Android ADK application for STM32
RS232 client for Android

Beretta Michele
Matr. 782936, (michele3.beretta@mail.polimi.it)

Report for the master course of Embedded Systems
Reviser: PhD. Patrick Bellasi (bellasi@elet.polimi.it)

Received: September, 25 2013

Abstract

The goal of this project is to design and implement an Android accessory that enables to an Android device to communicate through the standard RS232 serial port. The project consist of two components: the first is the ADK accessory, based on the evaluation board STM32F4DISCOVERY [8], and the second is an Android application used to communicate with the accessory.

1 Introduction

1.1 What is an accessory

The ADK (Accessory Development Kit) is a reference implementation for building accessories for Android devices. An accessory can be any external hardware device that provides some kind of functionality to the Android system (audio dock station, weather station, ...) and it is connected through USB port (or Bluetooth). The connected accessory acts as the USB host and the Android device acts as a USB device. The reason why the accessory is the USB host is manly because not all the Android devices support the USB host mode.
Between the accessory and the Android device is established a point-to-point packet-based communication channel through which is possible to communicate.
The protocol that regulates the communication is called AOA and its complete specification is freely available online [5]. Android provides a simple API for interfacing with an accessory, communication is handled using only two streams as is the case for network programming.
The accessory can be any hardware device capable of communicating using the AOA protocol, so its implementation is strongly tied to the particular hardware used.
There are implementations of different types of accessory designed for different platforms. For example there is a particular version of the popular Arduino board specifically designed to easily create new accessories [3].

1.2 RS232 accessory

The project aims to the development of an accessory that enables to an Android device to connect and communicate using other interfaces in addition to those available on the device.
Most of the Android devices are smartphones or tablets, the number of wired interfaces available on those devices is very limited. There is hardly any device with any other interface in addition to USB, headphone jack and some kind of video output.
It would be interesting to expand the connectivity of these smartphones using some kind of external device, something like the docking stations used for notebooks.
The accessory that has been developed permits the connection and the bidirectional communication with a standard RS232 serial interface any Android device. The possibility to have a serial port directly on an Android device can be useful to connect to some particular peripheral that still use that type of interface.

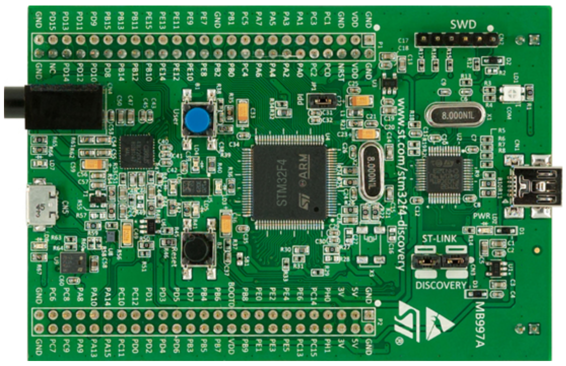


Figure 1: STM32F4DISCOVERY evaluation board

For the implementation of this type of accessory is necessary to develop a dedicated hardware and software. The hardware of the accessory must be equipped with a serial interface and a USB host

interface. The choice was to use the evaluation board STM32F4DISCOVERY (Figure 1). This board is equipped with the STM32F407VGT6 microcontroller, it is a 32-bit ARM Cortex-M4 CPU running at 168MHz with 1 MB Flash and 192 KB RAM. This microcontroller provides a lot of interfaces among which as many as 6 serial ports (configurable in different modes of operation and transfer speed) and a USB host port.
In addition to the accessory it is also necessary to develop an application that allows an easy management of the device and of course provide bidirectional communication through the serial port.
The rest of the report is organized as follows: section 2 briefly describes the existing implementation of the project used, section 3 describes in detail the implementation of the accessory, section 4 describes in detail the implementation of the Android application, section 5 explains how to compile and run the various components, section 6 shows a use case of the accessory, section 7 analyzes the possible future works for this project.

