no pino 36 do terminal A/D



Fonte: Autoria própria

Figura 35: Imagens de aquisição da palmilha com 3.3V no pino 39 do terminal A/D



Fonte: Autoria própria

7 CONCLUSÃO

Os testes realizados demonstram que a programação dos ESP32s e a aplicação Java estão em pleno funcionamento. Cada módulo ESP está programado para realizar o controle dos demultiplexadores e FETs N de sua placa de aquisição correspondente, captando os dados dos sensores de sua palmilha. O ESP slave envia seus dados captados para o master por um cabo de comprimento 2,5m, que sai de sua placa em um tornozelo, passa pelas pernas e cintura do indivíduo e chega na placa no outro tornozelo, em que o master está localizado. O master, então, envia com sucesso os dados de ambas palmilhas via Bluetooth, os quais são visualizados em tempo real na tela da aplicação Java, com uma frequência de 3 a 4 imagens por segundo. Todas as imagens visualizadas são salvadas em uma pasta do computador em que a aplicação Java se encontra. Assim, demonstrou-se que os microcontroladores e a aplicação Java estão prontos para serem utilizados no sistema completo, na aplicação real de uso com pacientes.

Conforme mencionado na na seção 6.2, a placa de aquisição física não funcionou de acordo com o esperado. Dentre os motivos observados, destaca-se como principais causas possíveis:

- Mal contato dos conectores de linhas e colunas da palmilha com a placa de aquisição;
- Mal contato ou falta de contato entre trilhas e componentes, podendo ser causados por:
  - Soldas não funcionando corretamente;
  - Componentes defeituosos;
  - Erros de continuidade nos conectores;
- Trilhas pequenas, podendo causar curto circuitos.

A placa do circuito de aquisição foi realizada de forma manual e prensada, método o qual, nos dias atuais, é considerado rudimentar. Portanto, para sanar esses problemas, recomenda-se fabricar uma placa de aquisição com materiais e métodos mais precisos, automáticos e tecnológicos, fazendo com que diminua as falhas humanas de solda e prensa, permitindo um circuito mais preciso e com menos erros, mesmo com muitos componentes e trilhas próximas entre si.

Apesar do não funcionamento devido a manufatura, o circuito projetado tem um custo muito inferior a um sistema similar chamado F-Scan™(TEKSCAN,), oferecido pela Tekscan, empresa que fornece as palmilhas utilizadas neste projeto. O sistema F-Scan tem a proposta muito semelhante a deste projeto, fornecendo o mapeamento em tempo real

da distribuição da pressão plantar obtida pelas palmilhas. Tal sistema é avaliado em US$ 13.295, equivalente a aproximadamente R$ 54.000. Com isso, nota-se que, caso se fabrique a placa desse projeto utilizando materiais mais robustos, o preço final do projeto ficara muito abaixo do vendido pela empresa Tekscan, oque é um benefício para o projeto, já que o objetivo é que ele funcione e tenha um custo baixo para os usuários.

Uma sugestão de melhoria seria a implementação no código Java de funcionalidades extras para auxílio na análise da distribuição das pressões plantares, tais como:

- Gráfico de pressão média em dado intervalo de tempo;
- Centro de pressão na planta do pé;
- Identificação do momento atual da marcha (contato inicial com o solo, fase pré-oscilatória, oscilação);
- Escala de cores diversificada para identificação de pressões.

Por fim, outra sugestão seria encontrar alguma outra forma de comunicar um ESP32 com o outro que não seja comunicação serial via UART, tornando a aplicação totalmente sem fios. Para tanto, poderia ser avaliada a utilização do Wi-fi integrado na placa ou o envio por Bluetooth, em que seria necessário a possibilidade do ESP master conectar-se primeiramente com um dispositivo, que seria o slave, e posteriormente com outro, a aplicação Java no computador.

REFERÊNCIAS

BRASIL. Ministério da Saúde. **Diretrizes de Atenção à Pessoa com Lesão Medular**. Brasília: Ministério da Saúde, 2013. 7-8 p. Disponível em: <http://bvsms.saude.gov.br/bvs/publicacoes/diretrizes_atencao_pessoa_lesao_medular.pdf>.

CLIQUET, A.; ABREU, D. C. de; RONDINA, J. M.; CENDES, F. Electrical stimulation during gait promotes increase of muscle cross-sectional area in quadriplegics: a preliminary study. **Clinical orthopaedics and related research**, p. 553–557, 02 2009. Disponível em: <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2628524/>.

CLIQUET, A.; CARVALHO, D.; GARLIPP, C.; BOTTINI, P.; AFAZ, S.; MODA, M. Effect of treadmill gait on bone markers and bone mineral density of quadriplegic subjects. **Brazilian Journal of Medical and Biological Research**, scielo, v. 39, p. 1357 – 1363, 10 2006. ISSN 0100-879X. Disponível em: <http://www.scielo.br/scielo.php?script=sci_arttext&pid=S0100-879X2006001000012&nrm=iso>.

CRITTER, M. Sistema de transdutores para dispositivo de auxilio a marcha de tetraplegicos e paraplegicos. p. 1–63, 06 2015. Disponível em: <http://www.tcc.sc.usp.br/tce/disponiveis/18/180450/tce-03022016-155816/?&lang=br>.

DATASHEET ESP32-WROOM-32. Espressif Systems. Disponível em: <https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32_datasheet_en.pdf>. Acesso em: 20 mar. 2019.

F-SCAN System. Tekscan. Disponível em: <https://www.tekscan.com/products-solutions/systems/f-scan-system>. Acesso em: 15 mai. 2019.

GETTING Started with the ESP32 Development Board. Random Nerd Tutorials, 2016. Disponível em: <https://randomnerdtutorials.com/getting-started-with-esp32/>. Acesso em: 10 mar. 2019.

RAZAK, A. H.; ZAYEGH, A.; BEGG, R. K.; WAHAB, Y. Tfoot plantar pressure measurement system: a review. **Sensors (Basel, Switzerland)**, London, p. 1–29, 2012.

VAROTO, R.; OLIVEIRA, G. C.; LIMA, A. V. F. de; CRITTER, M. M.; CLIQUET, A. A low cost wireless system to monitor plantar pressure using insole sensor: Feasibility approach. 2017.

