

Como funciona nosso App de comunicação *bluetooth*?



O App responde a eventos (ações de usuário ou ‘coisas’ que acontecem, tais como mensagens de sensores, passar do tempo...).

Um evento pode ser acionado com uma situação de sistema ou pelo usuário. Por exemplo, iniciar um aplicativo ou clicar em um ícone.

Em nosso App o primeiro evento que ocorre é a inicialização da tela, ou seja, o início. Este evento é utilizado pelo nosso App. Ele ocorrerá assim que o usuário iniciar o App:

```
quando Screen1 .Inicializar  
fazer chamar esconderBotoes
```

- ‘quando’ representa a ocorrência do evento, que é o ‘Inicializar’ a tela inicial (‘Screen1’ – esta SEMPRE é a primeira tela inicializada em Apps no MIT Inventor)
- ‘fazer’ é a ação que será realizada em função da ocorrência do evento; neste caso, será chamada a FUNÇÃO ‘esconderBotoes’

Quando colocamos objetos na tela de nosso App podemos modificar suas propriedades. Alguns objetos possuem a opção de ficarem ou não visíveis (eventualmente, visíveis porém não habilitados). A função ‘esconderBotoes’ tem a tarefa de tornar não visíveis alguns objetos do tipo ‘botão’:

```
para esconderBotoes  
fazer  
  ajustar btnDesconectar . Ativado para falso  
  ajustar btnDesliga13 . Ativado para falso  
  ajustar btnLiga13 . Ativado para falso
```

- Quando for chamada, esta função irá ‘ajustar’ a propriedade ‘Ativado’ dos objetos ‘btnDesconectar’, ‘btnDesliga13’ e ‘btnLiga13’ para ‘falso’. Isto fará com que os botões de desconectar e de ligar e desligar o LED da placa do Arduino fiquem inacessíveis ao usuário. Por quê? Para garantir que as operações a realizar não sejam chamadas aleatoriamente, porém com um pensamento lógico: como o usuário irá desconectar o aplicativo se ele não foi antes conectado? Ou, como ele irá controlar o LED para ligá-lo ou desligá-lo se não há conexão? Desta forma, garantimos que o aplicativo funcione de forma a não dar ao usuário opções incorretas.
 - Cabe uma observação – poderíamos ter iniciado o App com um desenho tal que os botões já estivessem desativados. Mas, em caso de uma conexão e posterior desconexão eles ficariam ativados e teríamos que ter uma função para desativá-los de qualquer forma. Assim, iniciamos no App controlando por programação o que acontece em seu comportamento.

Neste ponto, o usuário terá a seu dispor uma tela na qual só pode realizar duas ações:

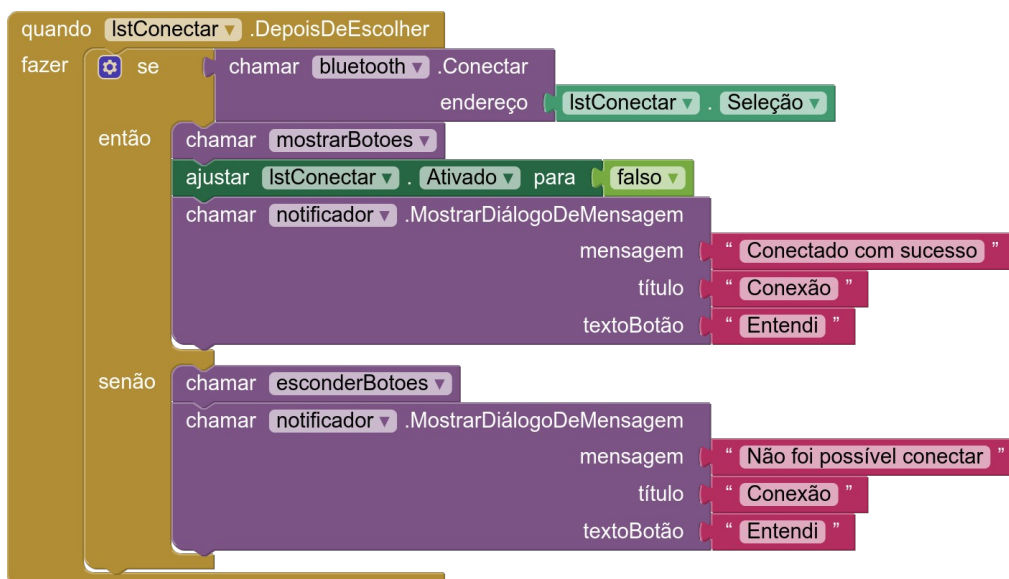


- clicar (evento ‘Clique’) no botão ‘btnSair’ e, com isso, provocar a execução (‘fazer’) do comando ‘fechar aplicação’ – obviamente, encerrando o App