2 Existing accessory implementation

The project was realized as an extension of an accessory already implemented for the evaluation board. The starting point was the implementation of an existing accessory for the evaluation board STM32F4DISCOVERY. With that accessory it was possibile to turn on or off a LED of the evaluation board and to read the status of a button.
The source code is divided into two projects, one for the accessory STM32F4-ADK [2] and one for the Android application HelloADK [1]. Both projects have the source code freely available on Github.

3 RS232 accessory

The project STM32F4-ADK implements a bare metal system that offers a simple way to communicate with an Android device via AOA protocol.
The library's interface used to communicate is the following

```
void USBH_ADK_Init(char* manufacture,
        char* model, char* description,
        char* version, char* uri, char* serial);

USBH_Status USBH_ADK_write(USB_OTG_CORE_HANDLE *pdev,
        uint8_t *buff, uint16_t len);

uint16_t USBH_ADK_read(USB_OTG_CORE_HANDLE *pdev,
        uint8_t *buff, uint16_t len);

ADK_State USBH_ADK_getStatus(void);
```

USBH_ADK_Init initializes the communication between accessory and Android device, during this phase it is established the communication channel and are exchanged information of the accessory to the Android device (serial number, description, vendor name, optional url of a website. . .)

USBH_ADK_Read used to read packets sent from the Android device

USBH_ADK_Write sends a packet to the Android device

USBH_ADK_getStatus is an utility function that is used to determine the status of communication

This implementation has been kept and has been used as the basis for the project because it was simple and worked well.
Only later were identified problems with that library which have somehow affected the project.
The most severe problem is a bug in the function `USBH_ADK_Read`, this bug consists in the fact that the function occasionally returns a packet that has been already received previously.
This bug does not compromises the operation of the application because the multiple reception of a packet containing the command "turn on the LED" or "turn off the LED" has no side effects.

3.1 Serial port abstraction layer

To develop a system that can be easily ported to another platform without changing too much code the first thing to do is to define an abstraction layer for the various serial ports offered by the microcontroller.
The way in which the serial port hardware is controlled strongly depends on the hardware used, each vendor typically provides within the BSP of the microcontroller also the code required to use the serial port. Being bound, in this case, to the libraries provided by ST would have affected the portability of the system to other platforms.
First of all has been defined the way how to represent a serial port. A serial port is characterized by a series of parameters: *baud rate*, *stop bits*, *parity type* and *hardware flow control* and a *number* that uniquely identifies it.
These characteristics are summarized in the following data structure

```
typedef struct _SerialPortConfig {
        uint32_t baudRate;
        uint8_t number;
        uint8_t stopBits;
        uint8_t parity;
        uint8_t hardwareFlowControl;
}SerialPortConfig;
```

Each field of the data structure (excluding port number) is used to save the various options supported by *bit fields*.
For example, for the field `stopBits` and `hardwareFlowControl` are defined the following possible values

```
enum StopBits {
        BITS_0_5 = 0x1,
        BITS_1_0 = 0x2,
        BITS_1_5 = 0x4,
        BITS_2_0 = 0x8
};

enum HardwareFlowControl {
    NO_HFC = 0x1,
    RTS = 0x2,
    CTS = 0x4,
    RTS_CTS = 0x8
};
```

Also the other fields have preset values, the definitions of all the these values are contained in the file `inc/serialport_config.h`.
With this data representation a `SerialPortConfig` object can express all the supported configurations of a serial port. This is done by setting or clearing every single bit to express whether or not the port supports a particular configuration. For example we can expose a port with only a possible configuration, this could be the case when to the serial port there is attached a device so that its configuration cannot be changed. A port can be exposed with all, or a part, of its capabilities so that it can be used with different devices.
Here there is an example of port that supports only a particular configuration

```
SerialPortConfig availablePorts[] =
{
    {
        .number = 0,
        .baudRate = BR_9600,
        .stopBits = BITS_1_0,
        .parity = NO_PARITY,
        .hardwareFlowControl = NO_HFC
    }
};
```