Apêndices

APÊNDICE A – CÓDIGO ESP32 - SLAVE

```
#include "Arduino.h"

// Controle dos circuitos externos
int dmux_crtl[4] = {21, 19, 18, 5}; //Pinos D-C-B-A do demux
const int d_en1 = 4;
const int d_en2 = 15;
int nmos_crtl[4] = {26, 14, 12, 13}; //NMOS 1-2-3-4
int adc_crtl[8] = {36, 39, 34, 35, 32, 33, 25, 27};
// Serial 17-16 (TX-RX)

// Controle da memória
const int colunas[42]={0x00,0x00,0x00,0x71,0x00,0x63,0x01,0x62,0x02,0x54,
         0x03,0x52,0x74,0x44,0x04,0x43,0x11,0x51,0x73,0x42,
         0x12,0x34,0x13,0x41,0x72,0x33,0x14,0x32,0x21,0x53,
         0x64,0x31,0x22,0x24,0x00,0x61,0x00,0x23,0x00,0x00,
         0x00,0x00};

const int linhas[60]={0x06,0x04,0x02,0x00,0x09,0x0A,0x0C,0x0E,0x0F,0x0D,
          0x10,0x12,0x11,0x13,0x14,0x0B,0x15,0x16,0x17,0x18,
          0x08,0x19,0x01,0x1A,0x1B,0x03,0x1C,0x05,0x1D,0x1E,
          0x1F,0x07,0x0F,0x0D,0x10,0x12,0x11,0x13,0x14,0x0B,
          0x15,0x16,0xFF,0x17,0x18,0x08,0x19,0xFF,0x01,0x1A,
          0x1B,0x03,0xFF,0x1C,0x05,0x1D,0x1E,0xFF,0x1F,0x07};

int selec_demux(int linha);
int captura_dados(int coluna);

const int led_verif = 2;
void verif_sync();
char verificador;

void setup() {
  analogReadResolution(9);


  //Setup serial
```

```
  pinMode(led_verif, OUTPUT);
  digitalWrite(led_verif, LOW);
  Serial.begin(115200); // Essa serial utiliza os pinos 1 e 3 (USB)
  Serial2.begin(115200); // Essa serial utiliza os pinos 16 e 17 (ESP-ESP)

  // DEBUG
  delay(3000);


  // Setup controle do demux
  for(int i = 0; i < 4; i++){
    pinMode(dmux_crtl[i], OUTPUT);
    digitalWrite(dmux_crtl[i], HIGH);
  }
  //Setup do enables (desativando os demux)
  pinMode(d_en1, OUTPUT);
  digitalWrite(d_en1, HIGH);
  pinMode(d_en2, OUTPUT);
  digitalWrite(d_en2, HIGH);

  //Setup dos pinos do nmos e desativação dos mesmos
  for(int i = 0; i < 4; i++){
    pinMode(nmos_crtl[i], OUTPUT);
    digitalWrite(nmos_crtl[i], LOW);
  }
  //Setup dos pinos responsáveis pela leitura de dado
  for(int i = 0; i < 8; i++){
    pinMode(adc_crtl[i], INPUT);
  }
}

void loop() {
  int val;
  int verif;
  int c;
  char data2[1261];
  int count = 0;

  //Varredura linhas e colunas
```

```
c = 1;
for(int i = 0; i  < 42; i++){
  if((i%2) == 0){
    for (int j = 0; j < 32; j++){
      verif = selec_demux(linhas[j]);
      if(verif != 500){
        val = captura_dados(colunas[i]);
        if(val > 250){
          val = 250;
        }
      } else {
        val = 0;
      }
      data2[c] = val;
      c++;
    }
  } else {
    for (int j = 32; j < 60; j++){
      verif = selec_demux(linhas[j]);
      if(verif != 500){
        val = captura_dados(colunas[i]);
        if(val > 250){
          val = 250;
        }
      } else {
        val = 0;
      }
      data2[c] = val;
      c++;
    }
  }
}
data2[0] = 254; //start byte slave

verif_sync();

//Envio de dados para ESP master
for(int i = 0; i < 1261; i++){
  Serial2.print(data2[i]);
```

```
  }

  //Serial.println("Vetor enviado");
  digitalWrite(led_verif, LOW);

}

//Seleção dos demux
int selec_demux(int linha){
  int aux_msb;
  int aux_lsb;
  int stack = 8;
  //Controle dos demux
  //Determinar pela tabela qual demux está operando
  aux_msb = (linha & 0xF0) >> 4;
  if(aux_msb == 0x00){
    digitalWrite(d_en1, LOW);
    digitalWrite(d_en2, HIGH);
  } else if(aux_msb == 0x01){
    digitalWrite(d_en1, HIGH);
    digitalWrite(d_en2, LOW);
  }
  //Verificando se a posição é uma posição válida, se for executa a seleção
  if(aux_msb == 0x0F){
    return 500;
  } else {
    aux_lsb = linha & 0x0F;
    for(int k = 0; k < 4; k++){
      if(aux_lsb >= stack){
        digitalWrite(dmux_crtl[k], HIGH);
        aux_lsb = aux_lsb - stack;
      } else {
        digitalWrite(dmux_crtl[k], LOW);
      }
      stack = stack / 2;
    }
  }
  return 0;
}
```

```
//Seleção dos NMOS e leitura dos dados
int captura_dados(int coluna){
  int val;
  int aux_msb;
  int aux_lsb;
  //Controle dos NMOS e leitura de dados
  aux_lsb = (coluna & 0x0F);
  if(aux_lsb != 0){
    aux_lsb--;
    digitalWrite(nmos_crtl[aux_lsb], HIGH);
    aux_msb = ((coluna & 0xF0) >> 4);
    val = analogRead(adc_crtl[aux_msb])/2;
    //delay(50); //(!!) - Verificar se esse delay é alto/baixo
    digitalWrite(nmos_crtl[aux_lsb], LOW);
    return val;
  } else {
    // Retorno de erro caso seja uma posição vazia
    return 0;
  }
}

//Sincronização ESPs
void verif_sync(){
  verificador = 'e';
  while(verificador != 's'){ // Espera receber a palavra de sincronismo do slave
    verificador = Serial2.read(); // Lê a mensagem recebida
  }
  Serial2.print('s');
  digitalWrite(led_verif, HIGH);
  return;
}
```

APÊNDICE B – CÓDIGO ESP32 - MASTER

```
#include "Arduino.h"
#include "BluetoothSerial.h"

BluetoothSerial SerialBT;

// Controle dos circuitos externos
int dmux_crtl[4] = {21, 19, 18, 5}; //Pinos D-C-B-A do demux
const int d_en1 = 4;
const int d_en2 = 15;
int nmos_crtl[4] = {26, 14, 12, 13};// NMOS 1-2-3-4
int adc_crtl[8] = {36, 39, 34, 35, 32, 33, 25, 27};
// Serial 17-16 (TX-RX)

// Controle da memória
const int colunas[42]={0x00,0x00,0x00,0x71,0x00,0x63,0x01,0x62,0x02,0x54,
         0x03,0x52,0x74,0x44,0x04,0x43,0x11,0x51,0x73,0x42,
         0x12,0x34,0x13,0x41,0x72,0x33,0x14,0x32,0x21,0x53,
         0x64,0x31,0x22,0x24,0x00,0x61,0x00,0x23,0x00,0x00,
         0x00,0x00};

const int linhas[60]={0x06,0x04,0x02,0x00,0x09,0x0A,0x0C,0x0E,0x0F,0x0D,
          0x10,0x12,0x11,0x13,0x14,0x0B,0x15,0x16,0x17,0x18,
          0x08,0x19,0x01,0x1A,0x1B,0x03,0x1C,0x05,0x1D,0x1E,
          0x1F,0x07,0x0F,0x0D,0x10,0x12,0x11,0x13,0x14,0x0B,
          0x15,0x16,0xFF,0x17,0x18,0x08,0x19,0xFF,0x01,0x1A,
          0x1B,0x03,0xFF,0x1C,0x05,0x1D,0x1E,0xFF,0x1F,0x07};

int selec_demux(int linha);
int captura_dados(int coluna);

const int led_verif = 2;
void verif_sync();
char verificador;

void setup() {
  analogReadResolution(9);
```

```
  //Setup serial
  pinMode(led_verif, OUTPUT);
  digitalWrite(led_verif, LOW);
  Serial.begin(115200); // Essa serial utiliza os pinos 1 e 3 (USB)
  Serial2.begin(115200); // Essa serial utiliza os pinos 16 e 17 (ESP-ESP)

  //Setup Bluetooth
  SerialBT.begin("ESP32");

  // DEBUG
  delay(3000);

  //Setup controle do demux
  for(int i = 0; i < 4; i++){
    pinMode(dmux_crtl[i], OUTPUT);
    digitalWrite(dmux_crtl[i], HIGH);
  }
  //Setup do enables (desativando os demux)
  pinMode(d_en1, OUTPUT);
  digitalWrite(d_en1, HIGH);
  pinMode(d_en2, OUTPUT);
  digitalWrite(d_en2, HIGH);

  //Setup dos pinos do nmos e desativação dos mesmos
  for(int i = 0; i < 4; i++){
    pinMode(nmos_crtl[i], OUTPUT);
    digitalWrite(nmos_crtl[i], LOW);
  }
  //Setup dos pinos responsáveis pela leitura de dado
  for(int i = 0; i < 8; i++){
    pinMode(adc_crtl[i], INPUT);
  }
}

void loop() {
  int val;
  int verif;
```

```
int c;
char data1[1261];
char data2[1261];
int count = 0;

//Varredura linhas e colunas
c = 1;
for(int i = 0; i  < 42; i++){
  if((i%2) == 0){
    for (int j = 0; j < 32; j++){
      verif = selec_demux(linhas[j]);
      if(verif != 500){
        val = captura_dados(colunas[i]);
        if(val > 250){
          val = 250;
        }
      } else {
        val = 0;
      }
      data1[c] = val;
      c++;
    }
  } else {
    for (int j = 32; j < 60; j++){
      verif = selec_demux(linhas[j]);
      if(verif != 500){
        val = captura_dados(colunas[i]);
        if(val > 250){
          val = 250;
        }
      } else {
        val = 0;
      }
      data1[c] = val;
      c++;
    }
  }
}
data1[0] = 255; //start byte master
```

```
  verif_sync();

  //Recebimento dos dados da ESP slave
  while(count < 1261){
    if(Serial2.available() > 0){
      data2[count] = Serial2.read();
      count++;
    }
  }

  //Serial.println("Vetor recebido");
  digitalWrite(led_verif, LOW);

  //Envio dos dados da master via Bluetooth
  for(int i = 0; i < 1261; i++){
    SerialBT.print(data1[i]);
  }

  //Envio dos dados da slave via Bluetooth
  for(int i = 0; i < 1261; i++){
    SerialBT.print(data2[i]);
  }
}

//Seleção dos demux
int selec_demux(int linha){
  int aux_msb;
  int aux_lsb;
  int stack = 8;
  //Ontrole dos demux
  //Determinar pela tabela qual demux está operando
  aux_msb = (linha & 0xF0) >> 4;
  if(aux_msb == 0x00){
    digitalWrite(d_en1, LOW);
    digitalWrite(d_en2, HIGH);
  } else if(aux_msb == 0x01){
    digitalWrite(d_en1, HIGH);
    digitalWrite(d_en2, LOW);
```

```
  }
  //Verificando se a posição é uma posição válida, se for executa a seleção
  if(aux_msb == 0x0F){
    return 500;
  } else {
    aux_lsb = linha & 0x0F;
    for(int k = 0; k < 4; k++){
      if(aux_lsb >= stack){
        digitalWrite(dmux_crtl[k], HIGH);
        aux_lsb = aux_lsb - stack;
      } else {
        digitalWrite(dmux_crtl[k], LOW);
      }
      stack = stack / 2;
    }
  }
  return 0;
}

//Seleção dos NMOS e leitura dos dados
int captura_dados(int coluna){
  int val;
  int aux_msb;
  int aux_lsb;
  //Controle dos NMOS e leitura de dados
  aux_lsb = (coluna & 0x0F);
  if(aux_lsb != 0){
    aux_lsb--;
    digitalWrite(nmos_crtl[aux_lsb], HIGH);
    aux_msb = ((coluna & 0xF0) >> 4);
    val = analogRead(adc_crtl[aux_msb])/2;
    //delay(50); //(!!) - Verificar se esse delay é alto/baixo
    digitalWrite(nmos_crtl[aux_lsb], LOW);
    return val;
  } else {
    // Retorno de erro caso seja uma posição vazia
    return 0;
  }
}
```

```
//Sincronização ESPs
void verif_sync(){
  verificador = 'e'; // empty
  //Serial.println('s');
  Serial2.print('s');

  while(verificador != 's'){ // Espera receber a palavra de sincronismo do slave
    verificador = Serial2.read(); // Lê a mensagem recebida
    //Serial2.print('s');
  }
  digitalWrite(led_verif, HIGH);
  //Serial.println("ESPs sincronizadas");
  return;
}
```