- clicar no objeto de lista ‘lstConectar’; será utilizado o método ‘AntesDeEscolher’. O objeto de lista, ou botão de lista, apresentará na tela uma lista na qual os elementos podem ser selecionados com um ‘clique’. A lista será escolhida e preenchida por meio do comando ‘fazer’: a lista irá ‘ajustar’ seus ‘Elementos’ utilizando ‘EndereçosENomes’ da conexão ‘bluetooth’. O usuário verá na tela uma lista de dispositivos *bluetooth* e poderá selecionar um deles.
 - Observações: 1) se não houver conexão bluetooth disponível o comando não funcionará; 2) se a conexão bluetooth estiver ativada mas não houver nenhum dispositivo pareado, a lista estará vazia.
 - Portanto, para que tudo funcione teremos que estar com o *bluetooth* ativado e com os dispositivos pareados.
 - Poderíamos ter mais blocos de comandos em nosso App para que pudessemos testar/ habilitar o *bluetooth* e suas conexões, mas deixaremos esta melhoria para outra oportunidade.

O que ocorre após a seleção de uma conexão *bluetooth*?



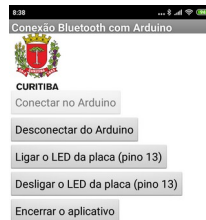
- O evento ‘DepoisDeEscolher’ no objeto ‘lstConectar’ provoca a execução do bloco acima. Foi inserido no comando ‘fazer’ um teste lógico, formado pelo bloco SE/ENTÃO/SENÃO: tudo começa

com a execução dos blocos de código que seguem o teste ‘SE’, o qual retorna um verdadeiro ou um falso em função de sua execução. O comando relacionado ao teste é o de ‘chamar’ o objeto ‘bluetooth’ e solicitar a função de ‘Conectar’, para a qual é passado como parâmetro o ‘endereço’ de conexão (que foi selecionado pelo usuário por meio da lista de dispositivos exibida, e está disponível na forma ‘lstConectar.Seleção’.

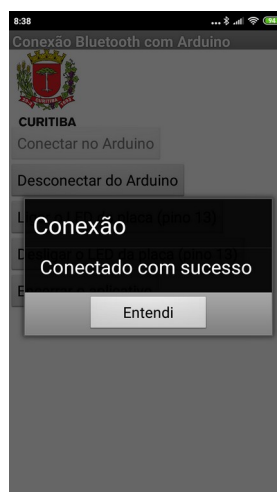
- SE deu certo (a conexão foi bem sucedida), será executado o bloco ‘ENTÃO’:
 - será chamada a função mostrarBotoes:



- a função ‘mostrarBotoes’ será executada até o final, retornando depois ao próximo passo da lista de comandos relacionados ao bloco ‘ENTÃO’. Isto é chamado de desvio de fluxo, já que o fluxo normal é a execução do bloco de teste SE, e estamos fora deste bloco, em outra função.
 - a função ‘mostrarBotoes’ irá ‘ajustar’ a propriedade ‘Ativado’ dos botões ‘btnDesconectar’, ‘btnDesliga13’ e ‘btnLiga13’ para ‘verdadeiro’. Isto fará com que os mesmos fiquem disponíveis ao usuário (pois a conexão deu certo e, agora, o usuário poderá comandar o LED da placa ou então desconectar o dispositivo *bluetooth*).
 - O objeto de lista ‘lstConectar’ terá sua propriedade ‘Ativado’ mudada para ‘falso’, fazendo com que o usuário não possa conectar em outro dispositivo sem desconectar antes do atual.