While this is an example of a port that can used be in different modes

```
SerialPortConfig availablePorts[] =
{
    {
        .number = 1,
        .baudRate = BR_19200 | BR_38400 | BR_57600,
        .stopBits = BITS_1_0 | BITS_2_0,
        .parity = EVEN_PARITY | ODD_PARITY,
        .hardwareFlowControl = RTS | CTS
    },
};
```

This way of representing a serial port is completely independent from the microcontroller.
The abstraction layer for the use of the serial port is defined by the following interface

```
extern SerialPortConfig activeConfig;
int getPortsConfig(SerialPortConfig **portConfig);
int openPort(SerialPortConfig* config);
int closePort();
int sendChar(uint8_t portNumber, uint8_t data);

void dataReceived(uint16_t data, int serialPort);
```

`activeConfig` variable containing the active serial port configuration, if no serial port is opened this number is set to the constant NO_SERIAL_PORT_ACTIVE

`getPortsConfig` initializes an array with all the available serial ports with their configurations, each field can have multiple flags set at the same time indicating that more than one mode is supported

`openPort` opens communication with a serial port specifying a particular configuration, in this case, the configuration must have passed only a flag set for each field

`closePort` closes the communication with the serial port currently active, if no port is opened this function does nothing

`sendChar` sends a single character on the serial port currently active, if no port is opened this function does nothing

`dataReceived` is an already implemented method that must be called when data is received on the serial port

An implementation of this interface is contained in the file `src/uart_hardware_stm32.c`, this implementation employs the libraries provided by ST for the use of the serial port.

3.2 Communication protocol

The communication channel between the accessory and the Android device is used for both to exchange data passing on the serial port and to exchange control commands, therefore it is necessary to define a communication protocol to differentiate the two types of flows.
The approach used is to create a small communication protocol based on different packets that regulates all the possible operations between the two devices.
This is the structure of the packet used

header	payload

The maximum length of the packet is 64 bytes, this is due to a limitation of the AOA protocol.4 bytes are used for the header and up to 60 bytes for the payload. The structure of the header is the following

type	length	id	reserved

type 1byte, identifies the type of the packet which can be one value of this enumeration

```
enum PacketType {
        REQUEST_PORT = 1,
        AVAILABLE_PORT = 2,
        OPEN_PORT = 3,
        CLOSE_PORT = 4,
        SEND_DATA = 8,
        RECEIVE_DATA = 9
};
```

length 1 byte, length of the payload of the packet

id 1byte, the `id` was introduced because of the strange behavior of the accessory described before and it is used to recognize duplicate packets. Each package is uniquely identified by this id number, in this way it is possible to determine if a received packet is new or is a duplicate of the previous

reserved this field is not used and is fixed to `0xFE`. The main reason is to align to 32 bits the structure of the packet

Here there are the details of all the packet types

REQUEST_PORT Accessory ← Android
Asks for all the serial ports available

`header`

AVAILABLE_PORT Accessory → Android
List of serial ports with their available configurations

`header`	`SerialPortConfig1`	`...`

OPEN_PORT Accessory ← Android
Opens a serial port with a specific configuration

`header`	`SerialPortConfig`

CLOSE_PORT Accessory ← Android
Closes the serial port with the specified number

`header`	`serialPortNumber`

SEND_DATA Accessory ← Android
Packet of data from the application to the accessory

`header`	`data`

RECEIVE_DATA Accessory → Android
Packet of data from the accessory to the application

`header`	`data`

With this simple protocol the Android device is capable of connecting to the accessory, determine what are the available serial ports with their various configuration, initiate the connection with a serial port and then communicate.