APÊNDICE C – CÓDIGO JAVA - JFRAME2

O código abaixo é o principal da aplicação Java, responsável por criar toda a interface gráfica, e utiliza o código "Bluetooth2.java", descrito no apêndice D.

```
package insolesensor;

// Import libraries
import InsoleBluetooth.Bluetooth2;
import java.awt.Color;
import java.awt.Graphics;
import java.awt.image.BufferedImage;
import java.io.File;
import javax.imageio.ImageIO;
import javax.swing.BorderFactory;
import javax.swing.DefaultListModel;
import javax.swing.JList;
import javax.swing.JPanel;
import java.awt.Dimension;
import javax.swing.border.Border;
//NOVO
/**
 * JFrame: specifies all the methods and components that will be
 * available in the program's GUI (graphic user interface).
 */
public class JFrame2 extends javax.swing.JFrame{

    /**
     * Arrays to hold the sensor's current pressure values.
     */
    public int[][] pressure;

    /**
     * Arrays to hold the sensor's reference pressure values.
     */
    public int[][] reference;
```

```
    /**
     * Lists to hold strings that will be displayed in the scroll pane.
     */
    public DefaultListModel listModel;

    // Java Swing: an API (set of subrotine definitions, protocols and tools)
    // for building a GUI in Java.

    // Create JButton fields for the GUI. A JButton is an implementation
    //of a "push" button.
    // Buttons to control 1st insole sensor:
    private javax.swing.JButton jButton11;  // "Search" button.
    private javax.swing.JButton jButton12;  // "Connect" button.
    private javax.swing.JButton jButton13;  // "Disconnect" button.
    private javax.swing.JButton jButton14;  // "Refresh Values" button.
    private javax.swing.JButton jButton15;  // "Set Pressure to Zero" button.
    private javax.swing.JButton jButton16; //"Start/Stop" image recording button

    /**
     Displays messages to the user about current program status (1st insole sensor)
     */
    public javax.swing.JLabel userMessage1;

    // Create JLabel fields for the GUI. A JLabel is an area that can
    //display an image, a short text string, or both.
    //Does not react to input events.
    private javax.swing.JLabel jLabel1; // "Insole Sensor System" title.

    // Labels for buttons and sensor information regarding 1st insole sensor:
    private javax.swing.JLabel jLabel10; //Label for 1st insole sensor.
    private javax.swing.JLabel jLabel11; //Label for buttons that control
                                            // Bluetooth connection.
    private javax.swing.JLabel jLabel12;    // Displays BT connection info.
    private javax.swing.JLabel jLabel13; //Label for control display buttons.
    private javax.swing.JLabel jLabel14; //Label for system message.
    private javax.swing.JLabel jLabel15; //Label for image recording.

// Labels for describing which  insole sensor:
    private javax.swing.JLabel jLabel16; //Label for 1st (left) insole sensor
```

```
private javax.swing.JLabel jLabel17; //Label for 2nd (right) insole sensor



/**
Hold messages and name of devices that the user can connect to; 1st insole sens
 */
public javax.swing.JList jList1;


// Create JPanel fields for the GUI. A JPanel is a general-purpose
container to hold components in a GUI.
private javax.swing.JPanel jPanel1; //System name
private javax.swing.JPanel jPanel2; //Panel for 1st insole sensor region.
private javax.swing.JPanel jPanel3;  //Controls for 1st insole sensor.
private javax.swing.JPanel jPanel4; //Image of 1st insole sensor.
private javax.swing.JPanel jPanel7; //Image of 2nd insole sensor.

// Create a JScrollPane field for the GUI. A JScrollPane provides a
//scrollable view of a component. It's used to display components
that are large or whose size can change dinamically.
// Will display messages and names of Bluetooth devices
//that the user can connect to.
private javax.swing.JScrollPane jScrollPane1;   // 1st insole sensor.

// Time delay between saved images (in milliseconds).
private static final int DELAY = 100;
// Create variable to keep track of the number of images saved.
private long[] counter;
// Create variables to keep track of time (used to save images).
private long[] previousTime;
private long[] currentTime;
// Holds user decision to enable/disable image recording.
private boolean[] imgRecording;


// Threads: A Thread is a thread of execution in a program. Thread objects
// are used in Java to allow parallel processing.
```

```
/**
 * GUI's main thread, used to update pixels representing the sensors.
 */
public Thread tmain1;
public Thread tmain2;
// bluetooth threads, used to read values coming from the Bluetooth devices.
Thread bt1;
Thread bt2;

/**
 * Object of the InsoleBluetooth.Bluetooth2 class that handles
 bluetooth communication.
 */
public Bluetooth2 bluetooth1;
public Bluetooth2 bluetooth2;

/**
 * Images that will be displayed in the GUI, representing
 the pressure values.
 */
public BufferedImage[] pressureImage;

/**
 * Used to create objects of the insolesensor.JFrame class.
 */
public JFrame2(){

    // Call the constructor of javax.swing.JFrame class and set window's title.
    super("Insole Sensor System");
    // Initialize pressure data vectors.
    pressure = new int[2][1260];
    reference = new int[2][1260];
    // Initialize list of strings that will be displayed in the GUI.
    listModel = new DefaultListModel();
    // Initialize image objects.
    pressureImage = new BufferedImage[2];
    pressureImage[0] = new BufferedImage(60 * 15, 21 * 15,
    BufferedImage.TYPE_INT_ARGB);
    pressureImage[1] = new BufferedImage(60 * 15, 21 * 15,
```

```
        BufferedImage.TYPE_INT_ARGB);
        // Initialize image counter.
        counter = new long[2];
        counter[0] = 0;
        counter[1] = 0;
        // Initialize time variables and capture start time.
        previousTime = new long[2];
        currentTime = new long[2];
        previousTime[0] = System.currentTimeMillis();
        previousTime[1] = System.currentTimeMillis();
        // Initialize image recording decision variables;
        imgRecording = new boolean[2];
        imgRecording[0] = false;
        imgRecording[1] = false;
        // Check to see if directory to save image files exists.
        File dir1 = new File("C:\\InsoleSensorSystem/LeftSensor");
        File dir2 = new File("C:\\InsoleSensorSystem/RightSensor");
        if(!dir1.exists())   dir1.mkdirs();
        if(!dir2.exists())   dir2.mkdirs();
        // Initialize components of the JFrame.
        initComponents();
// Re-sizing the GUI?
        this.setSize(1518,1000);
        this.setResizable(true);
// Create threads using jPanel4 and jPanel7 components.
This means that when this
// thread is started, the "run" method of these components will be invoked.
        // In this case, the "run" method updates pixels representing
        the sensors.
        tmain1 = new Thread((Runnable) jPanel4);
        tmain2 = new Thread((Runnable) jPanel7);
// Remove all elements from listModel.
        listModel.clear();
// Add two string messages to the end of listModel, instructions to the user
        // that will be displayed in the scroll panel.
        listModel.addElement("Here you can select the device.");
        listModel.addElement("Please run the Search first!");
        // Initialize bluetooth object.
        bluetooth1 = new Bluetooth2(this, (byte) 0);
```

```
        //bluetooth2 = new Bluetooth2(this, (byte) 1);

    } // End JFrame2 constructor.

    // Initialize components of the JFrame. Configure the GUI's layout.
    private void initComponents(){

        // Create objects for each field of the JFrame.
        jButton11 = new javax.swing.JButton();
        jButton12 = new javax.swing.JButton();
        jButton13 = new javax.swing.JButton();
        jButton14 = new javax.swing.JButton();
        jButton15 = new javax.swing.JButton();
        jButton16 = new javax.swing.JButton();

        jLabel1 = new javax.swing.JLabel();
        jLabel10 = new javax.swing.JLabel();
        jLabel11 = new javax.swing.JLabel();
        jLabel12 = new javax.swing.JLabel();
        jLabel13 = new javax.swing.JLabel();
        jLabel14 = new javax.swing.JLabel();
        jLabel15 = new javax.swing.JLabel();
        jLabel16 = new javax.swing.JLabel();
        jLabel17 = new javax.swing.JLabel();

        jList1 = new JList(listModel);

        jPanel1 = new javax.swing.JPanel();
        jPanel2 = new javax.swing.JPanel();
        jPanel3 = new javax.swing.JPanel();
        jPanel4 = new ImagePanel(0); // Sensor 1, index 0. Left Insole Sensor

        jPanel7 = new ImagePanel(1); // Sensor 2, index 1. Right Insole Sensor
        jScrollPane1 = new javax.swing.JScrollPane();

        userMessage1 = new javax.swing.JLabel();


        // Set background color for each panel.
```

```
        jPanel1.setBackground(new Color(195, 230, 250));
        jPanel2.setBackground(new Color(195, 230, 250));
        jPanel3.setBackground(new Color(195, 230, 250));
        jPanel4.setBackground(new Color(195, 230, 250));
        jPanel7.setBackground(new Color(195, 230, 250));

        // Set borders for each region of the GUI.
        Border blackline = BorderFactory.createLineBorder(Color.black);
        jPanel1.setBorder(blackline);
        jPanel3.setBorder(blackline);
        jPanel4.setBorder(blackline);
      //  jPanel6.setBorder(blackline);
        jPanel7.setBorder(blackline);

