



PocketLab

Um Laboratório de Bolso para Processamento Digital de Sinais com
Android e Arduino

Dr. Ulisses Martins Dias
Dr. Cristhof Johann Roosen Runge

```
/*
 * Faculdade de Tecnologia - Unicamp
 * CET0500 - Programação para Dispositivos Móveis
 * Professor: Ulisses Martins Dias
 */

#include <Stepper.h>

int in1Pin = 1;
int in2Pin = 1;
int in3Pin = 1;
int in4Pin = 9;

Stepper motor(512, in1Pin, in2Pin, in3Pin, in4Pin);

void setup()
{
    pinMode(in1Pin, OUTPUT);
    pinMode(in2Pin, OUTPUT);
    pinMode(in3Pin, OUTPUT);
    pinMode(in4Pin, OUTPUT);
}

while (true)
{
    // Your code here
}
```

A diagram illustrating the PocketLab system. It shows an Arduino Uno connected to a smartphone via a USB cable. The smartphone screen displays two plots: one showing a square wave signal and another showing a noisy signal. A small icon of a smartphone with a robot head is also present.





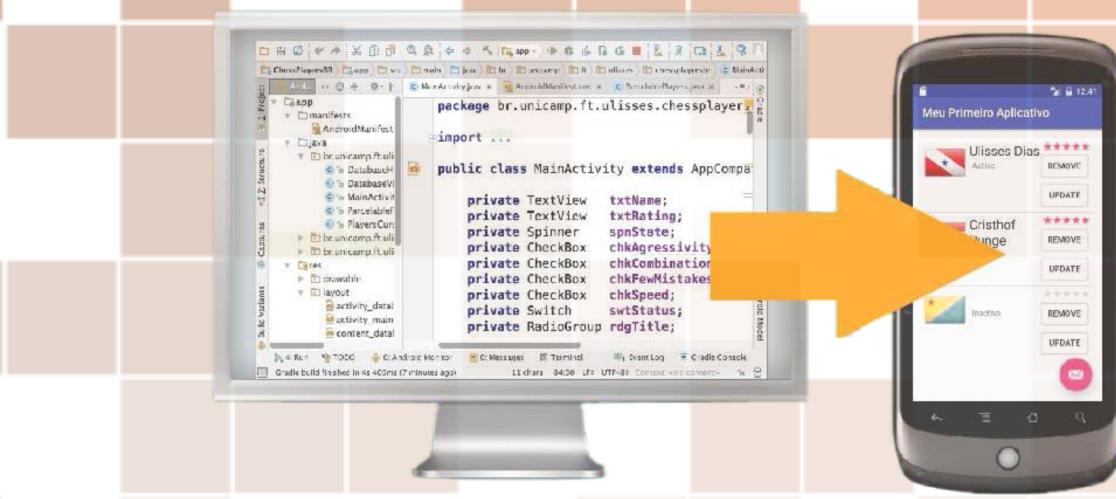
Apresentação

Esse material tem como finalidade apresentar os conceitos e procedimentos necessários para a elaboração do projeto “Pocket Lab”.

O “Pocket Lab” tem como principal objetivo desenvolver laboratórios a partir do desenvolvimento de um aplicativo de uso real que terá a função de um osciloscópio, da confecção de ponteiras de prova para realizar o monitoramento de sinais elétricos, e do uso de uma placa ARDUINO, que funcionará como gerador de sinais a serem observados através do aplicativo utilizando juntamente com a ponta de prova. A elaboração desse projeto visa mostrar uma aplicação prática dos conceitos de programação de dispositivos móveis , processamento de sinais, e de circuitos elétricos. É considerado que o aluno possui os conceitos básicos em programação de dispositivos móveis, processamento digital de sinais e em circuitos elétricos. Dessa forma, o material é focado na explicação da utilização desses conceitos dentro dos desenvolvimentos propostos, não correspondendo a um curso introdutório desses conceitos.

O material é dividido em quatro seções:

- Criação do aplicativo Osciloscópio para Android.
- Confeção da ponteira de prova para realização das medidas.
- Criação do programa para geração de estímulos no Arduino.
- Laboratórios de testes e medidas.



Conteúdo

I Parte 1: Construindo o PocketLab para Android

1	Android Studio	9
1.1	Configurando o Projeto	9
1.2	Configurando um Dispositivo	10
2	As Linguagens Java e XML	11
2.1	Java Convencional	11
2.2	Java para Dispositivos Móveis	13
2.3	XML para Dispositivos Móveis	15
2.4	O Método onCreate	16
2.5	Resumo do Capítulo	19
3	A Estrutura do Projeto	21
4	Construindo as Activities	23
4.1	MainActivity	23
4.1.1	O Elemento ListView	23
4.1.2	Ativação Explícita	25
4.2	ModuleActivity	26
5	Recebendo Dados do Microphone	27

6	Criando a Superfície de Desenho	31
6.1	MicrophoneView	31
6.2	MicrophoneCallback	33
7	Os Módulos do PocketLab	37
7.1	AbstractModule	37
7.2	Oscilloscope	40
7.3	Complex, FFT e SpectrumAnalyzer	42

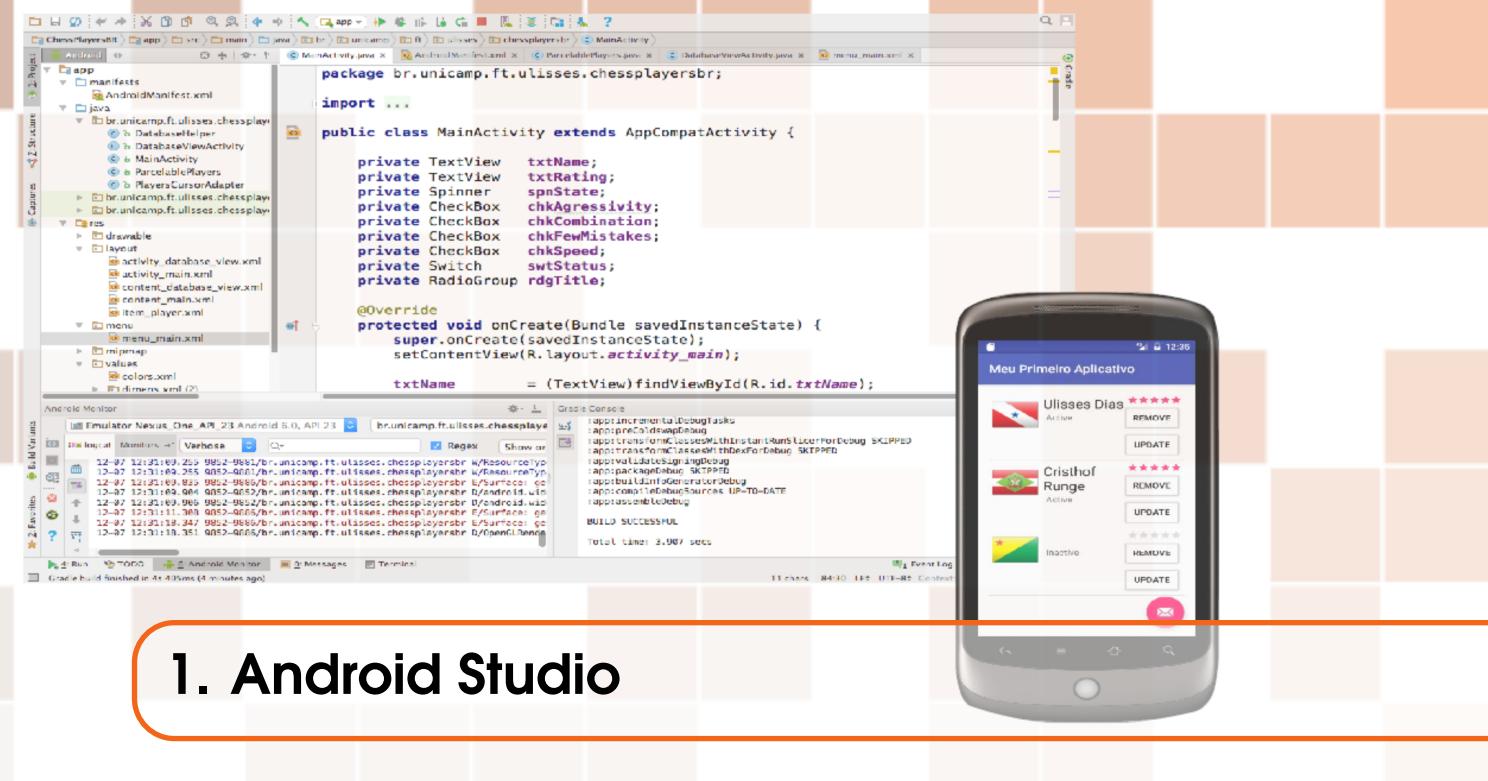
II

Confecção da Ponteira de Prova

8	Montando a ponta de prova	55
9	Utilização do Arduino	57
9.1	Considerações de Hardware	57
9.2	Considerações de Software	57
9.3	Programação	57
9.4	Geração de sinais para monitoração	58
9.5	Código para geração de ondas quadradas de diferentes frequências.	58
9.5.1	Programa	60
9.5.2	IDE Arduino	60
10	Laboratórios de testes e medidas	63
11	Referências	65

Parte 1: Construindo o PocketLab para Android

1	Android Studio	9
1.1	Configurando o Projeto	
1.2	Configurando um Dispositivo	
2	As Linguagens Java e XML	11
2.1	Java Convencional	
2.2	Java para Dispositivos Móveis	
2.3	XML para Dispositivos Móveis	
2.4	O Método onCreate	
2.5	Resumo do Capítulo	
3	A Estrutura do Projeto	21
4	Construindo as Activities	23
4.1	MainActivity	
4.2	ModuleActivity	
5	Recebendo Dados do Microphone ...	27
6	Criando a Superfície de Desenho	31
6.1	MicrophoneView	
6.2	MicrophoneCallback	
7	Os Módulos do PocketLab	37
7.1	AbstractModule	
7.2	Oscilloscope	
7.3	Complex, FFT e SpectrumAnalyzer	



1. Android Studio

Para criar os aplicativos nesta disciplina, usaremos o Android Studio, a IDE oficial do Android. Dentro os recursos principais oferecidos pelo Android Studio estão a edição de código, tanto Java como XML, depuração do código e ferramentas de desempenho.

Não daremos maiores informações sobre a instalação do Android Studio porque não difere muito da instalação de qualquer outro programa ou plataforma de desenvolvimento. Além disso, como novas versões do Android Studio são lançadas com bastante frequência, corre-se o risco de o que for apresentado aqui ficar desatualizado muito rapidamente.

Para instalar o android studio, acessem <http://developer.android.com> e procurem pelo link "Get Android Studio". Feito isso, a versão apropriada para o seu sistema operacional estará disponível para download.

As próximas seções explicam as configurações usadas pelos desenvolvedores deste material para executar os códigos que serão utilizados como exemplo.

1.1 Configurando o Projeto

Ao acessar o Android Studio pela primeira vez será exibida a tela de boas-vindas. Várias opções estão disponíveis nessa tela como, por exemplo, criar um novo projeto, abrir um projeto existente e importar um projeto do Eclipse/ADT. A opção sugerida é criar um novo projeto no Android Studio. Feito isso, as próximas telas que o Android Studio fornecerá serão de configuração do novo projeto, onde você deverá selecionar algumas das seguintes opções:

Application Name: nome do seu aplicativo que aparecerá no dispositivo do usuário.

Company Name: domínio da sua empresa. Caso o seu projeto seja pessoa, você pode usar as suas iniciais ou qualquer outro nome que achar conveniente.

Package Name: este campo não pode ser preenchido diretamente. Observe que ele é criado a partir do Application Name e do Company Name. Este campo serve de identificador único do seu aplicativo, ou seja, não pode haver dois aplicativos com o mesmo nome de pacote instalados em um determinado aparelho ao mesmo tempo.

Project Location: pasta onde o projeto será salvo.

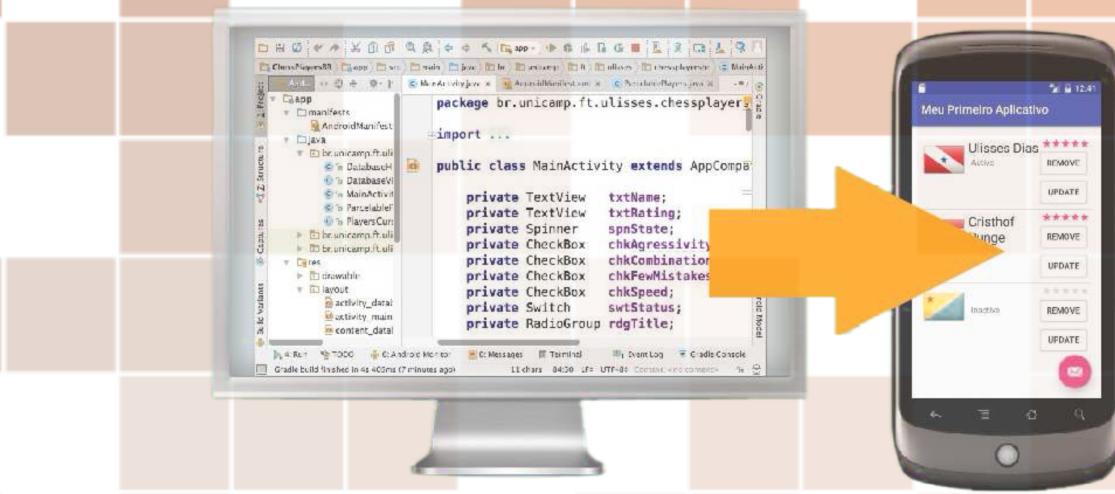
Minimum required SDK: versão mínima do Android que o aparelho deve possuir para executar o seu aplicativo. Quanto menor a versão, mais dispositivos podemos suportar, porém o conjunto de APIs disponíveis será menor. Os códigos deste material foram testados na API 23 da opção Phone and Tablet.

Activity Inicial: O Android Studio pode criar a primeira activity automaticamente para você. Dentre as opções disponíveis, temos Black Activity, Empty Activity e muitas outras. O código gerado neste material foi criado com a opção Empty Activity. Nos campos Activity Name e Layout Name foram mantidas as opções fornecidas por padrão pelo Android Studio.

1.2 Configurando um Dispositivo

Os códigos mostrados neste material foram executados em vários dispositivos com Sistema Operacional Android 6.0 (Marshmallow) ou superior. Para executar os códigos, você precisa inicialmente habilitar a opção de menu Opções do desenvolvedor nas configurações do aparelho. Para isso, selecione a opção Sobre o telefone e clique sete vezes seguidas sobre a opção Número da versão ou Build Number. Volte para tela anterior e a aba Opções do desenvolvedor estará disponível.

Acesse as Opções do desenvolvedor e habilite USB Debugging. Após isso, basta conectar seu aparelho ao computador por meio de um cabo USB e o dispositivo será reconhecido automaticamente pelo Android Studio nas plataformas Mac OS X e Linux. Caso você esteja usando o Windows e não puder usar nenhuma das plataformas previamente mencionadas, pode ser necessário instalar o driver do dispositivo no site do fabricante. É importante ressaltar que os códigos mostrados neste material funcionam apenas em dispositivos reais.



2. As Linguagens Java e XML

Java é uma linguagem de programação orientada a objetos bastante difundida atualmente. O osciloscópio apresentado neste material foi desenvolvido utilizando essa linguagem de programação, mas está fora do escopo do nosso escopo o ensino propriamente dito da linguagem. Para entender os conceitos apresentados, você precisa ter conhecimentos básicos tanto de programação quanto da linguagem.

Um outro ponto a ser levantado é que este material não foi desenvolvido para ser uma introdução ao desenvolvimento de aplicativos móveis. Dito isso, acreditamos que não seja didático que o osciloscópio seja o primeiro aplicativo para android que você irá programar na sua vida. É fortemente recomendável que você desenvolva alguns aplicativos básicos comuns em livros ou tutoriais na internet antes de se aventurar nas próximas páginas deste livro.

Nós nos comprometemos em sempre deixar claro os passos que precisam ser seguidos para acompanhar este material, mas em alguns casos você precisará de alguma habilidade com a linguagem Java, com o padrão XML e com a plataforma de desenvolvimento Android Studio para acompanhar esses passos.

2.1 Java Convencional

Iniciamos este material com um programa em java convencional para que o ponto de partida da programação para dispositivos móveis ocorra dentro de sua zona de conforto.

