POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: Informatyka (INF)
SPECJALNOŚĆ: Internet Engineering (INE)

PRACA DYPLOMOWA MAGISTERSKA

Smartfon z systemem Android
jako wysokopoziomowy sterownik robota

Android smartphone
as a high-level controller of a robot

AUTOR:
Michał Kowalski

PROWADZĄCY PRACĘ:
dr inż. Marek Woda

OCENA PRACY:

WROCŁAW 2016

Contents

List of Figures

Chapter 1

Introduction

1.1 Description of problem

Nowadays, popularity of robots is on the raise. It's not hard to built a simple one, and Internet is full of tutorials how to build them. They are built using specially programmed microcontrollers (MCUs), and simplest ones even without any. However, MCUs have some limitations:

1. They have limited memory and computational capability.
2. They require a lot of low-level configuration and programming.
3. Each MCU model requires (at least) slightly different configuration.
4. It's hard to look for help (e.g. on Stack Overflow, [7]) for specific MCU.

Therefore, usage of Android smartphones as high-level controllers, sending commands to MCU as low-level one, should be worth considering, because of:

1. Lot of memory and powerful processors.
2. Many built-in sensors.
3. Many ways to communicate with surroundings, especially - with MCU.
4. Popularity of Android platform:
    - tutorials,
    - devices,
    - solutions on Stack Overflow,
    - external libraries.
5. Compatibility between smartphones and Android versions.
6. High-level programming and reduced low-level configuration.

1.2 Goal of a project

Goal of this project is to check, if Android smartphone:

- can communicate with microcontroller,
- can extend functionality of robots using its built-in sensors.

Found solutions should be analyzed with attention to:

- compatibility,
- performance,
- difficulty of implementation.

Two Android smartphones (with different performance and Android version) will be used: Sony Ericsson Xperia Neo and Motorola Moto G LTE. Communication with MCU will be realized through USB cable. Face detection using built-in camera will be used as an example of extending MCU's capabilities - it requires both computing power and sensors not available in MCU, and there exists several ways to implement this.

As MCU, a Freescale FRDM KL26Z will be used. [1] is a blog dedicated to development on Freescale platform (mostly KL25Z, predecessor of KL26Z), and even contains an article how to built a mobile robot on that platform (img. 1.1). It has articles how to use most of those MCUs features, however from smartphone's point of view, only communication using USB port (KL25Z and KL26Z have two of them) is required.

Because of already working solutions on Freescale's MCUs (described in State-of-Art section), there is no need to build an actual, working robot - it's proven, that those MCUs can be used as a controlling unit of a robot. Therefore, this thesis focuses only on making use of smartphone for detecting position of face (as in robot on img. 1.5, but with Android instead of PC) and sending text command to MCU (also as in 1.5, but on Android, not PC).

1.3 Content of thesis

This thesis contains:

1. Introduction - this chapter.
2. Platforms - chapter describing MCU, Android, and used equipment.
3. Communication - chapter about communication between smartphone and MCU.
4. Sensors - chapter about extending robot's capabilities by using one of smartphone's sensors - camera.
5. Summary - chapter with conclusions, whether Android smartphone can be used as high-level controller.
6. Bibliography.

All code used in this thesis can be found on disc attached to archive version, or in git repository on Github: https://github.com/Koval92/MastersThesis