3.3 Protocol implementation

The implementation of the protocol described above handles incoming packets and sends packets to the Android device when necessary. The code has been split into two separete manager: *PacketManager* and *ProtocolManager*.
The *Packet Manager* is responsible of interpreting the incoming packets and create outgoing packets. That manager provides a number of methods to create each type of packet defined in the protocol (es. `createAvailablePortPacket`, `createSendDataPacket`, ...), and another set of function that extracts the data from the various types of packets (es. `handleReceivedData`, `handleOpenPort`, ...).
The *Protocol Manager* is responsible to perform the actions associated with each particular type of packet (eg. upon receipt of package `OpenPort` must match the opening of the serial port) and send packets when certain events occur (eg. when data is received on the serial port must then be sent to your Android device).
The simple interface of the protocol handler is shown below

```c
extern int packetToSend;
void packetReceived(uint8_t* packet, uint16_t length);
```

packetToSend is a boolean variable that indicates whether a packet should be sent, this variable is set when there is the need to send a packet. The main loop of the application periodically reads the value of this variable and if it is *true* sends a packet

packetReceived is an already implemented method that must be called when a packet is received from the Android application

`src/packet_manager.c` and `src/protocol_manager.c` contains the implementation of the two managers.

3.4 Components interaction

The application components and their interactions are summarized in figure 2.

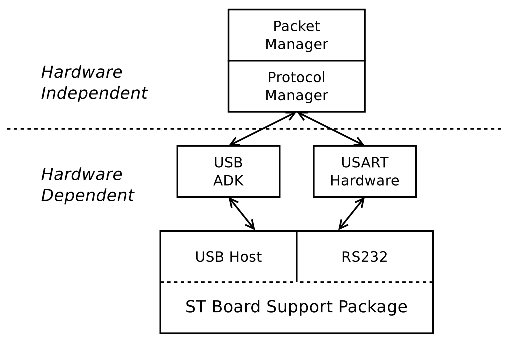


Figure 2: RS232 accessory components

The code is divided mainly into two parts: a hardware dependent part and an hardware independent part. The former includes all the component that interact directly with the hardware peripherals, which are: the USB Host stack, the USB ADK implementation and the Serial port abstraction layer.
The latter is the is the code that implements the functionality of the accessory, which are: protocol manager and packet manager.

4 Android application

The application HelloADK was a very simple program, which purpose was to control a LED and read the status of a button. In this case it has not been possible to reuse the

source code of the application because of its bad design. It was evidently created just for example purposes. All the source code is contained within a single activity which makes it a single not-modular entity, due to this design choice the application had a series of problems.
The most important was the fact of not being able to maintain the connection with the accessory when the device is put into stand-by mode or when the user changes the application.
Because of Android's design it is not possible to guarantee a persistent connection using only one activity, it is necessary to use a separate entity to manage the communication. This separate entity has to be a Service.

4.1 Application structure

The application was designed from scratch, its structure is shown in figure 3

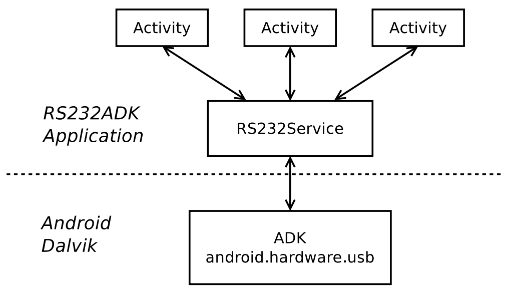


Figure 3: Android application components

The main component is the `RS232Service` and it takes care of handling the accessory, the rest of the application is a set of activities used by the user to interact with the accessory.