- Será chamado o ‘Notificador’. O notificador exibe uma mensagem na tela, estilo ‘pop up’ (uma pequena janela sobre o aplicativo). Quando ele for executado, por meio de ‘MostrarDiálogoDeMensagem’, serão estabelecidos três parâmetros: ‘mensagem’, ‘título’ e ‘textoBotão’, os quais farão com que a pequena janela exibida tenha o título ‘Conexão’, com texto ‘Conectado com sucesso’; e, haverá um botão com o texto ‘Entendi’.
- Esta janela tem a função de manter o usuário informado quanto às ações do App e seus resultados. Como não foi estabelecida nenhuma ação correspondente, ao clicar no botão ‘Entendi’ a pequena janela será fechada e o programa continuará seu fluxo normal de execução. Neste caso, como o bloco ‘ENTÃO’ não possui outros comandos a execução do bloco ‘SE’ termina (o bloco ‘SENÃO’ não será executado).

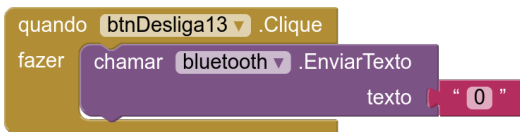


- SE não deu certo (a conexão não foi bem sucedida), será executado o bloco ‘SENÃO’:
 - será chamada a função `esconderBotoes`, já vista, e que tem por finalidade esconder os botões de ação (já que a conexão não deu certo).

Supondo que deu tudo certo, agora o usuário terá as opções:

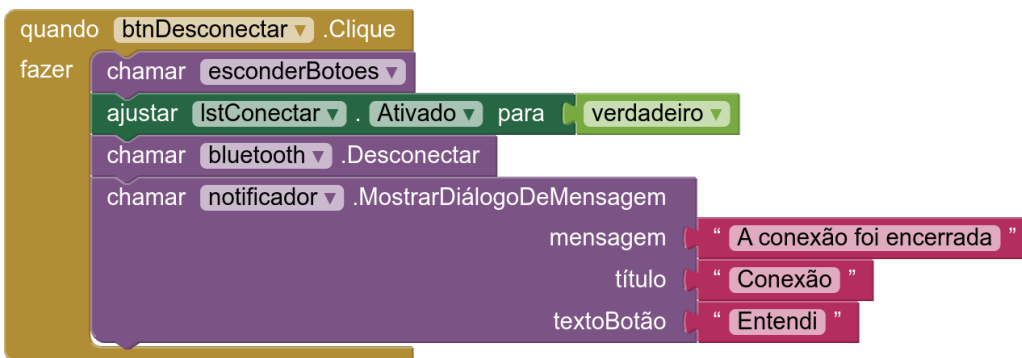


- Quando houver um clique neste botão o comando será o de ‘chamar’ o objeto ‘bluetooth’ e utilizar o método de ‘EnviarTexto’, enviando o texto ‘1’. Isto fará com que o Arduino receba o comando ‘1’, e irá realizar algo em correspondência à sua programação (para nosso exemplo, ligar o LED da placa, pino 13).



- Quando houver um clique neste botão o comando será o de ‘chamar’ o objeto ‘bluetooth’ e utilizar o método de ‘EnviarTexto’, enviando o texto ‘0’. Isto fará com que o Arduino receba o comando ‘0’, e irá realizar algo em correspondência à sua programação (para nosso exemplo, ligar o LED da placa, pino 13).
 - Notar que é exatamente mesmo comportamento anterior (o do ligar): só mudou o ‘texto’ enviado.

Finalmente, a última opção disponível ao usuário será:



- Ao clicar no ‘btnDesconectar’, primeiramente será chamada a função ‘esconderBotoes’, já vista. Depois, o objeto ‘lstConectar’ terá sua propriedade ‘Ativado’ mudada para ‘verdadeira’, tornando disponível a opção de conexão com novo dispositivo, quando selecionado. O objeto ‘bluetooth’ será desconectado (permitindo a escolha de nova conexão posteriormente). E, será chamado o notificador, o qual avisará o usuário de que a conexão foi encerrada.

E o que o Arduino faz com a mensagem (‘1’ ou ‘0’) recebida?

