UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ENGENHARIA DE SÃO CARLOS
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

Plataforma baseada em *Android* para auxílio na eletroencefalografia

Autor: João Pedro Berti Ligabô

Orientador: Dr. Evandro L. L. Rodrigues

São Carlos
2014

JOÃO PEDRO BERTI LIGABÔ

Plataforma baseada em *Android* para auxílio na eletroencefalografia

Trabalho de Conclusão de Curso apresentado à
Escola de Engenharia de São Carlos
Universidade de São Paulo

Curso de Engenharia Elétrica com ênfase em Eletrônica

Orientador: Dr. Evandro L. L. Rodrigues

São Carlos
2014

Ligabô, João Pedro Berti
L723p Plataforma baseada em Android para auxílio na eletroencefalografia / João Pedro Berti Ligabô; orientador Evandro Luís Linhari Rodrigues. São Carlos, 2014.

Monografia (Graduação em Engenharia Elétrica com ênfase em Eletrônica) -- Escola de Engenharia de São Carlos da Universidade de São Paulo, 2014.

1. EEG. 2. OpenEEG. 3. Android. 4. Bluetooth. I. Título.

FOLHA DE APROVAÇÃO

Nome: João Pedro Berti Ligabô

Título: "Plataforma baseada em Android para auxílio na eletroencefalografia"

Trabalho de Conclusão de Curso defendido e aprovado em 25/11/2014,

com NOTA 10,0 (dez, zero), **pela Comissão Julgadora:**

Prof. Associado Evandro Luís Linhari Rodrigues - (Orientador - SEL/EESC/USP)

Prof. Dr. Marcelo Andrade da Costa Vieira - (SEL/EESC/USP)

Prof. Dr. José Marcos Alves - (SEL/EESC/USP)

Coordenador da CoC-Engenharia Elétrica - EESC/USP:
Prof. Associado Homero Schiabel

Agradecimentos

Agradeço a todas as pessoas que fizeram parte diretamente e indiretamente na minha breve existência.

Agradeço, em particular, a todos aqueles que frequentam a Universidade de São Paulo e a todas as pessoas que pude conhecer durante o tempo em que estudei na França.

Agradeço a todos aqueles que colocam a educação e o conhecimento como prioridades em suas vidas.

Agradeço aos grandes nomes da história que construíram as bases do nosso conhecimento e que são a razão por nossos princípios atuais.

Agradeço a Deus.

"Só sei que nada sei."
(Sócrates)

Resumo

Este trabalho faz a transmissão dos sinais cerebrais obtidos no couro cabeludo por meio de um dispositivo *opensource* chamado OpenEEG para um celular com *Android*. Primeiramente os sinais fisiológicos cerebrais interessantes, que são da ordem de microvolts, foram obtidos no couro cabeludo por meio do OpenEEG e em seguida transmitidos via serial por meio da USB para uma plataforma móvel onde foi feito o tratamento de sinais necessários, em seguida estes sinais foram transferidos via *bluetooth* para um celular com *Android* onde foram, finalmente, plotados no tempo com o auxílio da biblioteca *Androidplot*.

Palavras chaves: EEG, *OpenEEG*, *Android*, *Bluetooth*, *Androidplot*

Abstract

This work aims to develop the transmission of the electrical activity along the scalp to an Android application. First, the useful physiological signals, which measure about microvolts, were obtained along the scalp with the help of an opensource hardware called OpenEEG and then transfered through the serial communication using the USB to a mobile platform, where the signal processing occurred, after, these signals were transfered via Bluetooth to a smartphone with Android where they were finally plotted with the help of a library called androidplot.

Keywords: EEG, OpenEEG, Android, Bluetooth, Androidplot

Lista de Figuras

Lista de Tabelas

Siglas

EEG = Eletroencefalografia / Eletroencefalograma

EOG = Eletrooculograma / Eletrooculografia

DIY = *Do It Yourself*

USB = *Universal Serial Bus*

FTDI = *Future Technology Devices International*

CC = Corrente Contínua

AC = Corrente Alternada

UUID = *Universally Unique IDentifier*

Sumário

Capítulo 1

Introdução

1.1 Explicações Iniciais

Este trabalho visa o estudo da EEG (Eletroencefalografia) através de uma aplicação utilizando sinais cerebrais de modo não invasivo, ou seja, utilizando eletrodos no couro cabeludo para medir os sinais fisiológicos provenientes da comunicação entre os neurônios.

Para o projeto do EEG, foi considerado um aparelho *opensource* DIY (*Do It Yourself*) onde o intuito foi utilizar um aparelho com todas as etapas do processamento de sinais utilizadas em hardwares robustos profissionais mas sem empregar grandes recursos financeiros, o que é bastante difícil para este nível de projeto pois os sinais medidos são da ordem de microvolts. Em seguida foi implementada uma aplicação onde o sinal captado no couro cabeludo foi exibido em um celular com android.

Tudo começou em 1929 quando um psiquiatra alemão chamado Hans Berger (que trabalhava em Jena/Alemanha) anunciou ser possível registrar as atividades cerebrais em um papel sem a necessidade de uma intervenção cirúrgica (figura 1.1), ele chamou este método de eletroencefalografia. Ele também descobriu que estes sinais variavam se a pessoa estivesse dormindo ou não, se a pessoa tivesse certas doenças como a epilepsia e no caso da falta de oxigênio (hipóxia).

Essas descobertas foram revolucionárias e Berger conseguiu assim inovar a medicina, criando um ramo inteiramente novo, a neurofisiologia clínica. [1]

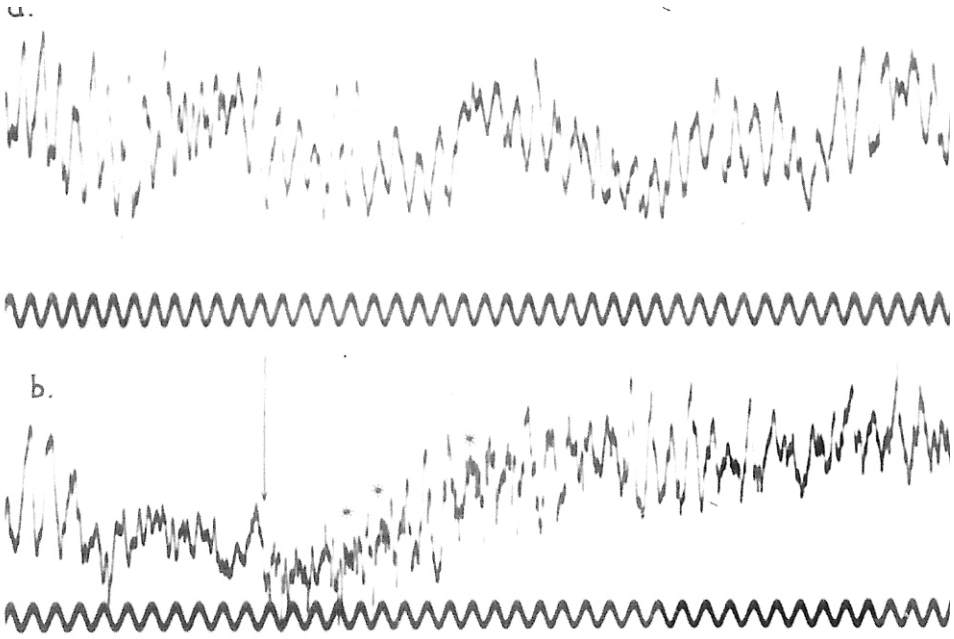


Figura 1.1: Primeiro EEG por Hans Berger [2]

1.2 Relevância

O uso da EEG está ainda muito distante de ser uma tecnologia a ser empregada de forma corriqueira uma vez que a medição das ondas cerebrais não é uma tarefa fácil pois para se ter uma precisão razoável é necessário fazer o uso de eletrodos a gel (visto que a região capilar dificulta a utilização de eletrodos secos) assim como também os sinais lá medidos não são significantes perto da grande quantidade de ruído que pode ser gerado se tal sistema fosse utilizado de forma móvel. Há também um certo tempo de calibração para a utilização do sistema e também outros cuidados que tornam este tipo de sistema pouco robusto ainda.

Por outro lado, a EEG pode ser usado e também é promissor nas áreas: médica (para a detecção de diversas patologias), jogos (consegue-se hoje manipular completamente um personagem dentro de um jogo), deficientes físicos (é possível que deficientes físicos se locomovam e também escrevam uma mensagem através de um editor de texto a base de estímulos neurais), e diversas outras interações homem-máquina. [3], [4]

O uso da eletroencefalografia é bastante difundido no caso de pacientes no estado *"locked in"* onde há uma lesão extensa das conexões neurais do cérebro com os movimentos do corpo, estado onde o paciente está consciente mas é incapaz de

demonstrar isso pois perdeu todos os movimentos do corpo, exceto o movimento vertical dos olhos. [5]

Este trabalho pode ser utilizado em casos em que o uso do EEG se faz necessário para uma avaliação prévia do estado do sujeito, como por exemplo: avaliação rápida da presença de epilepsia no paciente, acompanhamento do estado mental do paciente em uma anestesia, identificação de regiões no cérebro onde o paciente apresenta traumas, convulsões e tumores, entre outras aplicações. [6]

1.3 Objetivos

- Entender e manipular como é feito o sensoriamento da EEG encaminhando este sinal para uma plataforma móvel, ou seja, responder à questão de como captar as ondas no couro cabeludo e enviá-las até um celular com *Android*.

1.4 Princípios

1.4.1 Sinais Fisiológicos

Pode-se observar que os sinais medidos em torno do neurocrânio são da ordem de alguns microvolts e variam entre 1 à 40 Hz, visto que a medição foi feita de forma não invasiva, os eletrodos se encontram muito distantes de onde o sinal é produzido sendo assim necessária uma grande quantidade de neurônios transportando um sinal (gerando assim uma onda) para uma medição relevante do sinal cerebral. Além dos problemas de atenuação existem também problemas de precisão pois a condução dos sinais gerados pelos neurônios até o couro cabeludo se dá de forma caótica por meio da condução através dos diferentes materiais orgânicos. Por exemplo pode-se observar numa topografia cerebral 2D/3D que uma mancha na parte esquerda frontal/central do cérebro aparece (indicando que esta área está ativa) quando move-se a mão direita (ou quando imagina-se esse movimento).[7] [8] [9]

1.4.2 Processamento Digital

Para se fazer uma medição relevante e em tempo real de tais sinais é preciso certos cuidados com o tratamento de sinais, visto que eles são complexos, não

estacionários, possuem alta dimensão (dependendo da quantidade de eletrodos utilizados) e possuem muito ruído. Sendo assim é necessário saber a melhor maneira de explorar o sinal medido como por exemplo: explorar a resposta temporal a um estímulo externo ou explorar a densidade de potência do sinal em uma determinada banda de frequências, entre outras. Deste modo é necessário aproveitar as informações geométricas da geração do sinal (onde ele é gerado) e também sua natureza (frequência e forma).

Para a utilização de uma forma satisfatória e em tempo real do sinal é extremamente recomendado por diversos autores atuais [8] uma primeira etapa de calibração do sinal. Esta etapa se mostra importantíssima pois cada pessoa é única, sendo que a localidade dos sinais gerados assim como a percepção há certos estímulos se dá de uma maneira desigual entre as pessoas.

Após a etapa de calibração pode-se utilizar o sistema *online*, sendo que assim sua performance será muito maior pois o sistema já foi treinado para reconhecer o tipo de resposta de cada pessoa.

Há também uma etapa de remoção do ruído onde a relação sinal/ruído pode ser melhorada, pois a influência dos olhos (um piscar de olhos pode gerar sinais da ordem de milivolts em torno da cavidade ocular sendo propagada através da pele e atingindo principalmente os eletrodos frontais da EEG, como visto na figura 1.2) assim como a influência de vários outros fatores como: a fricção da mandíbula, o movimento dos braços, a própria tensão natural da pessoa, entre diversos outros sinais que podem ser muito superiores em amplitude que os sinais provenientes da EEG. [8], [9]

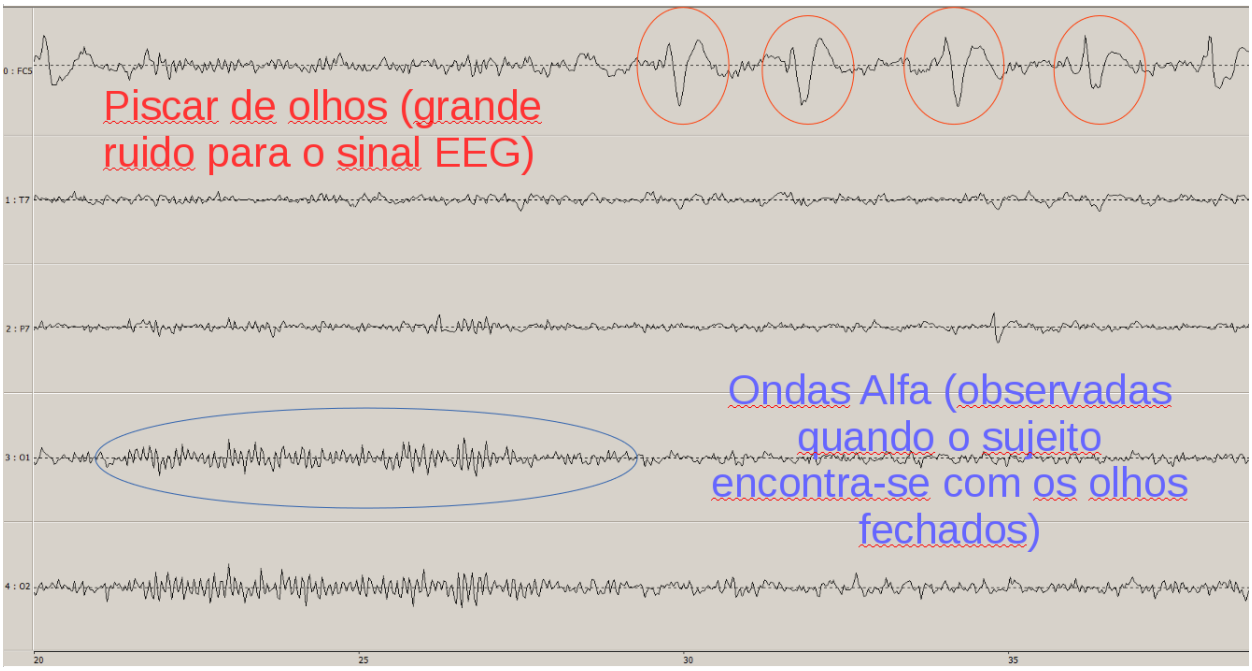


Figura 1.2: Visualização de um EEG

1.4.3 Estudo do *Hardware*

Vale ressaltar desde já que o nível das aplicações e resultados obtidos com um *hardware* mais barato são bastante inferiores comparados com aqueles resultados obtidos com um *hardware* mais robusto (caro) porém o princípio de funcionamento e as etapas para a confecção destes são bastante semelhantes, sendo que o estudo realizado sobre o OpenEEG proporcionou um grande conhecimento sobre as etapas necessárias para a instrumentação e processamento do sinal.

