

Visual Andruino

Mobile System Engineering
Free University of Bozen, Italy
Faculty of Computer Science

GitHub Repository:

<https://github.com/F4b1-/visual-andruino>

Author: Gabriel Alonso Sanz Salas
and
Fabian Gand

Introduction

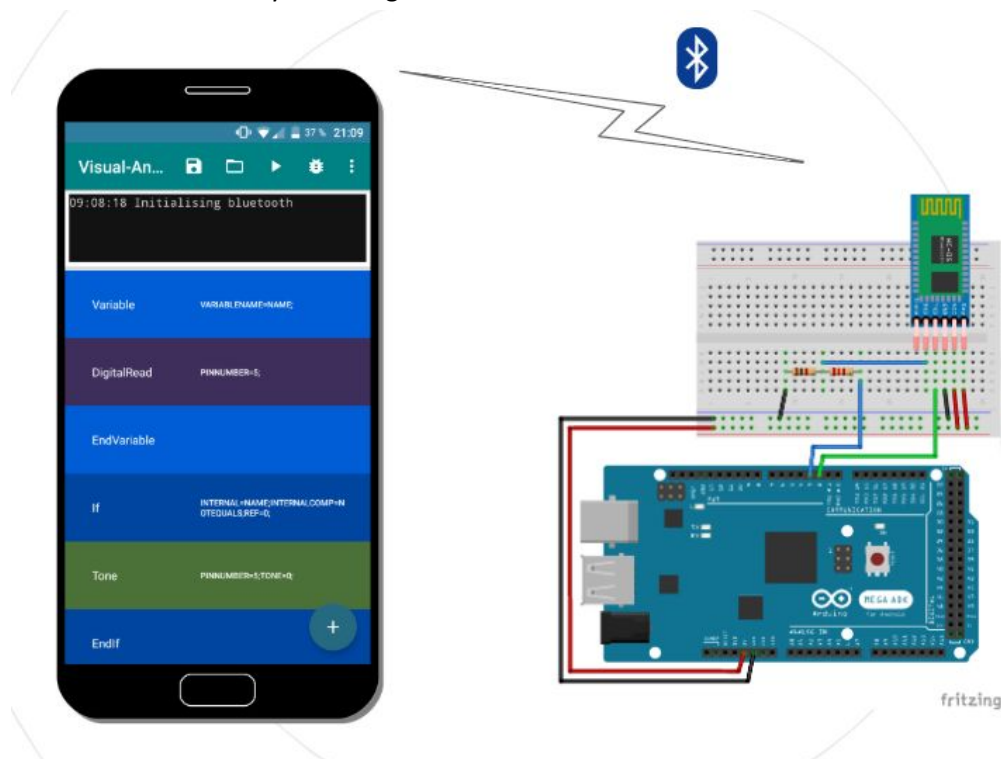
The Arduino is a widely used microcontroller that is especially helpful when introducing people to the basic concepts of programming and electronics. Unfortunately it can only be programmed in a dialect of C/C++: A language family that is not very appealing to beginners. Even experienced programmers complain about the arduino sketches that are often confusing and misleading. Finding one particular line of code in a complex sketch can be a tedious task.

Our idea is therefore a visual programming platform for the arduino microcontroller on android. The UI will be based on code blocks that can be visually combined to create an arduino sketch.

A sketch should be running on the arduino that executes the commands sent to it.

In addition to arduino commands we also want to create code blocks that wrap mobile android logic. This for example should enable the user to create code that controls the brightness of a led connected to the arduino by pressing a button on the phone.

- Bluetooth connection (HC-05/HC-06 Bluetooth adapter for arduino)
- Arduino only receives low level commands, high level logic is on the smartphone.
- Easy to use visual interface to create code logic on the smartphone
- Innovative way to arrange code blocks



Benefits

The visual nature of the project enables inexperienced programmers to take their first steps in programming for the arduino/any microcontroller.

Because of the widespread use of smartphones, many users interested in taking their first steps with the arduino, will have the option to create and test the sketches without having to use a computer.

Even experienced users can make use of the visual interface that provides a better overview of the code than a traditional arduino sketch.

Processing data can be done on the smartphone's processor that is more powerful than the arduino resulting in a quicker and more efficient execution.