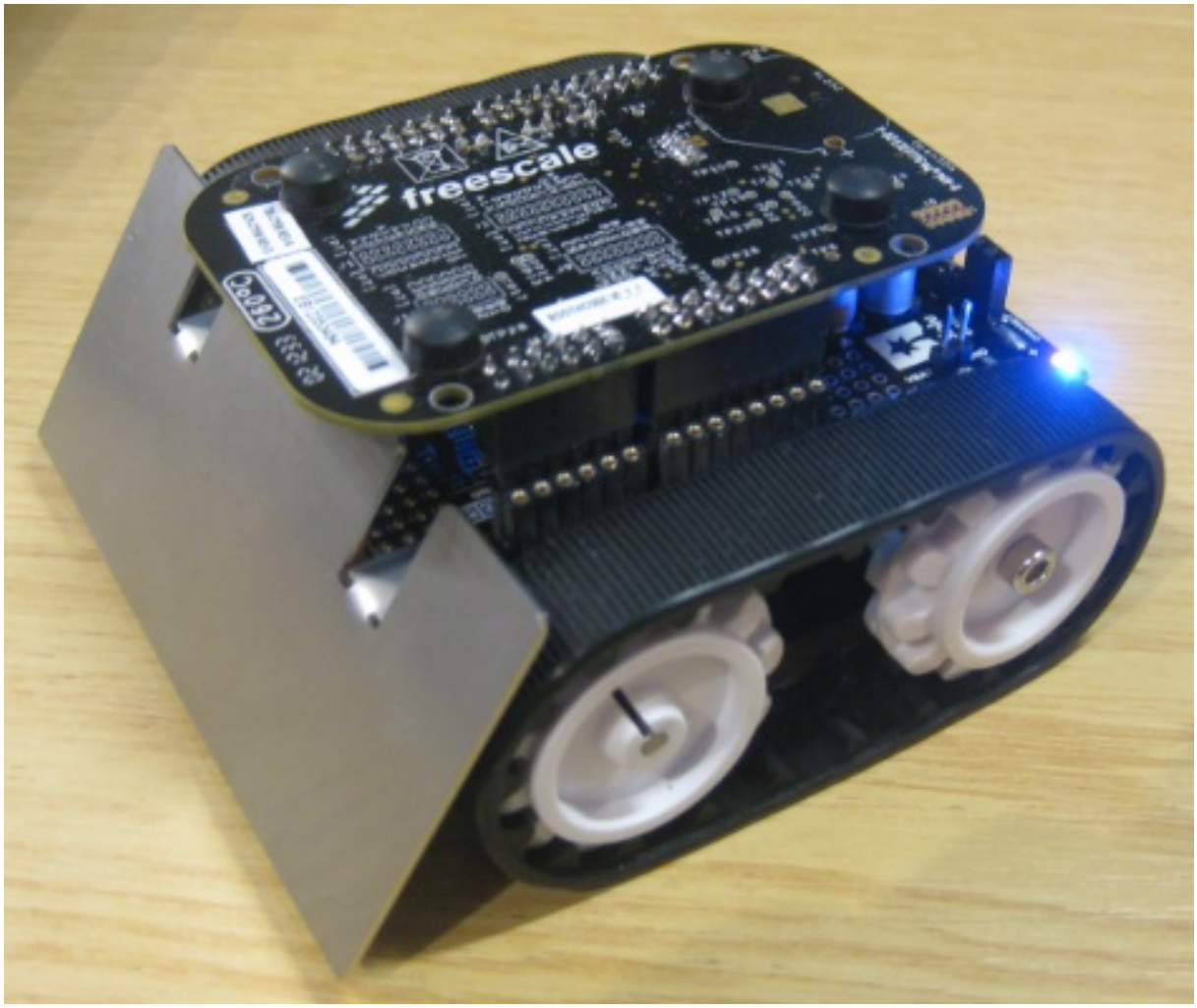


Figure 1.1: FRDM Zumo Robot, [1]

1.4 State of Art

State of Art can be divided into three different areas:

1. Robots working on Freescale's MCU.
2. Robots controlled by mobile phones.
3. "Followers".

1.4.1 Robots working on Freescale's MCU

Good example of such robot can be Freedom Zumo Robot, described in one of posts on [1] (img. 1.1). It's built on FRDM KL25Z, and Zumo Robot Kit for Arduino - KL25Z (and KL26Z probably too) is compatible with it.

Another example could be a robot built for one of previous projects - it will be described later. Therefore, it's proven, that this MCU can be used for building (at least simple) robots.