Para o estudo do *hardware* foram consideradas as etapas da figura 1.3

Nesta etapa do trabalho foi feito a comunicação serial do OpenEEG para o computador, a comunicação via *bluetooth* do computador para o celular e o desenvolvimento de um aplicativo para a visualização das ondas cerebrais. Para isso foi necessário compreender o funcionamento de todas as etapas do processamento de sinais fisiológicos.

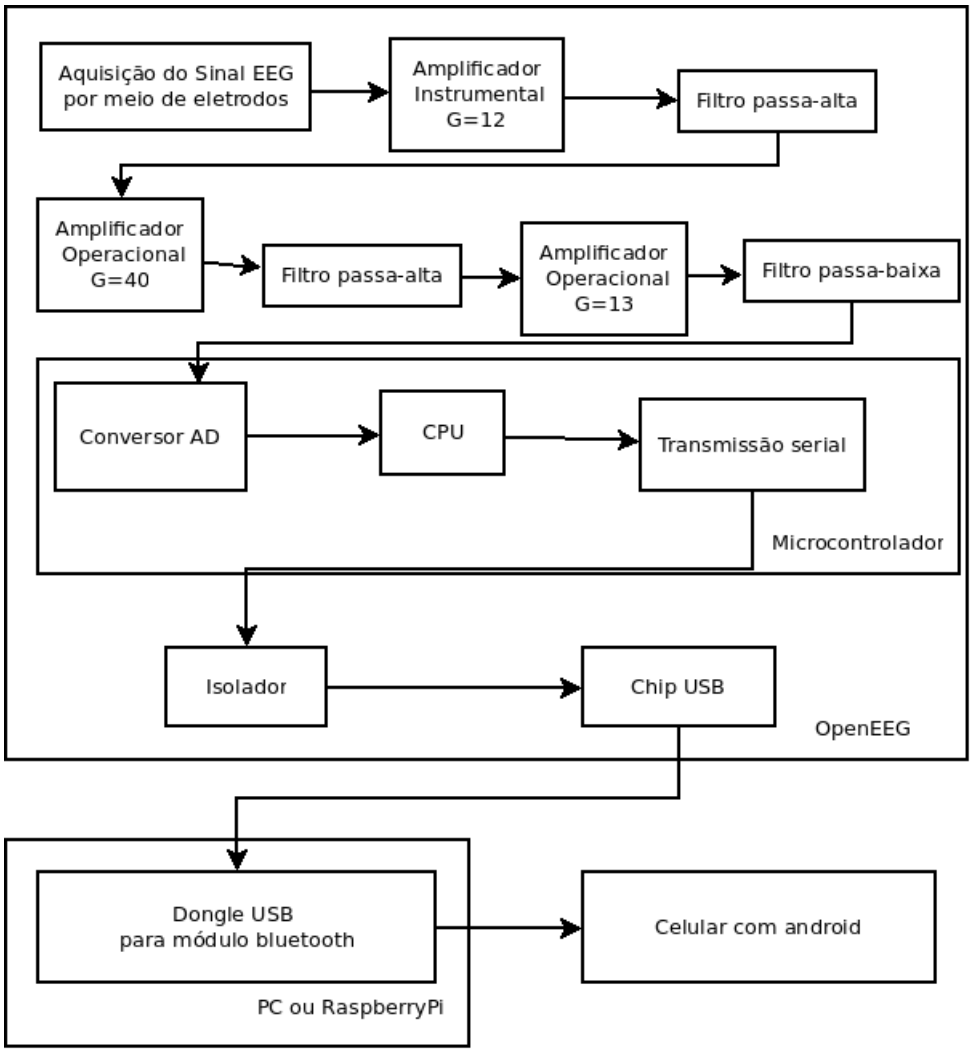


Figura 1.3: Fluxograma da transmissão dos dados

Capítulo 2

Embasamento Teórico

Para uma melhor compreensão deste assunto é preciso utilizar um vocabulário comum, sendo assim, abaixo seguem algumas convenções para facilitar o entendimento:

2.1 Sinais Fisiológicos

Os sinais presentes no couro cabeludo se encontram numa faixa de frequências que vai de 0.1 a 40 Hz aproximadamente, sendo que estes são classificados em bandas de acordo com a tabela 2.1. Estas informações são importantes pois é muito comum explorar a densidade de potência do sinal nas diferentes bandas para assim extrair informações importantes do sinal.

Banda	Domínio de frequência
Delta	0.1-3,5
Theta	4-7,5
Alfa	8-13
Beta	14-30
Gamma	> 30

Tabela 2.1: Banda de frequências em EEG [7]

Não serão discutidos aqui o porquê da frequência de tais sinais fisiológicos e nem o porquê de sua localidade pois seria preciso um conhecimento mais específico sobre o funcionamento do cérebro mas podemos obter da literatura atual uma grande quantidade de informações pertinentes ao projeto atual sem a necessidade de uma tal especialização. A figura 2.1 ilustra o Sistema Internacional de posicionamento dos eletrodos nas diferentes partes da cabeça para a exploração do sinal na área desejada.

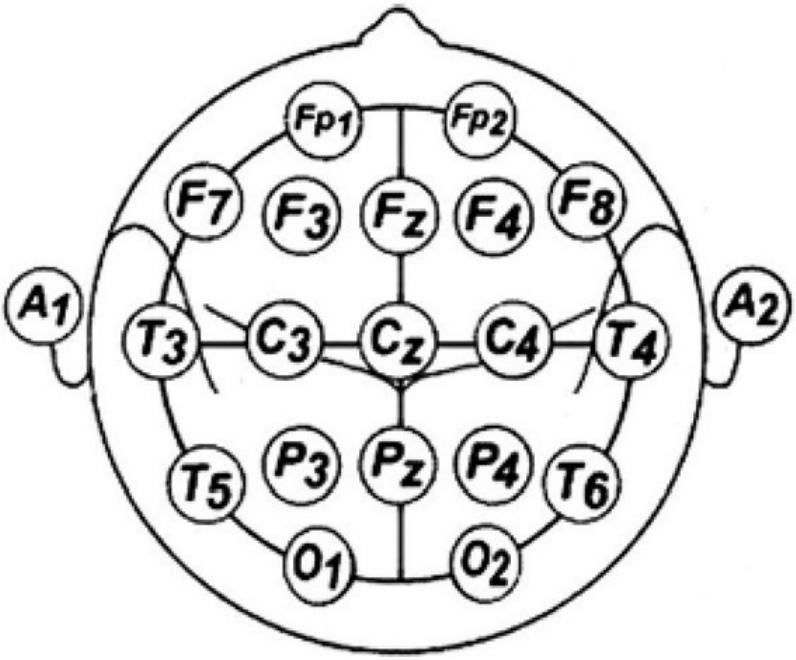


Figura 2.1: Sistema Internacional 10-20 [10]

2.2 Estudo do *hardware* envolvido no projeto

2.2.1 Dos sensores até o sistema embarcado

Um sinal EEG é normalmente captado por meio de eletrodos banhados a cloreto de prata embora as vezes possa ser utilizados outros materiais como a prata pura e o ouro. No caso do hardware baseado no projeto OpenEEG aqui estudado, o sinal é capturado por dois eletrodos e passa pelo circuito de proteção que tem a função de proteger o circuito contra descargas eletrostáticas.

Antes do sinal ser tratado ele precisa ser amplificado algumas milhares de vezes pois sua amplitude é da ordem de microvolts, devido à essa sua natureza frágil o sinal está exposto a introdução de diferentes fontes de ruído como por exemplo a indução elétrica da corrente que passa nos fios ao redor do aparelho.

Para cuidar disso o sinal é primeiramente amplificado por um amplificador instrumental (com ganho por volta de 12) de qualidade no qual se mede a diferença de potencial entre duas regiões do couro cabeludo, muitos ruídos externos são eliminados nesta parte pois ambos os eletrodos estão suscetíveis à mesma fonte de ruído, o que será cancelado pelo amplificador operacional posteriormente.

Em seguida o sinal passa por filtros passa-alta (com polos em 0.16 Hz) com o intuito de bloquear o sinal contínuo propagado até aqui pois muitos eletrodos são polarizáveis, ou seja, eles podem acumular carga elétrica em sua superfície, gerando assim um grande potencial CC, sendo que algumas vezes este sinal é da ordem de mV saturando assim completamente o amplificador e inviabilizando a captura do sinal EEG.

Em seguida o sinal é amplificado por meio de amplificadores normais (com ganho por volta de 40 e 13) e são utilizados filtros passa-baixa para minimizar a distorção causada por 'aliasing' quando o sinal for amostrado para se tornar um sinal digital. A figura 2.2 representa esta etapa inicial. [11]

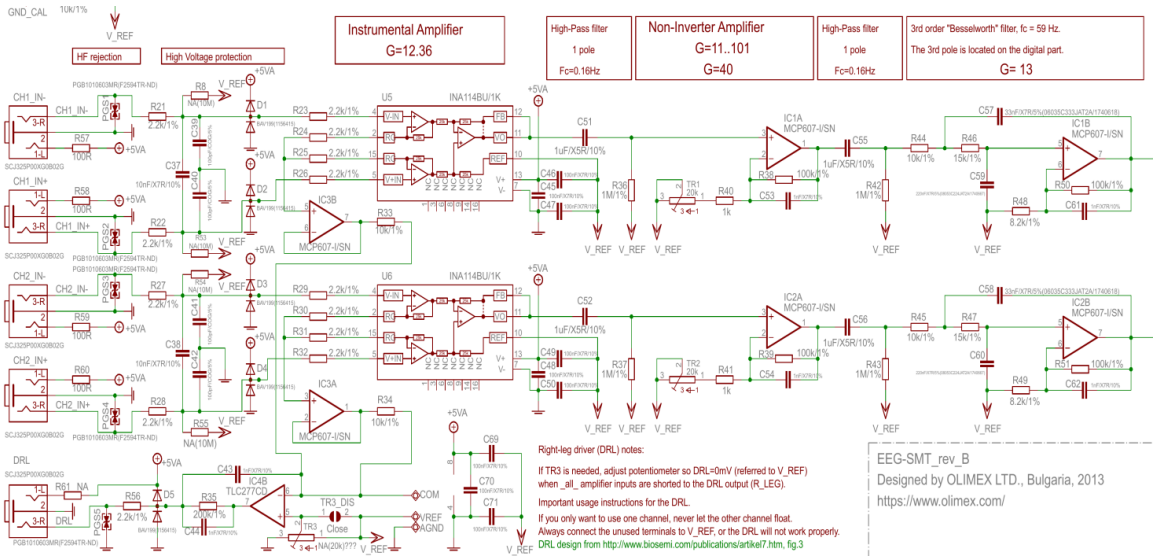


Figura 2.2: Etapa inicial do processo de aquisição dos sinais EEG [12]

Abaixo dos amplificadores de sinais e com a orientação contrária a esses, se encontra o DRL, cujo propósito é a rejeição do sinal de modo comum como por exemplo os sinais da rede elétrica.

Após esta etapa analógica o sinal é amostrado pelo microcontrolador (neste caso foi usado um ATMEGA16) e enviado para o computador de forma serial através de uma porta USB. Para proteger o usuário de problemas elétricos o sistema é isolado do computador e de fontes externas de energia. De acordo com a figura 2.3.