Derived Features

Total: **16 weeks** (8 weeks * 2 developers)

1. Code is wirelessly send to and executed on the arduino. **(4 weeks)**
 - a. Arduino Sketch: Commands receiver/sender. (2 weeks)
 - b. Android: Commands sender/receiver. (2 weeks)
2. Predefined low-level code blocks for common commands as well as higher-level blocks for creating own complex functionalities. **(5 weeks)**
 - a. Low-level blocks. (3 weeks)
 - b. High-level blocks. (Optional) (2 weeks)
3. Drag & Drop mechanism to arrange blocks of code. **(3 weeks)**
4. Functionality to use features of the mobile device to interact with the arduino. (Optional) **(1 week)**
5. Tutorials (beginners/intermediate) with direct feedback on the arduino and on the mobile device. **(3 weeks)**

Use Cases

As a user I want to be able to create a sketch by using a drag & drop mechanism with code blocks.

As a user I want to be able to automatically upload and execute the sketch on the arduino.

As a user I want to be able to see the status of the upload and execution process on the arduino.

As a user I want to complete predefined tutorial.

As a user I want to get feedback on the tutorials.

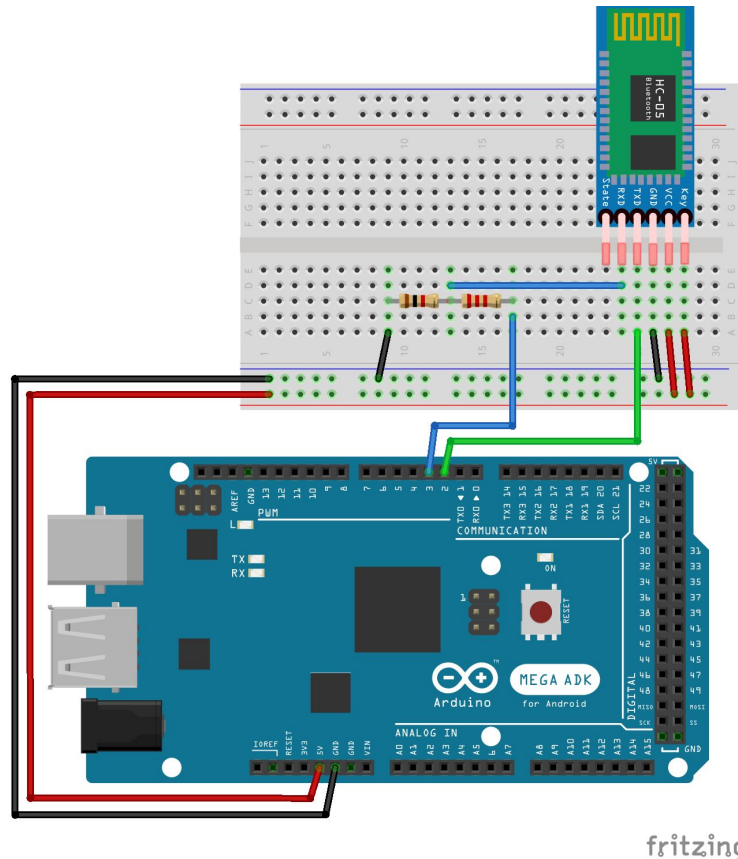
Final Features

1. Code is wirelessly send to and executed on the arduino. (4 weeks)
 - a. Arduino Sketch: Commands receiver/sender. (2 weeks)
 - b. Android: Commands sender/receiver. (2 weeks)
2. Predefined low-level code blocks for common commands as well as higher-level blocks for creating own complex functionalities. (5 weeks)
 - a. Low-level blocks. (3 weeks)
 - b. High-level blocks. (Optional) (2 weeks)
3. Drag & Drop mechanism to arrange blocks of code. (3 weeks)
4. ~~Functionality to use features of the mobile device to interact with the arduino. (Optional) (1 week)~~
5. ~~Tutorials (beginners/intermediate) with direct feedback on the arduino and on the mobile device. (3 weeks)~~
6. **Added:** Debug Mode to execute a sketch brick by brick follow the flow of the execution. Including highlighting of executed brick. (3 weeks)
7. **Added:** Terminal to show information about the current state of the application and the sketch. (1 week)

General Setup

The only prerequisite is a HC-05 or HC-06 bluetooth module that is connected to the arduino. There's a voltage divider for the RX pin of the bluetooth module to reduce the voltage from 5V to 3.3V. This is done to ensure that the module is not damaged over time.