4.2 RS232 Service

The service handles the communication with the accessory managing it exclusively, all activities must use the service to communicate with the accessory. By structuring the application in this way all the problems affecting the application HelloADK are resolved and the system is much more modular and more easily expandable.
Being always active, the service can keep the connection with the accessory even when the Android system is put into standby mode. It is also possible to use the accessory by more than one activity at the same time.
The service communicates with the accessory using the protocol described in section 3.2, while uses a different method to communicate with the activities.
There are several possible methods of communication between a service and an activity, each of them fits specific situations, the one used consists of using a Messenger object between each activity.
This method allows an activity to exchange Message objects with the service. The choice of this method was almost inevitable because it is necessary to establish a bidirectional communication channel between activity and service and the use of a Messenger to exchange Message is the only method that provides bidirectional communication. The need for bi-directionality is caused by the fact that the service may need to contact the activity in order to notify events.

Since every activity of the application needs to communicate with the service it has been created an utility class that allows the establishment of the connection to the service and the subsequent communication.

```
// instantiate a Service Manager object
new ServiceManager(this, Service.class, new Handler() {
   @Override
   public void handleMessage(Message msg) {
      switch (msg.what) {
         case Service.MSG_ID:
            // handle message
            break;

         default:
            super.handleMessage(msg);
      }
   }
});

// send a message to the service
Message msg = new Message();
...
mServiceManager.send(msg);
```

Through the `ServiceManager` class it is possible to receive messages from the service by overriding the `handleMessage` method, it is also possible to send messages to the service using the method `send`. This system to communicate with the service allows a quick and easy application development.

The various activity just need to connect to the accessory using a `ServiceManager` and: map the actions of the user into messages to send to the accessory, interpret and then somehow display the messages sent from the accessory.

The messages exchanged between service and activity aren't the same of the communication protocol, but they are very similar as showed below

```
// id of messages that can be only received
// by the service
public static final int MSG_CONNECT = 3;
public static final int MSG_DISCONNECT = 4;

public static final int MSG_MESSAGE_RECEIVED = 6;

public static final int MSG_OPEN_PORT = 7;
public static final int MSG_CLOSE_PORT = 8;

public static final int MSG_GET_PORTS_CONFIGS = 10;

// id of messages that can be only sent by the service
public static final int MSG_CONNECTION_OK = 1;
public static final int MSG_CONNECTION_FAILED = 2;

public static final int MSG_SEND_MESSAGE = 5;

public static final int MSG_PORTS_CONFIGS = 9;
```

4.3 Activities

The application was designed with the idea of creating a simple graphical interface to use the accessory. It was therefore chosen to use a few simple activities to make the application easy to use.
The structure, in terms of activity and the various interaction, of the application is shown in figure 4.

RS232ADK is the main activity, displays the connection status with the accessory and show its information. From here it is possible to connect or disconnect to the accessory

SelectPort is the activity that displays the serial ports that can be used. From here it is possible to select a port and change or update its configuration. It is also possible to open the communication with a port that has been configured

ConfigPort is the activity that allows the configuration of a serial port. The accessory can expose different options for the various parameters of the serial port, here those parameters can be selected

CommunicationActivity is the activity used to communicate via the selected serial port. It is possible to send messages by entering the text inside the line edit, and then press the button *Send*. Received messages will be displayed in the text box, to differentiate the messages received from those sent it is used in a different color

5 Installation

For the accessory setup is necessary to have the following hardware: an Android device, an evaluation board STM32F4DISCOVERY, a *microUSB-A* to *microUSB-B* cable. That cable may not be easy to find because of the fact that the type of connection is quite strange. However it is possible to use two cables to build an equivalent one, this is done by connecting together a *microUSB-B* $< - >$*USB* cable with a *microUSB OTG* cable.
Once procured the necessary hardware must compile and install the firmware and the Android application.

5.1 Accessory

In order to build the firmware to flash on the microcontroller of the evaluation board it is necessary to have the appropriate toolchain. The toolchain must be a GNU ARM one for bare metal systems, you can use an existing precompiled one or you can compile your own. This project has been compiled with the precompiled toolchaing of Ubuntu for bare metal ARM systems [6].
The source has a GNU `Makefile` that automates the compilation process, it is sufficient to run the `make` command and the generated firmware will be saved in the file `build/FLASH_RUN/project.bin`.
To flash the firmware on the evaluation board the `Makefile` has a particular target names `program` that allows programming with STLINKv2 or OpenOCD.