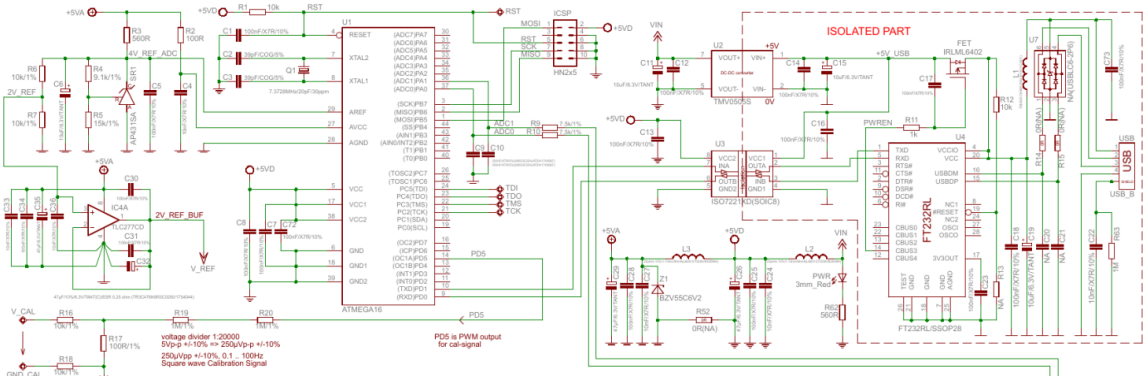


Figura 2.3: Amostragem e envio do sinal EEG do microcontrolador para a USB [12]

O circuito completo encontra-se no apêndice, assim como outras especificações dos componentes, mas as informações técnicas principais do projeto estão ilustradas na tabela 2.2

Número de canais	2 - 6(somente 4 utilizados)
Resolução	10 bits
Resolução da tensão de entrada	0.5 uV
Máxima tensão de entrada	+/-256 uV
Ruído de banda	1 uVp-p
Corrente fornecida (5V ou 9 - 12V tensão)	70 mA
Tensão de isolamento	2500V por 1 minuto
Tensão contínua de isolamento	480V

Tabela 2.2: Especificações gerais do projeto OpenEEG [11]

2.2.2 Do sistema embarcado até o celular

O sinal serial transmitido por meio da USB é recebido por um computador ou um dispositivo embarcado como por exemplo uma *Raspberry Pi*, este dispositivo deverá conter um módulo de recepção serial como ilustrado com os comandos abaixo:

```
joao@joao-pc:~ > lsusb
Bus 002 Device 004: ID 0403:6001 Future Technology Devices International,
Ltd FT232 USB-Serial (UART) IC
```

No caso aqui o módulo FTDI (FTDI é uma empresa escocesa de semicondutores especializada na manipulação de dados via USB) está corretamente instalado e

pronto para ser usado para a transmissão serial via USB, esse módulo é necessário pois a transmissão dos dados seriais enviados pelo microprocessador são primeiro convertidos para o protocolo USB por meio de um chip da FTDI localizado na última etapa da placa do kit da Olimex. [13]

```
joao@joao~ > lsmod
Module                  Size  Used by
ftdi_sio               48930  0
usbserial              45014  1 ftdi_sio
```

Com os dados sendo enviados na porta serial, pode-se recuperá-los de diversas formas, no linux eles estão normalmente disponíveis no arquivo/dispositivo /dev/ttyUSB0, sendo que para visualizar os sinais pode-se usar os comandos (como superusuário):

```
#Configurar a recepção serial para a porta /dev/ttyUSB0 e taxa de
#transmissão de 57600 bps
stty -F /dev/ttyUSB0 sane 57600

#Recepção de 17 bytes por pacote
od /dev/ttyUSB0 -tx1 -w17
```

Nota-se que para a recepção correta do sinal é necessário que a transmissão seja feita a cada 17 *bytes* pois o pacote possui este tamanho. Esta informação se encontra no *firmware* do microcontrolador, a seguir um pequeno extrato deste:

```
////////// Packet Format Version 2 ////////////

// 17-byte packets are transmitted from the ModularEEG at 256Hz,
// using 1 start bit, 8 data bits, 1 stop bit, no parity, 57600 bits
// per second.

// Minimial transmission speed is 256Hz * sizeof(modeeg_packet) * 10
// = 43520 bps.

struct modeeg_packet
{
```

```
uint8_t sync0; // = 0xa5
uint8_t sync1; // = 0x5a
uint8_t version; // = 2
uint8_t count; // packet counter. Increases by 1 each
// packet.
uint16_t data[6]; // 10-bit sample (= 0 - 1023) in big
// endian (Motorola) format.
uint8_t switches; // State of PD5 to PD2, in bits 3 to 0.
};

// Note that data is transmitted in big-endian format.
// By this measure together with the unique pattern in sync0 and
// sync1 it is guaranteed that re-sync (i.e after disconnecting
// the data line) is always safe.
```

Algumas considerações sobre a transmissão são importantes aqui:

1. A velocidade mínima de transmissão dos dados: O microcontrolador vai enviar os pacotes com uma frequência de 256 Hz (256 pacotes por segundo) sendo que a velocidade mínima então para a recepção será de: $256\frac{pacotes}{s} * 17\frac{palavras}{pacotes} * 10\frac{bits}{palavras} = 43520bps$ pois a amostragem é feita utilizando 10 bits.

2. sync0 e sync1 serão utilizados posteriormente para a sincronização do sinal. Sendo possível recuperar os dados exatos de cada canal graças à posição relativa aos *bytes* de sincronização.

3. Os dados serão transmitidos em dois *bytes* consecutivos seguindo o padrão big-endian (na ordem decrescente dos seus pesos numéricos) ou seja, para transmitir o valor 200 é necessário 2 palavras, sendo a decomposição igual à 16*12 + 08, por exemplo.

4. A placa suporta até seis canais, porém foram utilizados 4 neste projeto. [11]

Após todo esse procedimento de regulagem da recepção serial é preciso configurar o sistema *bluetooth* para o envio dos dados para o celular (neste caso foi utilizado um sistema android).

Neste projeto foi utilizado um *dongle USB* para a conexão *bluetooth* entre o computador e o celular, como ilustrado na figura 2.4

Figura 2.4: USB Bluetooth Dongle [14]

Para se fazer uma conexão *bluetooth* entre dois dispositivos é preciso que um deles atue como servidor e o outro cliente, para o usuário final não importa qual papel cada dispositivo vai desempenhar pois os dados podem ser transmitidos em ambos os sentidos (do servidor ao cliente ou vice-versa) porém é preciso respeitar a função de cada um deles como mostrado abaixo: [15]

- Função do servidor *bluetooth*
  O objetivo do servidor é ficar escutando inquisições de conexões (por meio da variável *BluetoothServerSocket*), para isto ele deve manter um canal aberto onde é informado o nome do servidor e uma UUID, quando uma conexão é aceita, é criado uma variável (*BluetoothSocket*) no servidor contendo os dados da conexão. Sendo que após a conexão a variável *BluetoothServerSocket* pode ser descartada a não ser que o servidor queira ficar procurando por novas conexões.

- Função do cliente *bluetooth*
  O objetivo do cliente é procurar por dispositivos que estejam oferecendo uma conexão, após ser encontrado um dispositivo o cliente tentará uma conexão caso os parâmetros especificados pelo servidor sejam os mesmos do cliente (o nome do servidor e o UUID), no caso dos dados serem os mesmos a conexão será estabelecida e o cliente armazenará os dados da conexão na variável *BluetoothSocket*.

Após a configuração tanto do servidor como do cliente *bluetooth* é necessário começar a transmissão e tratamento dos dados utilizando um aplicativo para android.

2.3 Programação em Java para android

Agora serão abordados tópicos necessários para a programação e compreensão de um aplicativo para android (baseado em java). Não serão abordados a história, estado de arte e programação orientada a objetos pois estes tópicos serão considerados já compreendidos pelo leitor que não quer se fatigar com sua revisão, sendo assim apenas o que é interessante para o entendimento do projeto será exposto aqui.

Primeiramente é preciso ter em mente a definição do que é uma *activity* e o que é um *thread*. [16]

Começando pelas explicações da *activity*: Uma *activity* é um componente da aplicação que fornece uma tela na qual o usuário pode interagir com o intuito de executar uma ação, como por exemplo fazer uma ligação, tirar uma foto ou enviar um e-mail. Cada *activity* possui seu próprio design sendo que dentro de um aplicativo é comum haver várias *activities* que são ou não ligadas entre si. A *activity* principal é chamada de *main activity* e deve cumprir alguns requisitos especiais como por exemplo: ela nunca pode ser bloqueada durante sua execução por mais de 5 segundos senão o programa causará um erro e o aplicativo será fechado. Mais detalhes sobre as *activities* serão informados na seção de 'aplicações'. Segue na figura 2.5 o ciclo de vida de uma *activity* [17] [18]:

Uma *thread* pode ser definida por uma unidade de processamento que pode ser executada concomitantemente com outros processamentos. Toda aplicação possui pelo menos uma *thread* que será executada no início da aplicação. As *threads* são importantes pois permitem que vários processos sejam executados ao mesmo tempo liberando o sistema para a execução de outras atividades.

No caso da aplicação aqui presente, é fundamental a utilização de uma *thread* para a utilização do *bluetooth* pois este serviço requer uma chamada que bloqueia o serviço, sendo assim a *thread* que vai tratar da conexão *bluetooth* ficará em modo de escuta e portanto bloqueada até que uma conexão seja estabelecida, desta forma bloqueia-se uma *thread* mas o aplicativo continua rodando e o usuário nem percebe o que se passa pois a *main thread* estará sempre liberada para o uso. [20]

O importante a ser considerado aqui é responder à seguinte questão: como fazer a comunicação entre diferentes *activities*?

Não existe uma resposta simples para esta pergunta que satisfaça todas as aplicações pois os objetivos de cada comunicação são diferentes, por exemplo pode-se usar uma *ActivityforResult*, um *Broadcast Receiver*, um *Handler* entre outras opções.

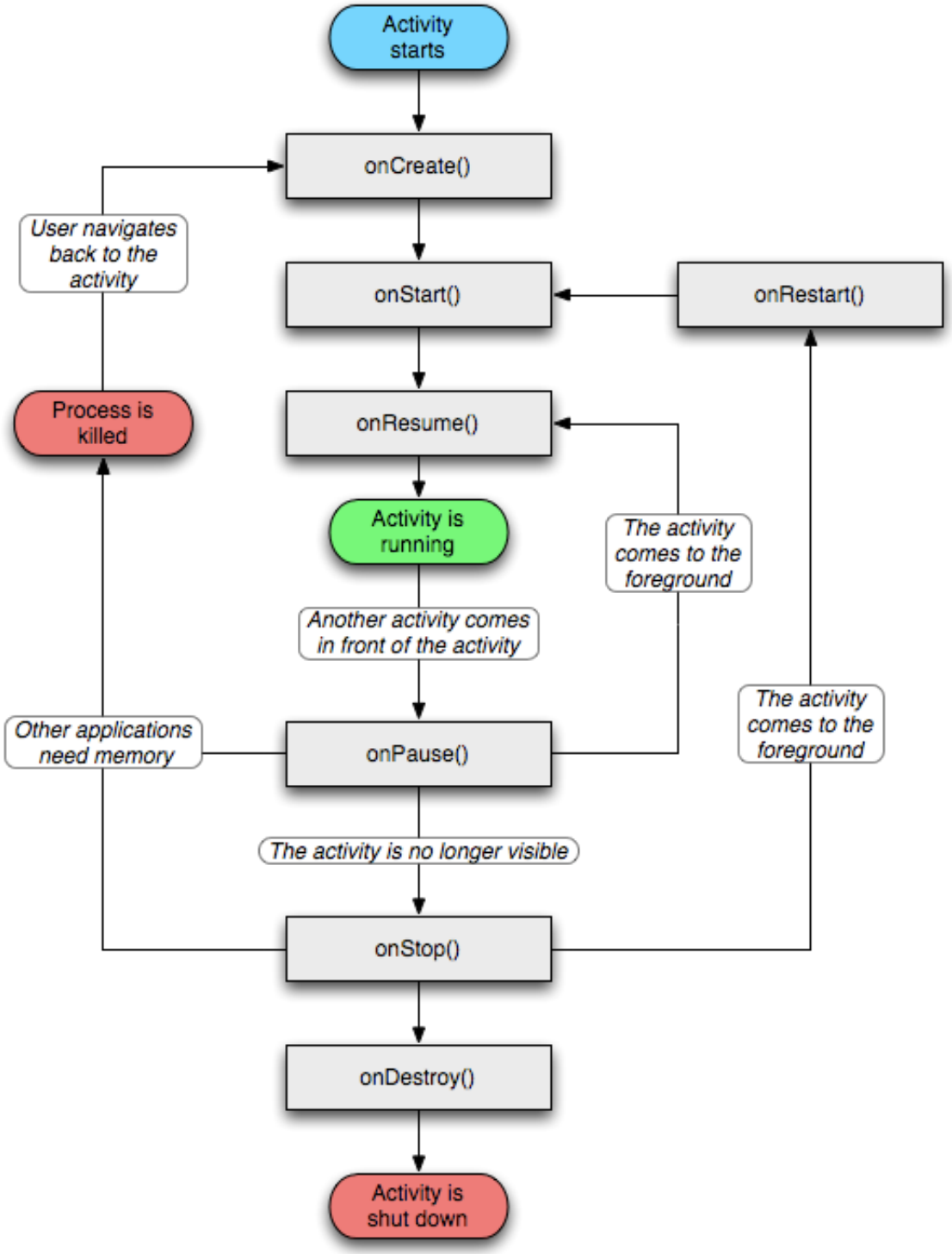


Figura 2.5: Ciclo de vida de uma *activity* [19]

Não serão detalhados todos os métodos para a comunicação entre as *activities* mas será justificado o método utilizado para a aplicação. [21]

Um indício para a resposta desta pergunta está em saber o que é uma *intent*. *Intents* são mensagens assíncronas que permitem que componentes de uma aplicação requisitem funcionalidades de outras aplicações android e permitam interagir também com componentes da mesma aplicação. Por exemplo, uma *activity* pode requisitar executar uma música utilizando um outro aplicativo feito para a execução de músicas. Um *intent* pode também conter dados utilizando um *bundle* sendo que esses dados podem ser usados pelo componente receptor.