The sketch that comes with the project needs to be uploaded and running on the arduino. All other devices (senors, actuators ...) can be connected to the arduino at will. In our example setup we used a small breadboard for the bluetooth setup (see image below) and a larger one for the sensors and actuators.



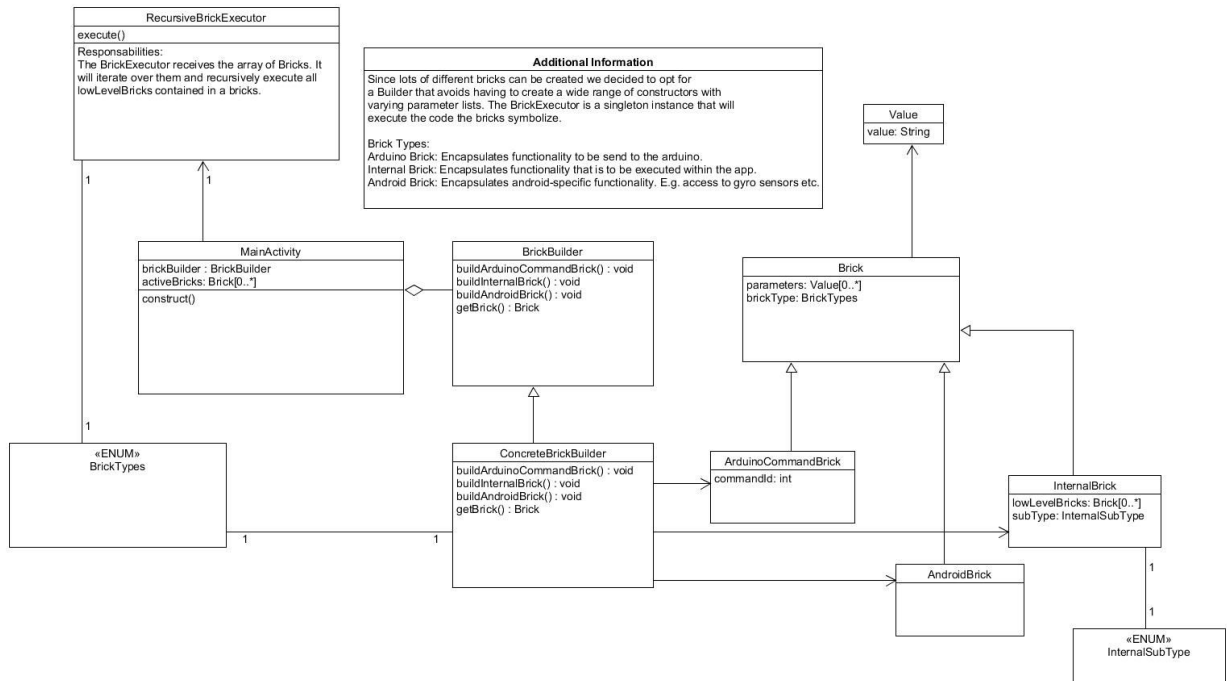
General Setup of the bluetooth module.

Connection

Communication between the devices is done using standardized commands (e.g. `12 11 1`; equals `DigitalWrite(11, HIGH)`, 12 being the command id. We do this to reduce the size of the messages that need to be exchanged. Positive command ids are reserved for write functionality. Negative Values are set for commands like `DigitalRead` where the application is expecting a return value.

Functionalities

General Overview



Class diagram: Overview over the most important classes.

Implemented Bricks

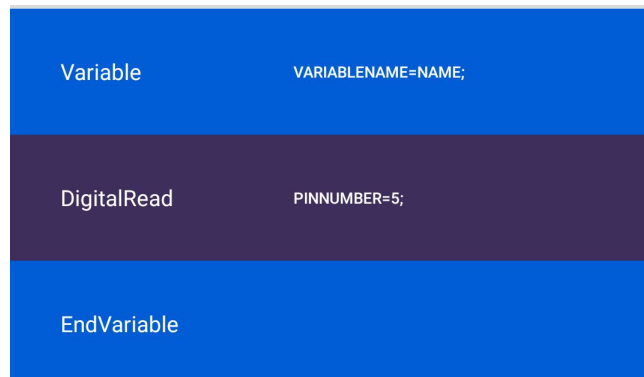
There are two kinds of bricks. The Arduino Command bricks represent single arduino commands like `DigitalWrite` or `AnalogRead`. The InternalCommand bricks represent simple coding structures like an if statement or a for loop that are executed on the android device. The InternalCommand Bricks contain sub bricks that are executed within the structure: This could for example be a `DigitalRead` within an if statement. The sub bricks are limited by a *Start* and *End* brick (see example below). All Bricks contain parameters. A future improvement could be the implementation of AndroidBricks to encapsulate functionality and information of the android device like the pitch/yaw angles.