5.2 Android application

The Android application is a normal application and is built using the tools provided by the SDK of Android itself. The only note is that the compilation target must be one version of the Google APIs, this is because the package `android.hardware.usb` (or `com.android.future.usb`) that contains the API for interfacing with the accessory ADK is present only in the Google APIs and not in the standard API.
With the Android command line tools building the application from source is done with these commands

```
$ android update project
    --subprojects
    --target "Google Inc.:Google APIs:14"
    --path .
$ ant release
```

It will be created the `bin` folder which contains the generated program ready to be installed on any Android device.

6 Usage example

In the following section there is a real example of use of the application.
To use the application you must connect the accessory to a serial port on which you can read and write in order to control the data that is exchanged.
It was used a serial USB adapter [4] to be able to use the serial port directly from a PC.

6.1 Connection

By launching the program, the screen that is presented is shown in figure 5, when the application starts it will not connect to the accessory.

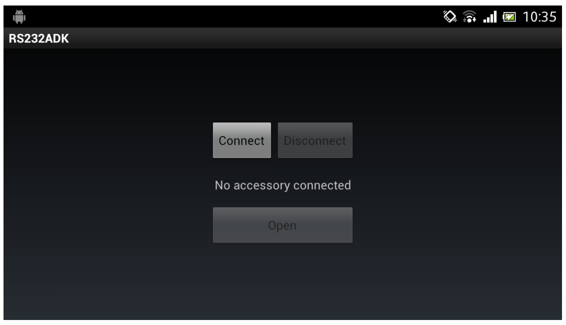


Figure 5: Application start

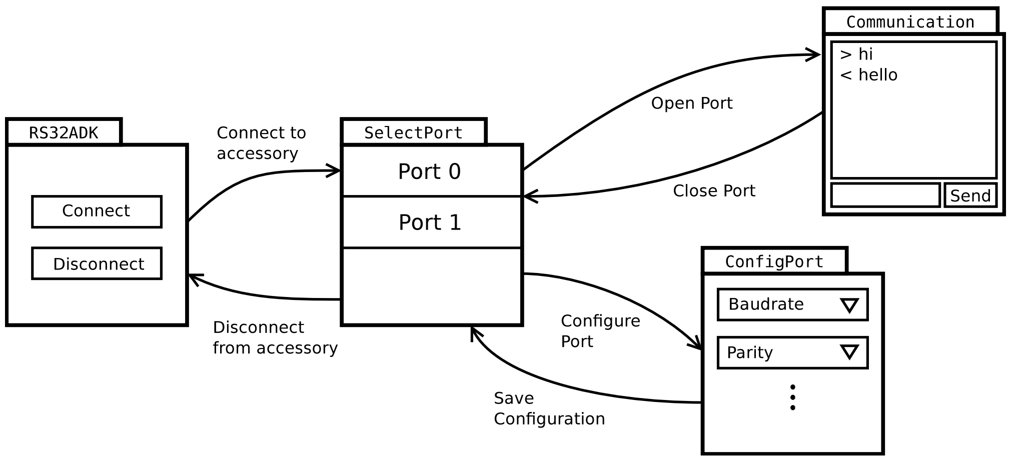
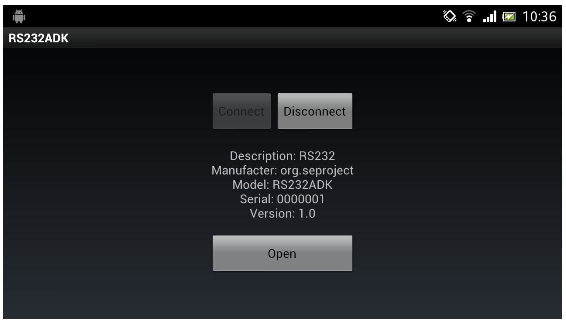


Figure 4: Activity interaction

By pressing the *Connect* button the application tries to connect to the accessory, if this is actually connected an Android notification appears, asking for permission to use the accessory. This permit is requested each time you want to use the accessory, however there is the possibility to remind the choice in the application settings.