O código a seguir contém apenas uma classe que possui o nome de Operações. Duas variáveis do tipo float, chamadas de a e b, são atributos dessa classe e foram recebidas como parâmetro pelo método construtor. Como um único construtor foi definido, qualquer instância da classe "Operações" terá obrigatoriamente esses dois números instanciados.

```

1 package Operacoes;
2
3 public class Operacoes {
4
5     private float a;
```

```

6     private float b;
7
8     public Operacoes(float a, float b){
9         this.a = a;
10        this.b = b;
11    }
12
13    public float soma(){
14        return a + b;
15    }
16
17    public float subtracao(){
18        return a - b;
19    }
20
21    public float multiplicacao(){
22        return a * b;
23    }
24
25    public float divisao(){
26        return a / b;
27    }
28

```

A classe Operações possui métodos para aplicar as quatro operações matemáticas básicas nos atributos "a" e "b". O resultado dessas operações é devolvido ao final de cada método.

Em um programa java convencional, a compilação ocorreria invocando o comando javac na linha de comando ou deixando que a sua IDE de preferência invoque o comando para você de uma maneira mais abstrata. Vale ressaltar que a classe Operações não possui um método main e, nesse caso, não pode ser executada por conta própria. Assim, para executar o programa, precisamos de um ponto de partida, um método especial que será invocado e iniciará a execução.

Neste nosso código, o ponto de partida foi colocado em uma outra classe chamada PontoDePartida. Essa classe possui o método que iniciará toda a execução do código. Esse método é chamado de `main` e a sua assinatura é sempre semelhante ao método `main` da classe `PontoDePartida`, que possui os identificadores `public`, `static` e `void`. Se você não sabe o que significam esses identificadores, é de suma importância que comecem a estudar java antes de prosseguir com este material.

```

1 package Operacoes;
2
3 public class PontoDePartida {
4
5     public static void main(String[] args){
6
7         float number1 = 6;
8         float number2 = 4;
9
10        Operacoes operacoes = new Operacoes(number1, number2);
11        System.out.println(operacoes.soma());
12        System.out.println(operacoes.subtracao());

```

```

13     System.out.println(operacoes.multiplicacao());
14     System.out.println(operacoes.divisao());
15 }
16 }
```

Observem que o método main está declarando duas variáveis do tipo float e atribuindo os valores 6 e 4. Em seguida, essas duas variáveis são usadas para gerar uma instância de nome operacoes (iniciando com o minúsculo) da classe Operacoes (que foi declarada iniciando com o o maiúsculo). Neste momento, o método construtor da classe Operações será chamado. Vale ressaltar que declarar instâncias com nomes que iniciam com letra minúscula é uma convenção, assim como declarar as classes com nomes que começam com letras maiúsculas.

Feita a instanciação do novo objeto, os métodos que realizam operações básicas são invocados em sequência e os resultados são enviados para a saída padrão.

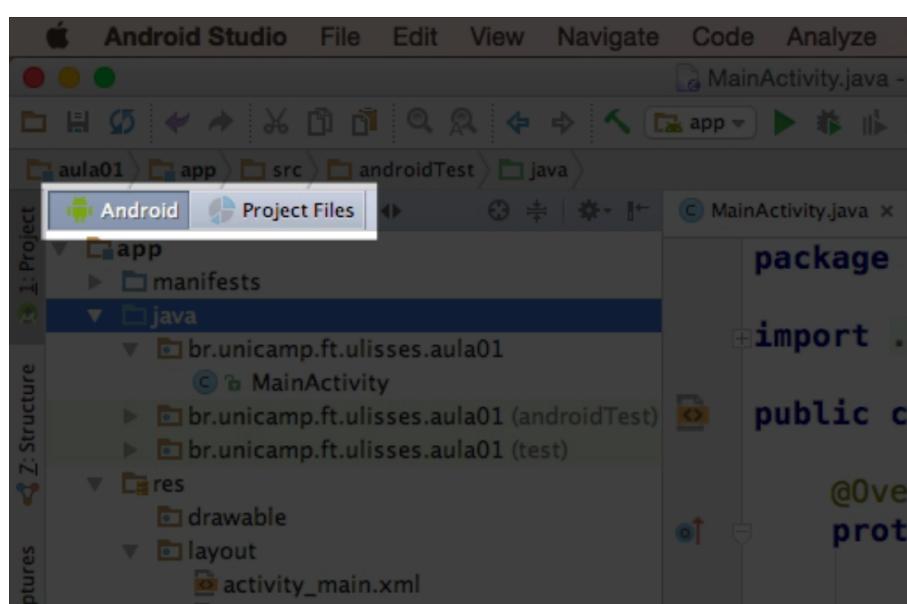
Para executar esse código em java convencional, vocês inicialmente compilariam usando java e posteriormente invocariam o comando java, ou usariam algum botão na IDE de sua preferência:

```
javac PontoDePartida.java
java PontoDePartida
```

O resultado apropriado aparece na saída padrão.

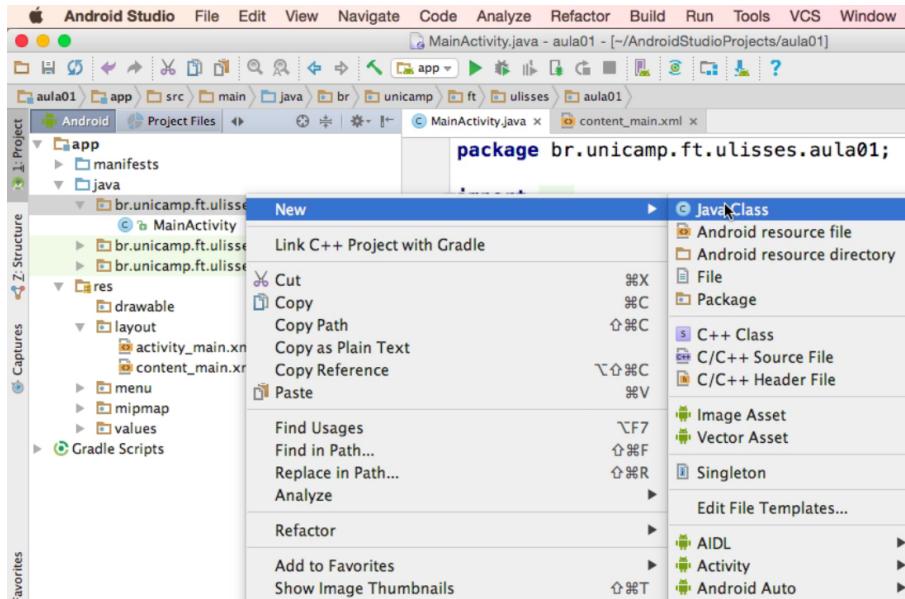
2.2 Java para Dispositivos Móveis

Vamos agora entender como a classe "Operações" seria executada em um dispositivo móvel. Do lado esquerdo da IDE, é possível visualizar uma lista dos arquivos que são criados automaticamente pelo Android Studio e várias formas de visualizar esses arquivos. É possível, por exemplo, visualizar esses arquivos da forma como as pastas são realmente organizadas no computador, bastando selecionar a opção Project Files, ou da forma resumida, bastando selecionar a opção Android.



Na forma de visualização Android, nós podemos obter acesso ao lugar onde os códigos em java devem ser adicionados. É possível, por exemplo, adicionar a classe Operações pressionando o botão direito do mouse nesse lugar e usando o menu suspenso para selecionar a opção desejada

conforme visto na figura. O próximo passo seria chamar essa classe de Operacoes e depois transportar o código para a classe recém-criada. Observem que nenhuma mudança é necessária no código da classe Operacoes. Não esqueça, no entanto, de manter o nome do pacote conforme criado pelo Android Studio automaticamente.



Analisando os códigos em java gerados automaticamente, percebemos a existência de uma classe chamada de `MainActivity`. Essa classe é uma `Activity` e, por isso, ela possui uma janela na qual é possível adicionar botões, caixas de textos, imagens e outros elementos. Observe que a nossa `MainActivity` possui um método chamado de `OnCreate`. Este método é o ponto de partida que usaremos neste nosso primeiro código. Aqui não temos um método `main` como no java convencional e método `onCreate` é onde a activity é inicializada.

Para que este aplicativo tenha um comportamento no Android parecido com o que tinha em java convencional. Podemos adicionar o trecho que estava em `PontoDePartida.java` no final do método `onCreate`, sem a necessidade de alterar nada do que foi gerado automaticamente pelo Android Studio. Feito isso, basta compilar essa classe e executar no dispositivo. Quando a activity for carregada, o método `onCreate` será chamado e o mesmo resultado que no código em java convencional era mandado para a saída padrão do sistema será enviado para o LogCat. Observe abaixo a `MainActivity` após as alterações.

```

1 package br.unicamp.ft.ulisses.aula01;
2
3 import android.app.Activity;
4
5 public class MainActivity extends Activity {
6
7     @Override
8     protected void onCreate(Bundle savedInstanceState) {
9         super.onCreate(savedInstanceState);
10        setContentView(R.layout.activity_main);
11
12        float number1 = 6;
13

```

```

14     float number2 = 4;
15
16     Operacoes operacoes = new Operacoes(number1, number2);
17     System.out.println(operacoes.soma());
18     System.out.println(operacoes.subtracao());
19     System.out.println(operacoes.multiplicacao());
20     System.out.println(operacoes.divisao());
21
22 }
23
24 }
```

2.3 XML para Dispositivos Móveis

Em um programa escrito em java convencional, a interface gráfica é gerada principalmente usando Swing ou AWT. Em um código para Android, é muito comum usarmos a linguagem XML para criar a Interface Gráfica. Esses códigos em XML serão inflados em classes java posteriormente.

XML significa *EXtensible Markup Language* e, caso você tenha uma ideia de HTML, então você não terá problemas em entender o que é XML. Primeiramente, vejamos onde podemos localizar o código em XML. Ele está acessível usando a estrutura de arquivos do lado esquerdo da IDE, dentro da pasta `res` e dentro da subpasta `layout`. O Android Studio cria automaticamente o código XML que gera uma tela com o texto Hello World.

De maneira geral, o XML organiza as informações usando tags, que são blocos delimitados por parênteses angulares. Você utiliza as tags para fazer marcações. Por exemplo, no código em XML criado automaticamente pelo Android Studio mostrado abaixo, a tag `<TextView ... />` é uma marcação que indica que uma caixa de texto será adicionada na interface gráfica. A tag inicia com o parêntese angular de abertura seguido do identificador `TextView` e termina com uma barra e um outro parêntese angular. O android verá essa tag `TextView` e entenderá que precisa adicionar uma caixa de texto seguindo as configurações detalhadas na própria marcação.

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
3   xmlns:app="http://schemas.android.com/apk/res-auto"
4   xmlns:tools="http://schemas.android.com/tools"
5   android:id="@+id/content_main"
6   android:layout_width="match_parent"
7   android:layout_height="match_parent"
8   android:paddingBottom="@dimen/activity_vertical_margin"
9   android:paddingLeft="@dimen/activity_horizontal_margin"
10  android:paddingRight="@dimen/activity_horizontal_margin"
11  android:paddingTop="@dimen/activity_vertical_margin"
12  app:layout_behavior="@string/appbar_scrolling_view_behavior"
13  tools:context="br.unicamp.ft.ulisses.myapplication.MainActivity"
14  tools:showIn="@layout/activity_main">
15
16  <TextView
17    android:layout_width="wrap_content"
18    android:layout_height="wrap_content"
19    android:text="Hello World!" />
```

20 </RelativeLayout>

Note que uma marcação pode possuir outras marcações encadeadas, como é o caso da marcação `RelativeLayout`, que possui a TAG `TextView` como uma marcação interna. Nesse caso, precisamos de uma outra maneira para delimitar o alcance dessa marcação. A marcação é iniciada com os parênteses angulares, sendo que não estamos usando aqui a barra para fechar o parêntese angular do modo como fizemos na `<TextView ... />`. A ausência dessa barra indica que a marcação ainda não terminou.

Nesse caso, para finalizar a marcação, precisaremos escrever novamente o identificador delimitado pelos parênteses angulares e a barra será posicionada na frente do identificador, resultando em `</RelativeLayout>`. Em resumo, temos a abertura de um parêntese, seguido da barra, seguido do nome do identificador e finalmente seguido de um fechamento de parêntese.

A última coisa que precisamos entender sobre XML é que as marcações podem possuir atributos. Lembrando sempre que o valor atribuído ao atributo deve sempre estar entre aspas. No caso específico do android, já existe um esquema de XML que fixa tanto os identificadores usados nas tags quanto nos atributos. Você não pode inventar nomes para as tags e nomes para os atributos, porque não será gerada um código em XML válido para o Android, você terá que memorizar as identificadores das tags e os atributos principais. A boa notícia é que esses atributos se repetem com uma certa frequência. Notem que os atributos `layout_width` e `layout_height` apareceram nas duas tags deste nosso exemplo.

Agora que vocês estão começando a programar para Android, é aceitável utilizar o Android Studio para gerar o código em XML para vocês. Você pode clicar na aba Design que está embaixo da janela e arrastar os componentes da paleta para a representação gráfica do dispositivo. Isso gerará o código XML correspondente. Assim, você poderá ver o código XML e aprender as principais construções antes de tentar escrever diretamente no código em XML.

2.4 O Método `onCreate`

Vamos dar prosseguimento ao nosso estudo tentando entender o código que foi gerado automaticamente pelo Android Studio no método `onCreate` e adicionando um pouco mais de responsabilidades a este método.

A primeira linha de código no corpo do método `onCreate` será sempre `super.onCreate()`. Isso avisa ao sistema operacional que, além do código que vamos inserir aqui, queremos que o código que está no método `onCreate` da classe pai seja também executado. Em particular, queremos que o método `onCreate` da classe pai seja executado antes de qualquer linha de código que vamos inserir. Não vamos entrar em detalhes sobre o que essa chamada está fazendo porque as classes do android são muito complexas. Entenda apenas que, caso você não insira essa chamada, a exceção `SuperNotCalledException` será disparada.

A segunda linha de código do método `onCreate` será normalmente uma chamada ao método `setContentView`. Essa chamada recebe como parâmetro a ID do código em XML que servirá de interface gráfica para esta activity. Neste momento, o código em XML será inflado, o que significa que ele será transformado em código Java. Objetos serão gerados a partir das marcações e dos atributos e, posteriormente, adicionados na hierarquia. Observe na chamada de `setContentView` que o parâmetro é o nome do arquivo XML precedido de `R.layout`.

Após a chamada a `setContentView`, podemos acessar os elementos que estão no arquivo XML usando o método `findViewById`. É uma prática comum usar o método `findViewById` para todos os elementos da interface gráfica que serão usados em algum lugar do código. Nesse caso, o método `findViewById` irá criar uma referência para o elemento da interface gráfica e nós adicionaremos essa referência como atributo da classe. Abaixo nós adicionamos um código em

XML com vários elementos da interface gráfica e um código em java que acessa esses elementos usando o método findViewById.

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3     xmlns:app="http://schemas.android.com/apk/res-auto"
4     xmlns:tools="http://schemas.android.com/tools"
5     android:orientation="vertical"
6     android:id="@+id/content_main"
7     android:layout_width="60dp"
8     android:layout_height="match_parent"
9     android:paddingBottom="@dimen/activity_vertical_margin"
10    android:paddingLeft="@dimen/activity_horizontal_margin"
11    android:paddingRight="@dimen/activity_horizontal_margin"
12    android:paddingTop="@dimen/activity_vertical_margin"
13    app:layout_behavior="@string/appbar_scrolling_view_behavior"
14    tools:context="br.unicamp.ft.ulisses.aula01.MainActivity"
15    tools:showIn="@layout/activity_main">
16
17    <EditText
18        android:layout_width="match_parent"
19        android:layout_height="wrap_content"
20        android:id="@+id/txtA"    />
21
22    <EditText
23        android:layout_width="match_parent"
24        android:layout_height="wrap_content"
25        android:id="@+id/txtB"    />
26
27    <Button
28        android:layout_width="match_parent"
29        android:layout_height="wrap_content"
30        android:onClick="onSoma"    />
31
32    <TextView
33        android:layout_width="wrap_content"
34        android:layout_height="wrap_content"
35        android:text="Hello World!"
36        android:id="@+id/textView"    />
37 </LinearLayout>
```

```
1 package br.unicamp.ft.ulisses.aula01;
2
3 import ... ;
4
5 public class MainActivity extends Activity {
6     TextView textView;
7     EditText txtA;
8     EditText txtB;
```

```

9
10    @Override
11    protected void onCreate(Bundle savedInstanceState) {
12        super.onCreate(savedInstanceState);
13        setContentView(R.layout.activity_main);
14
15        textView = (TextView)findViewById(R.id.textView);
16        System.out.println(textView.getText());
17
18        txtA = (EditText)findViewById(R.id.txtA);
19        txtB = (EditText)findViewById(R.id.txtB);
20
21        float number1 = 6;
22        float number2 = 4;
23
24        Operacoes operacoes = new Operacoes(number1, number2);
25        System.out.println(operacoes.soma());
26        System.out.println(operacoes.subtracao());
27        System.out.println(operacoes.multiplicacao());
28        System.out.println(operacoes.divisao());
29    }
30
31    public void onSoma(View view){
32        int a = Integer.parseInt(txtA.getText().toString());
33        int b = Integer.parseInt(txtB.getText().toString());
34        Operacoes operacoes = new Operacoes(a,b);
35        textView.setText(String.valueOf(operacoes.soma()));
36    }
37}

```

Para usar o método `findViewById`, temos que garantir que uma id foi configurada no código em XML na marcação da view. Por exemplo, na `TextView` da linha 36 do código de layout em XML temos a declaração de uma id chamada de `textView` usando o atributo `android:id` e na linha 15 do código em java temos uma chamada ao método `findViewById` que cria uma referência a essa `TextView` usando a id. Observe que a id criada em XML com o valor `@+id/textView` deve ser chamada no código em java com `R.id.textView`.