A classe *intent* tem várias aplicações e sua documentação é bastante extensa sendo que aqui será explorado apenas o seu uso mas é de extrema importância para o desenvolvedor de aplicativos android conhecer algumas de suas utilidades. [22]

Capítulo 3

Materiais

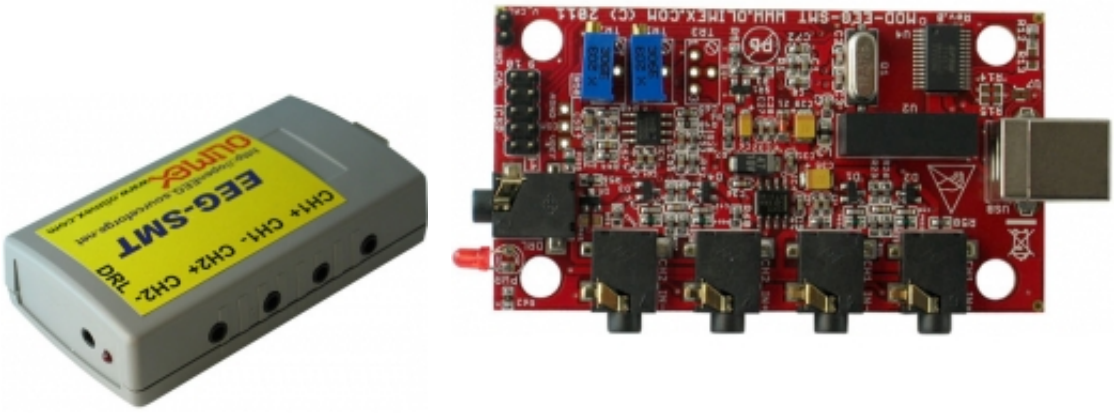
Atualmente os aparelhos comerciais para medição dos sinais cerebrais são muito caros inviabilizando assim o seu uso para este projeto, restando apenas a opção da utilização de um aparelho que não seja tão caro e que atenda satisfatoriamente à aplicação desejada.

3.1 OpenEEG

Inicialmente foi adquirido uma placa da Olimex, representado na figura 3.1 (o projeto pode ser construído ou comprado por uma média de 100 euros), juntamente com alguns eletrodos passivos e ativos, baseada no projeto OpenEEG, um projeto open-source que visa a criação de um dispositivo completo de até 6 canais para a medição de tais ondas (em anexo mais detalhes).

Este dispositivo possui várias limitações (pouca quantidade de canais, seus eletrodos (figura 3.2) são desconfortáveis e de difícil implementação e o sinal obtido apresenta muito ruído) mas foi a única solução de hardware viável para a elaboração deste projeto. [23]

Ele foi utilizado durante a etapa da coleta dos sinais no couro cabeludo para sua posterior visualização em um aplicativo *Android*.



(a) Box do kit (b) Circuito do kit

Figura 3.1: Kit EEG Olimex [24]

(a) Eletrodo ativo da Olimex [25] (b) Kit Olimex OpenEEG pronto para ser usado

Figura 3.2: OpenEEG com eletrodos

Capítulo 4

Aplicações

4.1 Estudo do *Hardware*

O intuito do projeto foi de aprender sobre o hardware a ser utilizado no processo de obtenção do sinal EEG, sendo que o sinal nesta parte do trabalho não foi tratado numericamente da mesma maneira e intensidade quando se faz o uso de softwares específicos para o seu tratamento, como por exemplo o OpenVibe [26].

A aplicação aqui apresentada consiste em extrair o sinal EEG por meio de um hardware de baixo custo (OpenEEG) e exibí-los em tempo real em um aplicativo *Android*.

Para médicos ou pessoas com experiência na leitura de um EEG, é relativamente fácil a visualização de certas patologias como a epilepsia sendo que esta aplicação apresenta uma relevância na área de instrumentação médica no intuito de diminuir os custos para certos exames que não demandam uma grande precisão.

A visualização do EEG em tempo real também permite o acompanhamento da eficácia de uma anestesia e algumas outras aplicações médicas.

Nesta seção será explicado em mais detalhes o percurso da informação desde sua captura no couro cabeludo até sua visualização em um aplicativo *Android*.

Primeiramente o sinal extraído se encontra na faixa de -256 μV 256μV , pois a resolução do sistema é de 0,5μV , a amostragem é feita utilizando 10 bits e os amplificadores possuem ganhos para optimizar a recepção do sinal nesta faixa de valores. O sinal é recebido por meio da transmissão serial USB e se encontra na forma a seguir: (representação da recepção de 5 pacotes de informação) [11]

```
joao@joao-PC > sudo od /dev/ttyUSB0 -tx1 -w17
a5 5a 02 3e 01 fe 01 fe 01 f5 01 ed 01 e5 01 df 0f
a5 5a 02 3f 01 fe 01 fe 01 f7 01 f0 01 e9 01 e2 0f
a5 5a 02 40 01 fd 01 fe 01 fb 01 f7 01 f2 01 ec 0f
a5 5a 02 41 01 fe 01 fe 01 fa 01 f6 01 f1 01 eb 0f
a5 5a 02 42 01 fe 01 ff 01 f7 01 ef 01 e8 01 e2 07
```

Conforme explicado na seção de embasamento teórico, o pacote se apresenta com 17 palavras, sendo que as informações importantes são as seguintes (em ordem):

```
a5: representa o sync0
5a: representa o sync1
02: representa a versão do firmware
xx: contador
CH1p_H: byte mais significativo do canal1 positivo
CH1p_L: byte menos significativo do canal 1 positivo
CH1n_H: byte mais significativo do canal1 negativo
CH1n_L: byte menos significativo do canal 1 negativo
CH2p_H: byte mais significativo do canal2 positivo
CH2p_L: byte menos significativo do canal 2 positivo
CH2n_H: byte mais significativo do canal2 negativo
CH2n_L: byte menos significativo do canal 2 negativo
CH3p_H: byte mais significativo do canal3 positivo
CH3p_L: byte menos significativo do canal 3 positivo
CH3n_H: byte mais significativo do canal3 negativo
CH3n_L: byte menos significativo do canal 3 negativo
xx: byte não importante para o projeto.
```

Após extrair as informações úteis utilizando um computador ou um sistema embarcado, é preciso transferir estes pacotes para o aplicativo *Android* por meio do *bluetooth*, sendo assim a configuração utilizada neste projeto, é a seguinte:

- Celular faz o papel do servidor, sendo que ele vai receber os dados.
- Computador ou sistema embarcado faz o papel do cliente, sendo que ele vai enviar os dados.

O lado do cliente, foi feito utilizando um algoritmo em *python* para facilitar esta etapa de transmissão. Segue abaixo um trecho do algoritmo utilizado (no apêndice encontra-se o algoritmo completo):

```
import sys
from bluetooth import *
from time import sleep
service_matches = find_service( name = "android_servidor",
uuid = "5b553d50-4b4f-11e4-916c-0800200c9a66" )
if len(service_matches) == 0:
  print "servico não encontrado!"
  sys.exit(0)
first_match = service_matches[0]
port = first_match["port"]
name = first_match["name"]
host = first_match["host"]
print "connecting to ", host
sock=BluetoothSocket( RFCOMM )
sock.connect((host, port))
k=int(1)
while True:
  sock.send(str(k))
  sleep(1)
  k=k+1
  if k==15:
    break
sock.close()
```

O cliente procura primeiramente por um servidor com o nome *android_servidor* e a *UUID = 5b553d50-4b4f-11e4-916c-0800200c9a66*, no caso do serviço encontrado, o cliente vai criar uma variável *sock* do tipo *BluetoothSocket* e vai conectar com o servidor, fazendo com que o serviço de envio de dados possa ser estabelecido. Neste exemplo, o programa vai enviar os algarismos de 1 a 15 a cada segundo, porém na aplicação real é preciso enviar o sinal obtido a uma frequência de pelo menos 128 Hz pois os sinais EEG visualizados contém frequências menores que 64 Hz. Após o envio dos dados a conexão é fechada.

O lado do servidor (localizado no aplicativo android e programado em Java) segue o fluxo da figura 4.1

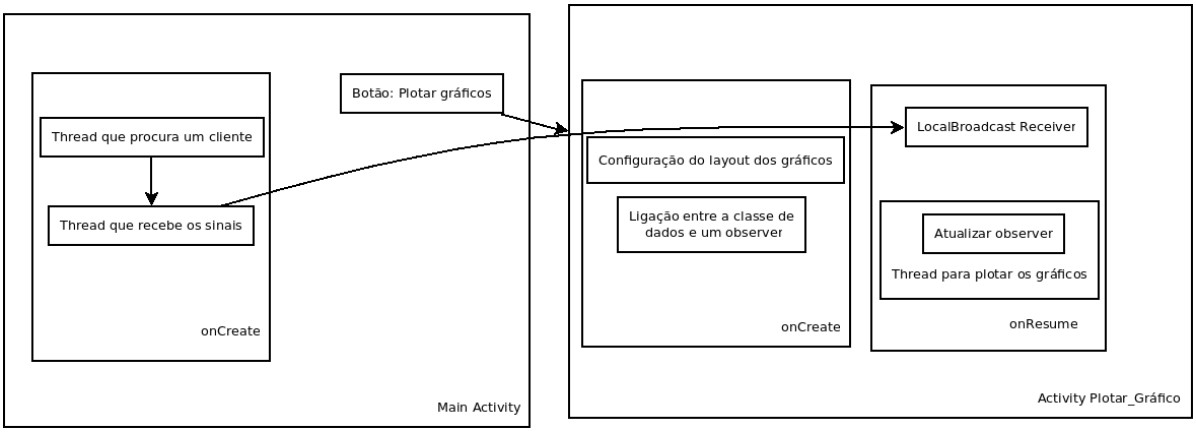


Figura 4.1: Diagrama das activities e threads do aplicativo

Sendo que as etapas importantes são explicadas a seguir:

1. habilitar o descobrimento do *bluetooth* (tarefa que vai automaticamente habilitar o bluetooth)

   O celular ficará visível para todos os dispositivos *bluetooth* durante um tempo de 300 segundos (figura 4.2), este é o tempo padrão que um dispositivo fica em estado de descoberta e pode ser mudado por meio de algumas opções nesta função.

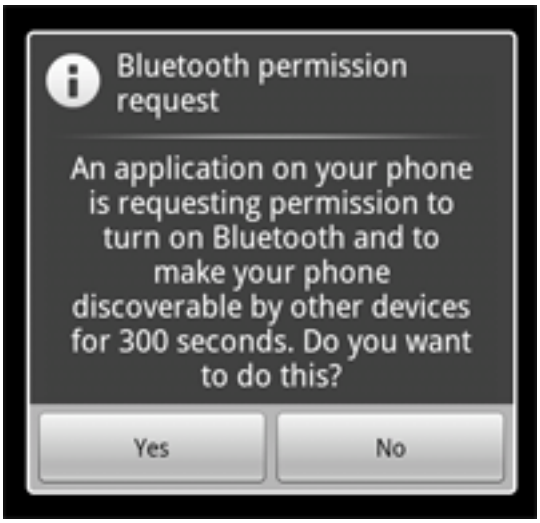


Figura 4.2: Requisição de acesso ao modo descoberta [15]

2. criação de uma *thread* para obter o socket de comunicação com o cliente

   Esta *thread* pode ser fechada assim que o *socket* for obtido, a não ser que o sistema queira procurar por novas conexões. [15]

3. criação de uma *thread* para receber os dados do cliente.

   Esta *thread* faz a recepção dos dados do cliente e os envia por meio de um *LocalBroadcast* à segunda *activity* responsável pela plotagem dos gráficos, ela também envia a um *handler* uma mensagem com os dados que podem ser visualizados tanto na *main activity* como na segunda *activity*. Esta *thread* só será interrompida, ou seja, o processo de envio dos dados só serão interrompidos por ordem explícita do usuário ou quando o programa for fechado.

Tudo isso ocorre sem o usuário perceber o que está acontecendo pois a conexão *bluetooth* é feita automaticamente (pois neste caso tanto o servidor como o cliente possuem os mesmos dados para conexão, que são o nome do servidor e a UUID).