        jPanel3.setPreferredSize(new Dimension(900,1000));
        jPanel4.setPreferredSize(new Dimension(900,500));
        jPanel7.setPreferredSize(new Dimension(900,500));

        // Define the behavior of the GUI when the user hits the close button.
// Here, it will exit the application using the System "exit" method.
        setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);

        // Define initial message to user.
        userMessage1.setText("Sensors not connected!");
    //    userMessage2.setText("Sensor 2 not connected!");

        // Only one list index in jList1 can be selected at a time.
        jList1.setSelectionMode(javax.swing.ListSelectionModel.
        SINGLE_SELECTION);
// Specifies jList1 as the component that will be displayed in jScrollPane1.
        jScrollPane1.setViewportView(jList1);


        // Define font on jLabel1 label.
        jLabel1.setFont(new java.awt.Font("Verdana",
        java.awt.Font.BOLD, 24));
// Define text on jLabel1.
        jLabel1.setText("Insole Sensor System");
```

```
        // Define font on jLabel10 label.
        jLabel10.setFont(new java.awt.Font("Verdana",
        java.awt.Font.BOLD, 20));
// Define text on jLabel10.
        jLabel10.setText("Sensors Control");

        // Define font on jLabel11 label.
        jLabel11.setFont(new java.awt.Font("Verdana", 0, 18));
// Define text on jLabel11.
        jLabel11.setText("Bluetooth Connection");

        // Define font color on jLabel12 label.
        jLabel12.setForeground(Color.black);
// Define text on jLabel12.
        jLabel12.setText("Bluetooth not connected");

        // Define font on jLabel13 label.
        jLabel13.setFont(new java.awt.Font("Verdana", 0, 18));
// Define text on jLabel13.
        jLabel13.setText("Display");

        // Define font on jLabel14 label.
        jLabel14.setFont(new java.awt.Font("Verdana", 0, 18));
// Define text on jLabel14.
        jLabel14.setText("System Message");

        // Define font on jLabel15 label.
        jLabel15.setFont(new java.awt.Font("Verdana", 0, 18));
// Define text on jLabel15.
        jLabel15.setText("Image Recording");

     // Define font on jLabel16 label.
        jLabel16.setFont(new java.awt.Font("Verdana",
        java.awt.Font.BOLD, 20));
// Define text on jLabel16.
        jLabel16.setText("Left Insole Sensor");

     // Define font on jLabel17 label.
        jLabel17.setFont(new java.awt.Font("Verdana",
```

```
        java.awt.Font.BOLD, 20));
// Define text on jLabel17.
        jLabel17.setText("Right Insole Sensor");

     // Define font on jButton11.
        jButton11.setFont(new java.awt.Font("Verdana", 0, 14));
// Define text on jButton11.
        jButton11.setText("Search");
// Adds an Action Listener to jButton11. An Action Listener is implemented
// to define what should be done when an user performs an action.
// When jButton11 is pressed, the program runs "jButton11ActionPerformed".
        jButton11.addActionListener(new java.awt.event.ActionListener() {
            @Override
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                jButton11ActionPerformed(evt);
            }
        });

        // Define font on jButton12.
        jButton12.setFont(new java.awt.Font("Verdana", 0, 14));
// Define text on jButton12.
        jButton12.setText("Connect");
// Adds an Action Listener to jButton12. An Action Listener is implemented
// to define what should be done when an user performs an action.
// When jButton12 is pressed, the program runs "jButton12ActionPerformed".
        jButton12.addActionListener(new java.awt.event.ActionListener() {
            @Override
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                jButton12ActionPerformed(evt);
            }
        });

        // Define font on jButton13.
        jButton13.setFont(new java.awt.Font("Verdana", 0, 14));
// Define text on jButton13.
        jButton13.setText("Disconnect");
        // Disable "Disconnect" button.
        jButton13.setEnabled(false);
// Adds an Action Listener to jButton13. An Action Listener is implemented
```

```
// to define what should be done when an user performs an action.
// When jButton13 is pressed, the program runs "jButton13ActionPerformed".
        jButton13.addActionListener(new java.awt.event.ActionListener() {
            @Override
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                jButton13ActionPerformed(evt);
            }
        });

        // Define font on jButton14.
        jButton14.setFont(new java.awt.Font("Verdana", 0, 14));
// Define text on jButton14.
        jButton14.setText("Refresh Values");
// Adds an Action Listener to jButton14. An Action Listener is implemented
// to define what should be done when an user performs an action.
// When jButton14 is pressed, the program runs "jButton14ActionPerformed".
        jButton14.addActionListener(new java.awt.event.ActionListener() {
            @Override
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                jButton14ActionPerformed(evt);
            }
        });

        // Define font on jButton15.
        jButton15.setFont(new java.awt.Font("Verdana", 0, 14));
// Define text on jButton15.
        jButton15.setText("Set Pressure to Zero");
// Adds an Action Listener to jButton15. An Action Listener is implemented
// to define what should be done when an user performs an action.
// When jButton15 is pressed, the program runs "jButton15ActionPerformed".
        jButton15.addActionListener(new java.awt.event.ActionListener() {
            @Override
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                jButton15ActionPerformed(evt);
            }
        });

        // Define font on jButton16.
        jButton16.setFont(new java.awt.Font("Verdana", 0, 14));
```

```
// Define text on jButton16.
        jButton16.setText("Start");
// Adds an Action Listener to jButton16. An Action Listener is implemented
// to define what should be done when an user performs an action.
// When jButton16 is pressed, the program runs "jButton16ActionPerformed".
        jButton16.addActionListener(new java.awt.event.ActionListener() {
            @Override
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                jButton16ActionPerformed(evt);
            }
        });


        // This next part defines the position of the
        components inside the GUI.
        // First, position the main containers of the GUI - jPanel1,
        // jPanel2 and jPanel5 - inside the JFrame.

        // Create a GroupLayout object. GroupLayout hierarchically groups
        components in order to
// position them inside a container. Implements LayoutManager, which is the
interface
// for classes that know how to lay out components inside a container.
        // Create a GroupLayout object to manage components in the JFrame.
        javax.swing.GroupLayout layout = new
        javax.swing.GroupLayout(getContentPane());
        // Set layout as the Layout Manager for the JFrame
        getContentPane().setLayout(layout);
        // GroupLayout works with the horizontal and vertical layouts
        separately. This means each
// component needs to be defined twice in the layout, once in each
direction.
// Here, "setHorizontalGroup" defines the Group that positions and sizes
components along the horizontal axis.
        layout.setHorizontalGroup(
            // GroupLayout uses two types of arrangements. Sequential
            arrangement places the components one after another.
   // Parallel arrangement places components on top of each other in the
   same space. They can be baseline-, top-, or bottom-aligned
```

```
// along the vertical axis. Along the horizontal axis, they can be left-,
right-, or center-aligned if the components are not all the same size.

// Usually, components placed in parallel in one dimension are in
sequence in the other, to avoid overlap.
// Here, a parallel group is created with "LEADING" alignment, which
means elements should be aligned to the origin.
         layout.createParallelGroup(javax.swing.GroupLayout.
         Alignment.CENTER)
             // Add sequential groups that contain jPanel1, jPanel2 and
             jPanel5, the main containers of the GUI.
             .addGroup(layout.createSequentialGroup()
                     .addComponent(jPanel1,0,1400, 1500))
             .addGroup(layout.createSequentialGroup()
                     .addComponent(jPanel2,0,1400, 1500))
       //      .addGroup(layout.createSequentialGroup()
       //              .addComponent(jPanel5))
     );
     // Now, "setVerticalGroup" defines the Group that positions and
     sizes components along the vertical axis on jPanel1.
     layout.setVerticalGroup(
         // Create sequential group.
         layout.createSequentialGroup()
             // The same components added in the horizontal axels should
             be added in the vertical axis as well.
             .addGroup(layout.createSequentialGroup()
                     .addComponent(jPanel1))
             .addGroup(layout.createSequentialGroup()
                     .addComponent(jPanel2))
         //    .addGroup(layout.createSequentialGroup()
           //          .addComponent(jPanel5))
     );

     // Now configure each one of the main containers.

     // Create a GroupLayout object to manage components in the jPanel1.
     javax.swing.GroupLayout jPanel1Layout = new
     javax.swing.GroupLayout(jPanel1);