É interessante notar que o método `findViewById` retorna um objeto da classe `View` e precisamos sempre converter esse objeto para o tipo específico correspondente. Na linha 15 convertemos o objeto para a classe `TextView` antes de atribuir a referência ao atributo `textView`. Feito isso, a referência é um atributo e podemos obter o texto da `TextView` em qualquer lugar do código invocando o método `getText()` da instância. No código acima, na linha 16, obtivemos o texto e imprimimos no logcat.

Agora que você comprehende um pouco mais sobre java e XML, vamos entender o que o aplicativo android cujos códigos foram mostrados acima estão fazendo. O aplicativo possuirá uma interface gráfica contendo duas caixas de texto e um botão. As caixas de texto serão usadas para inserir valores numéricos e o botão será usado como gatilho para que uma soma seja efetuada.

Primeiramente, verifiquem que o código em XML possui uma marcação externa às outras marcações chamada de `LinearLayout`, que é um gerenciador de layout. Três gerenciadores de layout são muito comuns em programação para android:

RelativeLayout a principal particularidade desse gerenciador é que os elementos da interface gráfica são posicionados tendo os outros elementos como referência.

LinearLayout a principal particularidade desse gerenciador é que views são posicionadas na ordem em que aparecem no arquivo xml, sendo que essas views podem ser posicionadas horizontalmente ou verticalmente usando o atributo `android:orientation`, que é um atributo obrigatório quando estamos usando um LinearLayout.

GridLayout a principal particularidade desse gerenciador é que os elementos da interface gráfica são posicionados em uma tabela contendo linhas e colunas. Um atributo obrigatório dentro de um GridLayout é `android:columnCount`, que informa o número de colunas.

O gerenciador de layout LinearLayout exige o atributo `android:orientation` inserido na linha 5. Duas EditTexts foram inseridas na linhas 17 – 25 para entrada de dados. Em ambas, configuramos os atributos obrigatórios `layout_width` e `layout_height` como `match_parent` e `wrap_content`, respectivamente. Por fim, observem que as EditTexts possuem identificadores, o identificador `txtA` foi utilizado na primeira EditText (linha 20) e o identificador `txtB` foi utilizando na segunda (linha 25).

Logo abaixo das EditTexts foi adicionado um botão. Observem que, novamente, precisamos adicionar os atributos `layout_width` e `layout_height`. Um atributo que é específico para Views que herdam de botões é o `android:onClick` (linha 30). Esse atributo, permite especificar o método, dentro do código java da activity, que será disparado quando o botão for pressionado. Isso facilita bastante a programação. O método foi chamado de `onSoma` e foi criado nas linhas 31 – 36 do código java.

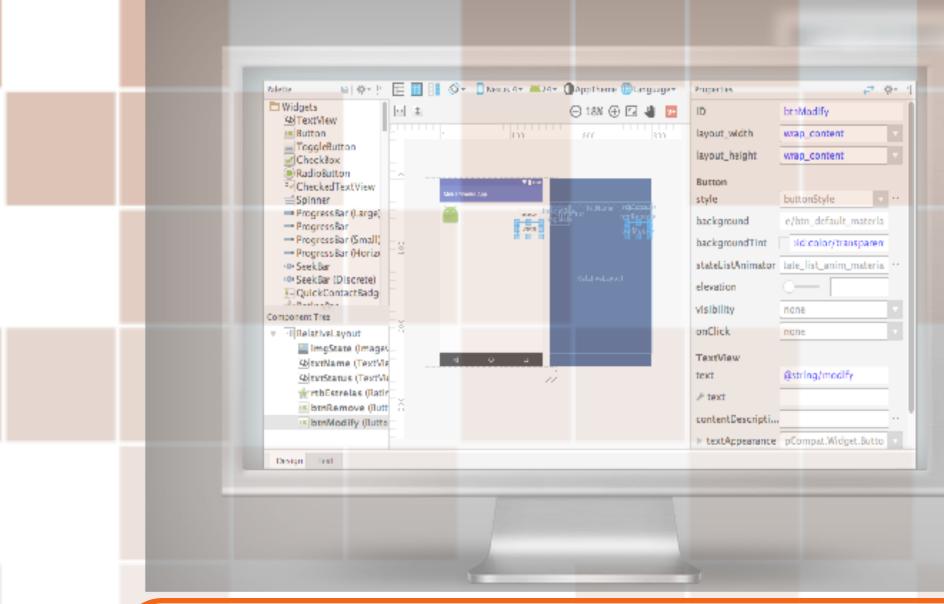
Dentro do código java, nas linhas 18 e 19, foi usado o método `findViewById` para criar referências para as caixas de entrada de texto (EditTexts). Ambas foram usadas no método `onSoma` para receber valores do usuário antes de criar o objeto `operacoes` da classe `Operacoes`. Essa instância é usada para realizar a operação de soma, que informada ao usuário na TextView que anteriormente continha a frase "Hello World".

A sua função agora é reproduzir esse código no seu computador antes de prosseguir para os próximos capítulos.

2.5 Resumo do Capítulo

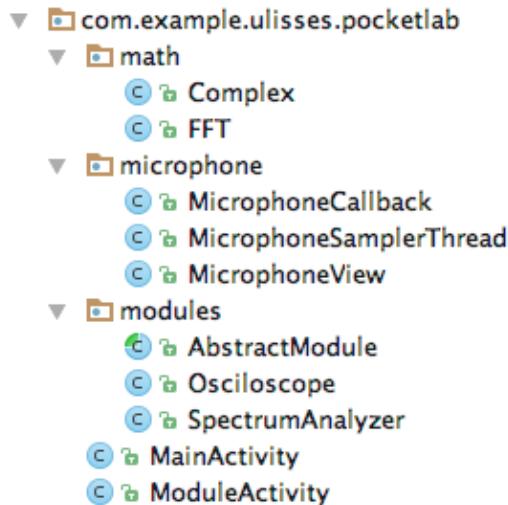
Nesta semana, nós aprendemos três informações importantes da programação para dispositivos móveis.

- A primeira se refere a como adicionar código em java no Android Studio. Nós inserimos no Android Studio uma nova classe que é responsável pelas quatro operações matemáticas básicas. Além disso, nós criamos a nossa primeira activity, que é um dos *building blocks* para a criação de programas para dispositivos móveis.
- O segundo aspecto de programação para dispositivos móveis que aprendemos esta semana foi que a interface gráfica pode ser criada usando código em XML, sendo essa a forma mais usada. Aprendemos também a obter os dados que estão na Interface Gráfica em XML dentro do nosso código Java.
- A terceira informação que aprendemos é que, na programação para dispositivos móveis, o método `onCreate` da activity que está em execução é o primeiro método a ser chamado. Assim, em dispositivos móveis, nós não usamos a função `main` que era comum na programação em java convencional.



3. A Estrutura do Projeto

Começaremos agora a construção do aplicativo **PocketLab**. O aplicativo não é tão simples quanto os códigos mostrados nas seções anteriores deste material, dado que foram necessárias 10 classes para fazê-lo funcionar. As classes foram colocadas no pacote **com.example.ulisses.pocketlab**. Dentro desse pacote, algumas subdivisões foram necessárias conforme mostrado na figura abaixo.



Na pasta **math**, colocamos as classes **Complex** e **FFT**. A primeira é uma representação do conjunto dos números complexos e a segunda implementa a transformada rápida de fourier, um algoritmo muito importante para a geração do módulo de análise de espectro do PocketLab.

Na pasta **microphone**, colocamos as seguintes classes:

MicrophoneView Essa classe é um componente que pode ser adicionado na interface gráfica diretamente nos arquivos XML de layout. Essa classe herda de **SurfaceView**, o que proporciona uma superfície de desenho utilizada pelos módulos do PocketLab.

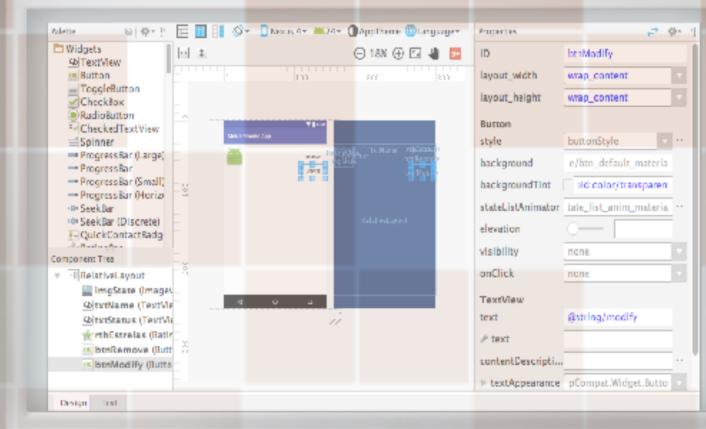
MicrophoneCallback Essa classe recebe informações sobre mudanças que ocorrem nas **SurfaceViews** as quais estão associadas. No nosso caso, a **MicrophoneCallback** está associada a **Micropho-**

neView e age nos momentos em que a superfície de desenho é criada ou destruído, alocando ou desalocando recursos.

MicrophoneSamplerThread Essa classe age diretamente sobre o microfone do dispositivo e obtém os dados que são lidos. Após ler os dados, a classe armazena em um buffer que fica à disposição dos módulos do PocketLab.

Na pasta `modules`, implementamos os dois módulos do nosso sistema: `Oscilloscope` e `SpectrumAnalyzer`, ambos os módulos herdam de `AbstractModule`. As classes que herdam de `AbstractModule` são threads que continuamente desenham na superfície criada por `MicrophoneView`. Nesse caso, a única diferença entre um módulo e outro são o modo como eles desenham os dados recebidos do microfone.

As duas classes que não estão em nenhuma subpasta são a `MainActivity` e a `ModuleActivity`. Você deve ter notado pelo sufixo *Activity* que essas classes são as telas de interação com o usuário.



4. Construindo as Activities

4.1 MainActivity

O nosso projeto terá uma classe `MainActivity` que será o ponto de partida do nosso aplicativo. A tela inicial mostrada ao usuário conterá uma `ListView` com as opções do `PocketLab`. Vamos inicialmente detalhar como essa tela inicial foi construída. Na Seção 4.1.1 daremos detalhes da implementação de nossa `ListView`.

4.1.1 O Elemento ListView

`ListViews` são parte integrante da interface gráfica de diversos aplicativos que você já deve ter visto na PlayStore. Uma `ListView` pode ser entendida como um menu onde o usuário pode escolher uma dentre várias opções. Como ela lida com interações diretas do usuário, será preciso adicionar um `listener` para tratar a opção selecionada pelo usuário.

Uma `ListView` é adicionada no código XML usando a tag `ListView`. Observem abaixo o código do nosso arquivo `content_main.xml`, que é a interface gráfica da nossa `MainActivity`. Na tag `<ListView>` nós adicionamos alguns atributos obrigatórios como `layout_height` e o `layout_width` e, além deles, colocamos o atributo `entries` para informar onde a `ListView` deverá buscar a lista de elementos para colocar no menu.

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <LinearLayout
3     xmlns:android="http://schemas.android.com/apk/res/android"
4     xmlns:tools="http://schemas.android.com/tools"
5     android:layout_width="match_parent"
6     android:layout_height="match_parent"
7     android:orientation="vertical"
8     tools:context="com.example.ulisses.pocketlab.MainActivity">
9
10    <ListView
11        android:id="@+id/listView"

```

```
12         android:layout_width="match_parent"
13         android:layout_height="match_parent"
14         android:entries="@array/modulos"
15     />
16
17 </LinearLayout>
```

A lista de elementos foi adicionada como um array de nome `modulos` no arquivo `strings.xml` que está dentro do diretório `layout/res`. Nós adicionamos itens a esse array da forma mostrada abaixo. Esses itens serão as opções do nosso menu.

```
1 <resources>
2     <string name="app_name">PocketLab</string>
3
4     <string-array name="modulos">
5         <item>Osciloscópio</item>
6         <item>Analizador de Espectro</item>
7     </string-array>
8 </resources>
```

O próximo passo será adicionar um `listener` para o evento de selecionar um item da lista. Esse `listener` foi adicionado no método `onCreate` da nossa classe `MainActivity` e o tipo de `listener` que estamos interessados deve implementar a interface java `onItemClickListener`. Esta interface nos pede para implementar o método `onItemClick` como mostrado no código abaixo.

```
1 package com.example.ulisses.pocketlab;
2
3 import android.content.Intent;
4 import android.support.v7.app.AppCompatActivity;
5 import android.os.Bundle;
6 import android.view.View;
7 import android.widget.AdapterView;
8 import android.widget.ListView;
9
10 public class MainActivity extends AppCompatActivity {
11
12     ListView listView;
13
14     @Override
15     protected void onCreate(Bundle savedInstanceState) {
16         super.onCreate(savedInstanceState);
17         setContentView(R.layout.activity_main);
18
19         listView = (ListView) findViewById(R.id.listView);
20
21         AdapterView.OnItemClickListener itemClickListener = new
22             AdapterView.OnItemClickListener() {
23                 public void onItemClick(AdapterView<?> listview,
24                                     View view,
```

```
24             int position,
25             long id){
26     Intent intent = new Intent(MainActivity.this,
27         ↪ ModuleActivity.class);
28     intent.putExtra("MODULE", position);
29     startActivity(intent);
30   }
31   listView.setOnItemClickListener(itemClickListener);
32 }
33 }
```

O método `onItemClick` possui o seguinte comportamento. A posição no menu do item clicado pelo usuário é recebida como parâmetro e usada na chamada do método `putExtra` de um objeto da classe `Intent`. Na seção 4.1.2, detalharemos esse objeto e o uso que fazemos dele para ativação da nova activity quando o usuário seleciona um item do menu.

4.1.2 Ativação Explícita

Nosso aplicativo precisa mostrar mais de uma tela ao usuário, então precisaremos da habilidade de invocar uma nova activity e de enviar dados a essa nova activity. Primeiramente, crie no seu projeto uma segunda activity pressionando o botão direito do mouse na pasta onde estão localizados os códigos java e depois selecionando a opção new e depois activity. Nesse momento, abrirá uma tela de configurações e você deverá selecionará a opção Empty Activity. A tela seguinte apresenta uma série de configurações onde podemos dar detalhes de como queremos que a activity seja criada. No nosso caso, mudaremos o nome da activity para `ModuleActivity` e manteremos as outras opções.

Feito isso, aparecerá uma nova activity na pasta java, novos arquivos XML na pasta de layout e uma nova seção no arquivo `AndroidManifest.xml`, que é obrigatório em todos os aplicativos android e contém informações essenciais como quais activities ele possui, quais são as bibliotecas requeridas e outras declarações. O Android Studio cria o `manifest` automaticamente quando você inicia um projeto e atualiza esse arquivo sempre que necessário.

Para ativar essa segunda activity, primeiro criamos um objeto da classe `Intent`. Esse objeto representa uma ação que o usuário deseja realizar. No trecho mostrado abaixo usamos apenas três linhas de código para ativar uma nova activity e repassar uma parâmetro. A primeira linha é a instanciação do intent que nesse caso usa um construtor com dois parâmetros.