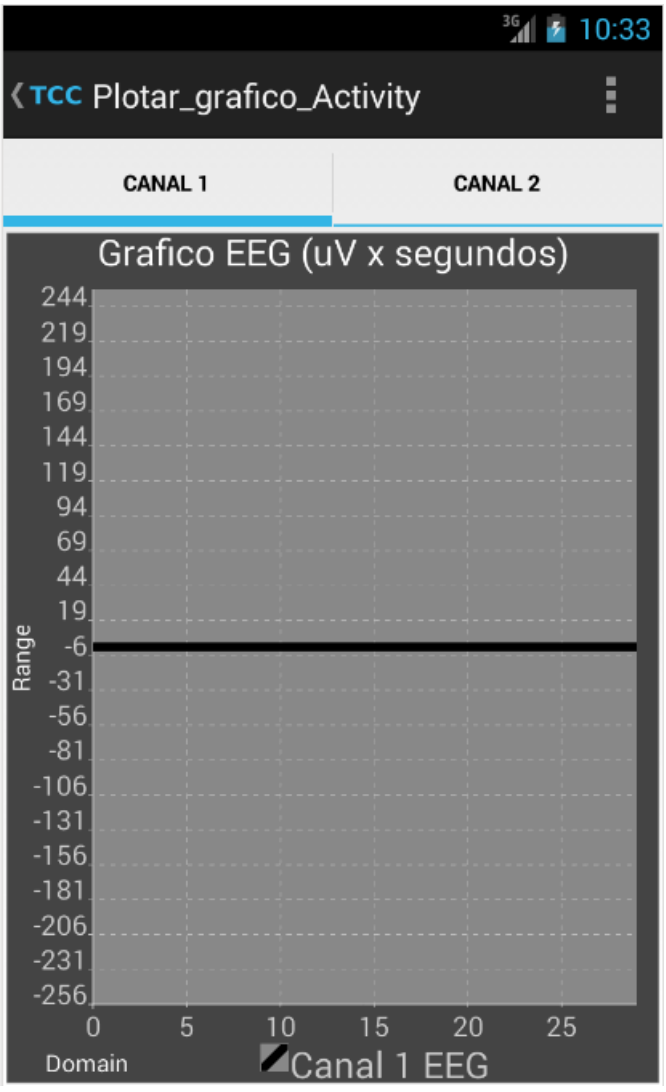


Figura 4.3: Layout da *activity* para plotar o sinal EEG

Em seguida o usuário pode começar a segunda *activity* onde será plotado um gráfico para cada canal do EEG.

Visto que neste trabalho foram utilizados apenas dois canais, o layout dos gráficos foi escolhido para ser o da figura 4.3

Sendo que no caso de mais canais utilizados, seria possível incluí-los em mais abas fixas ou móveis.

Para exibir os gráficos foi utilizada uma biblioteca chamada *androidplot* que é responsável pela criação de diversos tipos de gráficos, sendo necessário para esse projeto a configuração das diversas variáveis desta biblioteca para se fazer a visualização correta dos dados.

Encontra-se no apêndice como fazer a visualização dinâmica dos dados, sendo necessário o conhecimento de *notifications* em android para a compreensão do funcionamento da atualização dos dados. [27]

Capítulo 5

Resultados

5.1 Resultados obtidos

A etapa de transferência do sinal EEG para um aplicativo *Android* foi executada com sucesso, sendo que os objetivos propostos foram atendidos conforme propostos incialmente.

Pode-se observar o resultado do aplicativo na figura 5.1, onde os dados foram plotados em tempo real durante uma janela de 10 segundos (pois a amostragem do sinal para esta figura foi de 64 Hz) e com um filtro de Butterworth passa faixa (entre 3 e 20 Hz), conforme código de envio do sinal presente no apêndice C.

Nota-se também a presença do ruído proveniente do EOG (os dois picos da figura 5.1 que representam dois piscares de olhos), pois os eletrodos estão localizados na região frontal da cabeça.

A etapa referente ao estudo do *hardware* foi concluída onde começou o estudo referente ao processamento digital do sinal, mostrando assim a completude do trabalho, como pode ser visualizado na figura 5.2.

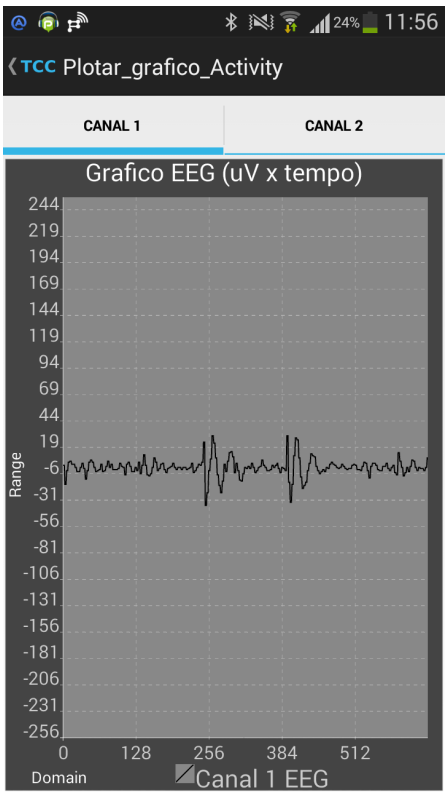


Figura 5.1: Aplicativo sendo executado em tempo real

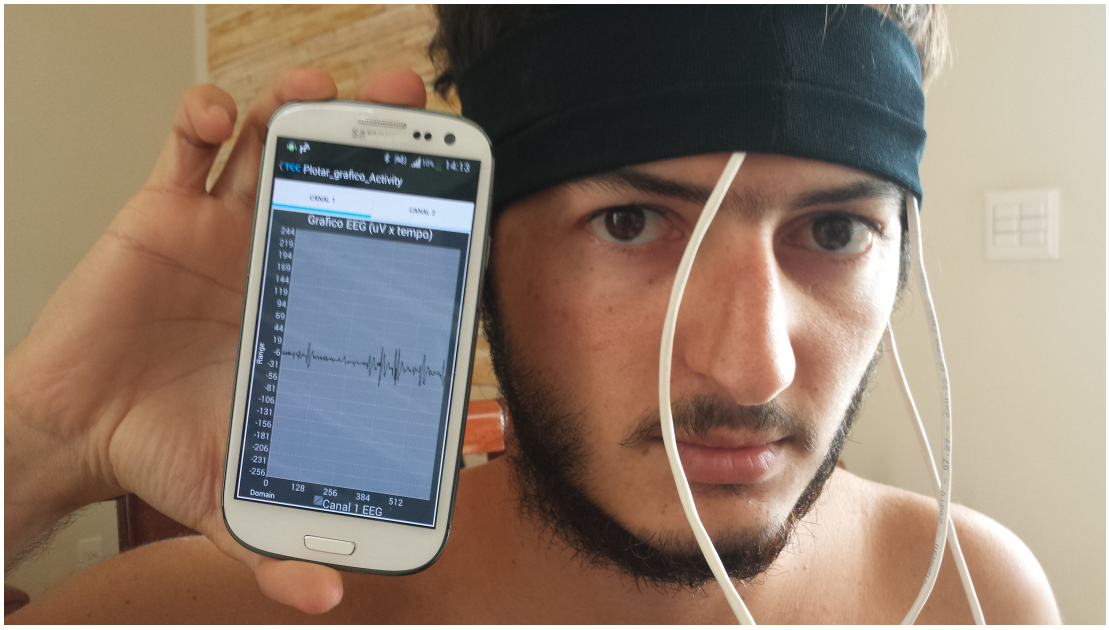


Figura 5.2: Ilustração do processo completo

Capítulo 6

Conclusões

6.1 Conclusões finais

Quando a idéia de fazer a transferência do sinal EEG para um aplicativo *Android* foi lançada, a única coisa que estava clara era o produto final (a visualização das ondas cerebrais no visor do celular) sendo que todo o processo para sua realização era desconhecido. Desta forma, este projeto se mostrou bastante eficiente do ponto de vista dos resultados alcançados pois estes estavam bem claros desde o início.

Visto que o caminho a ser percorrido para a obtenção do resultado final foi difícil e trabalhoso, este foi também bastante enriquecedor para o aprendizado do processo como um todo, que forneceu a compreensão desde o tratamento analógico do sinal até a criação de um aplicativo *Android*.

Vale ressaltar aqui algumas melhorias que podem ser feitas ao projeto mas que não foram feitas pela limitação dos recursos disponíveis (tempo e recursos financeiros), ficando deste modo para aperfeiçoamentos futuros.

- Melhoria do processo de obtenção do sinal EEG: para esta etapa foi utilizada uma placa da Olimex, sendo que o desempenho do sistema foi confiado à ela, visto que é bastante importante a etapa de amplificação do sinal EEG, seria melhor se esta etapa fosse melhorada.

- Melhoria da integração dos *hardwares*: considerando que o projeto fosse a criação de um produto comercial, a melhoria da integração das diferentes etapas do fluxo do sinal seria imprescindível para o custo e performance do projeto. Aqui foi utilizada uma placa da Olimex que envia o sinal digital do microcon-

trolador para uma porta USB e em seguida é necessário recuperar o sinal USB por meio de um computador ou sistema embarcado para o envio do sinal via *bluetooth*, sendo que este processo poderia ter sido otimizado (envio do sinal do microcontrolador ao *bluetooth* diretamente) se o kit da Olimex fornecesse este grau de liberdade. Do ponto de vista do aprendizado foi bastante interessante o que foi feito.

- Melhoria dos códigos a serem utilizados: para este projeto foram utilizados códigos em *python* que não é a linguagem mais eficiente a ser utilizada porém a performance final obtida foi condizente com o planejado. O programa para *Android* feito em Java pode ser melhorado para evitar possíveis falhas de vazamento de memória e os métodos utilizados para se fazer a transferência de dados entre duas *activities* poderiam ser melhores estudados para se ter um maior desempenho, porém o objetivo final foi cumprido pois não foram feitas especificações mais severas quanto ao aplicativo a ser produzido, sendo que para um projeto comercial, estas considerações deveriam ter sido feitas.

- Melhoria do programa para *Android* para fornecer diversas liberdades como: a mudança nas escalas x(temporal) e y(valores de tensão), filtros temporais e frequenciais para uma melhor visualização do sinal, reconhecimento de outros dispositivos e não só o OpenEEG e etc. Vale ressaltar que foi feita neste trabalho toda a base para o estudo do *hardware* envolvido no processamento de sinal, sustentando assim a premissa utilizada no estudo do *software*.

Referências Bibliográficas

[1] *A História do Eletroencefalograma*. Disponível em: http://www.cerebromente.org.br/n03/tecnologia/historia_ p.htm Acessado dia 20/10/2014.

[2] Berger H. *Uber das Elektrenkephalogramm des Menchen*. Archives für Psychiatrie. 1929; 87:527-70.

[3] *The Applications of EEG*. Disponível em: http://www.neuroelectrics.com/about_ eeg/applications Acessado dia 20/10/2014.

[4] *Medical Applications of EEG Wave Classification*. Disponível em: http://pages.cs.wisc.edu/ gangluo/eegII_chance.pdf Acessado dia 20/10/2014.

[5] *Síndrome de Locked-in*. Disponível em: http://www.neuroelectrics.com/about_ eeg/applications Acessado dia 20/10/2014.

[6] *What Can EEG Show?* Disponível em: http://www.epilepsy.com/learn/diagnosis/eeg Acessado dia 20/10/2014.

[7] *How is EEG studied*. Disponível em: http://www.neuroelectrics.com/about_ eeg/how-is-EEG-studied Acessado dia 20/10/2014.

[8] F. Lotte. *A tutorial on EEG Signal Processing Techniques for Mental State Recognition in Brain-Computer Interfaces*. Springer book.

[9] Jonathan R. et al. WOLPAW. *Brain–computer interfaces for communication and control.* Clinical neurophysiology, v. 113, n. 6, p. 767-791, 2002.

[10] *Sistema internacional 10-20*. Disponível em: http://what-when-how.com/neuroanaesthesia-and-neurointensive-care/neurophysiology-monitoring-and-imaging-part-1/ Acessado 17/10/2014.

[11] *Modular EEG Design*. Disponível em: http://openeeg.sourceforge.net/doc /modeeg/modeeg_ design.html Acessado dia 17/10/2014.

[12] *Olimex EEG-SMT Schematic*. Disponível em: https://www.olimex.com/Products/EEG/OpenEEG/EEG-SMT/resources/EEG-SMT-SCHEMATIC-REV-B.pdf Acessado 17/10/2014.

[13] *FTDI*. Disponível em: http://www.ftdichip.com/ Acessado dia 20/10/2014.

[14] *USB Bluetooth Dongle*. Disponível em: http://www.icisp.net.au/store/product_print.php?products_id=255 Acessado dia 17/10/2014.

[15] *Bluetooth Guide for Android*. Disponível em http://developer.android.com/guide/topics/connectivity/bluetooth.html Acessado dia 17/10/2014.

[16] *Activity*. Disponível em: http://developer.android.com/reference/android/app/Activity.html Acessado dia 20/10/2014.

[17] *Activities*. Disponível em: http://developer.android.com/guide/components/activities.html Acessado dia 20/10/2014.

[18] *Android Development Tutorial*. Disponível em: http://developer.android.com/guide/components/activities.html Acessado dia 20/10/2014.

[19] *Ciclo de vida de uma activity*. Disponível em: http://androideity.com/2011/07/06/ciclo-de-vida-de-una-actividad/ Acessado dia 17/10/2014.

[20] *Threads and Processes*. Disponível em: http://developer.android.com/guide/components/processes-and-threads.html Acessado dia 20/10/2014.

[21] *Painless threading*. Disponível em: http://android-developers.blogspot.com.br/2009/05/painless-threading.html Acessado dia 20/10/2014.