O código a seguir exemplifica o tratamento que ocorrerá no arduino (pressupondo que há um modem/ módulo *bluetooth* ligado nas portas/ pinos ‘0’ – Rx e ‘1’ – Tx do Arduino):

```

1 char controle = 0; //define uma variável tipo 'char' e inicializa com '0'
2
3 void setup() {
4   Serial.begin(9600);
5   pinMode(LED_BUILTIN, OUTPUT); //este é o LED da placa, porta digital 13
6   digitalWrite(LED_BUILTIN, LOW); //inicia com o LED desligado
7 }
8
9 void loop() {
10  if(Serial.available()) { //se houver dados na interface serial
11    controle = Serial.read(); //lê os dados da serial e coloca na variável 'controle'
12  }
13  if(controle == '0') //controle é zero?
14  {
15    digitalWrite(LED_BUILTIN, LOW); //desliga o LED
16  }
17  if(controle == '1') //controle é um?
18  {
19    digitalWrite(LED_BUILTIN, HIGH); //liga o LED
20  }
21 }

```

No exemplo acima:

- na linha 1 é declarada uma variável do tipo caracter (`char controle`) que receberá o texto enviado pelo App;
- na linha 4 a comunicação serial é inicializada com velocidade de 9600 bps;
- na linha 5 o LED da placa (`LED_BUILTIN`, pino 13) é inicializado como saída, e na linha 6 há uma escrita de nível baixo (`LOW`) que faz com que ele apague;
- entre as linhas 10 e 12 ocorre um teste lógico: se houver dados na interface serial, estes dados são lidos e colocados na variável 'controle' (definida na linha 1);
- entre as linhas 13 e 16 o valor da variável 'controle' é lido e testado para saber se é igual a '0'; se for, o LED será desligado;
- entre as linhas 17 e 20 o valor da variável 'controle' é lido e testado para saber se é igual a '1'; se for, o LED será ligado.

O código exibido utiliza a interface padrão para comunicação serial (portas '0' - Rx e '1' - Tx). Isto simplifica a programação mas tem uma desvantagem: se a comunicação serial por USB estiver sendo utilizada, haverá conflito entre a comunicação com o computador e com o modem *bluetooth*.

Também podemos criar um objeto de comunicação serial e utilizar outras portas de conexão para realizar a comunicação:

```

1 #include <SoftwareSerial.h>           //biblioteca de comunicação serial
2 #define RX 2                         //pino 2 (será usado como RX pelo software)
3 #define TX 3                         //pino 3 (será usado como TX pelo software)
4 SoftwareSerial bluetooth(RX, TX);     //cria um objeto de comunicação serial
5 char controle = '0';                 //para armazenar o caractere recebido
6
7 void setup() {
8   Serial.begin(9600);                 //inicializa o monitor serial a 9600bps
9   bluetooth.begin(9600);              //inicializa o objeto bluetooth a 9600bps
10  pinMode(LED_BUILTIN, OUTPUT);        //este é o LED da placa, porta digital 13
11  digitalWrite(LED_BUILTIN, LOW);      //inicia com o LED desligado
12 }
13
14 void loop() {
15   if(bluetooth.available()) {         //se houver dados na interface serial na qual está o bluetooth
16     controle = bluetooth.read();      //lê os dados vindos do bluetooth e coloca na variável 'controle'
17   }
18   if(controle == '0')                 //controle é zero?
19   {
20     digitalWrite(LED_BUILTIN, LOW);  //desliga o LED
21   }
22   if(controle == '1')                 //controle é um?
23   {
24     digitalWrite(LED_BUILTIN, HIGH); //liga o LED
25   }
26 }

```

O código exibido acima tem um controle similar ao anterior, porém com uma nova interface serial para ligação do modem *bluetooth*: a linha 1 declara o uso de uma biblioteca (`#include <SoftwareSerial.h>`), as linhas 2 e 3 definem as portas que serão utilizadas para receber e transmitir dados, o que será realizado pelo objeto (`'bluetooth'`) declarado na linha 4. A comunicação serial continua sendo inicializada (linha 8), permitindo trabalhar dados via USB; mas agora também está sendo inicializado o objeto (`'bluetooth'`) na linha 9. No código da linha 15, o teste será realizado não na comunicação padrão (`'Serial.available()'`) como antes, mas sim no novo objeto (`'bluetooth.available()'`). O restante dos testes continua sendo realizado na variável que recebe o texto (`'controle'`).