```
1 Intent intent = new Intent(MainActivity.this, ModuleActivity.class);
2 intent.putExtra("MODULE", position);
3 startActivity(intent);
```

O primeiro parâmetro avisa ao android qual o contexto de onde o intent é enviado, podemos passar a referência da activity atual. O segundo parâmetro é a classe que será ativada e receberá o intent.

O intent pode ser visto como uma mensagem de uma activity para outra e um conteúdo pode ser acoplado a essa mensagem usando o método `putExtra`. O método `putExtra` possui dois parâmetros: o primeiro parâmetro do tipo `String` é uma chave para acessar o segundo parâmetro que é o conteúdo. O método `putExtra` utiliza sobrecarga para que você tenha várias opções no segundo parâmetro, podendo ser um tipo primitivo como `boolean` ou `int`; um array de primitivos ou uma `string`. Você pode chamar o método `putExtra` repetidamente para adicionar valores, garantindo

é claro que você está dando chaves diferentes para esses valores. A segunda linha do trecho de código que mostramos acima faz uso do método `putExtra` para enviar para a `ModuleActivity` qual o item do menu que foi selecionado.

A terceira linha de código poderia ser tanto uma chamada para `startActivity` quanto para `startActivityForResult`. No nosso caso, usamos `startActivity`. Isso avisa ao Sistema Operacional para iniciar a activity especificada pelo intent. Uma vez que o Sistema recebe o intent, ele verifica se não há nenhum problema e permite que a nova activity inicie.

O ativação feita dessa forma é dita explícita, porque informamos no próprio intent qual activity precisa ser disparada. Está fora do escopo deste material as ativações implícitas, que usamos quando não sabemos qual a activity que será carregada e deixamos essa decisão para o sistema operacional.

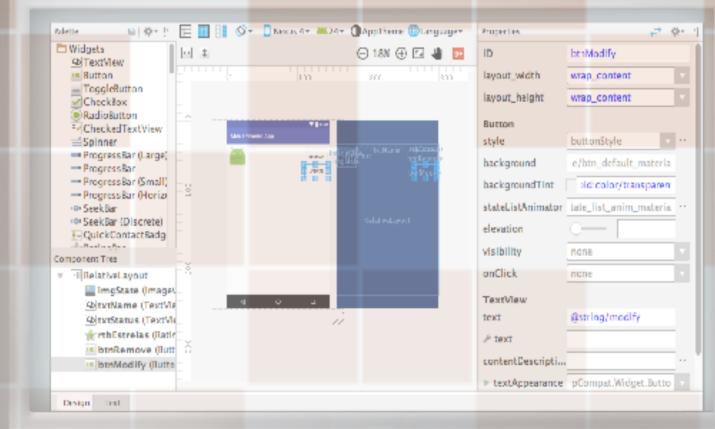
4.2 ModuleActivity

Para receber o valor que foi passado, a `ModuleActivity` precisa primeiramente obter o intent que foi usado para ativá-la. Isso é feito com a ajuda do método `getIntent()` invocado no `onCreate()` conforme mostrado na linha 19 do trecho de código abaixo.

Obtida essa instância, é possível invocar o método `getIntExtra` (linha 22) e passar a chave como parâmetro. Vale ressaltar que eu poderia receber não apenas inteiros, mas objetos de outros tipos por meio de métodos como `getStringExtra`, `getFloatExtra` e muitos outros.

```

1 package com.example.ulisses.pocketlab;
2
3 import android.content.Intent;
4 import android.support.v7.app.AppCompatActivity;
5 import android.os.Bundle;
6
7 import com.example.ulisses.pocketlab.microphone.MicrophoneCallback;
8 import com.example.ulisses.pocketlab.microphone.MicrophoneView;
9 import com.example.ulisses.pocketlab.modules.AbstractModule;
10
11 public class ModuleActivity extends AppCompatActivity {
12
13     @Override
14     protected void onCreate(Bundle savedInstanceState) {
15         super.onCreate(savedInstanceState);
16         setContentView(R.layout.activity_module);
17
18         MicrophoneView view =
19             (MicrophoneView) findViewById(R.id.microphoneView);
20         Intent intent = getIntent();
21         MicrophoneCallback microphoneCallback;
22         if (intent != null) {
23             microphoneCallback = new MicrophoneCallback(this,
24                 intent.getIntExtra("MODULE", 0));
25         } else {
26             microphoneCallback = new MicrophoneCallback(this);
27         }
28         view.addCallback(microphoneCallback);
29     }
30 }
```



5. Recebendo Dados do Microphone

Para capturarmos o áudio do microfone do aparelho, a primeira coisa que devemos fazer é adicionar a permissão RECORD_AUDIO no `AndroidManifest.xml`. Outras edições menores também foram feitas nesse arquivo e você pode comparar com o manifest criado automaticamente pelo Android Studio para o seu projeto.

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3     package="com.example.ulisses.pocketlab">
4     <application
5         android:allowBackup="true"
6         android:icon="@mipmap/ic_launcher"
7         android:label="@string/app_name"
8         android:roundIcon="@mipmap/ic_launcher_round"
9         android:supportsRtl="true"
10        android:theme="@style/AppTheme">
11            <activity android:name=".MainActivity">
12                <intent-filter>
13                    <action android:name="android.intent.action.MAIN" />
14                    <category android:name="android.intent.category.LAUNCHER" />
15                </intent-filter>
16            </activity>
17            <activity android:name=".ModuleActivity"></activity>
18        </application>
19        <uses-permission android:name="android.permission.RECORD_AUDIO"/>
20        <uses-permission
21             android:name="android.permission.MODIFY_AUDIO_SETTINGS"/>
</manifest>
```

Veremos como exploramos os recursos de áudio do android. Duas APIs são frequentemente

utilizadas quando queremos fazer algum tipo de processamento de áudio em dispositivos android: `AudioRecorder` e `MediaRecorder`.

`AudioRecorder` gerencia os recursos de áudio do dispositivo. A leitura dos dados é feita usando o método `read`. Esses dados serão lidos em blocos e nenhum pré-processamento é realizado. `MediaRecorder` normalmente utilizada quando queremos copiar os dados para um arquivo. Essa classe pode ser melhor entendida como uma caixa-preta que fornece dados na saída com algum tipo de compressão.

Como estamos interessados em processar diretamente os dados fornecidos pelo hardware de leitura, usamos a classe `AudioRecorder` e salvamos blocos de leitura em um buffer temporário. A classe `MicrophoneSamplerThread` mostrada abaixo é responsável por essa tarefa:

```
1 package com.example.ulisses.pocketlab.microphone;
2
3 import android.media.AudioFormat;
4 import android.media.AudioRecord;
5 import android.util.Log;
6
7
8 public class MicrophoneSamplerThread extends Thread {
9
10    // Classe para obtenção dos dados de áudio
11    private AudioRecord ar;
12    // Buffer onde os dados de áudio são armazenados.
13    private short[] buffer;
14    // Configuração para gravação de áudio no formato mono.
15    private int channelConfiguration = AudioFormat.CHANNEL_IN_MONO;
16    // Configuração para gravação de áudio em 16bit
17    private int audioEncoding = AudioFormat.ENCODING_PCM_16BIT;
18    // Tamanho do buffer em bytes
19    private int bufferSizeBytes;
20    // Variável que define se a thread está em execução.
21    private boolean threadIsRunning = true;
22
23    @Override
24    public void run() {
25        try {
26            if (threadIsRunning) {
27                /*
28                 * Configurando a quantidade de bytes que será lida
29                 * a cada momento
30                 */
31                bufferSizeBytes = AudioRecord.getMinBufferSize(44100,
32                    channelConfiguration, audioEncoding);
33                bufferSizeBytes = (int) Math.pow(2, (int)
34                    Math.ceil(Math.log(bufferSizeBytes) / Math.log(2)));
35                buffer = new short[bufferSizeBytes];
36
37                /*
38                 * Inicializando a interface de áudio
39                 */
```

```

38         ar = new AudioRecord(1, 44100, channelConfiguration,
39             ↵ audioEncoding,
40             buffersizebytes);
41
42     if (ar.getState() != AudioRecord.STATE_INITIALIZED) {
43         Log.e("MicrophoneSampler", "Audio Record can't
44             ↵ initialize!");
45         return;
46     }
47     ar.startRecording();
48
49     /*
50      Obtendo os dados do microfone e salvando no
51      atributo "buffer"
52      */
53     while (threadIsRunning) {
54         ar.read(buffer, 0, buffersizebytes);
55     }
56
57     /*
58      A amostragem terminou, liberando os recursos de áudio.
59      */
60     ar.stop();
61     ar.release();
62 }
63
64 }
65
66 public short[] getBuffer() {
67     return buffer;
68 }
69
70 public boolean isThreadRunning() {
71     return threadIsRunning;
72 }
73
74 public void setRunning(boolean threadIsRunning) {
75     this.threadIsRunning = threadIsRunning;
76 }
77
78 }
79

```

Começaremos a análise da classe `MicrophoneSamplerThread` pelos atributos declarados nas linhas 10–21. O atributo `ar` da classe `AudioRecord` será usado para obtenção dos dados de áudio. Os dados serão amostrados e salvos no atributo `buffer`. O atributo `channelConfiguration` define a configuração do canal de audio que, no nosso caso, foi configurado para mono para garantirmos que

o PocketLab irá funcionar em uma gama maior de dispositivos. O atributo `audioEncoding` define qual método utilizaremos para representar digitalmente uma sinal analógico. Nós vamos utilizar o método mais comum que é o Pulse-Code Modulation (PCM) com uma profundidade de 16 bits. Por fim, definimos uma variável booleana chamada de `threadIsRunning` para determinar se a thread está ou não em execução.

Condensamos todo o processo de inicialização do hardware de áudio, captura das amostras e desalocação dos recursos no método `run` da nossa `thread` para facilitar a apresentação dos conceitos. Entretanto, você pode achar interessante modularizar esse método em várias rotinas.

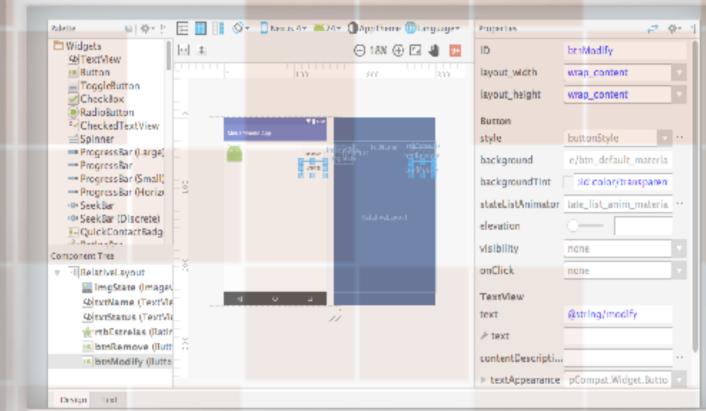
O método `run` inicia com uma verificação padrão de `threadIsRunning` antes da alocação de recursos realizada nas linhas 31–33. Um método muito útil chamado de `getMinBufferSize()` calcula para nós qual o tamanho mínimo que o nosso buffer deve ter dependendo do dispositivo onde o código está sendo utilizado e das configurações fornecidas. Obtido esse tamanho mínimo, nós computamos qual a menor potência de dois maior do que ele na linha 31 e inicializamos o buffer com esse valor na linha 32.

Após isso, inicializamos a interface de áudio na linha 37. Observe que configuramos uma taxa de amostragem de 44100Hz , o que significa que 44100 serão obtidas por segundo. Caso a inicialização da interface de áudio tenha êxito, passamos a capturar dados com a chamada ao método `startRecording` na linha 44.

O próximo passo é ler os dados que estão sendo gravados pelo `AudioRecorder`, o que é feito enquanto a flag `threadIsRunning` for verdadeira pelo método `read`. Os dados são lidos e imediatamente gravados no atributo `buffer` que está sendo passado como parâmetro para o método `read`. Observe que esta classe não toma nenhuma atitude além de salvar os dados na variável `buffer`. Na verdade, esses dados serão utilizados por outras classes que terão acesso à `buffer` por chamadas ao método `getBuffer` implementado nas linhas 64–66.

No momento em que `threadIsRunning` se tornar falso por uma eventual chamada ao método `setRunning` implementado nas linhas 72–74, o laço de repetição das linhas 49–51 será finalizado e os recursos de audio que alocamos serão liberados nas linhas 56 e 57.

No nosso aplicativo, nós optamos por implementar duas threads, uma para gerenciar os dados do microfone e outra para desenhar na interface gráfica. Entretanto, você poderia optar por implementar apenas uma Thread que fizesse ambas as tarefas. Nesse caso, você deveria acionar na linha 51, após a chamada para o método `read`, algum gatilho que pré-processasse os dados e resultasse no desenho desses dados na interface gráfica de forma apropriada.



6. Criando a Superfície de Desenho

Você pode criar uma visualização personalizada estendendo a classe `View` diretamente ou uma de suas subclasses, como fazemos com a classe `MicrophoneView`, que estende `SurfaceView`. Para adicionar um componente personalizado ao arquivo XML de um layout, você precisa informar o seu nome completo (nome do pacote + nome da classe). Na Seção 6.1 explicamos alguns detalhes adicionais.

Os elementos de uma `SurfaceView` são manipulados por meio de um objeto da classe `SurfaceHolder`, que fornece um objeto `Canvas` no qual elementos gráficos podem ser desenhados. O acesso ao `Canvas` é exclusivo, ou seja, duas classes não podem acessá-lo simultaneamente. Esse acesso exclusivo será usado pelos módulos que implementaremos no Capítulo 7.

Cada subclasse de `SurfaceView` deve implementar a interface `SurfaceHolder.Callback`, que nos obriga a criar alguns métodos que serão chamados em eventos do ciclo de vida de `SurfaceView`, como criação, modificação e destruição do `SurfaceView`. A Seção 6.2 descreve a nossa implementação da interface `SurfaceView.Callback`.

Vamos agora desenvolver as classes que farão o desenho na tela. Optamos por escrever as classes de modo a facilitar a adição de novos módulos no futuro. Iniciaremos com a definição do componente que será adicionado na interface gráfica, que é a classe chamada de `MicrophoneViewer` e depois passaremos para as classes auxiliares.

6.1 MicrophoneView

O `PocketLab` desenha os gráficos manualmente na tela ao atualizar os elementos dos módulos em uma thread de execução separada. Para isso, todos os módulos são subclasses de `AbstractModule` que por sua vez é uma subclasse de `Thread`. De um modo geral, todas as atualizações na interface gráfica devem ser feitas no que chamamos de `Main Thread`, que é a thread principal de execução. É importante minimizar o volume de trabalho feito na thread principal para garantir que o usuário não receba caixas de diálogo com a mensagem *Application Not Responding*.

Entretanto, os módulos do `PocketLab` podem possuir lógica complexa e o desenho na tela ocorre após o resultado dessa lógica ser obtido. Por exemplo, o módulo `SpectrumAnalyzer` precisa computar a transformada rápida de Fourier antes de desenhar na tela. Para esses casos, o

Android fornece a classe `SurfaceView`, que é uma subclasse de `View` na qual uma thread pode desenhar em um objeto `Canvas` e depois indicar que o desenho deve ser exibido na thread principal.

Observe que a classe `MicrophoneView` foi colocada dentro da pasta `microphone`. Nesse caso, o pacote dessa nova classe será `com.example.ulisses.pocketlab.microphone`. A classe `MicrophoneViewer` foi declarada herdando de `SurfaceView`. De modo geral, uma `SurfaceView` é associada a uma thread que indica como os resultados devem ser mostrados ao usuário. No `PocketLab`, todos os módulos usarão um mesmo objeto de `MicrophoneView`. Nesta seção, falaremos sobre esta `SurfaceView` que é compartilhada com todos os módulos. O código dessa classe é mostrado abaixo.

```

1 package com.example.ulisses.pocketlab.microphone;
2
3 import android.content.Context;
4 import android.util.AttributeSet;
5 import android.view.SurfaceView;
6
7 public class MicrophoneView extends SurfaceView {
8
9     /*
10      Recebe chamadas de métodos que indicam quando o objeto
11      SurvaceView é criado, atualizado ou destruído.
12      */
13     private MicrophoneCallback callBack;
14
15     public MicrophoneView(Context context, AttributeSet attrs){
16         super(context, attrs);
17     }
18
19     public void addCallback(MicrophoneCallback callBack){
20         this.callBack = callBack;
21         getHolder().addCallback(callBack);
22     }
23
24     public void stopDrawer(){
25         if (callBack.getDrawerThread() != null){
26             callBack.getDrawerThread().setRunning(false);
27         }
28     }
29     public void stopSampler(){
30         if (callBack.getSamplerThread() != null){
31             callBack.getSamplerThread().setRunning(false);
32         }
33     }
34 }
```

Começaremos a detalhar essa classe a partir do atributo `MicrophoneCallback` declarado na linha 13, que serve para implementar o comportamento do aplicativo quando ocorrem mudanças na `SurfaceView`. Por exemplo, é de suma importância que esse callback garanta que só usaremos a `SurfaceView` quando esta estiver visível ao usuário.

O construtor da classe `MicrophoneView` recebe um parâmetro do tipo `Context` que nos passa informações de onde esta `SurfaceView` foi adicionada e um parâmetro do tipo `AttributeSet` que nos diz quais atributos foram configurados na tag XML que adicionou esta View no layout. Esta View foi adicionada no código de layout da classe `ModuleActivity` que é mostrado abaixo. Observe que não adicionamos nada além dos atributos obrigatórios.