[22] *Android Background Processing with Handlers and AsyncTask and Loaders - Tutorial*. Disponível em: http://www.vogella.com/tutorials/AndroidBackgroundProcessing/article.html Acessado dia 20/10/2014.

[23] *OpenEEG Project*. Disponível em: http://openeeg.sourceforge.net/doc/ Acessado dia 20/10/2014.

[24] *OLIMEX PRODUCTS*. Disponível em: https://www.olimex.com/Products/EEG/OpenEEG/EEG-SMT/open-source-hardware Acessado dia 17/10/2014.

[25] *Low cost open source EEG device active electrodes with 1m shielded cable*. Disponível em: https://www.olimex.com/Products/EEG/Electrodes/EEG-AE/ Acessado dia 20/10/2014.

[26] *OpenViBE User Documentation*. Disponível em: http://openvibe.inria.fr/documentation-index/# User+Documentation Acessado dia 20/10/2014.

[27] *Android Notifications - Tutorial*. Disponível em: http://www.vogella.com/tutorials/AndroidNotifications/article.html Acessado dia 20/10/2014.

Apêndice A

Esquema completo do OpenEEG

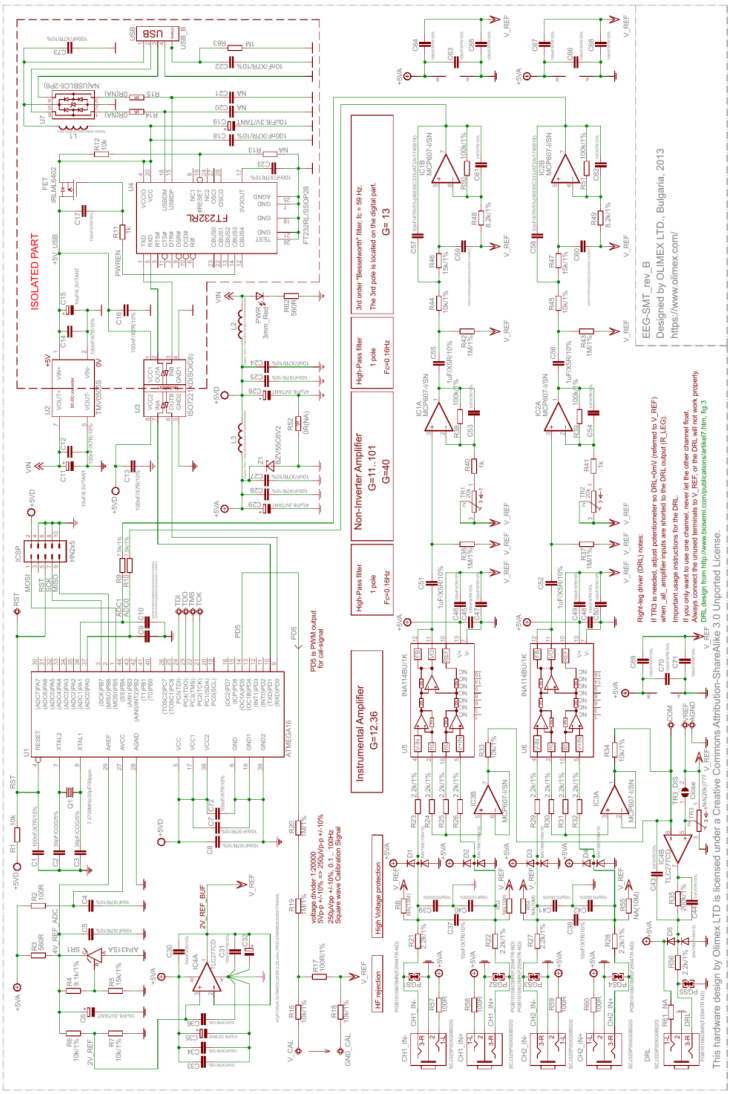


Figura A.1: Esquema completo do OpenEEG da Olimex

Apêndice B

Firmware do OpenEEG

```
/*
 * ModularEEG firmware for one-way transmission, v0.5.4-p2
 * Copyright (c) 2002-2003, Joerg Hansmann, Jim Peters, Andreas Robinson
 * License: GNU General Public License (GPL) v2
 * Compiles with AVR-GCC v3.3.
 *
 * Note: -p2 in the version number means this firmware is for packet version 2.
 */
////////////////////////////////////////////////////////////
/*
////////// Packet Format Version 2 ////////////
// 17-byte packets are transmitted from the ModularEEG at 256Hz,
// using 1 start bit, 8 data bits, 1 stop bit, no parity, 57600 bits per second.

// Minimial transmission speed is 256Hz * sizeof(modeeg_packet) * 10 = 43520 bps.
struct modeeg_packet
{
    uint8_t     sync0;      // = 0xa5
    uint8_t     sync1;      // = 0x5a
    uint8_t     version;    // = 2
    uint8_t     count;      // packet counter. Increases by 1 each packet.
    uint16_t    data[6];    // 10-bit sample (= 0 - 1023) in big endian (Motorola) format.
    uint8_t     switches;   // State of PD5 to PD2, in bits 3 to 0.
};

// Note that data is transmitted in big-endian format.
// By this measure together with the unique pattern in sync0 and sync1 it is guaranteed,
// that re-sync (i.e after disconnecting the data line) is always safe.
// At the moment communication direction is only from Atmel-processor to PC.
// The hardware however supports full duplex communication. This feature
// will be used in later firmware releases to support the PWM-output and
// LED-Goggles.
*/
////////////////////////////////////////////////////////////
/*
 * Program flow:
 * When 256Hz timer expires: goto SIGNAL(SIG_OVERFLOW0)
 * SIGNAL(SIG_OVERFLOW0) enables the ADC
 * Repeat for each channel in the ADC:
```

```
 * Sampling starts. When it completes: goto SIGNAL(SIG_ADC)
 * SIGNAL(SIG_ADC) reads the sample and restarts the ADC.
 * SIGNAL(SIG_ADC) writes first byte to UART data register
 * (UDR) which starts the transmission over the serial port.
 * Repeat for each byte in packet:
 * When transmission begins and UDR empties: goto SIGNAL(SIG_UART_DATA)
 * Start over from beginning.
 */
// ======================================================
// Last Built with WinAVR-20100110
// Supported processors: ATmega8, ATmega16 and AT90S4434
// ======================================================

#include <avr/io.h>
#include <inttypes.h>
#include <avr/interrupt.h>
#include <compat/deprecated.h>
//#include <avr/signal.h>
#define Debug_Led_On    PORTB |= 0x20;      DDRB |= 0x20;
#define Debug_Led_Off   PORTB &= (~0x20);   DDRB |= 0x20;

#if defined (__AVR_ATmega328P__)
    #define Crystal_Freq_16MHz 16 //16MHz external crystal oscillator - especially for
        Olimexino 328!
#else
    #undef  Crystal_Freq_16MHz  // In all other cases 7.3728MHz crystal is used!
#endif

#define NUMCHANNELS 6
#define HEADERLEN 4
#define PACKETLEN (NUMCHANNELS * 2 + HEADERLEN + 1)
#define SAMPFREQ 256
#define TIMER0VAL 256 - ((7372800 / 256) / SAMPFREQ)
//char const channel_order[]= { 0, 3, 1, 4, 2, 5 };
char const channel_order[]= { 0, 1, 2, 3, 4, 5 };
/** The transmission packet */
volatile uint8_t TXBuf[PACKETLEN];
/** Next byte to read or write in the transmission packet. */
volatile uint8_t TXIndex;
/** Current channel being sampled. */
volatile uint8_t CurrentCh;

/** Sampling timer (timer 0) interrupt handler */
//SIGNAL(SIG_OVERFLOW0)
ISR(TIMER0_OVF_vect)
{
    //Debug_Led_On;
    outb(TCNT0, TIMER0VAL); //Reset timer to get correct sampling frequency.
    CurrentCh = 0;
    // Write header and footer:
    // Increase packet counter (fourth byte in header)
    TXBuf[3]++;
    //Get state of switches on PD2..5, if any (last byte in packet).
    TXBuf[2 * NUMCHANNELS + HEADERLEN] = (inp(PIND) >> 2) &0x0F;
#if defined (__AVR_ATmega8__)
    cbi(UCSRB, UDRIE);      //Ensure UART IRQ's are disabled.
    sbi(ADCSR, ADIF);       //Reset any pending ADC interrupts
    sbi(ADCSR, ADIE);       //Enable ADC interrupts.
```

```
97  #elif defined (__AVR_ATmega16__)
        cbi(UCSRB, UDRIE);      //Ensure UART IRQ's are disabled.
99      sbi(ADCSRA, ADIF);      //Reset any pending ADC interrupts
        sbi(ADCSRA, ADIE);      //Enable ADC interrupts.
101 #elif defined (__AVR_ATmega328P__)
        cbi(UCSR0B, UDRIE0);     //Ensure UART IRQ's are disabled.
103     sbi(ADCSRA, ADIF);       //Reset any pending ADC interrupts
        sbi(ADCSRA, ADIE);       //Enable ADC interrupts.
105     sbi(ADCSRA, ADSC) ;      // Start conversion!!!!!!!!!!!!!!!!!!
    #else//(__AVR_AT90S4434__)
107     cbi(UCR, UDRIE);      //Ensure UART IRQ's are disabled.
        sbi(ADCSR, ADIF);       //Reset any pending ADC interrupts
109     sbi(ADCSR, ADIE);       //Enable ADC interrupts.
    #endif
111 }
    /** AD-conversion-complete interrupt handler. */
113
    //SIGNAL(SIG_ADC)
115 ISR(ADC_vect)
    {
117     volatile uint8_t i;
        i = 2 * CurrentCh + HEADERLEN;
119     TXBuf[i+1] = inp(ADCL);
        TXBuf[i] = inp(ADCH);
121     CurrentCh++;
        //Debug_Led_On;
123     if (CurrentCh < NUMCHANNELS)
        {
125         outb(ADMUX, (channel_order[CurrentCh]));  //Select the next channel.
            //The next sampling is started automatically.
127 #if defined (__AVR_ATmega328P__)
            sbi(ADCSRA, ADSC) ;      // Start conversion!!!!!!!!!!!!!!!!!!
129 #endif
        }
131     else
        {
133         outb(ADMUX, channel_order[0]);      //Prepare next conversion, on channel 0.
            // Disable ADC interrupts to prevent further calls to SIG_ADC.
135 #if defined (__AVR_ATmega8__) | defined (__AVR_AT90S4434__)
            cbi(ADCSR, ADIE);
137 #elif defined (__AVR_ATmega16__) | defined (__AVR_ATmega328P__)
            cbi(ADCSRA, ADIE);
139 #endif
    #if     defined (__AVR_ATmega328P__)
141         outb(UDR0, TXBuf[0]);
    #else
143         outb(UDR, TXBuf[0]);
    #endif
145
    #if defined (__AVR_ATmega8__) | defined (__AVR_ATmega16__)
147         sbi(UCSRB, UDRIE);
    #elif defined (__AVR_ATmega328P__)
149         sbi(UCSR0B, UDRIE0);
    #else//(__AVR_AT90S4434__)
151         sbi(UCR, UDRIE);
    #endif
153         TXIndex = 1;
            //Debug_Led_On;
```

```
155     }
}
157 /*** UART data transmission register-empty interrupt handler ***/
#if defined (__AVR_ATmega328P__)
159     ISR(USART_UDRE_vect)        /* USART, Data Register Empty interrupt */
#else
161     SIGNAL(SIG_UART_DATA)
#endif
163 {
#if defined (__AVR_ATmega328P__)
165     //Debug_Led_On;
        outb(UDR0, TXBuf[TXIndex]);  //Send next byte
167 #else
        outb(UDR, TXBuf[TXIndex]);  //Send next byte
169 #endif
        TXIndex++;
171     if (TXIndex == PACKETLEN)   //See if we're done with this packet
        {
173 #if defined (__AVR_ATmega8__) | defined (__AVR_ATmega16__)
            cbi(UCSRB, UDRIE);      //Disable SIG_UART_DATA interrupts.
175                                 //Next interrupt will be a SIG_OVERFLOW0.
#elif defined (__AVR_ATmega328P__)
177         cbi(UCSR0B, UDRIE0);    //Disable SIG_UART_DATA interrupts.
                                    //Next interrupt will be a SIG_OVERFLOW0.
179 #else//(__AVR_AT90S4434__)
            cbi(UCR, UDRIE);        //Disable SIG_UART_DATA interrupts.
181                                 //Next interrupt will be a SIG_OVERFLOW0.
#endif
183
        }
185 }
/** Initialize PWM output (PB1 = 14Hz square wave signal) */
187
void pwm_init(void)
189 {
    // Set timer/counter 1 to use 10-bit PWM mode.
191 // The counter counts from zero to 1023 and then back down
    // again. Each time the counter value equals the value
193 // of OCR1(A), the output pin is toggled.
    // The counter speed is set in TCCR1B, to clk / 256 = 28800Hz.
195 // Effective frequency is then clk / 256 / 2046 = 14 Hz

197 #if defined (__AVR_ATmega8__) | defined (__AVR_ATmega16__) | defined (__AVR_ATmega328P__)
    outb(OCR1AH,2); // Set OCR1A = 512
199 outb(OCR1AL,0);
    outb(TCCR1A, ((1<<COM1A1) + (1<<WGM11) + (1<<WGM10))); // Set 10-bit PWM mode
201 outb(TCCR1B, (1 << CS12));  // Start and let run at clk / 256 Hz.
#else // __AVR_AT90S4434__
203 outb(OCR1AH,2); // Set OCR1 = 512
    outb(OCR1AL,0);
205 outb(TCCR1A, ((1<<COM1A1) + (1<<PWM11) + (1<<PWM10))); // Set 10-bit PWM mode
    outb(TCCR1B, (1 << CS12));  // Start and let run at clk / 256 Hz.
207
#endif
209 }
int main( void )
211 {
#if defined (Crystal_Freq_16MHz)
```

```
213     //Devide system clock by two to acheive the same performance and time intervals
        sbi(CLKPR,CLKPCE);  // Clock Prescaler Change Enable
215     outb(CLKPR,0x01);   // Set prescaller to 1:2
    #endif
217     //Write packet header and footer
        TXBuf[0] = 0xa5;        //Sync 0
219     TXBuf[1] = 0x5a;        //Sync 1
        TXBuf[2] = 2;           //Protocol version
221     TXBuf[3] = 0;           //Packet counter
        //Set up the ports.
223 #if defined (__AVR_ATmega8__) | defined (__AVR_ATmega328P__)
        outb(DDRD,  0xc2);
225     outb(DDRB,  0x07);
        outb(PORTD, 0xff);
227     outb(PORTB, 0xff);
    #elif defined (__AVR_ATmega16__)
229     DDRD |= 0x22;           // Enable PWM output (PD5/OC1A) and TxD output
        PORTD |= 0x22;          // Set outputs to high
231 #endif
        //Select sleep mode = idle.
233 #if defined (__AVR_ATmega8__) | defined (__AVR_ATmega16__) | defined (__AVR_ATmega328P__)
        outb(MCUCR,(inp(MCUCR) | (1<<SE)) & (~(1<<SM0) | ~(1<<SM1) | ~(1<<SM2)));
235 #else // __AVR_AT90S4434__
        outb(MCUCR,(inp(MCUCR) | (1<<SE)) & (~(1<<SM0)) & (~(1<<SM1)) );
237 #endif
        //Initialize the ADC
239     // Timings for sampling of one 10-bit AD-value:
        // prescaler > ((XTAL / 200kHz) = 36.8 =>
241     // prescaler = 64 (ADPS2 = 1, ADPS1 = 1, ADPS0 = 0)
        // ADCYCLE = XTAL / prescaler = 115200Hz or 8.68 us/cycle
243     // 14 (single conversion) cycles = 121.5 us (8230 samples/sec)
        // 26 (1st conversion) cycles = 225.69 us
245     outb(ADMUX, 0);         //Select channel 0
        //Prescaler = 64, free running mode = off, interrupts off.
247 #if defined (__AVR_ATmega8__) | defined (__AVR_AT90S4434__)
        outb(ADCSR, ((1<<ADPS2) | (1<<ADPS1)));
249     sbi(ADCSR, ADIF);       //Reset any pending ADC interrupts
        sbi(ADCSR, ADEN);       //Enable the ADC
251 #elif defined (__AVR_ATmega16__) | defined (__AVR_ATmega328P__)
        outb(ADCSRA, ((1<<ADPS2) | (1<<ADPS1)));
253     sbi(ADCSRA, ADIF);      //Reset any pending ADC interrupts
        sbi(ADCSRA, ADEN);      //Enable the ADC
255 #endif
        //Initialize the UART
257 #if defined (__AVR_ATmega8__) | defined (__AVR_ATmega16__)
        outb(UBRRH,0);              //Set speed to 57600 bps
259     outb(UBRRL,7);
        outb(UCSRA, 0);
261     outb(UCSRC, ((1<<URSEL) | (1<<UCSZ1) | (1<<UCSZ0)));    // 8 bits data length
        outb(UCSRB, (1<<TXEN));                                 // Transmitter Enabled
263 #elif defined (__AVR_ATmega328P__)
        #if defined (Crystal_Freq_16MHz)
265         outb(UBRR0, 8);             //Set speed to 57600 bps
            outb(UCSR0B, (1<<TXEN0));
267     #else
            outb(UBRR0, 7);             //Set speed to 57600 bps
269         outb(UCSR0B, (1<<TXEN0));
        #endif
```

```
271 #else // __AVR_AT90S4434__
    outb(UBRR, 7);                //Set speed to 57600 bps
273     outb(UCR, (1<<TXEN));
#endif
275     //Initialize timer 0  -> used to sample ADC
#if defined (__AVR_ATmega328P__)
277     outb(TCNT0, 0);             //Clear it.
    outb(TCCR0B, 4);            //Start it. Frequency = clk / 256
279     outb(TIMSK0, (1<<TOIE0));   //Enable the interrupts.
#else
281     outb(TCNT0, 0);             //Clear it.
    outb(TCCR0, 4);             //Start it. Frequency = clk / 256
283     outb(TIMSK, (1<<TOIE0));    //Enable the interrupts.
#endif
285     //Initialize PWM (optional)
    pwm_init();
287     //while(1){ Debug_Led_On; outb(UDR0, 0x55); }   // Just for debug purpose!
    sei();  // Enable all interrupts
289     //Now, we wait. This is an event-driven program, so nothing much
    //happens here.
291     while (1)
    {
293         __asm__ __volatile__ ("sleep");
    }
295 }
```