     // Set layout as the Layout Manager for the jPanel1.
```

```
jPanel1.setLayout(jPanel1Layout);
// In jPanel1, add the title label and gaps surrounding it.
jPanel1Layout.setHorizontalGroup(
        jPanel1Layout.createParallelGroup(javax.swing.GroupLayout.
        Alignment.CENTER)
            .addGap(1400)
            .addComponent(jLabel1)
            .addGap(1400)
);
jPanel1Layout.setVerticalGroup(
        jPanel1Layout.createSequentialGroup()
            .addGap(25)
            .addComponent(jLabel1)
            .addGap(25)
);

// Create a GroupLayout object to manage components in the jPanel2.
javax.swing.GroupLayout jPanel2Layout = new
javax.swing.GroupLayout(jPanel2);
// Set layout as the Layout Manager for the jPanel2.
jPanel2.setLayout(jPanel2Layout);
// In the horizontal axis of jPanel2, jPanel3 and jPanel4 should be
// positioned side-by-side, with a gap in-between.
jPanel2Layout.setHorizontalGroup(
        jPanel2Layout.createSequentialGroup()
            .addComponent(jPanel3)
            .addGroup(jPanel2Layout.createParallelGroup(javax.swing.
            GroupLayout.Alignment.CENTER)
            .addComponent(jLabel16)
            .addComponent(jPanel4,1200,1200, 1300)
            .addComponent(jLabel17)
            .addComponent(jPanel7,1200,1200, 1300))
);
jPanel2Layout.setVerticalGroup(
        jPanel2Layout.createSequentialGroup()
        .addGroup(jPanel2Layout.createParallelGroup(javax.swing.
        GroupLayout.Alignment.LEADING)
            .addComponent(jPanel3,0,500, 1000)
            .addComponent(jLabel16)
```

```
            .addComponent(jPanel4,0,500, 1000))
        .addComponent(jLabel17)
        .addComponent(jPanel7,0,500, 1200)
);


// Inside jPanel3 and jPanel6, position the items used to control
// the first and second sensors, respectively.

// Create a GroupLayout object to manage components in the jPanel3.
javax.swing.GroupLayout jPanel3Layout = new
javax.swing.GroupLayout(jPanel3);
// Set layout as the Layout Manager for the jPanel3.
jPanel3.setLayout(jPanel3Layout);
jPanel3Layout.setHorizontalGroup(jPanel3Layout.createParallelGroup(
javax.swing.GroupLayout.Alignment.CENTER)
        .addComponent(jLabel10)
        .addGroup(jPanel3Layout.createSequentialGroup()
                .addGap(20)
                .addGroup(jPanel3Layout.createSequentialGroup()
            .addGroup(jPanel3Layout.createParallelGroup(javax.swing.
            GroupLayout.Alignment.CENTER)
                        .addComponent(jLabel11)
                        .addComponent(jLabel12)
                        .addComponent(jButton11, 185, 185, 185)
                        .addComponent(jScrollPane1,
                        javax.swing.GroupLayout.PREFERRED_SIZE, 200,
                        javax.swing.GroupLayout.PREFERRED_SIZE)
                        .addComponent(jButton12, 185, 185, 185)
                        .addComponent(jButton13, 185, 185, 185)
                    )
                    .addGap(30)
                .addGroup(jPanel3Layout.createParallelGroup(javax.
                sing.GroupLayout.Alignment.CENTER)
                        .addComponent(jLabel13)
                        .addComponent(jButton14, 185, 185, 185)
                        .addComponent(jButton15, 185, 185, 185)
                        .addComponent(jLabel15)
                        .addComponent(jButton16, 185, 185, 185)
```

```
                            .addComponent(jLabel14)
                            .addComponent(userMessage1, 185, 185, 185)
                        )
                    )
                    .addGap(30)
            )
        );
        jPanel3Layout.setVerticalGroup(jPanel3Layout.createParallelGroup()
                .addGroup(jPanel3Layout.createSequentialGroup()
                    .addGap(30)
                    .addComponent(jLabel10)
                    .addGap(5)
                )
                .addGroup(jPanel3Layout.createSequentialGroup()
                        .addGap(100)
                        .addComponent(jLabel11)
                        .addGap(10)
                        .addComponent(jLabel12)
                        .addGap(15)
                        .addComponent(jScrollPane1,
                        javax.swing.GroupLayout.PREFERRED_SIZE, 70,
                        javax.swing.GroupLayout.PREFERRED_SIZE)
                        .addGap(15)
                        .addComponent(jButton11)
                        .addGap(10)
                        .addComponent(jButton12)
                        .addGap(10)
                        .addComponent(jButton13)
                        .addGap(60)
                )
                .addGroup(jPanel3Layout.createSequentialGroup()
                        .addGap(100)
                        .addComponent(jLabel13)
                        .addGap(10)
                        .addComponent(jButton14)
                        .addGap(10)
                        .addComponent(jButton15)
                        .addGap(30)
                        .addComponent(jLabel15)
```

```
                            .addGap(10)
                            .addComponent(jButton16)
                            .addGap(30)
                            .addComponent(jLabel14)
                            .addGap(10)
                            .addComponent(userMessage1)
                            .addGap(20)
                    )
        );

        // Causes the JFrame to be sized to fit the preferred size and
        layouts of its subcomponents.
        pack();

    } // End initComponents.

    // "Search" button for 1st insole sensor
    private void jButton11ActionPerformed(java.awt.event.ActionEvent evt){
        // Disable all buttons.
        this.ButtonSettings(0, (byte) 0);
        // Create thread using the bluetooth component. This means that when
        this thread is started, the "run" method of bluetooth
        will be invoked.
        bt1 = new Thread(bluetooth1);
// Start the bluetooth thread.
        bt1.start();
    } // End jButton11ActionPerformed.

    // "Connect" button for 1st insole sensor
    private void jButton12ActionPerformed(java.awt.event.ActionEvent evt){
        // Check to see if the bluetooth is connected.
        if ( bluetooth1.connectToServer(this) ){
   // Display "bluetooth connected successfully" message on jLabel3,
   // disable jButton1 and jButton 5, enable jButton2, jButton3 and
   jButton4.
            this.ButtonSettings(2, (byte) 0);
   // Create thread using the bluetooth object. This means that when this
   // thread is started, the "run" method of bluetooth will be invoked.
            bt1 = new Thread(bluetooth1);
```

```
   // Remove all elements from listModel.
            listModel.clear();
   // Add instructions to be displayed at the scroll panel.
            listModel.addElement("Here you can select the device,");
            listModel.addElement("but please run the Search first!");
   // Set message in userMessage component.
            userMessage1.setText("Connected successfully!");
   // Start the bluetooth thread.
            bt1.start();
        }
    } // End jButton12ActionPerformed.

    // "Disconnect" button for 1st insole sensor
    private void jButton13ActionPerformed(java.awt.event.ActionEvent evt){
        // Disable all buttons.
        this.ButtonSettings(0, (byte) 0);
// End bluetooth connection.
        bluetooth1.close(this);
// Display "bluetooth not connected" message on jLabel3,
// disable jButton2, enable jButton1, jButton3, jButton4 and jButton5.
        this.ButtonSettings(1, (byte) 0);
// Set message in userMessage component.
        userMessage1.setText("Disconnected successfully!");
    } // End jButton13ActionPerformed.

    // "Refresh Values" button for 1st insole sensor
    private void jButton14ActionPerformed(java.awt.event.ActionEvent evt){
        // Clear contents on reference array and update difference array.
        for(int j=0;j<=1;j++) {
        for(int i=0;i<reference[j].length;i++){
                reference[j][i]=0;
            }
        }

// Executes "run" method of tmain1 thread to update image.
        tmain1.run();
        tmain2.run();
// Set message in userMessage1 component.
        userMessage1.setText("Pixels refreshed successfully!");
```

```
    } // End jButton14ActionPerformed.

    // "Set Pressure to Zero" button for 1st insole sensor
    private void jButton15ActionPerformed(java.awt.event.ActionEvent evt){
        // Copy contents of pressure array in the reference array.
    for(int j=0;j<=1;j++) {
    System.arraycopy(pressure[j], 0, reference[j], 0, pressure[j].length);
    }
// Execute "run" method of tmain1 thread to update image.
        tmain1.run();
        tmain2.run();
// Set message in userMessage1 component.
        userMessage1.setText("Pixels set to Zero successfully!");
    } // End jButton15ActionPerformed.

    // "Start/Stop" image recording button for 1st insole sensor.
    private void jButton16ActionPerformed(java.awt.event.ActionEvent evt){
    for(int j=0;j<=1;j++) {
    if(imgRecording[j]){
                jButton16.setText("Start");
                imgRecording[j] = false;
                userMessage1.setText("Image Recording Stopped!");
            }
            else{
                jButton16.setText("Stop");
                imgRecording[j] = true;
                userMessage1.setText("Image Recording Started!");
            }