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <LinearLayout
3     xmlns:android="http://schemas.android.com/apk/res/android"
4     xmlns:tools="http://schemas.android.com/tools"
5     android:layout_width="match_parent"
6     android:layout_height="match_parent"
7     tools:context="com.example.ulisses.pocketlab.ModuleActivity">
8
9     <com.example.ulisses.pocketlab.microphone.MicrophoneView
10        android:id="@+id/microphoneView"
11        android:layout_width="match_parent"
12        android:layout_height="match_parent"/>
13
14
15 </LinearLayout>
```

Na classe `MicrophoneView` criamos um método `addCallback` (linhas 19–22) para registrar um objeto que implementa a interface `SurfaceHolder.Callback`. Observe a chamada ao método herdado `getHolder` de `SurfaceView`, que retorna o objeto `SurfaceHolder` correspondente para gerenciar o `SurfaceView`. O método `addCallback` de `SurfaceHolder` armazena o objeto que implementa a interface `SurfaceHolder.Callback`.

No nosso projeto, definimos duas threads (além da Main Thread), uma para a captura do som e outra para desenhar os elementos gráficos na tela. Criamos os métodos `stopSampler` e `stopDrawer` para permitir que essas threads sejam finalizadas (linhas 24–34).

6.2 MicrophoneCallback

Nesta seção, apresentamos o código da classe `MicrophoneCallback` que implementa a interface `SurfaceHolder.Callback`. Esta classe é responsável por acionar a thread de captura de dados de áudio e a thread do módulo do PocketLab para desenho na tela, sendo o primeiro representado pelo atributo `samplerThread` declarado na linha 14 e o segundo representado pelo atributo `abstractModule` declarado na linha 16 do código abaixo.

```
1 package com.example.ulisses.pocketlab.microphone;
2
3 import android.content.Context;
4 import android.graphics.Bitmap;
5 import android.graphics.Bitmap.Config;
6 import android.view.SurfaceHolder;
7
8 import com.example.ulisses.pocketlab.modules.AbstractModule;
9 import com.example.ulisses.pocketlab.modules.Oscilloscope;
10 import com.example.ulisses.pocketlab.modules.SpectrumAnalyzer;
```

```
11
12 public class MicrophoneCallback implements SurfaceHolder.Callback {
13     // Thread de captura de áudio
14     private MicrophoneSamplerThread samplerThread;
15     // Módulo do PocketLab
16     private AbstractModule abstractModule;
17     // Referência à ModuleActivity
18     private Context context;
19     // Parâmetro que identifica qual módulo deverá ser usado
20     private int moduleType = 0;
21
22     /*
23         Construtor padrão. Usando esse construtor o módulo
24         Osciloscópio do PocketLab será utilizado.
25     */
26     public MicrophoneCallback (Context context){
27         this.context = context;
28     }
29
30     /*
31         Construtor invocado quando queremos que o módulo utilizado
32         seja diferente do padrão (Osciloscópio).
33     */
34     public MicrophoneCallback (Context context, int moduleType){
35         this.context      = context;
36         this.moduleType   = moduleType;
37     }
38
39     /*
40         Permitimos que alguém obtenha a thread do módulo do PocketLab
41     */
42     public AbstractModule getDrawerThread()
43     {
44         return abstractModule;
45     }
46
47     /*
48         Permitimos que alguém obtenha a thread de captura de áudio
49     */
50     public MicrophoneSamplerThread getSamplerThread()
51     {
52         return samplerThread;
53     }
54
55     @Override
56     public void surfaceDestroyed(SurfaceHolder holder) {
57         /*
58             Finalizamos as duas threads que estão em execução
59             para que nenhum acesso ao SurfaceView seja feito
60         */
61     }
62 }
```

```
60         após o surfaceDestroyed ter sido chamado.
61     */
62     abstractModule.setRunning(false);
63     samplerThread.setRunning(false);
64     try {
65         /*
66         Esperamos ambas as threads terminarem.
67         */
68         abstractModule.join();
69         samplerThread.join();
70     } catch (InterruptedException e){
71
72     }
73 }
74
75 /**
76 Método chamado quando o tamanho da superfície muda.
77 A implementação desse método é obrigatória. Este método tem
78 um corpo vazio, pois sempre exibimos a orientação retrato
79 conforme configuramos no manifest.
80 */
81 @Override
82 public void surfaceChanged(SurfaceHolder holder, int format, int
83     ↪ width, int height) {
84
85
86 @Override
87 public void surfaceCreated(SurfaceHolder holder) {
88     /*
89         Iniciando a thread que captura dados do microfone.
90         */
91     samplerThread = new MicrophoneSamplerThread();
92     samplerThread.start();
93
94     /*
95         Iniciando a thread que irá desenhar na tela.
96
97         Temos uma thread por módulo. Neste momento, temos
98         dois módulos implementados: o osciloscópio e o analisador
99         de espectro. Iniciaremos o módulo correto de acordo com o
100        atributo moduleType recebido no construtor.
101    */
102    if (moduleType == 0) {
103        abstractModule = new Oscilloscope(holder, samplerThread,
104            ↪ context);
105    } else {
106        abstractModule = new SpectrumAnalyzer(holder, samplerThread,
107            ↪ context);
```

```
106     }
107     abstractModule.start();
108 }
109 }
```

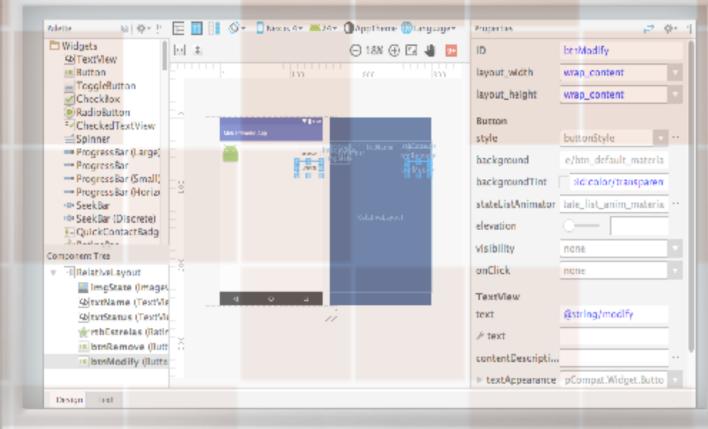
A classe `MicrophoneCallback` precisa conhecer qual o módulo do PocketLab que será responsável por desenhar na tela do usuário e o atributo `moduleType` armazena essa informação. É importante recapitular de onde obtemos essa informação.

1. Você deve lembrar que a `MainActivity` possuia uma `ListView` contendo dois itens (Osciloscópio e Analisador de Espectro).
2. Quando o usuário clicava em algum desses itens, a posição do elemento clicado era adicionada no objeto da classe `Intent` usando o método `putExtra`. Este objeto ativava a `ModuleActivity`.
3. A `ModuleActivity` recebia essa posição e inicializava um objeto da classe `MicrophoneCallback` usando o construtor que agora está sendo mostrado nas linhas 34–37 do código acima.
4. Caso a `ModuleActivity` não recebesse nenhuma informação sobre a posição clicada pelo usuário, o construtor das linhas 26–28 seria invocado e o atributo `moduleType` não seria alterado, mantendo o padrão 0.

A interface `SurfaceHolder.Callback` nos pede para implementar três métodos: `surfaceChanged`, `surfaceCreated` e `surfaceDestroyed`. O método `surfaceChanged` (linhas 79–81) tem corpo vazio, pois sempre exibimos a orientação retrato conforme configuramos no manifest.

O método `surfaceCreated` é chamado quando o objeto `SurfaceView` é criado, por exemplo, quando `ModuleActivity` é chamada pela primeira vez ou quando é retomada do segundo plano. Usamos `surfaceCreated` para iniciar as threads a fim de dar início ao módulo que foi selecionado pelo usuário (linhas 85–105).

O método `surfaceDestroyed` é chamado quando o objeto `SurfaceView` é destruído. Usamos esse método para garantir que os recursos que alocamos sejam finalizados corretamente. Primeiramente, as linhas 61 e 62 invocam `setRunning(false)` nas threads que estão em execução para indicar que elas devem parar. Após isso, as linhas 63–71 esperam que as threads terminem. Isso garante que não seja feita qualquer tentativa de desenhar na `SurfaceView` uma vez que `surfaceDestroyed` termine de ser executado.



7. Os Módulos do PocketLab

Neste capítulo, apresentaremos os módulos que já estão implementados no PocketLab e o que o usuário deverá fazer caso necessite adicionar novos módulos. A estrutura de módulos do PocketLab é bastante simples. Temos uma classe `AbstractModule` que implementa as funcionalidades comuns a todos os módulos, mas que não realiza desenho nenhum na tela. Dessa forma, a classe `AbstractModule` não pode ser instanciada. Apresentaremos essa classe na Seção 7.1.

A classe `Oscilloscope` apresenta o primeiro módulo implementado no PocketLab. Nesse módulo, recebemos os dados do microfone e colocamos na tela. Nenhum processamento é feito além da transformação necessária para fazer com que os dados caibam na tela do dispositivo. Esse módulo é apresentado na Seção 7.2.

A classe `SpectrumAnalyzer` implementa um módulo um pouco mais complicado que o osciloscópio. Nesse módulo, é preciso processar uma série de Fourier com os dados de entrada para posteriormente exibir os dados na tela. Para o processamento da série de Fourier, utilizamos as classes `FFT` e `Complex`, sendo que a primeira é uma implementação do algoritmo propriamente dito e a segunda é uma representação dos números complexos. Apresentaremos todas essas classes na Seção 7.3.

7.1 AbstractModule

Começaremos esta seção ressaltando duas informações importantes sobre a classe `AbstractModule`.

1. A classe é abstrata e por isso não pode ser instanciada. Se ela não pode ser instanciada, para que serve? Serve para herança dos métodos e dos atributos definidos. O código que inserimos na classe `AbstractModule` só poderá ser executado quando um de seus filhos for instanciado.
2. A classe `AbstractModule` herda da classe `Thread`. Nesse caso, os módulos do PocketLab que herdam de `AbstractModule` também são threads. Isso significa que os módulos executarão em paralelo e eventualmente desenharão no objeto canvas.

O código da classe `AbstractModule` é mostrado abaixo. Para entender o funcionamento dessa classe, precisamos compreender que todo o código pode ser subdividido em três partes.

Na primeira parte temos o construtor e os atributos relacionados a `SurfaceView`, como aqueles das classes `SurfaceHolder` e `Paint`. Os atributos da classe `Paint` podem ser entendidos como folhas de estilo para que todos os módulos tenham a mesma aparência. Recomenda-se fortemente que as classes filhas de `AbstractModule` utilizem esses estilos para que o usuário não fique com a sensação de que mudou de aplicativo sem ter percebido. Por exemplo, o atributo `linePaint` define a cor das retas que serão impressas na tela e o atributo `backPaint` define a cor de fundo. Todos esses atributos são inicializados no construtor.

Na segunda parte do código temos o método `run`, que é o método que ficará continuamente executando em `background` até que a variável `threadIsRunning` receba o valor `false`. Primeiramente, obtemos o objeto `Canvas` para desenhar no elemento `SurfaceView` fazendo uma chamada a `lockCanvas`. Somente uma thread pode desenhar em um elemento `SurfaceView` e por isso usamos o bloco `synchronized`. Dentro do bloco, verificamos se tanto a `canvas` para desenho quanto o `buffer` com dados do microfone são válidas para finalmente chamar o método `doDraw`.

A terceira parte do código corresponde ao método `doDraw` que aqui foi declarado como `abstract`. Isso quer dizer ao compilador java que não iremos implementar esse método, mas queremos que as classes filhas (que não são abstratas) o implementem. O método `doDraw` receberá como parâmetro o objeto da classe `Canvas` que preparamos no método `run`.

```
1 package com.example.ulisses.pocketlab.modules;
2
3 import android.content.Context;
4 import android.graphics.Bitmap;
5 import android.graphics.Canvas;
6 import android.graphics.Paint;
7 import android.util.Log;
8 import android.view.SurfaceHolder;
9 import android.graphics.Typeface;
10
11 import com.example.ulisses.pocketlab.microphone.MicrophoneSamplerThread;
12
13
14 public abstract class AbstractModule extends Thread {
15     private SurfaceHolder surfaceHolder;
16     private Context context;
17
18     protected Paint linePaint;
19     protected Paint backPaint;
20     protected Paint stringPaint;
21
22     private boolean threadIsRunning;
23
24     protected short[] buffer;
25     private MicrophoneSamplerThread samplerThread;
26
27
28     public AbstractModule(SurfaceHolder holder, MicrophoneSamplerThread
29         → samplerThread, Context context){
30         this.threadIsRunning = true;
31         this.surfaceHolder = holder;
32         this.samplerThread = samplerThread;
33     }
34 }
```

```
32     this.context          = context;
33
34     this.linePaint         = new Paint();
35     this.linePaint.setAntiAlias(true);
36     this.linePaint.setARGB(255, 0, 0, 0);
37
38     this.backPaint = new Paint();
39     this.backPaint.setAntiAlias(true);
40     this.backPaint.setARGB(255, 255, 255, 255);
41
42     stringPaint = new Paint();
43     stringPaint.setAntiAlias(true);
44     stringPaint.setARGB(255, 255, 0, 0);
45     stringPaint.setStrokeWidth(5);
46     stringPaint.setTypeface(Typeface.create("Arial", Typeface.BOLD));
47     stringPaint.setTextSize(40);
48
49
50     this.buffer = new short[2048];
51 }
52
53 public void setRunning(boolean running){
54     threadIsRunning = running;
55 }
56
57 @Override
58 public void run() { // loop em background
59     Canvas canvas = null;
60
61     while(threadIsRunning){
62         try {
63             canvas = surfaceHolder.lockCanvas(null);
64             buffer = samplerThread.getBuffer();
65
66             synchronized (surfaceHolder){
67                 if (canvas != null && buffer != null) {
68                     doDraw(canvas);
69                 }
70             }
71             Thread.sleep(500);
72         }catch (InterruptedException e){
73
74         }finally {
75             /*
76              Liberamos a canvas no finally por que queremos
77              evitar um deadlock
78             */
79             if (canvas != null){
80                 surfaceHolder.unlockCanvasAndPost(canvas);
```

```

81         }
82     }
83 }
84
85
86 public abstract void doDraw(Canvas canvas);
87
88
89 }
```

7.2 Osciloscope

Na classe `Osciloscope` mostramos os dados obtidos do microfone diretamente na tela. Esta classe possui apenas o construtor e o método obrigatório `doDraw`. O construtor não realiza nenhum código extra além da chamada ao construtor da classe mãe (`AbstractModule`).

O método `doDraw` recebe um objeto do tipo `Canvas` por parâmetro e inicializa esse objeto com a cor de fundo que está em `backCanvas`. Isso tem o efeito de apagar tudo o que existia na `Canvas` antes dessa chamada.

A seguir, o método configura alguns valores que auxiliarão no processo de desenhar na tela. Lembrando que em uma `Canvas` o ponto $(0,0)$ está no canto superior esquerdo e os valores dos eixos X e Y crescem conforme nos movemos para a direita e para baixo, respectivamente. O canto inferior esquerdo é o ponto $(width, height)$, onde $width$ é a largura da tela do aparelho (assumindo que a canvas ocupa toda a tela) e $height$ é a altura. Nesse caso, queremos que a linha central do osciloscópio fique na altura $height/2$.

Por questões de desempenho, não desenhamos todos os dados do microphone na tela do dispositivo, mas apenas um conjunto de dados suficientes para ocupar toda a largura da tela. Cada informação do buffer será usada para posicionar um pixel na tela e desenharemos uma linha entre um pixel e outro. A variável `mBufferShift` guarda o índice do primeiro ponto que desenharemos na tela. Devemos escolher `mBufferShift` de maneira que cada pixel da tela possa ser associado a um ponto do microfone.

A segunda parte do método `doDraw` executa o algoritmo propriamente dito. Nós sempre dividimos o valor que está no buffer por `height` para que a função que descreve o osciloscópio tenha uma escala que caiba dentro da tela. Essa escala é depois ajustada pelo atributo `m_iScaler`. Dentro do laço `while`, os valores de X e Y do ponto inicial e do ponto final da reta são calculados. Vale ressaltar que, após mudar a escala dos pontos, nós fazemos uma translação somando com `midHeight` para que a linha central do Osciloscópio fique no centro da tela. Ao final de todo o processo, uma chamada ao método `drawLine` é realizada. Abaixo mostramos o código da classe `Osciloscope`.