codigos/main.c

Apêndice C

Transmissão do OpenEEG para o celular

Neste código foi utilizado um filtro passa faixa (de 3 a 20 Hz) com o intuito de melhorar a visualização do sinal pois o sinal bruto é muito ruidoso. Observa-se aqui o limite entre o tratamento do sinal analógico e digital.

```
import serial
import io
import sys
import math
from bluetooth import *
from time import sleep
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import freqz
from scipy.signal import butter, lfilter


def butter_bandpass(lowcut, highcut, fs, order=5):
    nyq = 0.5 * fs
    low = lowcut / nyq
    high = highcut / nyq
    b, a = butter(order, [low, high], btype='band')
    return b, a


def butter_bandpass_filter(data, lowcut, highcut, fs, order=5):
    b, a = butter_bandpass(lowcut, highcut, fs, order=order)
    y = lfilter(b, a, data)
    return y


#Conexao bluetooth
service_matches = find_service( name = "android_servidor", uuid = "5b553d50-4b4f-11e4-916c
    -0800200c9a66" )
if len(service_matches) == 0:
  print "couldnot find the service!"
```

```
  sys.exit(0)
first_match = service_matches[0]
port = first_match["port"]
name = first_match["name"]
host = first_match["host"]
print "connecting to ", host
sock=BluetoothSocket( RFCOMM )
sock.connect((host, port))
k=int(1)


#Estabelece a conexao serial
ser = serial.Serial('/dev/ttyUSB0',baudrate=57600)
print ser.name


i=int(0)
xx=[0]*128
y=[0]*128
fs=128
lowcut = 3
highcut = 20

#Recebe os dados da serial
while True:
  x=ser.read(34)
  data=x.encode('hex')
  #print "Data: ", data
  data_str=str(data)
  init_package= data_str.find("a55a02");
  counter=data_str[init_package+6]+data_str[init_package+7]
  data01p_h=data_str[init_package+8]+data_str[init_package+9]
  data01p_l=data_str[init_package+10]+data_str[init_package+11]
  data01n_h=data_str[init_package+12]+data_str[init_package+13]
  data01n_l=data_str[init_package+14]+data_str[init_package+15]
  data02p_h=data_str[init_package+16]+data_str[init_package+17]
  data02p_l=data_str[init_package+18]+data_str[init_package+19]
  data02n_h=data_str[init_package+20]+data_str[init_package+21]
  data02n_l=data_str[init_package+22]+data_str[init_package+23]


  #print "data lenght", len(data_str)

  ch1p = 255*int(data01p_h,16)+int(data01p_l,16)
  ch1n = 255*int(data01n_h,16)+int(data01n_l,16)
  ch2p = 255*int(data02p_h,16)+int(data02p_l,16)
  ch2n = 255*int(data02n_h,16)+int(data02n_l,16)

  #print ch1p
  #print ch1n
  #print ch2p
  #print ch2n

  ch1 = ch1p-ch1n
  ch2 = ch2p-ch2n

  #print ch1
  #print ch2
```

```
89    #print i
      xx[i]=int((ch1)/2)
91    #sock.send(str(ch1))
      #sleep(0.1)
93    # 128 pacotes por segundo
      #Envia dados bluetooth
95    if (i%4==0):
        if (y[i]>-256 and y[i]<256):
97        sock.send(str(int(y[i])))
          sleep(0.015625)
99    i=i+1
      if (i==128):
101     y = butter_bandpass_filter(xx, lowcut, highcut, fs, order=5)
        i=0
103 sock.close()
```

codigos/readusb_filter.py

Apêndice D

Códigos em *Java* do *android*

Main Activity

```
1  package pack1.tcc_v1;
2
3  import java.io.IOException;
4  import java.io.InputStream;
5  import java.io.OutputStream;
6  import java.nio.ByteBuffer;
7  import java.util.ArrayList;
8  import java.util.Arrays;
9  import java.util.Random;
10 import java.util.Set;
11 import java.util.UUID;
12 import android.app.Activity;
13 import android.bluetooth.BluetoothAdapter;
14 import android.bluetooth.BluetoothDevice;
15 import android.bluetooth.BluetoothServerSocket;
16 import android.bluetooth.BluetoothSocket;
17 import android.content.Intent;
18 import android.graphics.Color;
19 import android.os.Bundle;
20 import android.os.Handler;
21 import android.os.Looper;
22 import android.os.Message;
23 import android.support.v4.content.LocalBroadcastManager;
24 import android.view.Menu;
25 import android.view.View;
26 import android.widget.ArrayAdapter;
27 import android.widget.Button;
28 import android.widget.ListView;
29 import android.widget.TextView;
30 import android.widget.Toast;
31 import com.androidplot.xy.FillDirection;
32 import com.androidplot.xy.LineAndPointFormatter;
33 import com.androidplot.xy.SimpleXYSeries;
34 import com.androidplot.xy.XYPlot;
35 import com.androidplot.xy.XYSeries;
36
37 public class MainActivity extends Activity {
```

```
    private final static int REQUEST_ENABLE_BT = 1;
    private boolean cond;
    private TextView label1;
    private TextView lb_data;
    private TextView label3;
    private Button bt_conectar;
    private Button bt_receber;
    private Button bt_parar;
    private ListView lv;
    private ArrayAdapter<String> mArrayAdapter;
    private ArrayList<String> itemsList;
    private Message msg;
    private Thread mthread1;
    public BluetoothAdapter mBluetoothAdapter;
    private AcceptThread mAcceptThread;
    private receber_dados rd;

    private Handler mHandler = new Handler() {
        @Override
        public void handleMessage(Message msg) {
            super.handleMessage(msg);
            switch (msg.what) {
            case 1:
                byte[] readBuf = (byte[]) msg.obj;
                String string = new String(readBuf);
                lb_data.setText(string);
                 Intent intent = new Intent("custom-event-name");
                  // You can also include some extra data.
                 int transf = 90;
                 transf = Integer.valueOf(string);
                 intent.putExtra("data1", transf); //
                 LocalBroadcastManager.getInstance(getBaseContext()).sendBroadcast(intent);
                break;
            }
        }
    };

    private Handler handler_teste = new Handler() { //Looper.getMainLooper()
          @Override
          public void handleMessage(Message msg) {
              TextView myTextView = (TextView)findViewById(R.id.textView2);
              myTextView.setText((String)msg.obj);
             }
         };

    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        label1 = (TextView)findViewById(R.id.textView1);
        bt_conectar = (Button)findViewById(R.id.button1);
        bt_receber = (Button)findViewById(R.id.button2);
        bt_parar = (Button)findViewById(R.id.button3);
        lb_data = (TextView)findViewById(R.id.textView2);
        lv = (ListView)findViewById(R.id.listView1);
        label3 = (TextView)findViewById(R.id.textView3);
        mBluetoothAdapter = BluetoothAdapter.getDefaultAdapter();
```

```
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }

   public void conectar(View view)
    {
       boolean fluxo;

       if (habilitar_bluetooth())
       {
          label1.setText("Bluetooth habilitado");
          habilitar_descobrimento();
          aparelhos_pareados();
          conectar_servidor();
       }
       else
       {
           label1.setText("Bluetooth desabilitado");
       }
      }

   public void receber(View view)
   {

       Intent intent_plot = new Intent(this,Plotar_grafico_Activity.class);
       intent_plot.putExtra("data1", 25);  //String.valueOf(i1));
       startActivity(intent_plot);
   }


   public void parar(View view)
   {
       if (mAcceptThread!=null)
       {
          mAcceptThread.cancel();
       }

       if (mtesteThread!=null)
       {
          mtesteThread.cancel();
       }

       if (rd!=null)
       {
           rd.cancel();
       }
   }

   public boolean habilitar_bluetooth()
   {
    if (mBluetoothAdapter == null)
       {
       Toast.makeText(getBaseContext(),"Sistema nao suporta bluetooth",Toast.LENGTH_LONG).
```

```
            show();
            // Device does not support Bluetooth
155          return false;
            }
157         else
            {
159             if (!mBluetoothAdapter.isEnabled())
            {
161             Toast.makeText(getBaseContext(),"Bluetooth sera habilitado",Toast.LENGTH_LONG).
                    show();
            }
163             return true;
        }
165     }

167     public void habilitar_descobrimento()
        {
169         Intent discoverableIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_DISCOVERABLE);
            discoverableIntent.putExtra(BluetoothAdapter.EXTRA_DISCOVERABLE_DURATION, 300);
171         startActivity(discoverableIntent);
    }
173
        public void aparelhos_pareados()
175     {
            Set<BluetoothDevice> pairedDevices = mBluetoothAdapter.getBondedDevices();
177         itemsList = new ArrayList<String>();
            mArrayAdapter = new ArrayAdapter(this,android.R.layout.simple_list_item_1,itemsList);
179         if (pairedDevices.size() > 0) {
             // Loop through paired devices
181          for (BluetoothDevice device : pairedDevices) {
                // Add the name and address to an array adapter to show in a ListView
183             itemsList.add(device.getName() + "\n" + device.getAddress());
            }
185     }
        lv.setAdapter(mArrayAdapter);
187     }

189     public void conectar_servidor()
        {
191         mAcceptThread = new AcceptThread();
            mAcceptThread.start();
193     }

195 private class AcceptThread extends Thread {
        private final BluetoothServerSocket mmServerSocket;
197         public AcceptThread()
            {
199       //Comecar servidor
            BluetoothAdapter mBluetoothAdapter = BluetoothAdapter.getDefaultAdapter();
201         UUID meu_uuid = UUID.fromString("5b553d50-4b4f-11e4-916c-0800200c9a66");
            BluetoothServerSocket tmp = null;
203         try {
             // MY_UUID is the app's UUID string, also used by the client code
205          tmp = mBluetoothAdapter.listenUsingRfcommWithServiceRecord("android_servidor",
                    meu_uuid);
            } catch (IOException e) { }
207         mmServerSocket = tmp;
            }
```

```
209
        public void run()
211     {
            BluetoothSocket socket = null;
213         // Keep listening until exception occurs or a socket is returned
            while (true) {
215             lb_data.post(new Runnable() {
                    public void run() {
217                     lb_data.setText("Procurando conexao...");
                    }
219           });
              try {
221               socket = mmServerSocket.accept();
              } catch (IOException e) {
223               lb_data.post(new Runnable() {
                      public void run() {
225                     lb_data.setText("Erro na procura");
                      }
227             });
                  break;
229           }
              // If a connection was accepted
231           if (socket != null) {

233               lb_data.post(new Runnable() {
                      public void run() {
235                     lb_data.setText("Conexao aceita!");
                      }
237             });

239               label3.post(new Runnable() {
                      public void run() {
241                     label3.setText("Conexao aceita!!!");
                      }
243             });
                  // Do work to manage the connection (in a separate thread)
245               //manageConnectedSocket(socket);
                  rd = new receber_dados(socket);
247               rd.start();
              }
249         }
        }
251
        /* Will cancel the listening socket, and cause the thread to finish */
253     public void cancel() {
            try {
255             mmServerSocket.close();
            } catch (IOException e) { }
257     }
        }
259
private class receber_dados extends Thread
261   {
        private final BluetoothSocket mmSocket;
263     private final InputStream mmInStream;
        //private final OutputStream mmOutStream;
265
        public receber_dados(BluetoothSocket socket) {
```

```
267         mmSocket = socket;
            InputStream tmpIn = null;
269         try {
                tmpIn = socket.getInputStream();
271             //tmpOut = socket.getOutputStream();
            } catch (IOException e) { }
273         mmInStream = tmpIn;
            }
275
            public void run() {
277         int bytes_recebidos=0; // bytes returned from read()
            int availableBytes;
279         // Keep listening to the InputStream until an exception occurs
            while (true) {
281             try {

283             // Read from the InputStream
                availableBytes = mmInStream.available();
285             if (availableBytes>0)
                {
287
                byte[] buffer = new byte[availableBytes];  // buffer store for the stream
289             // Read from the InputStream
                bytes_recebidos = mmInStream.read(buffer);
291             mHandler.obtainMessage(1, bytes_recebidos, -1, buffer).sendToTarget();

293             }

295             }
                catch (IOException e) {
297             break;
                }
299         }
            }
301
            /* Call this from the main activity to shutdown the connection */
303         public void cancel()
            {
305         try {
                mmSocket.close();
307         } catch (IOException e) { }
            }
309   }
}
```