If the connection is successful the service for the management accessory is started. The first action taken by the service is to read the information from the accessory as shown in figure 6.

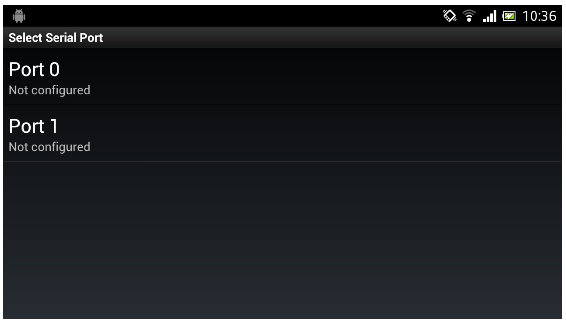


Description: RS232
Manufacter: org.seproject
Model: RS232ADK
Serial: 0000001
Version: 1.0

Figure 6: Accessory connected

6.2 Ports configurations

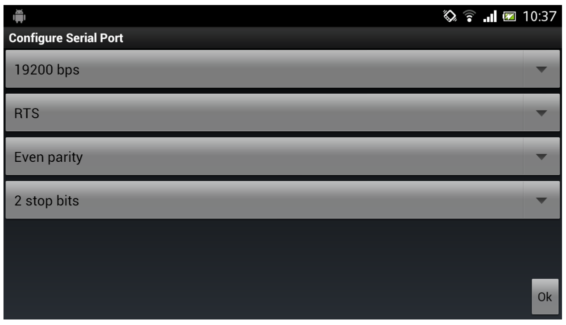
Once the accessory is connected you can switch to the port configuration activity by pressing on the *Open* button. Initially these ports do not have any configuration as shown in figure 7.



Port 0
Not configured
Port 1
Not configured

Figure 7: List of available ports

From here you can configure individual ports using the interface in figure 8, each entry contains all the configurations supported by the particular selected port. As mentioned earlier, these can have only one or many possible configuration for each field, this setting is contained in the firmware of the accessory.

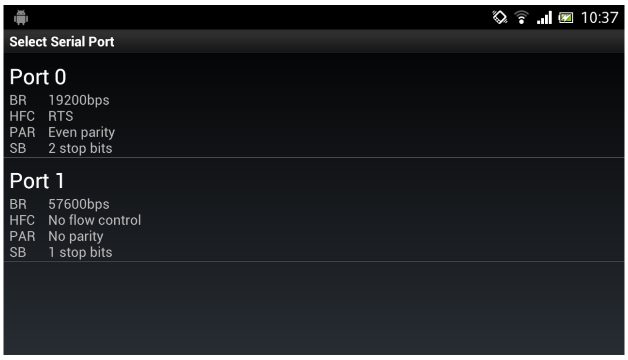


19200 bps
RTS
Even parity
2 stop bits

Figure 8: Port configuration

6.3 Communication

Once that the port has been configured it is sufficient to click on it to open it as shown in figure 9. The application passes to the communication activity that shows the data exchanged through the serial port. A different color is udes to distinguish sent messages from received messages , green for sent messages, gray for received messages.