Arduino Command

- DigitalWrite
- DigitalRead
- AnalogWrite
- AnalogRead

Internal Command

- Variable
 - Can be used to store a value.
- If
 - Can be used to compare a variable to a value. E.g. *if x > 5*
- For
 - Standard “for loop”.



Example usage of the variable bricks. After the execution, the value of name is set to whatever was returned from the read command.

UI Bricks

For implementation of the user interface of the bricks, the main functionality was the possibility of changing position, background, and text depending on the subtype of brick. For that reason the use of RecyclerView was taking on consideration, as is recommend on Android developers documentation (<https://developer.android.com/guide/topics/ui/layout/recyclerview>)

The class that extends recyclerview.adapter is named ItemBrickAdapter and overrides the method onBindViewHolder to set the text and background of the brick viewholder. For the texts the values are binded by the name and

parameters of the brick and for the background the colors of res/values/colors file are used.

The ItemBrickAdapter is instantiated at the ListFragment class. The use of Fragments als was the best option for the list of bricks, letting the correct user interaction of scrolling, drag, etc.

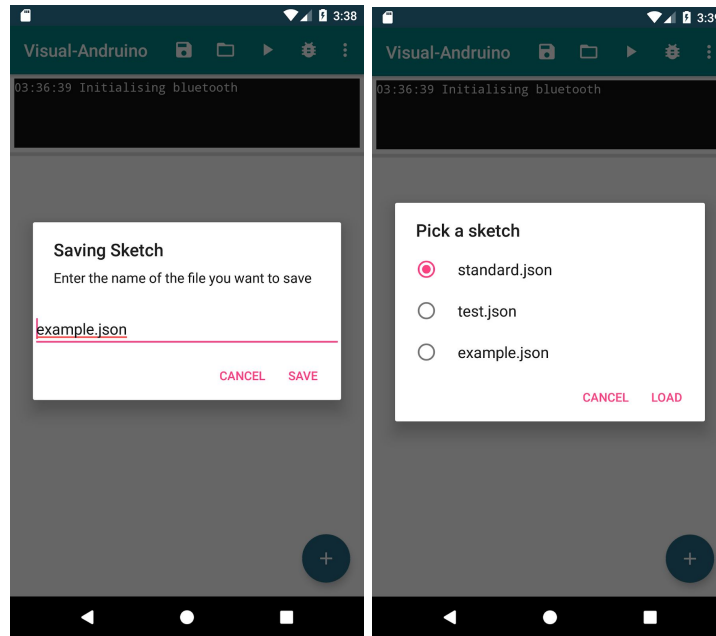
Final features

Loading/Saving

Sketches are stored as JSON files. An ArrayList of Bricks is transformed into a JSON Array (making use of the gson library) and saved in a file. The files are stored in the internal storage (in a special directory) of the device.

Loading/Saving opens a pop up menu. Here, the name of the file can be chosen (saving) or a list of available files is shown and the user is asked to pick one (loading). We have chosen this method since sketches can easily be exported and the transformation between a Java Array and a JSON Array is quick and relatively simple.