```

1 package com.example.ulisses.pocketlab.modules;
2
3 import android.content.Context;
4 import android.graphics.Canvas;
5 import android.view.SurfaceHolder;
6
7 import com.example.ulisses.pocketlab.microphone.MicrophoneSamplerThread;
8
9 public class Osciloscope extends AbstractModule {
```

```
11     private float m_iScaler = 2f;
12
13     public Osciloscope(SurfaceHolder holder, MicrophoneSamplerThread
14         ↪ samplerThread, Context context) {
15         super(holder, samplerThread, context);
16     }
17
18     public void doDraw(Canvas paramCanvas) {
19
20         /*
21             Aqui apagamos tudo o que estava na canvas.
22             */
23         paramCanvas.drawPaint(backPaint);
24
25         /**
26             Aqui configuramos alguns valores que auxiliarão no processo
27             de desenhar na tela. Lembrando que em uma Canvas o ponto
28             (0,0) está no canto superior esquerdo e o canto inferior
29             esquerdo é o ponto (width, height). Nesse caso, queremos que
30             a linha central do osciloscópio fique na altura height/2.
31
32             Por questões de desempenho, não desenhamos todos os dados do
33             microphone na tela do dispositivo, mas apenas um conjunto de
34             dados suficiente para ocupar toda a largura. Cada informação
35             do buffer será usada para posicionar um pixel na tela e
36             desenharemos uma linha entre um pixel e outro. A variável
37             mBufferShift guarda o índice do primeiro ponto que
38             desenharemos na tela. Devemos escolher mBufferShift de
39             maneira que cada pixel da tela possa ser associado a um ponto
40             do microfone.
41             */
42
43     int height = paramCanvas.getHeight();
44     int midHeight = height / 2;
45     int width = paramCanvas.getWidth();
46     int mBufferShift = 0;
47     if (width < buffer.length) {
48         mBufferShift = (buffer.length - width) / 4;
49     }
50
51
52     /**
53         Aqui o cálculo é efetivamente realizado. Nós sempre dividimos
54         o valor que está no buffer por height para que a função do
55         osciloscópio tenha uma escala que caiba dentro da tela. Essa
56         escala é depois ajustada pelo atributo m_iScaler. Dentro do
57         loop while, os valores de X e Y do ponto inicial e do ponto
```

```

59     final da reta são calculados. Vale ressaltar que, após mudar
60     a escala dos pontos, nós fazemos uma translação somando com
61     midHeight para que a linha central do Osciloscópio fique no
62     centro da tela. Ao final de todo o processo, uma chamada ao
63     método drawLine é realizada.
64 */
65
66     int mBuffIndex = 1;
67     float StartBaseY = m_iScaler * buffer[(mBuffIndex + mBufferShift -
68         ↵ 1)] / height;
69     while (mBuffIndex < width) {
70         float StopBaseY = m_iScaler * buffer[mBuffIndex +
71             ↵ mBufferShift] / height;
72
73         float StartY = StartBaseY + midHeight;
74         float StopY = StopBaseY + midHeight;
75         paramCanvas.drawLine(mBuffIndex, StartY, mBuffIndex + 1,
76             ↵ StopY, linePaint);
77         mBuffIndex++;
78         StartBaseY = StopBaseY;
79     }
80 }
```

Sugerimos que você passe um tempo lendo a classe `Oscilloscope` e tente entender todas as linhas de código. Para criar novos módulos do `PocketLab`, você deverá usar essa classe como referência, dado que ela é bastante resumida.

7.3 Complex, FFT e SpectrumAnalyzer

Nesta seção, apresentamos as três classes necessárias para desenvolver o `SpectrumAnalyzer`: `Complex`, `FFT` e `SpectrumAnalyzer`. Diferente do que fizemos nas seções anteriores, aqui não explicaremos o código em detalhes, dado que isso foge ao escopo deste material. Entretanto, você deve ser capaz de compreender a importância de cada uma dessas classes no contexto geral do `SpectrumAnalyzer` e entender o propósito de cada método e trecho de código com os comentários nos próprios códigos.

A primeira classe que apresentaremos é a `Complex`, que representa o conjunto dos números complexos. Essa classe possui apenas dois parâmetros `im` e `re` que representam as partes real e imaginária de um dado número complexo, respectivamente. A classe implementa operações básicas como soma, subtração e multiplicação; além de outras operações quer serão necessárias para a classe `FFT` e para a classe `SpectrumAnalyzer`, como `abs` e `conjugate`. Abaixo mostramos o código da classe `Complex`.

```

1 package com.example.ulisses.pocketlab.math;
2
3 public class Complex {
4     private final double re;    // Parte real
```

```
5     private final double im;    // Parte Imaginária
6
7     /*
8      Cria um novo objeto com as partes imaginária e real passadas
9      como parâmetro.
10     */
11    public Complex(double real, double imag) {
12        re = real;
13        im = imag;
14    }
15
16    /*
17     * Retorna o valor absoluto do número complexo.
18     */
19    public double abs() {
20        return Math.hypot(re, im);
21    }
22
23    /*
24     * Retorna um novo número complexo cujo valor é a soma deste
25     * número complexo com aquele passado como parâmetro.
26     */
27    public Complex plus(Complex b) {
28        double real = this.re + b.re;
29        double imag = this.im + b.im;
30        return new Complex(real, imag);
31    }
32
33    /*
34     * Retorna um novo número complexo cujo valor é a subtração
35     * deste número complexo com aquele passado como parâmetro.
36     */
37    public Complex minus(Complex b) {
38        double real = this.re - b.re;
39        double imag = this.im - b.im;
40        return new Complex(real, imag);
41    }
42
43    /*
44     * Retorna um novo número complexo cujo valor é a multiplicação
45     * deste número complexo com aquele passado como parâmetro.
46     */
47    public Complex times(Complex b) {
48        Complex a = this;
49        double real = a.re * b.re - a.im * b.im;
50        double imag = a.re * b.im + a.im * b.re;
51        return new Complex(real, imag);
52    }
53 }
```

```

54
55
56     /*
57      Retorna um novo número complexo cujo valor é a multiplicação
58      deste número complexo com uma constante alpha passada como
59      parâmetro.
60     */
61     public Complex scale(double alpha) {
62         return new Complex(alpha * re, alpha * im);
63     }
64
65     /*
66      Retorna um novo número complexo cujo valor é o conjugado desta
67      instância.
68     */
69     public Complex conjugate() {
70         return new Complex(re, -im);
71     }
72
73 }
74

```

A classe FFT pode ser vista como uma classe auxiliar que contém apenas um método, chamado de `fft`. Esse método é uma implementação do algoritmo Cooley-Tuckey radix-2. Você pode encontrar mais referências sobre esse algoritmo online. Um detalhe importante sobre esse método é que ele precisa que o sinal de entrada tenha tamanho igual a uma potência de dois. Caso contrário, uma exceção é gerada com uma chamada a `throw`. Como na classe `MicrophoneSampler` tomamos cuidade de gerar um buffer cujo tamanho é uma potência de dois, essa restrição não será um problema para nós.

```

1 package com.example.ulisses.pocketlab.math;
2
3 public class FFT {
4
5
6     /*
7      Computa a FFT de x[]. Nesse método assumimos que o tamanho de x
8      é uma potência de 2. Esse método é uma implementação do Algoritmo
9      Cooley-Tuckey radix-2.
10     */
11
12     public static Complex[] fft(Complex[] x) {
13         int n = x.length;
14
15         /*
16          O Algoritmo de Cooley Tukey precisa que o array de entrada
17          tenha tamanho igual a uma potência de dois.
18         */
19         if (n % 2 != 0) {

```

```

20         throw new RuntimeException("n is not a power of 2");
21     }
22
23     /*
24      Caso base
25    */
26     if (n == 1) {
27       return new Complex[]{x[0]};
28     }
29
30     /*
31      Implementação do Algoritmo de Cooley Tukey para o Cálculo da
32      Transformada Rápida de Fourier.
33    */
34
35     /*
36      Caso 1: Números pares.
37    */
38     Complex[] even = new Complex[n / 2];
39     for (int k = 0; k < n / 2; k++) {
40       even[k] = x[2 * k];
41     }
42     Complex[] q = fft(even);
43
44     /*
45      Caso 2: Números ímpares.
46    */
47     Complex[] odd = even;
48     for (int k = 0; k < n / 2; k++) {
49       odd[k] = x[2 * k + 1];
50     }
51     Complex[] r = fft(odd);
52
53     /*
54      Combinando os dois casos.
55    */
56     Complex[] y = new Complex[n];
57     for (int k = 0; k < n / 2; k++) {
58       double kth = -2 * k * Math.PI / n;
59       Complex wk = new Complex(Math.cos(kth), Math.sin(kth));
60       y[k] = q[k].plus(wk.times(r[k]));
61       y[k + n / 2] = q[k].minus(wk.times(r[k]));
62     }
63     return y;
64   }
65 }
```

A classe SpectrumAnalyzer se assemelha à classe Oscilloscope no sentido de que herda de AbstractModule e implementa o método doDraw para desenhar na tela. A classe também

implementa um método chamado de `computeFFT` que faz uso da classe `FFT` para gerar a série transformada. Feita a transformação, os dados originais deixam de ser necessários e passamos a trabalhar apenas com os módulos dos números complexos gerados. Nesse caso, o método `computeFFT` retorna um array de números reais que são esses módulos.

```
1 package com.example.ulisses.pocketlab.modules;
2
3
4 import com.example.ulisses.pocketlab.math.*;
5
6 import android.content.Context;
7 import android.graphics.Canvas;
8 import android.view.SurfaceHolder;
9
10
11 import com.example.ulisses.pocketlab.microphone.MicrophoneSamplerThread;
12
13 public class SpectrumAnalyzer extends AbstractModule
14 {
15
16     private int mCanvasHeight = 10000;
17     private int mCanvasWidth = 10000;
18     private float ratioX;
19     private float ratioY;
20     private float minFrequency = 10f;
21     private float maxFrequency = 1000f;
22     private float magnitudeThreshold = 80000f;
23
24     private float maxMagnitude = 10000000f;
25     private float minMagnitude = 1000;
26     private Context mContext;
27
28     public SpectrumAnalyzer(SurfaceHolder holder, MicrophoneSamplerThread
29         → samplerThread, Context context) {
30         super(holder, samplerThread, context);
31     }
32
33     public double[] computeFFT() {
34         /*
35             Criando um array de números complexos chamar o método fft da
36             classe FFT.
37         */
38         Complex[] fftTempArray = new Complex[buffer.length];
39         for (int i = 0; i < buffer.length; i++) {
40             fftTempArray[i] = new Complex(buffer[i], 0);
41         }
42
43         /*
44             Obtendo a FFT e gerando os módulos em um array do tipo double.
45         */
```

```
44     */
45     Complex[] fftArray = FFT.fft(fftTempArray);
46
47     double[] absArray = new double[fftArray.length];
48     for (int i = 0; i < fftArray.length; i++) {
49         absArray[i] = fftArray[i].abs();
50     }
51     return absArray;
52 }
53
54
55     public void doDraw(Canvas paramCanvas) {
56         if (paramCanvas.getHeight() < mCanvasHeight) {
57             mCanvasHeight = paramCanvas.getHeight();
58             mCanvasWidth = paramCanvas.getWidth();
59             ratioX = (maxFrequency - minFrequency) / mCanvasWidth;
60             ratioY = (maxMagnitude - minMagnitude) / mCanvasHeight;
61         }
62
63         /*
64             Configurando os intervalos que serão impressos na tela
65         */
66
67         int height = mCanvasHeight;
68         int width = mCanvasWidth;
69
70         maxFrequency = minFrequency + ratioX * mCanvasWidth;
71         maxMagnitude = minMagnitude + ratioY * mCanvasHeight;
72
73
74         /*
75             Computando a FFT
76         */
77         double[] fftArray = computeFFT();
78
79
80
81         /*
82             Definindo o intervalo de valores que serão impressos na tela
83         */
84         int x_start = (int) ((minFrequency * buffer.length) / 44100);
85         int x_end = (int) ((maxFrequency * buffer.length) / 44100);
86
87
88         int x_range = x_end - x_start;
89         float[] arrayDecibeis = new float[x_range];
90         float maxDecibeis = 0;
91         for (int i = x_start; i < x_end; i++) {
```

```
93     arrayDecibeis[i - x_start] = (float) fftArray[i];
94     if (arrayDecibeis[i - x_start] > maxDecibeis) {
95         maxDecibeis = arrayDecibeis[i - x_start];
96     }
97 }
98 if (maxDecibeis > magnitudeThreshold) {
99     paramCanvas.drawPaint(backPaint);
100
101 /*
102     Imprimindo o grid e os valores de referência.
103 */
104 float margin_y = height / 10;
105 float margin_x = width / 10;
106
107 float delta_mag = (maxMagnitude - minMagnitude) / 10;
108 float delta_freq = (maxFrequency - minFrequency) / 10;
109
110
111 for (int i = 0; i < 10; i++) {
112     paramCanvas.drawLine(i * margin_x, 0, i * margin_x,
113         ↵ height, linePaint);
114     paramCanvas.drawLine(0, i * margin_y, width, i * margin_y,
115         ↵ linePaint);
116
117     paramCanvas.drawText(" " + (int) (minMagnitude + i *
118         ↵ delta_mag), width / 2, height - i * margin_y,
119         ↵ stringPaint);
120     paramCanvas.drawText(" " + (int) (minFrequency + i *
121         ↵ delta_freq), i * margin_x, height / 2 - margin_y / 2,
122         ↵ stringPaint);
123 }
124
125 /*
126     Imprimindo as escalas na tela
127 */
128 float x0 = -1;
129 float y0 = -1;
130 for (int i = x_start; i < x_end; i++) {
131     float x = (float) (i - x_start) * width / (float) x_range;
132
133     float decibeis = arrayDecibeis[i - x_start];
134     float y = height * (decibeis - minMagnitude) /
135         ↵ (maxMagnitude - minMagnitude);
136     y = height - y;
137
138     paramCanvas.drawCircle(x, y, 5, linePaint);
139     if (x0 != -1 && y0 != -1) {
140         paramCanvas.drawLine(x0, y0, x, y, linePaint);
141     }
142 }
```