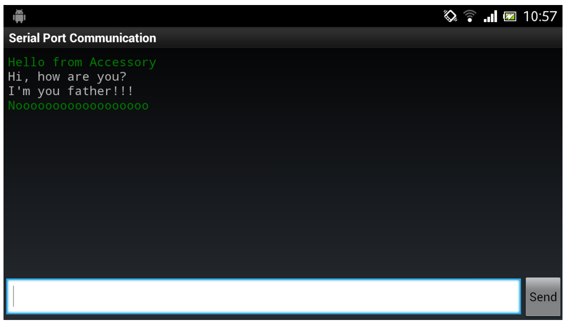
```
Select Serial Port

Port 0
BR   19200bps
HFC  RTS
PAR  Even parity
SB   2 stop bits

Port 1
BR   57600bps
HFC  No flow control
PAR  No parity
SB   1 stop bits
```

Figure 9: Ports configurated

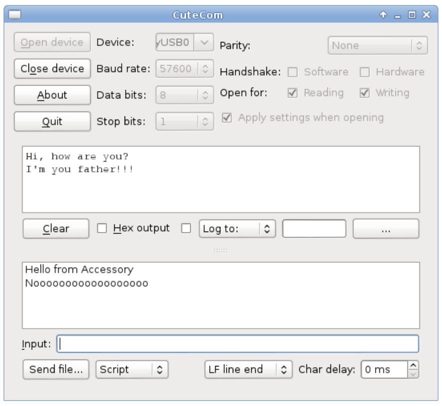
Figure 10 shows a short (and nice) exchange of text strings between accessory and application. There is a line edit where to write the message to sent, messages are sent only when the button *Send* is pressed.



```
Serial Port Communication
Hello from Accessory
Hi, how are you?
I'm you father!!!
Noooooooooooooooooo
```

Figure 10: Example of communication

In this example the program used to send and receive through the serial port is *Cutecom* [7] (a graphic serial terminal program for Linux, it is equivalent to Hyperterminal on Windows). Figure 11 shows a screenshot of that application used to send the messages showed in Figure 10.



```
CuteCom
Device: yUSB0        Parity: None
Baud rate: 57600     Handshake: ☐ Software ☐ Hardware
Data bits: 8         Open for: ☑ Reading ☑ Writing
Stop bits: 1         ☑ Apply settings when opening

Hi, how are you?
I'm you father!!!

Hello from Accessory
Noooooooooooooooooo

Input:
LF line end   Char delay: 0 ms
```

Figure 11: Cutecom

7 Future works

The project could be extended by adding other features.
With regard to the accessory there are mainly two major improvements: *porting to other hardware platforms* and *integration with an RTOS*. In this project it has been used the evaluation board STM32F4DISCOVERY, it could have been used other types of microcontrollers (AVR, Microchip ...) or other STM evaluation boards to implement the accessory. The code has been written so to abstract as much as possible the hardware of the microcontroller used, there should be no particular problems in a porting to a different platform.
The other main improvement is to switch form a bare metal system to a RTOS, this change requires the writing of a library for a specific RTOS such as FreeRTOS or ChibiOS. This would make the porting from an hardware to another even more simpler because the RTOS provides a lot of features of hardware abstraction.
From the point of view of the Android application there are several possible improvements.
For the time being the application only allows the exchange of printable characters, an extension could the integration of a hex viewer to "read" also generic stream of data, or the possibility to save the conversations to a file, or a better mechanism to select the configuration of a serial port... and other changes always related to the graphical interface.
This accessory adds a sort of virtual serial port to the Android device. There are other types of interfaces that could be added, for example an interface for the CAN bus or for the I^2C bus. Theoretically there are no particular constraints on the type of interface that the accessory can expose to the Android device. The main limitation is imposed by the computation power offered by the processor of the microcotroller and the speed limitation imposed by the AOA protocol.

References

[1] Yuuichi Akagawa. Helloadk for stm32f4-adk.

[2] Yuuichi Akagawa. Stm32f4-discovery with android adk.

[3] Arduino. Arduino adk.

[4] FTDI. Usb ttl serial cable ttl-232r-3v3-we.

[5] Google Inc. Aoa protocol.

[6] Launchpad. Gcc arm embedded.

[7] Alexander Neundorf. Cutecom.

[8] ST. Discovery kit for stm32 f4 series - with stm32f407 mcu.