    }

    }

    /**
     * Controls which buttons are enabled in the GUI and message on jLabel3,
     * according to the state value passed to the function.
     * @param state, sensorIndex
     */
    public void ButtonSettings(int state, byte sensorIndex){
```

```
switch (state) {
    case 0:
        // Disable all buttons.
        switch (sensorIndex){
            // Sensor 1
            case 0:
                jButton11.setEnabled(false);
                jButton12.setEnabled(false);
                jButton13.setEnabled(false);
                jButton14.setEnabled(false);
                jButton15.setEnabled(false);
                jButton16.setEnabled(false);
                break;
            // Sensor 2
            case 1:
          /*      jButton21.setEnabled(false);
                jButton22.setEnabled(false);
                jButton23.setEnabled(false);
                jButton24.setEnabled(false);
                jButton25.setEnabled(false);
                jButton26.setEnabled(false);*/
                break;
            default:
                break;
        }
        break;
    case 1:
        // Waiting for connection.
        switch (sensorIndex){
            // Display "Bluetooth not connected" message on
            jLabelx2,
            // disable "Disconnect" button, enable "Connect",
            "Refresh Values",
            // "Set Pressure to Zero", "Start/Stop" and "Search"
            buttons.
            case 0:
                jLabel12.setText("Bluetooth not connected...");
                jButton13.setEnabled(false);
                jButton12.setEnabled(true);
```

```
                jButton14.setEnabled(true);
                jButton15.setEnabled(true);
                jButton16.setEnabled(true);
                jButton11.setEnabled(true);
                break;
            case 1:
             /*   jLabel22.setText("Bluetooth not connected...");
                jButton23.setEnabled(false);
                jButton22.setEnabled(true);
                jButton24.setEnabled(true);
                jButton25.setEnabled(true);
                jButton26.setEnabled(true);
                jButton21.setEnabled(true);*/
                break;
            default:
                break;
        }
        break;
    case 2:
        // Connected to Bluetooth device.
        // Display "bluetooth connected successfully" message on
        jLabelx2,
        // disable "Connect" and "Search" buttons, enable the
        others.
        switch (sensorIndex){
            case 0:
                jLabel12.setText("Bluetooth connected
                successfuly!");
                jButton11.setEnabled(false);
                jButton12.setEnabled(false);
                jButton13.setEnabled(true);
                jButton14.setEnabled(true);
                jButton15.setEnabled(true);
                jButton16.setEnabled(true);
                break;
            case 1:
               /* jLabel22.setText("Bluetooth connected
               successfuly!");
                jButton21.setEnabled(false);
```

```
                    jButton22.setEnabled(false);
                    jButton23.setEnabled(true);
                    jButton24.setEnabled(true);
                    jButton25.setEnabled(true);
                    jButton26.setEnabled(true);*/
                    break;
                default:
                    break;
            }
            break;
        default:
            break;
    }
}

/**
 * Class used to create a visual representation of the sensor points.
 */
private class ImagePanel extends JPanel implements Runnable {

    /**
     * Index number representing the sensor in the GUI's arrays.
     * Sensor 1 = index 0.
     * Sensor 2 = index 1.
     */
    public int sensorIndex;

    /**
     * ImagePanel constructor. Runs every time an object of this class
     is created.
     * @param i, the index of the sensor represented in this panel.
     */
    public ImagePanel(int index){
        // Record sensor index.
        sensorIndex = index;
// Initialize pressure array with random numbers.
        for(int j = 0; j < pressure[sensorIndex].length; j++){
            pressure[sensorIndex][j] = (int) (Math.random()*255);
        }
```

```
    // Create pressure image to be displayed in the panel.
    pressureImage[sensorIndex] = updatePressurePixels();
} // End ImagePanel constructor.

@Override
public void run() {
    // Update pressure image.
    pressureImage[sensorIndex] = updatePressurePixels();
    repaint(50,45,1020,357);
    // Check to see if it's time to save a new image.
    // Update current time.
    currentTime[sensorIndex] = System.currentTimeMillis();
    // If elapsed time >= DELAY ms and user selected to save
    image...
    if(currentTime[sensorIndex] - previousTime[sensorIndex] >=
    DELAY &&
            imgRecording[sensorIndex]){
        boolean saved = saveImageFile();
        // If failed to save image, show error message.
        if(!saved){
            switch(sensorIndex){
                case 0:
                userMessage1.setText("Error Saving Image!");
                break;
                case 1:
                // userMessage2.setText("Error Saving Image!");
                break;
            }
        }
    }
} // End run method.

// Function used to save pressure images into .png files
boolean saveImageFile(){
    boolean success;
    // Create File object that is used to save image file.
    File imgFile;
    // Create string for image file name.
    String fileName;
```

```
        // Save image file
        try{
        if(sensorIndex==0) {
            fileName = "C:\\InsoleSensorSystem\\LeftSensor\\" +
            "LeftSensor_" + Long.toString(counter[sensorIndex]) +
            ".png";
            imgFile = new File(fileName);
            ImageIO.write(pressureImage[sensorIndex], "png", imgFile);
            // Update number of saved images.
            counter[sensorIndex]++;
            // Update time variable for next comparison.
            previousTime[sensorIndex] = System.currentTimeMillis();
            success = true;
        } else {

            fileName = "C:\\InsoleSensorSystem\\RightSensor\\" +
            "RightSensor_" + Long.toString(counter[sensorIndex]) +
            ".png";
            imgFile = new File(fileName);
            ImageIO.write(pressureImage[sensorIndex], "png", imgFile);
            // Update number of saved images.
            counter[sensorIndex]++;
            // Update time variable for next comparison.
            previousTime[sensorIndex] = System.currentTimeMillis();
            success = true;
        }

        } catch(Exception e){
            System.out.println("Error: " + e);
            success = false;
        }
        return success;
    }

    /**
     * Repaint the panel, using the current pressure image.
     * @param g
     */
    @Override
```

```
     protected void paintComponent(Graphics graph) {
// Updates graphics of ImagePanel's container.
         super.paintComponent(graph);
         // Display pressureImage on panel.
         graph.drawImage(pressureImage[sensorIndex], 50, 45, null);
     } // End paintComponent method.

     /**
      * Defines how the visual representation of the sensor
      points will be displayed based on the values of the
      pressure and reference arrays' elements.
      * @return img
      */
     public BufferedImage updatePressurePixels(){
         int position;
         int g;
         int pixel;
         int blankPixel = (255<<24) | (255<<16) | (255<<8) | 255;
         int difference;
         BufferedImage img = new BufferedImage(60 * 15, 21 * 15,
         BufferedImage.TYPE_INT_ARGB);
         for(int j=0; j < 21; j++){              // For all lines...
             for(int i=0; i < 60; i++){          // For all columns...
 // Calculate current position inside the pressure arrays
                 position = 60*j + i;
// Calculate the difference value, which means comparing the
current pressure
// value at this sensor point with its reference value.
                 difference = pressure[sensorIndex][position] -
                 reference[sensorIndex][position];
                 // Paint pixel according to the difference value.
// If the pressure value is bigger than the reference value:
// the more pressure, the more intense the red color of the pixel.
                 if(difference > 0){
                 g = 255 - difference;
                     pixel = (255<<24) | (255<<16) | (g<<8) | g;
                     // build pixel
                     // Fill 17 x 17 pixel rectangle on pressure image
                     for(int k = 0; k < 15; k++){
```

```
                        for(int l = 0; l < 15; l++){
                        img.setRGB((i * 15) + l, (j * 15) + k, pixel);
                            }
                         }
                    }
        // If the pressure is bellow the reference value, the pixel
        becomes white.
                    else{
                        for(int k = 0; k < 15; k++){
                            for(int l = 0; l < 15; l++){
                                img.setRGB((i * 15) + l, (j * 15) + k,
                                blankPixel);
                                }
                            }
                        }
                   }

             return img;
         }

    } // End ImagePanel class.

} // End JFrame2 class.
```