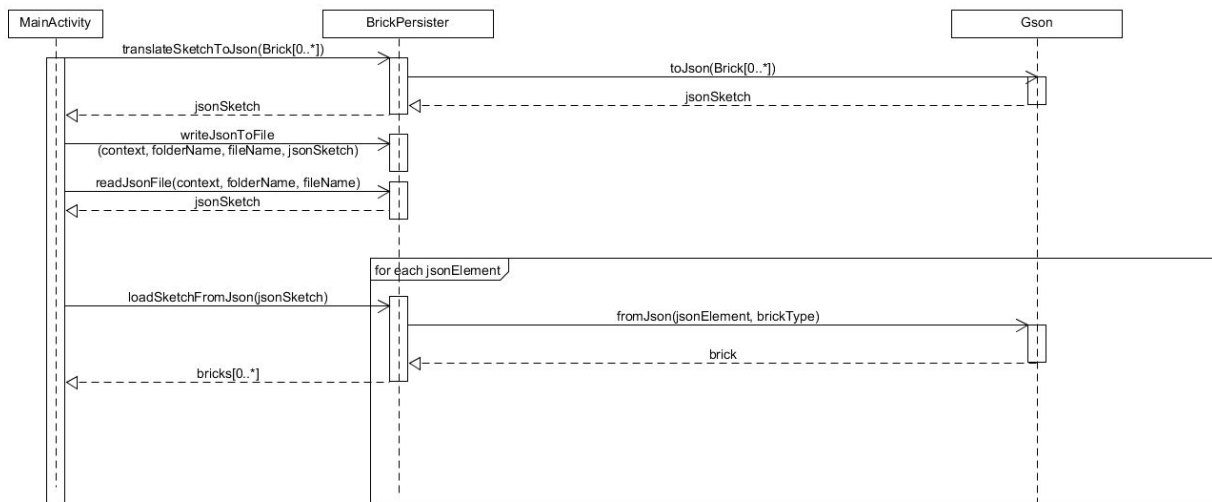
Problems faced: When turning the json array to an arraylist of bricks, we faced the problem that these Bricks had lost information concerning their actual class since all objects were simply transformed into the superclass that is Brick. To overcome this, we simply iterate over the json array manually, retrieve the information about the actual class from the json Object (using the *brickType* field) and transform it to the right subclass.



Additional Explanation

This sequence diagram explains how sketches can be persisted and retrieved. The MainActivity calls the BrickPersister that makes use of the Gson object for the transformation. Afterwards the json String can be written to a file.

The file content can be retrieved in the same way. To get the correct representation of the bricks again we iterate over the json array casting each jsonObject to the correct subtype of Brick (Arduino Command Brick, Internal Brick, Android Brick).

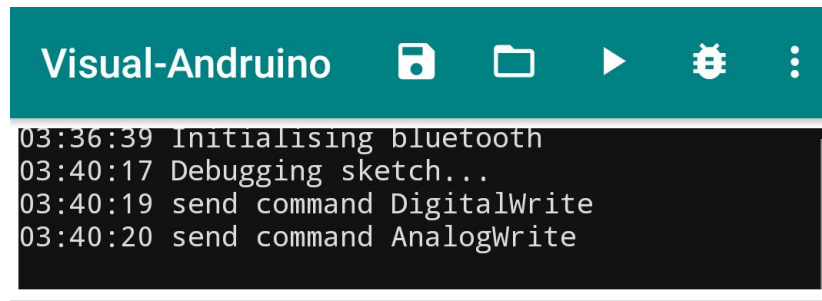


Sequence diagram: Highlights how sketches are saved and loaded.

Terminal

We added a terminal view that can be accessed from any part of the application for showing information on the current state of the application and the execution of the bricks. It is based on an open source library (<https://github.com/jraska/Console>). The terminal also shows:

- information on the status of the bluetooth connection.
- error messages.
- variables and their values that were set in the last execution of the sketch.

The image shows a screenshot of the Visual-Andruino application. At the top is a teal header bar with the text "Visual-Andruino" and several icons: a floppy disk, a folder, a play button, a gear, and a vertical ellipsis. Below the header is a black terminal window with white text showing the following log entries:

```
03:36:39 Initialising bluetooth
03:40:17 Debugging sketch...
03:40:19 send command DigitalWrite
03:40:20 send command AnalogWrite
```

Run Sketch

A sketch can be run by pressing the corresponding button. It is executed as quickly as possible.

The variables and their values that were set in the last run are shown in the terminal after the execution is finished.

When a sketch is run there are several steps:

- The UI Brick are turned into backend bricks: Their id value of the list is removed. We are left with a simple ArrayList of Bricks.
- *BrickExecutor.executeBlocks* is called in a new thread.
 - *BrickExecutor.executeBrick* is called for the first brick.
 - The logic of the Brick is executed. If it is a command Brick, the command is sent to the arduino. If it is an InternalBrick, *BrickExecutor.executeBrick* is called for its subBricks recursively.