codigos/MainActivity.java

Activity Plotar Gráfico

```
    package pack1.tcc_v1;
2
    import java.text.DecimalFormat;
4   import java.util.Arrays;
    import java.util.Observable;
6   import java.util.Observer;
    import android.app.Activity;
8   import android.content.BroadcastReceiver;
    import android.content.Context;
```

```
10 import android.content.Intent;
   import android.content.IntentFilter;
12 import android.graphics.Color;
   import android.graphics.DashPathEffect;
14 import android.os.Bundle;
   import android.os.Handler;
16 import android.os.Message;
   import android.support.v4.content.LocalBroadcastManager;
18 import android.util.Log;
   import android.view.Menu;
20 import android.view.MenuItem;
   import android.widget.TabHost;
22 import android.widget.Toast;
   import android.widget.TabHost.TabSpec;
24 import com.androidplot.Plot;
   import com.androidplot.util.PixelUtils;
26 import com.androidplot.xy.BoundaryMode;
   import com.androidplot.xy.LineAndPointFormatter;
28 import com.androidplot.xy.SimpleXYSeries;
   import com.androidplot.xy.XYPlot;
30 import com.androidplot.xy.XYSeries;
   import com.androidplot.xy.XYStepMode;
32
   public class Plotar_grafico_Activity extends Activity {
34     private BroadcastReceiver mMessageReceiver = new BroadcastReceiver(){
       @Override
36     public void onReceive(Context context, Intent intent) {
         int dx = intent.getIntExtra("data1",0);
38       data.receber_os_dados(dx);
   }
40 };

42 // redraws a plot whenever an update is received:
   private class MyPlotUpdater implements Observer {
44     Plot plot;

46     public MyPlotUpdater(Plot plot) {
       this.plot = plot;
48     }

50     @Override
       public void update(Observable o, Object arg) {
52     plot.redraw();
       }
54 }

56 private XYPlot dynamicPlot;
   private MyPlotUpdater plotUpdater;
58 SampleDynamicXYDatasource data;
   private Thread myThread;
60 private Intent intent_dados;
   private XYPlot xyplot_c1;
62 private int dados_recebidos=0;
   private TabHost tabhost;
64
   @Override
66 protected void onCreate(Bundle savedInstanceState) {
     super.onCreate(savedInstanceState);
```

```
68    setContentView(R.layout.activity_plotar_grafico_);
      tabhost = (TabHost)findViewById(android.R.id.tabhost);
70    tabhost.setup();
      TabSpec spec1=tabhost.newTabSpec("Canal 1");
72    spec1.setContent(R.id.tab1);
      spec1.setIndicator("Canal 1");
74    TabSpec spec2=tabhost.newTabSpec("Canal 2");
      spec2.setContent(R.id.tab2);
76    spec2.setIndicator("Canal 2");
      tabhost.addTab(spec1);
78    tabhost.addTab(spec2);
      LocalBroadcastManager.getInstance(this).registerReceiver(mMessageReceiver,
80    new IntentFilter("custom-event-name"));
      // get handles to our View defined in layout.xml:
82    dynamicPlot = (XYPlot) findViewById(R.id.plot_ch1);
      plotUpdater = new MyPlotUpdater(dynamicPlot);
84    // only display whole numbers in domain labels
      dynamicPlot.getGraphWidget().setDomainValueFormat(new DecimalFormat("0"));
86    // getInstance and position datasets:
      data = new SampleDynamicXYDatasource();
88    SampleDynamicSeries sine1Series = new SampleDynamicSeries(data, 0, "Canal 1 EEG");
      LineAndPointFormatter formatter1 = new LineAndPointFormatter(
90                Color.rgb(0, 0, 0), null, null, null);
      //formatter1.getLinePaint().setStrokeJoin(Paint.Join.ROUND);
92    formatter1.getLinePaint().setStrokeWidth(2);
      dynamicPlot.addSeries(sine1Series,
94        formatter1);
      // hook up the plotUpdater to the data model:
96    data.addObserver(plotUpdater);
      // thin out domain tick labels so they dont overlap each other:
98    dynamicPlot.setDomainStepMode(XYStepMode.INCREMENT_BY_VAL);
      dynamicPlot.setDomainStepValue(128);
100   dynamicPlot.setRangeStepMode(XYStepMode.INCREMENT_BY_VAL);
      dynamicPlot.setRangeStepValue(25);
102   dynamicPlot.setRangeValueFormat(new DecimalFormat("###.#"));
      // uncomment this line to freeze the range boundaries:
104   dynamicPlot.setRangeBoundaries(-256, 256, BoundaryMode.FIXED);
      // create a dash effect for domain and range grid lines:
106   DashPathEffect dashFx = new DashPathEffect(
      new float[] {PixelUtils.dpToPix(3), PixelUtils.dpToPix(3)}, 0);
108   dynamicPlot.getGraphWidget().getDomainGridLinePaint().setPathEffect(dashFx);
      dynamicPlot.getGraphWidget().getRangeGridLinePaint().setPathEffect(dashFx);
110

}
112

@Override
114 public boolean onCreateOptionsMenu(Menu menu) {
      // Inflate the menu; this adds items to the action bar if it is present.
116   getMenuInflater().inflate(R.menu.plotar_grafico_, menu);
      return true;
118 }

120 @Override
    public boolean onOptionsItemSelected(MenuItem item) {
122   // Handle action bar item clicks here. The action bar will
      // automatically handle clicks on the Home/Up button, so long
124   // as you specify a parent activity in AndroidManifest.xml.
      int id = item.getItemId();
```

```
126     if (id == R.id.action_settings) {
            return true;
128     }
        return super.onOptionsItemSelected(item);
130 }

132 
    @Override
134 protected void onDestroy() {
        // Unregister since the activity is about to be closed.
136     LocalBroadcastManager.getInstance(this).unregisterReceiver(mMessageReceiver);
        super.onDestroy();
138 }

140 
    @Override
142 public void onResume() {
        LocalBroadcastManager.getInstance(this).registerReceiver(mMessageReceiver,
144     new IntentFilter("custom-event-name"));
        myThread = new Thread(data);
146     data.receber_os_dados(dados_recebidos);
        Toast.makeText(getBaseContext(),"broadcastreceiver"+String.valueOf(dados_recebidos),Toast.
            LENGTH_LONG).show();
148     myThread.start();
        super.onResume();
150 }

152 @Override
    public void onPause() {
154     data.stopThread();
        super.onPause();
156 }

158 }

160 class SampleDynamicXYDatasource implements Runnable {
        // encapsulates management of the observers watching this datasource for update events:
162 class MyObservable extends Observable {
        @Override
164     public void notifyObservers() {
        setChanged();
166     super.notifyObservers();
        }
168 }

170 private static final int SAMPLE_SIZE = 640;
    private MyObservable notifier;
172 private boolean keepRunning = false;
    private int[] dadosr = new int[SAMPLE_SIZE];
174 private int cont=0;

176 {
        notifier = new MyObservable();
178 }

180 public void stopThread() {
        keepRunning = false;
182 }
```

```
184 public void receber_os_dados(int dados){
    if (cont<SAMPLE_SIZE)
186 {
        dadosr[cont]=dados;
188     cont++;
    }
190 else
    {
192     cont=0;
    }
194 }

196 //@Override
public void run() {
198 try {
    keepRunning = true;
200 boolean isRising = true;

202 while (keepRunning) {
        Thread.sleep(10); // decrease or remove to speed up the refresh rate.
204     notifier.notifyObservers();
    }
206 } catch (InterruptedException e) {
    e.printStackTrace();
208 }
}
210
public int getItemCount(int series) {
212 return SAMPLE_SIZE;
}
214
public Number getX(int series, int index) {
216 if (index >= SAMPLE_SIZE) {
    throw new IllegalArgumentException();
218 }
    return index;
220 }

222 public Number getY(int series, int index) {
    if (index >= SAMPLE_SIZE) {
224 throw new IllegalArgumentException();
    }
226 double angle = (index + (phase))/FREQUENCY;
    double amp = sinAmp * Math.sin(angle);
228 double y = dadosr[index];


230
    switch (series) {
232 case SINE1:
        return y;
234 case SINE2:
        return 1;
236 default:
        throw new IllegalArgumentException();
238 }
}
240
```

```
    public void addObserver(Observer observer) {
242     notifier.addObserver(observer);
    }
244
    public void removeObserver(Observer observer) {
246     notifier.deleteObserver(observer);
    }
248
    }
250
    class SampleDynamicSeries implements XYSeries {
252 private SampleDynamicXYDatasource datasource;
    private int seriesIndex;
254 private String title;

256 public SampleDynamicSeries(SampleDynamicXYDatasource datasource, int seriesIndex, String
        title) {
        this.datasource = datasource;
258     this.seriesIndex = seriesIndex;
        this.title = title;
260 }

262 @Override
    public String getTitle() {
264     return title;
    }
266
    @Override
268 public int size() {
        return datasource.getItemCount(seriesIndex);
270 }

272 @Override
    public Number getX(int index) {
274     return datasource.getX(seriesIndex, index);
    }
276
    @Override
278 public Number getY(int index) {
        return datasource.getY(seriesIndex, index);
280 }
    }
```

codigos/Plotar_grafico_Activity.java