APÊNDICE D – CÓDIGO JAVA - BLUETOOTH2

O código abaixo é utilizado pelo "JFrame2.java", descrito no apêndice C.

```
package InsoleBluetooth;

import com.intel.bluetooth.RemoteDeviceHelper;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.Enumeration;
import java.util.Vector;
import javax.bluetooth.BluetoothStateException;
import javax.bluetooth.DataElement;
import javax.bluetooth.DeviceClass;
import javax.bluetooth.DiscoveryAgent;
import javax.bluetooth.DiscoveryListener;
import javax.bluetooth.LocalDevice;
import javax.bluetooth.RemoteDevice;
import javax.bluetooth.ServiceRecord;
import javax.bluetooth.UUID;
import javax.microedition.io.Connector;
import javax.microedition.io.StreamConnection;
import insolesensor.JFrame2;
//NOVA

/**
 * Handles the Bluetooth communication of the Insole Sensor System.
 * Original Authors: Matheus Critter, Gustavo C. Oliveira.
 * New Authors: Natalia Aravequia, Cayna Ferraz.
 */
public class Bluetooth2 extends Thread {

    // It's too dangerous to go alone, take this:
//http://bluecove.org/bluecove/apidocs/javax/bluetooth/package-summary.html
    // http://www.oracle.com/technetwork/systems/bluetooth2-156149.html
    // http://www.jsr82.com/jsr-82-initialization-localdevice/
    // http://www.jsr82.com/jsr-82-sample-device-discovery/
```

```
// http://www.jsr82.com/jsr-82-sample-bluetooth-service-search/
// http://www.jsr82.com/jsr-82-sample-spp-server-and-client/

public DiscoveryAgent agent;
public DiscoveryListener listener;
public String btDeviceDiscovered;
public Vector devicesDiscovered = new Vector();
public Vector friendlyNameDiscovered = new Vector();
public Vector serviceFound = new Vector();
public Vector serviceNameFound = new Vector();
public Object inquiryCompletedEvent = new Object();
public Object serviceSearchCompletedEvent = new Object();
UUID serviceUUID = new UUID(0x0003);
UUID[] searchUuidSet = new UUID[] { serviceUUID };
int[] attrIDs =  new int[] { 0x0100 };
String message;
public InputStream input;
public OutputStream output;
public StreamConnection conn;
private boolean stateRecord = false;
private int pos = 0;
boolean connected = false;
int deviceDiscoveredIndex;
byte sensorIndex;
private JFrame2 originalGUI;

/**
 * Class constructor.
 * @param GUI, index
 */
public Bluetooth2(JFrame2 GUI, byte index){
    // Record sensor index.
    sensorIndex = index;
    // Save reference for original GUI
    originalGUI = GUI;
    // This might fail, so put inside a try-catch block to handle
    exceptions.
    try {
        // Retrieve the local Bluetooth device object. Provides
```

```
    device-management capabilities.
    LocalDevice local = LocalDevice.getLocalDevice();
    // Because wireless devices are mobile, they need a mechanism
    that allows them to find other devices
    // and gain access to their capabilities. The DiscoveryAgent
    class provides methods to perform device
    // and service discovery. The next line retrieves the discovery
    agent for the Bluetooth device.
    agent = local.getDiscoveryAgent();
} catch (BluetoothStateException e) {
    // If something goes wrong, show error message.
    switch(sensorIndex){
        case 0:
        GUI.userMessage1.setText("Error... " + e);
        break;
        case 1:
        //  GUI.userMessage2.setText("Error... " + e);
        break;
    }
}
// Create a listener. The DiscoveryListener interface allows an
application to receive device discovery and service discovery
events.
// Services are Bluetooth applications the local device can use to
perform useful tasks.
listener = new DiscoveryListener() {

    // Called when a device is found during an inquiry.
    @Override
    public void deviceDiscovered(RemoteDevice btDevice,
    DeviceClass cod) {
        // Construct message containing the device's address.
        message = message + "<br>" + "Device " +
        btDevice.getBluetoothAddress() + " found";
        // Show message in userMessage component of the GUI.
        switch(sensorIndex){
            case 0:
            GUI.userMessage1.setText("<html>" + message +
            "</html>");
```

```
        break;
        case 1:
        //GUI.userMessage2.setText("<html>" + message +
        "</html>");
        break;
    }
    // Add device to the vector of devices discoreved.
    devicesDiscovered.addElement(btDevice);
    try {
        // Try to retrive the Bluetooth device name and display
        it on userMessage component.
        message = message + "<br>" + "     name " +
        btDevice.getFriendlyName(false);
        switch(sensorIndex){
            case 0:
            GUI.userMessage1.setText("<html>" + message +
            "</html>");
            break;
            case 1:
         //GUI.userMessage2.setText("<html>" + message +
         "</html>");
            break;
        }
        friendlyNameDiscovered.addElement(btDevice.
        getFriendlyName(false));
    } catch (IOException cantGetDeviceName) {
        // Show error message if device's name is not found.
        message = message + "<br>" + "     name not Found!";
        switch(sensorIndex){
            case 0:
            GUI.userMessage1.setText("<html>" + message +
            "</html>");
            break;
            case 1:
       //  GUI.userMessage2.setText("<html>" + message +
       "</html>");
            break;
         }
         friendlyNameDiscovered.addElement("NONE");
```

```
    }
}

// Called when service(s) are found during a service search.
@Override
public void servicesDiscovered(int transID, ServiceRecord[]
servRecord) {
    // For each service discovered during the search request...
    for (int i = 0; i < servRecord.length; i++) {
        // Retrieve the String that can be used to
        connect to the service.
        String url =
        servRecord[i].getConnectionURL(ServiceRecord.
        NOAUTHENTICATE_NOENCRYPT, false);
        if (url == null) {
            // If url is not found, continue to the
            next service.
            continue;
        }
        // Add the url of the service to the list of services.
        serviceFound.add(url);
        // Retrive the service name.
        DataElement serviceName =
        servRecord[i].getAttributeValue(0x0100);
        if (serviceName != null) {
            // Display serviceName and url in the userMessage
            component of the GUI.
            message = message + "<br>" + "service " +
            serviceName.getValue() + " found " + url;
            switch(sensorIndex){
                case 0:
                GUI.userMessage1.setText("<html>" + message +
                "</html>");
                break;
                case 1:
                // GUI.userMessage2.setText("<html>" + message +
                "</html>");
                break;
            }
```

```
                // Include the service name in the service name
                list.
                String service = deviceDiscoveredIndex+"
                "+btDeviceDiscovered+": "+serviceName.getValue();
                if(service.length()>33)service =
                service.substring(0,33);
                serviceNameFound.add(service);
            } else {
                // Display service url in the userMessage component
                and add the url to the service name list.
                String service = deviceDiscoveredIndex+"
                "+btDeviceDiscovered+": "+"NULL";
                if(service.length()>33)service =
                service.substring(0,33);
                serviceNameFound.add(service);
                message = message + "<br>" + "service found " + url;
                switch(sensorIndex){
                    case 0:
                    GUI.userMessage1.setText("<html>" + message +
                    "</html>");
                    break;
                    case 1:
                    //GUI.userMessage2.setText("<html>" + message +
                    "</html>");
                    break;
                }
            }
        }
    }

    // Called when a service search is completed or was
    terminated because of an error.
    @Override
    public void serviceSearchCompleted(int i, int i1) {
        // Display "service search completed" message in the
        userMessage component of the GUI.
        message = message + " " + "service search completed!";
        switch(sensorIndex){
            case 0:
```

```
                GUI.userMessage1.setText("<html>" + message +
                "</html>");
                break;
                case 1:
            //  GUI.userMessage2.setText("<html>" + message +
            "</html>");
                break;
            }
            // Synchronize threads.
            synchronized(serviceSearchCompletedEvent){
                serviceSearchCompletedEvent.notifyAll();
            }
        }

        // Called when an inquiry is completed.
        @Override
        public void inquiryCompleted(int i) {
            // Display message in the userMessage component of the GUI.
            message = message + "<br>"+ "Device Inquiry completed!";
            switch(sensorIndex){
                case 0:
                    GUI.userMessage1.setText("<html>" + message
                    +"</html>");
                    break;
                case 1:
                //    GUI.userMessage2.setText("<html>" + message
                +"</html>");

                    break;
            }
            // Synchronize threads.
            synchronized(inquiryCompletedEvent){
                inquiryCompletedEvent.notifyAll();
            }
        }
    };
}

/**
```

```
 * Function responsible for finding Bluetooth devices, registering their
 names and services.
 * @param GUI
 */
public void bluetoothInquiry(JFrame2 GUI){
    // Clear all device, name and service records.
    boolean error = false;
    devicesDiscovered.clear();
    friendlyNameDiscovered.clear();
    serviceFound.clear();
    serviceNameFound.clear();
    message=null;
    try{
        // Synchronize threads.
        synchronized(inquiryCompletedEvent) {
        // Start process to discover Bluetooth devices.
        boolean started =
        LocalDevice.getLocalDevice().getDiscoveryAgent().
        startInquiry(DiscoveryAgent.GIAC, listener);
            if (started) {
                //If discovery process started with no problems,
                show a message indicating
                // the process has started in the userMessage
                component of the GUI.
                message = "wait for device inquiry to complete...";
                switch(sensorIndex){
                    case 0:
                    GUI.userMessage1.setText("<html>" + message
                    + "</html>");
                    break;
                    case 1:
                //   GUI.userMessage2.setText("<html>" + message
                + "</html>");
                    break;
                 }
                 try {
                    // Wait until search is over.
                    inquiryCompletedEvent.wait();
                 } catch (InterruptedException ex) {
```

```
                    //If any problems happen, show an error message
                    instead.
                    message = message + "Error: " + ex;
                    switch(sensorIndex){
                        case 0:
                        GUI.userMessage1.setText("<html>" + message
                        + "</html>");
                        break;
                        case 1:
                  //        GUI.userMessage2.setText("<html>" + message
                  + "</html>");
                            break;
                    }
                }