```
135 }  
136     x0 = x;  
137     y0 = y;  
138 }  
139 }  
140 }  
141 }
```


Confecção da Ponteira de Prova

8	Montando a ponta de prova	55
9	Utilização do Arduino	57
9.1	Considerações de Hardware	
9.2	Considerações de Software	
9.3	Programação	
9.4	Geração de sinais para monitoração	
9.5	Código para geração de ondas quadradas de diferentes frequências.	
10	Laboratórios de testes e medidas	63
11	Referências	65

Nesta segunda parte do material, discutiremos os aspectos conceituais e práticos que envolvem a confecção da ponteira de prova que será utilizada no "PocketLab".

Utilizaremos a entrada de microfone (plug P2), do smartphone para a realização da conexão de entrada com a ponteira que será confeccionada. Aqui temos uma consideração importante: primeiramente é necessário verificar se o seu celular possui entrada de microfone, (a maioria dos celulares mais modernos possuem uma conexão elétrica para entrada de microfone acessível através da entrada P2 do celular, a mesma utilizada para utilização do fone de ouvido). Havendo entrada de microfone, o plug P2 a ser utilizado difere um pouco dos plugs normalmente utilizados para fone de ouvido, pois ele possuirá um contato a mais, correspondendo justamente ao contato relativo ao microfone, conforme pode ser visto na figura 1, que mostra a imagem do plug macho que será utilizado para a confecção da nossa .

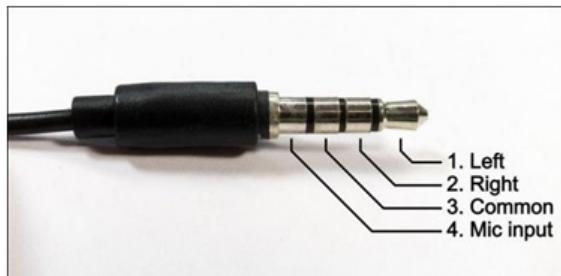
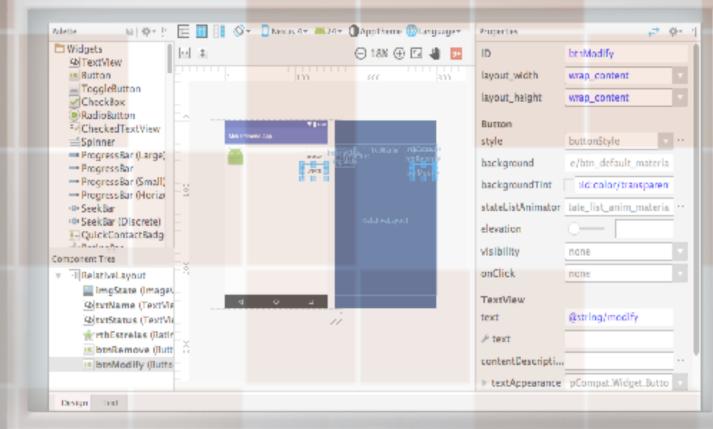


Figura 1. Plug P2 com contato para microfone



8. Montando a ponta de prova

Nesta seção confeccionaremos as ponteiras de prova que serão utilizadas para estabelecer a conexão entre o sinal que se deseja medir e o Smartphone, para isso algumas informações são necessárias. Primeiramente, a ponteira consiste em um circuito passivo, que precisa propiciar um casamento de impedância entre a entrada do celular e a impedância do ponto que se quer medir. Uma ponta de prova normalmente ainda terá a função de atenuar o sinal de entrada, o que pode ser conseguido através de um divisor resistivo. No nosso caso estaremos utilizando as saídas do Arduino, e teremos nossa saída sempre limitada a 5V. Porém, caso você tenha a intenção de realizar medidas em outras fontes de sinal, cuidado com magnitude do sinal medido para evitar danificar o Smartphone. Para sinais de maior magnitude utilize um divisor resistivo para realizar a atenuação de acordo com a magnitude esperada do sinal medido. A figura 2 mostrar a configuração geral da ponteira que iremos confeccionar com alguns valores possíveis. Para o cálculo aproximado do valore da atenuação para a configuração da figura 2, pode-se utilizar a seguinte formula:

Valor na entrada do Smartphone = Valor medido $(R1R2) / (R1R2 + R1R3 + R2R3)$

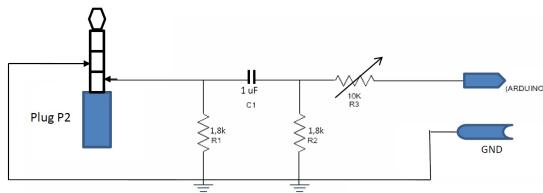


Figura 2. Esquemático da ponteira.

Os Smartphones que utilizam sistema Android, normalmente exibem uma tensão DC de polarização na entrada de microfone que é utilizada para criação de funções adicionais através do uso de botões no caminho elétrico da entrada de microphone, por meio do acionamento de botões podem-se estabelecer diferentes tensões geradas através de diferentes impedâncias criadas pelos botões, e que são lidas como comandos tais como play,pause,vol+,vol-, etc., estas funções estão detalhadas em [1], de maneira a evitarmos a interpretação incorreta na entrada de microphone,

acionando tais funções, o resistor variável da ponteira será ajustado em torno de 1,8k ohms, para manter a impedância equivalente fora dessa faixa de valores de acionamento.

Para bloquear o nível DC presente no pino de microfone do Smartphone, e também para evitar a passagem de níveis DC que eventualmente poderiam danificar a entrada do Smartphone utilizaremos um capacitor de desacoplamento no circuito da ponteira. (Como os sinais a serem monitorados não são de frequência muito elevada < 20kHz, e se pretende cortar a componente DC, será utilizado um capacitor de $1\mu F$).

O valor da impedância de entrada de microfone P2 do Smartphone será considerado como sendo $= 1.8k\Omega[2][3]$, esse valor é baseado em medidas empíricas. O pino do Arduino onde serão realizadas as medidas no nosso experimento, pode ser considerado com sendo uma fonte de tensão com uma impedância muito baixa. Dessa forma o circuito da ponta de prova terá a função de atenuar, desacoplar e casar a impedância de entrada do Smartphone. Dadas essas considerações a ponteira será construída conforme a especificação da figura 2. A figura 3 mostra uma ponteira real montada segundo essa especificação.

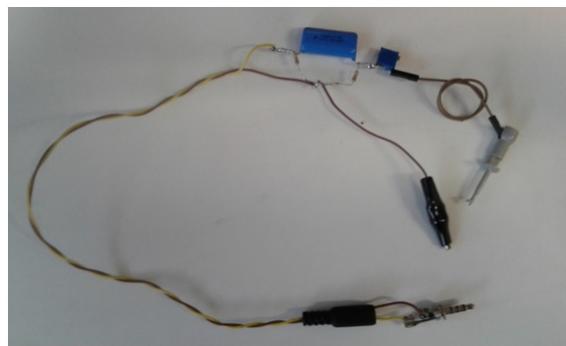
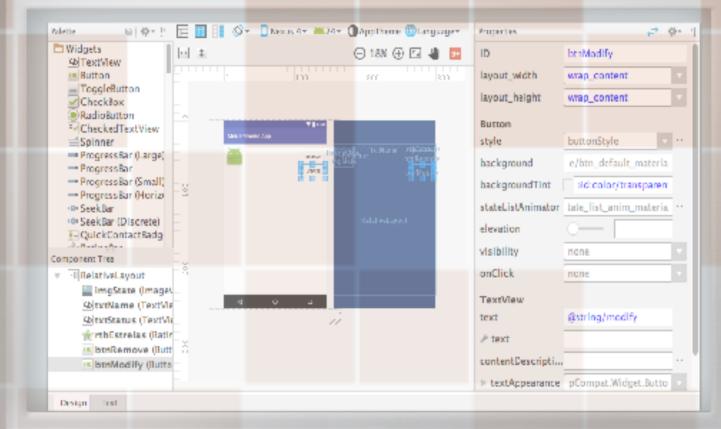


Figura 3 - Foto de uma ponteira de prova prática



9. Utilização do Arduino

Nessa seção discutiremos o uso da placa Arduino Mega para geração de sinais elétricos para serem monitorados no app Osciloscópio desenvolvido dentro do projeto "PocketLab".

O Arduino é um projeto aberto de software e hardware, que utiliza um programação bastante simples baseada nas linguagens C/C++.

9.1 Considerações de Hardware

O hardware que será utilizado possui basicamente um microcontrolador de 8bits da Atmel (ATmega2560) como elemento de programação. O microcontrolador é montado em uma placa que fornece ainda um conjunto básico de conectores que permitem a exteriorização de sinais elétricos gerados no microcontrolador ou ainda a leitura de sinais elétricos externo para processamento interno.

9.2 Considerações de Software

Conforme mencionado anteriormente, o software para programação das placas Arduino é aberto, podendo ser baixado gratuitamente diretamente no site do projeto Arduino, [4]. O site do projeto Arduino disponibiliza ainda tutoriais, fóruns, projetos, além de diversas informações relacionadas ao uso da plataforma.

9.3 Programação

A programação no Arduino é relativamente simples, sendo seus programas normalmente compostos por duas macro estruturas. A primeira macro estrutura é normalmente utilizada para a configuração inicial do modo de operação dos pinos que serão utilizados pelo programa. Estas configurações são reunidas dentro da função `setup()`. A segunda estrutura contém as instruções que descrevem tarefas que serão executados pelo programa e são agrupadas dentro da função `loop()`. Com relação à sintaxe de programação é bom sempre ter em mente que, o corpo de instruções dentro das funções é delimitado por chaves " ", e as instruções são sempre finalizadas com ";".

9.4 Geração de sinais para monitoração

Conforme mencionado anteriormente, o Arduino será utilizado apenas como ferramenta integrante do kit para realização dos laboratórios propostos no "PocketLab", não sendo objetivo deste material se aprofundar em questões relativas a programação e uso dessa plataforma, limitando seu uso apenas como um "coadjuvante" nos laboratórios.

Nas seção a seguir descreveremos o uso da placa Arduino para geração de ondas que serão monitorados no osciloscópio.

Para esse "PocketLab" criaremos um programa no Arduíno que permite a geração de ondas quadradas de diferentes frequência, e que serão disponibilizadas em um pino de saída da placa Arduino Mega, para que através da ponteira de prova confeccionada, possamos verificar as formas de onda no osciloscópio implementado no smartphone.

9.5 Código para geração de ondas quadradas de diferentes frequências.

O programa que será descrito a seguir, possibilita a geração de ondas quadradas com duty cycle de 50% e com frequência que varia de 0 até 20KHz. Para isso duas intruções dedicadas disponíveis na plataforma Arduino estão sendo utilizadas e serão detalhadas a seguir para posterior melhor compreensão do programa que será mostrado:

- Instruções dedicadas

analogRead(pinName) Lê o valor analógico,(dentro do intervalo de 0 Volts até 5 Volts) disponível no pino de entrada analógico, (informado no argumento da função) e retorna um valor entre 0 (0Volts) e 1023(5Volts).

tone(pinName, frequency) Gera uma onda quadrada com duty cycle de 50% no pino de saída informado no primeiro argumento da função e com a frequência informada pelo segundo argumento da função (os valores de frequencia podem variar de 3Hz até 65535Hz).

- Hardware: a fim de implementarmos este gerador na placa Arduino Mega, algumas considerações para definição das interfaces de entrada e saída na placa são necessárias e descritas a seguir.

Entradas analógicas Para entradas de tensões analógicas são disponibilizados 16 pinos identificados de A0 até A15 (barra de pinos ANALOG IN) e que aceitam tensões de entrada de 0Volts até 5Volts.

Entradas digitais A placa disponibiliza 32 pinos para utilização de sinais digitais (valor de 0Volts ou 5Volts), e que podem ser configurados através da programação como entradas ou saídas. Estes pinos estão identificados na barra de pinos DIGITAL e possuem numeração de 22 até 53.

Tensões disponíveis na placa É possível ainda ter acesso a Tensão de entrada de 5V(pino Vin) e ao terra comum (pino GND), (e.g barra de pinos POWER). A figura 4 mostra a placa Arduino Mega com a identificação do hardware descrito.

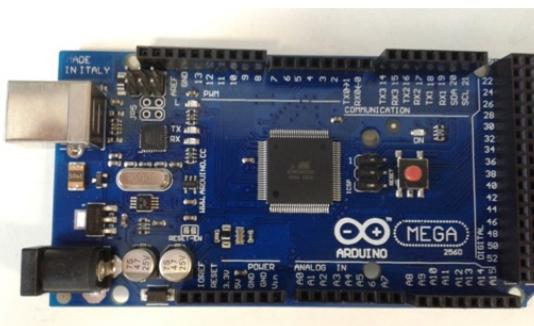


Figura 4 – Placa Arduino Mega

Para o nosso gerador, a geração de diferentes frequências de saída serão controlada externamente através do uso de um potenciômetro conectado entre uma saída de 5V disponibilizada pela própria placa (Vin) e um pino de terra da placa (gnd) (conforme figura 5).

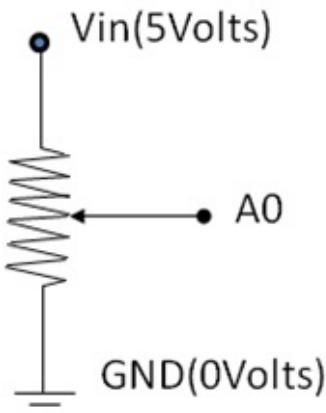


Figura 5 . Variação de tensão no pino A0

Utilizaremos um potenciômetro de $5k\Omega$ como um divisor resistivo, e através da uso da tensão do seu terminal intermediário, fornecida como entrada analógica para placa, através de um de seus pinos analógicos (A0), controlaremos a frequência desejada em um pino de saída digital (pino 53). Note que através da excursão do terminal intermediário do potenciômetro, a tensão lida no pino de entrada A0 da placa Arduino Mega pode ser variada desde 0V até 5V. Com esse fato em mente, a idéia é bastante simples: O valor analógico fornecido em A0 será convertido em um valor entre 0 e 1023, através da função `analogRead(pinName)`, discutida anteriormente, e esse valor será ajustado para fornecer uma onda quadrada de frequência de saída entre 31Hz e 20KHz através do pino 53, através da função `tone(pinName, frequency)`.

A figura 6 mostra a placa Arduino Mega configurada com o potênciometro para ajuste da frequência de saída que será monitorada no pino 53.

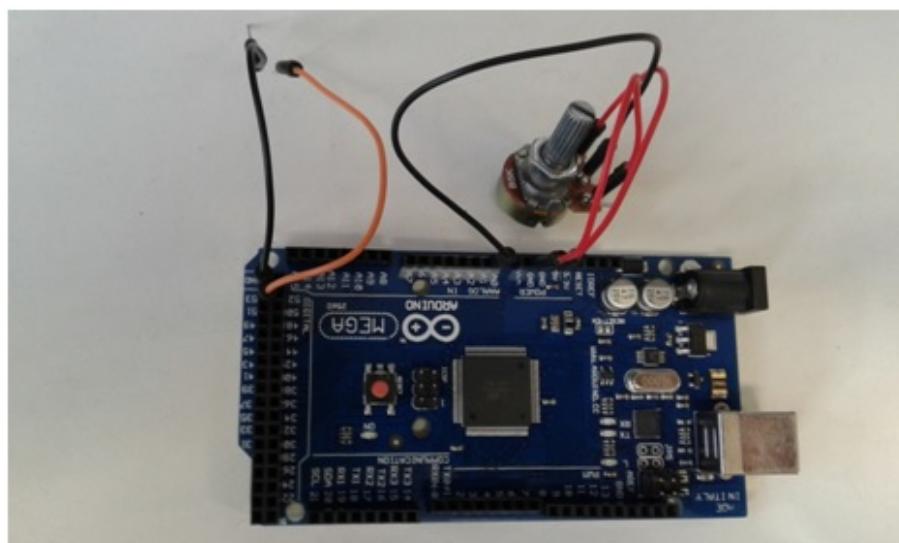


Figura 6 – Configuração do Hardware para ajuste de frequência utilizando potenciômetro externo e monitoração na saída do pino 53.

9.5.1 Programa

Baseado na descrição acima, o programa a seguir implementa as configurações e o conjunto de funções no Arduino necessárias pra criarmos o gerador de formas de onda utilizando a placa Arduino MEGA.

```

1 /**
2  Programa para implementação do gerador de forma de onda quadrada com
3  frequências entre 30Hz e 20KHz
4 */
5
6 void setup() {
7     pinMode(53, OUTPUT); //define o pino 53 como saída
8 }
9 void loop() {
10    int tensaoControle = analogRead(A0); // atribui o valor de tensão no
11                                // pino A0 a um variável
12                                // inteira.
13    int frequencia = 31+ tensaoControle*20; // ajusta o valor lido entre
14                                // 0 e 1023 para cair na
15                                // faixa entre 31Hz e
16                                // ~20kHz.
17    tone(53,frequencia); //gera uma onda quadrada com o valor da
18                                //variável frequencia.
19 }
20 /**
21  Programa para implementação do gerador de forma de onda quadrada
22  com frequências entre 30Hz e 20KHz
23 */

```

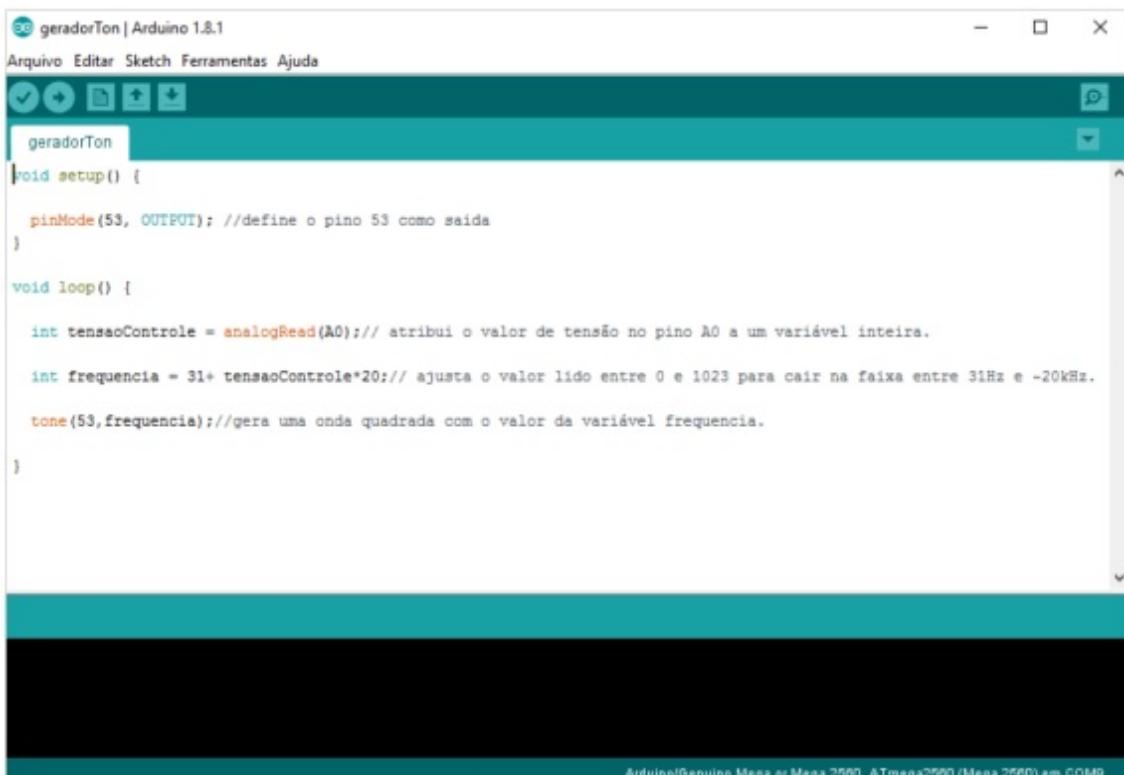
9.5.2 IDE Arduino

A IDE (Integrated Development Environment) para a programação da placa Arduino Mega pode ser baixada diretamente do site do projeto Arduino [4]. Siga os passos indicados para download e instalação fornecidos no site. Uma vez baixado e instalado o IDE, abra o aplicativo. Se tudo correu bem, você verá uma tela parecida com a da figura 7. Note que dependendo da época que você fizer o download, a sua versão pode diferir da versão desse documento.



Figura 7 – IDE Arduino

Em seguida será aberta automaticamente a tela do editor e você pode simplesmente inserir o código descrito anteriormente, como na figura 8 abaixo.



The screenshot shows the Arduino IDE interface with the following details:

- Title Bar:** geradorTon | Arduino 1.8.1
- Menu Bar:** Arquivo Editar Sketch Ferramentas Ajuda
- Toolbar:** Includes icons for Save, Open, New, Print, and others.
- Code Editor:** The sketch named "geradorTon" contains the following C++ code:

```
void setup() {
    pinMode(53, OUTPUT); //define o pino 53 como saída
}

void loop() {
    int tensaoControle = analogRead(A0); // atribui o valor de tensão no pino A0 a um variável inteira.
    int frequencia = 31 + tensaoControle*20; // ajusta o valor lido entre 0 e 1023 para cair na faixa entre 31Hz e -20kHz.
    tone(53, frequencia); // gera uma onda quadrada com o valor da variável frequencia.
}
```
- Status Bar:** Arduino/Genuine Mega or Mega 2560, ATmega2560 (Mega 2560) em COM9

Figura 8 – Tela de edição do IDE Arduino

Para que possamos fazer a programação da placa Arduino Mega, é necessário que o IDE esteja configurado para a programação da placa/entrada de programação correspondentes. Normalmente a IDE consegue identificar a placa quando conectada ao computador para a programação de forma automática. Conforme pode ser observado no canto inferior direito da figura 8. No entanto no menu ferramentas/placa , é possível verificar/alterar o hardware para programação, conforme mostrado na figura 9.

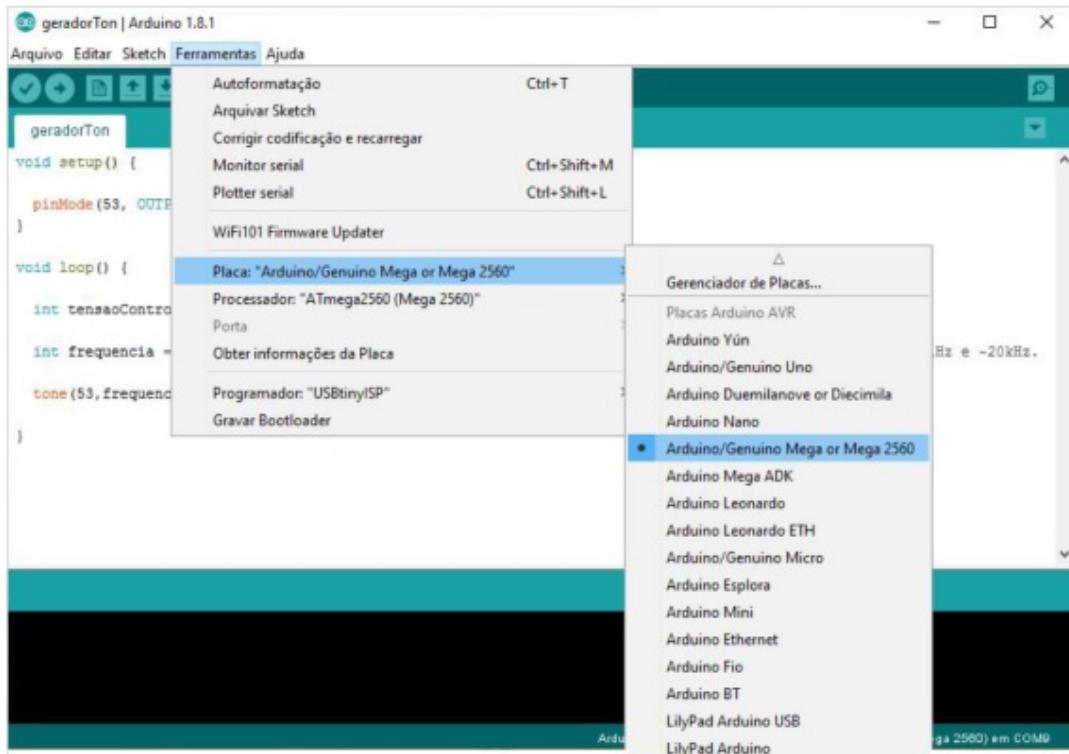
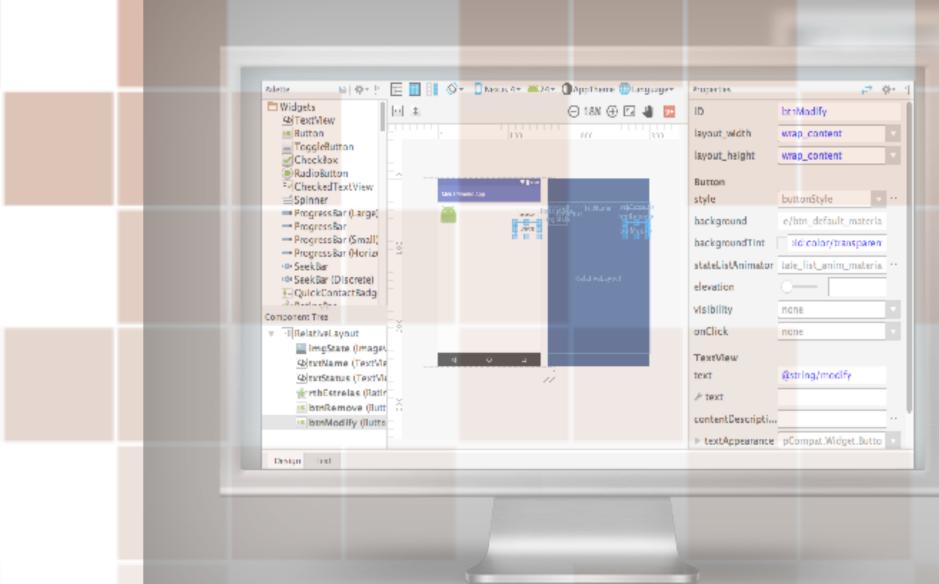


Figura 9 – Configuração do Hardware na IDE

Uma vez realizadas estas verificações, realize primeiramente a compilação do programa para verificação de eventuais erros, através do botão de verificação no menu superior esquerdo, e em seguida realize a carga na placa, através do botão ao lado conforme mostrado na figura 10.



Figura 10 – Verificação/Carga do programa no hardware.



10. Laboratórios de testes e medidas

Se todos os passos anteriores foram bem sucedidos, então podemos agora integrar o gerador de sinais (Placa Arduino Mega), a ponteira confeccionada que deverá estar plugada na entrada de microfone do smartphone, e com as pontas de medida conectadas ao terra (GND) da placa arduino e a nossa saída de medida (Pino 53) da placa.

Através da excursão do potenciômetro é possível monitorar a variação da frequência gerada na placa através da tela do smartphone que implementa o nosso osciloscópio. As figuras 10, 11 e 12 mostram diferentes médias da forma de onda na saída da placa Arduino para diferentes valores ajustados no potenciômetro.

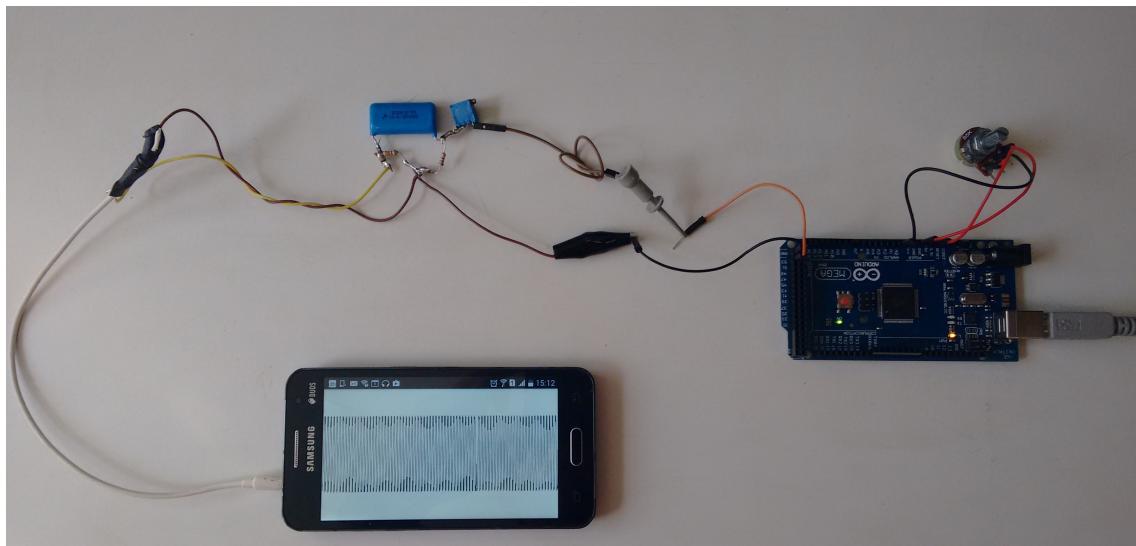


Figura 11 – Potenciômetro ajustado no fundo de escala $5k\Omega$ correspondendo a frequência máxima de saída no Arduíno.

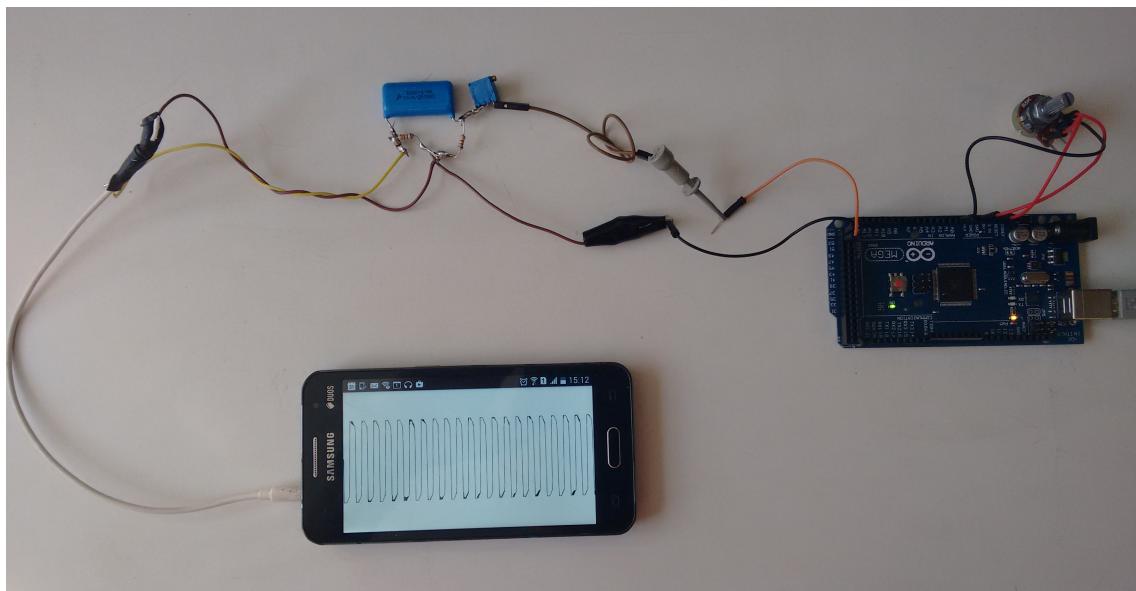


Figura 12 – Potenciômetro ajustado em meia escala $2,5\text{k}\Omega$ correspondente a frequência máxima/2 de saída no Arduíno.

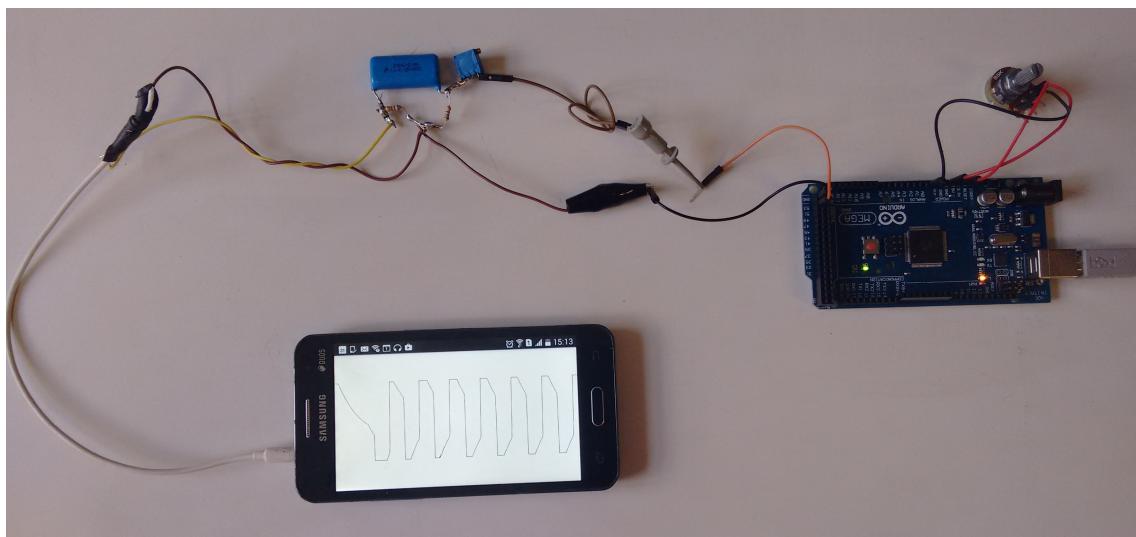
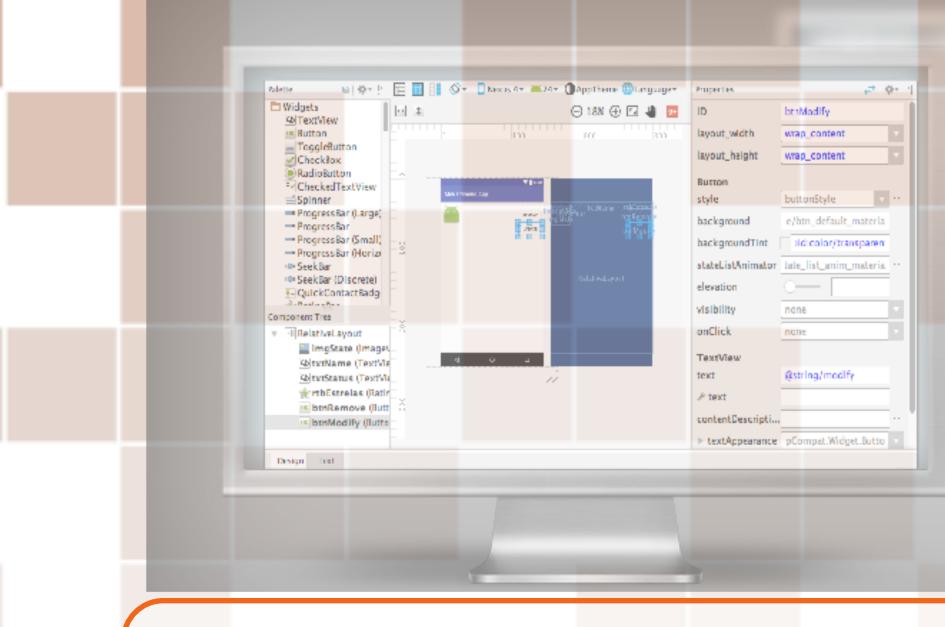


Figura 13 – Potenciômetro ajustado no início de escala = 0Ω correspondento a frequência mínima de saída no Arduíno.

Esse experimento busca apenas mostrar as potencialidades do esquema implementado. Através da diferentes programações da placa Arduino Mega, ou mesmo através de medição de valores na saída de outros dispositivos, é possível comprovar os conceitos envolvidos na monitoração de sinais através de um osciloscópio por meio do "PocketLab" criado nessa apostila.



11. Referências

- [1] <http://source.android.com/devices/accessories/headset/plug-headset-spec.html>
- [2] <http://electronics.stackexchange.com/questions/6846/how-important-is-impedance-matching-in-audio-applications>
- [3] <http://electronics.stackexchange.com/questions/38452/electronic-aspects-of-iphone-3-5mm-audio-output>
- [4] <https://www.arduino.cc/>