- The executed brick is removed from the ArrayList and *BrickExecutor.executeBlocks* is called recursively with the shortened ArrayList.

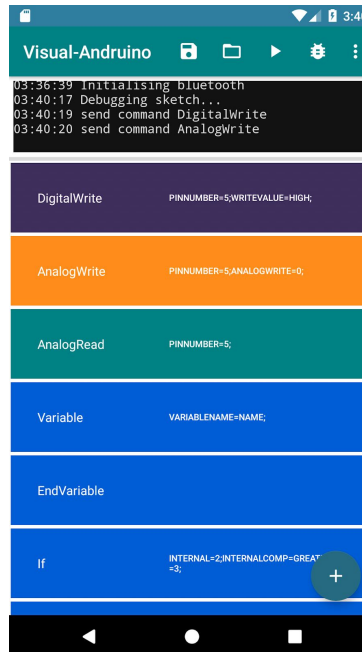
Problems faced: When waiting for the return values of a read CommandBrick the data from the arduino is received in a callback. The Executor will block until this data is received (or a timeout is triggered). Previously the callback as well as the execution were done on the same thread. The problem here is: The thread will block any execution, including the callback. Therefore the data is never received in the current execution. This was solved by starting the execution on a separate thread. This thread will block and the main thread will execute the callback. After this, the execution of the first thread resumes. This also solves the problem of the UI freezing during execution.

Debug Sketch

A sketch can be debugged by pressing the corresponding button.

In this case the sketch is executed brick by brick like a normal run but there is a delay between the execution of the bricks that allow a user to follow what is currently going on. The brick that is executed is highlighted by turning it green. The variables and their values that were set in the last run are shown in the terminal after the execution is finished. Under the hood debugging works as follows: Say we are inside the executeBrick method, currently executing a Brick. Before actually starting with any logic we update the brick status.

In this initial stage the brick status is set to *Starting*. Then, the UI is updated which will highlight the current brick. After the execution of the logic the brick Status is set to *Waiting* again and the steps are repeated for the next bricks.



Packages Overview

it.unibz.mobile.visualandruino

This package contains the main parts of the application such as the main activity and the fragments.

it.unibz.mobile.visualandruino.models

This package contains all entities such as the different Brick classes.

it.unibz.mobile.visualandruino.models.enums

Contains enums that are used as entities: for example the brick status.

it.unibz.mobile.visualandruino.utils

Contains utils that are used from all parts of the application.

Tests

Different Tests were created. This was especially useful because backend functionality was sometimes developed ahead of the corresponding UI components. In those cases tests were written to ensure that the functionality works as intended.

Regular Unit Test

Normal unit tests were used to test the behavior of regular classes and methods. We kept our methods and classes as cohesive as possible.

Mocked Unit tests

In order to test the *BrickExecutor* we needed to use mocked tests. Since we are not actually sending commands to the arduino when testing, we mocked the *BrickExecutor* by passing a mocked instance of the *BrickExecutor* class itself to the *BrickExecutor.executeInternal* method. This way we can verify if and how often the *executeBlocks* (for the internalBricks) method of this instance is called. This means we were able to test if and how often the sub bricks were called. This is done to check whether the if and while/for bricks work properly.

```
// create mock
BrickExecutor beMock = mock(BrickExecutor.class);
BrickExecutor be = new BrickExecutor();
be.executeInternal(item, null, beMock, false);
verify(beMock, times(2)).executeBlocks(item.getSubBricks(),
null, false);
```

Instrumented Tests

In order to test the loading and saving functionality we needed the context of a running application. We used instrumented tests for this that run on an actual or emulated device.

Additional Problems faced:

- Bricks were removed when rotating the device. This could be solved by saving the bricks Array in a Bundle and resetting it in the *onCreate* Method of the activity.
 - This caused a problem when stopping the app manually since the Array could not be serialized properly. We solved this by making use of the previously implemented persistence logic. The ArrayList was translated into JSON and saved as a string. The JSON was turned into a Java ArrayList again *onCreate*.
- To use a HC-06 as a backup device we implemented the settings menu to change the name of the desired bluetooth device. This can be done “on the fly”.
- The app continuously tries to reconnect to a found device if no connection could be established. If no device was found the user is asked to change the device in the settings menu to avoid unnecessary reconnects.