            // Display on userMessage component how many
            devices were found.
                message = message +"<br>"+devicesDiscovered.size() +
                " device(s) found";
                switch(sensorIndex){
                    case 0:
                      GUI.userMessage1.setText("<html>"+message+"</html>");
                        break;
                        case 1:
                 //
                 GUI.userMessage2.setText("<html>"+message+"</html>");
                        break;
                }
            }
        }
    } catch (BluetoothStateException ex) {
        //  If any problems happen, show an error message instead.
        error = true;
        message = message + "Error(1): " + ex;
        switch(sensorIndex){
            case 0:
                GUI.userMessage1.setText("<html>" + message +
                "</html>");
                break;
            case 1:
```

```
    //        GUI.userMessage2.setText("<html>" + message +
    "</html>");
            break;
    }
}
// Device counter.
deviceDiscoveredIndex = 0;
// For each Bluetooth device discovered...
for(Enumeration en = devicesDiscovered.elements();
en.hasMoreElements(); ) {
    // Create an object representing the Bluetooth device.
    RemoteDevice btDevice = (RemoteDevice)en.nextElement();
    // Synchronize threads.
    synchronized(serviceSearchCompletedEvent) {
        try {
            // Retrieve device's name.
            btDeviceDiscovered = btDevice.getFriendlyName(false);
        } catch (IOException ex) {
            //  If any problems happen, show an error message
            instead.
            error = true;
            message = message + "Error(2): " + ex;
            switch(sensorIndex){
                case 0:
                GUI.userMessage1.setText("<html>" + message +
                "</html>");
                break;
                case 1:
    //          GUI.userMessage2.setText("<html>" + message +
    "</html>");
                break;
            }
        }
        //Display message to indicate services will be
        searched in this device.
        message = message +"<br>"+ "search services on " +
        btDevice.getBluetoothAddress() + " " + btDeviceDiscovered;
        switch(sensorIndex){
            case 0:
```

```
                GUI.userMessage1.setText("<html>" + message +
                "</html>");
                break;
            case 1:
//              GUI.userMessage2.setText("<html>" + message +
"</html>");
                break;
        }
        try {
            // Start searching for services.
            LocalDevice.getLocalDevice().getDiscoveryAgent().
            searchServices(attrIDs, searchUuidSet,
            btDevice, listener);
        } catch (IOException ex) {
            //If any problems happen, show an error message instead.
            error = true;
            message = message + "Error(3): " + ex;
            switch(sensorIndex){
                case 0:
                GUI.userMessage1.setText("<html>" + message +
                "</html>");
                break;
                case 1:
//              GUI.userMessage2.setText("<html>" + message +
"</html>");
                break;
            }
            break;
        }
        try {
            // Wait until search is finished.
            serviceSearchCompletedEvent.wait();
        } catch (InterruptedException ex) {
            //If any problems happen, show an error message instead.
            error = true;
            message = message + "Error(4): " + ex;
            switch(sensorIndex){
                case 0:
                GUI.userMessage1.setText("<html>" + message
```

```
                + "</html>");
                break;
                case 1:
      //        GUI.userMessage2.setText("<html>" + message
      + "</html>");
                break;
            }
        }
    }
    // Update device counter.
    deviceDiscoveredIndex++;
}
// Display message showing how many services and service's
names were found.
System.out.println("serviceFound = " + serviceFound.size() +
"\nserviceNameFound = "+ serviceNameFound.size());
// Add service names to a list.
if(serviceFound.size()!=0) GUI.listModel.clear();
for(int i=0;i<serviceFound.size();i++){
    GUI.listModel.addElement(serviceNameFound.elementAt(i));
}
if(error == false){
    // Display message indicating search for devices and services
    was completed successfully.
    message = "<br>" + "SEARCH COMPLETED!";
    if(serviceFound.size() == 0) message = message + "<br>
    didn't find any service... please try again!";
}
// If problems happened, show an error message instead.
else message = message + "<br>" + "SEARCH COMPLETED WITH ERRORS...";
switch(sensorIndex){
    case 0:
        GUI.userMessage1.setText("<html>" + message + "</html>");
        break;
    case 1:
     //   GUI.userMessage2.setText("<html>" + message + "</html>");
        break;
}
// Now the system is waiting for connection. Display "bluetooth not
```

```
    connected" message on jLabel3,
    // disable jButton2, enable jButton1, jButton3, jButton4
    and jButton5.
    GUI.ButtonSettings(1, sensorIndex);
}

/**
 * Receives pressure values from Bluetooth device.
 * @param GUI
 */
public void updateValues(JFrame2 GUI){
    boolean state = false;  // Flag used to indicate the program is
                     //still reading a new group of pressure values.
    char zipState=0;
    int value = 0;  // Value observed in the multiple readings.
    int times = 0;  // How many multiple readings are being
                    //represented by "value" variable.
    int i=0;//Position where incoming value will be saved in vector.

    // While program is connected to the device...
    while(connected == true){
        // Reading values might fail, so wrap it in try-catch block.
        try {
            // Read next byte of data from the input stream
            //(coming via Bluetooth).
            // The value is returned as an int in the range 0 to 255.
            // If the end of the stream is reached, returns -1.
            int ch = input.read();
                if(ch != -1){
                    // Print incoming value.
                    // If starting to read a new group of pressure
                    values...
                    if(state == false){
                        // ch == 255 - received start byte for master
                        if(ch==255)
                        {
                            // Update flag and reset position.
                            state=true;
                            i=0;
```

```
            sensorIndex=0;
         // ch == 254 means the system received
         the start byte for slave module
        } else if(ch==254)
        {
        state=true;
            i=0;
            sensorIndex=1;
        }
    }
    // If group has already started...
    else{
        // If received the start byte, update flag
        //and reset position value.
        if((ch==255)&&(zipState==0)) //start byte master
        {
        state=true;
            i=0;
            sensorIndex=0;
            //ch = input.read();
        }else if((ch==254)&&(zipState==0))//start byte slave
        {
    //zipstate=0; //nova - dps apagar
        state=true;
            i=0;
            sensorIndex=1;
            //ch = input.read();
        }
        if(zipState==0){
        // If the byte read isn't 254, it means
        //the value received represents only one
        reading.
        // Save it in the pressao vector and
        //move to the next position.
        GUI.pressure[sensorIndex][i]=ch;
        System.out.println(GUI.pressure[sensorIndex][i])
        ;
        i++;
        System.out.println(i);
```

```
                        }
                        // When i == 1260, pressao vector is full.
                        if(i==1260)
                        {
                            // Update pixels in the GUI.
                            switch(sensorIndex){
                                case 0:
                                GUI.tmain1.run();
                                break;
                                case 1:
                                GUI.tmain2.run();
                                break;
                            }
                            // Mark flag.
                            state = false;
                        }
                    }
                }
            } catch (IOException ex) {
                // If an error occurs, display a message on userMessage
                // component of the GUI, change "connected" flag
                and exit loop.
                switch(sensorIndex){
                    case 0:
                    GUI.userMessage1.setText("Error: " + ex);
                    break;
                    case 1:
                    System.out.println("sensor index 1");
                   //GUI.userMessage2.setText("Error: " + ex);
                    break;
                }
                System.out.println("connected = false");
                connected = false;
                break;
            }
        }
    }
```

```
/**
 * Runs when bluetooth thread is started.
 */
public void run(){
    // If not connected, search for Bluetooth devices.
    if(connected == false){
        this.bluetoothInquiry(originalGUI);
    }
    // If connected to device, receive pressure values coming
    from the sensors.
    else{
        this.updateValues(originalGUI);
    }
}

/**
 * Establish a connection to the server.
 * @param GUI
 * @return
 */
public boolean connectToServer(JFrame2 GUI) {
    int ServiceIndex = 0;
    try {
        switch(sensorIndex){
            case 0:
            ServiceIndex = GUI.jList1.getSelectedIndex();
            break;
            case 1:
       //   ServiceIndex = GUI.jList2.getSelectedIndex();
            break;
        }
        String a = (String) GUI.listModel.elementAt(ServiceIndex);
        int connectionIndex = Integer.parseInt(a.split(" ")[0]);
        System.out.println("connectando a... " + connectionIndex + " :"
        + devicesDiscovered.elementAt(connectionIndex));
        System.out.println(friendlyNameDiscovered);
        RemoteDevice btDevice = (RemoteDevice)devicesDiscovered.
        elementAt(connectionIndex);
        String pin="1234";
```

```
            RemoteDeviceHelper.authenticate(btDevice, pin);
            conn = (StreamConnection)Connector.open((String)serviceFound
            .elementAt(ServiceIndex));
            input = conn.openInputStream();
            output = conn.openOutputStream();
            connected = true;
            return true;
        } catch (Exception e) {
            switch(sensorIndex){
                case 0:
                GUI.userMessage1.setText("Error: " + e);
                break;
                case 1:
                //GUI.userMessage2.setText("Error: " + e);
                break;
            }
            return false;
        }
    }

    /**
     * End bluetooth connection.
     * @param GUI
     * @return
     */
    public boolean close(JFrame2 GUI){
        try {
            connected = false;
            output.close();
            input.close();
            conn.close();
            return true;
        } catch (IOException ex) {
            switch(sensorIndex){
                case 0:
                GUI.userMessage1.setText("<html>" + "Error: " + ex +
                "</html>");
                break;
                case 1:
```

```
                // GUI.userMessage2.setText("<html>" + "Error: " + ex +
                "</html>");
                    break;
                }
                return false;
            }
        }
}
```