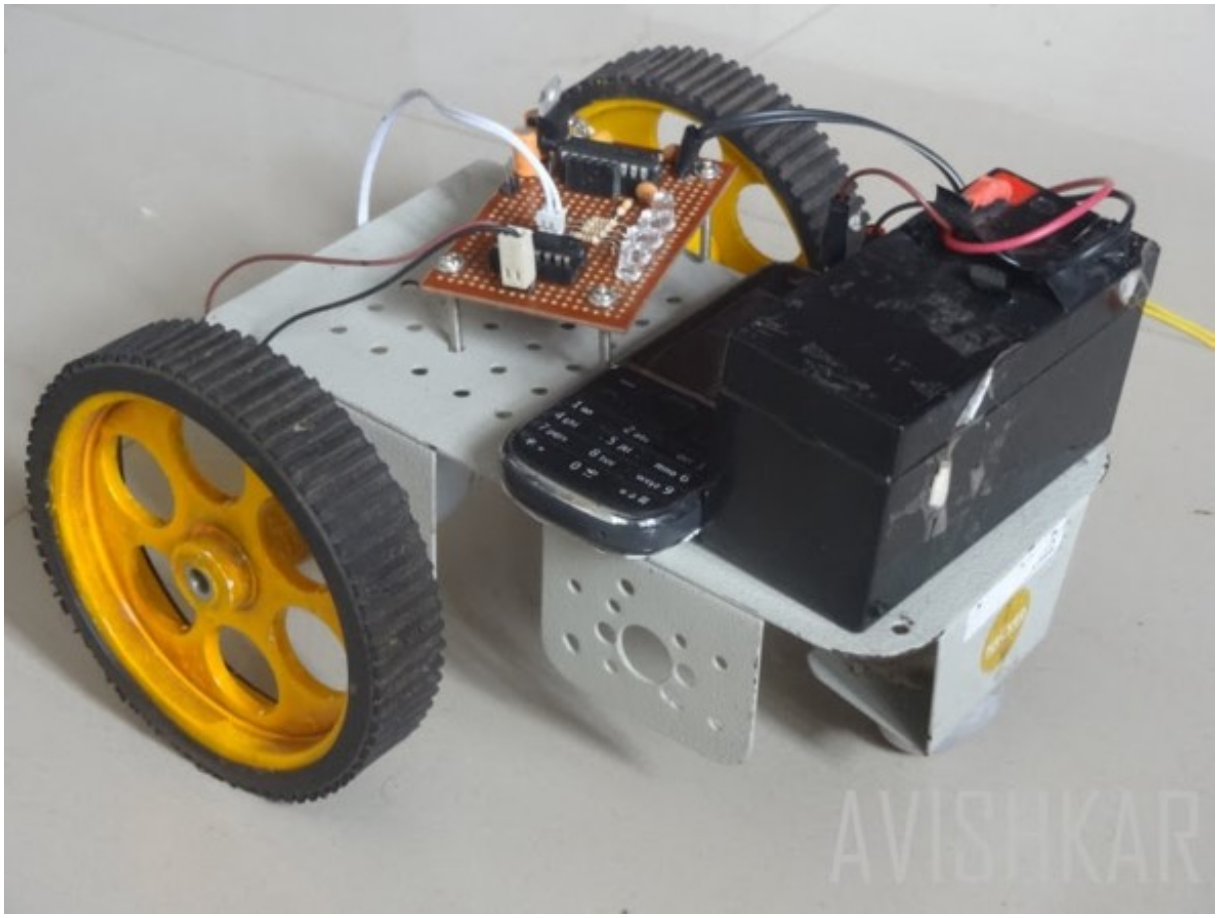


Figure 1.2: Mobile Controlled Robot by Ganeev Singh, [2]

1.4.2 Robots controlled by mobile phones

Next category are robots controlled by mobile phones, like ones shown on photos 1.2, 1.3 and 1.4. It seems, that all of them all remotely-controlled using another phone - clicking on button during phone call with receiver generates a sound, which is received and transformed into signal in headphones' port. None of examples have any (additional) logic in phone.

Some examples of robots controlled by additional program on smartphones still exist, but most of them either are poorly documented, or are using smartphone only as a remote, not part of the robot.

The best example was line-following (it was also an example of both other categories) robot developed by co-students Krzysztof Taborski and Kamil Szyc, but it wasn't published anywhere. Part of their code was even used as one of methods to communicate between Android and MCU.

Figure 1.3: Mobile Controlled Robot by Robotics Bible, [3]

Figure 1.4: Mobile Controlled Robot by Mayoogh Girish, [4]

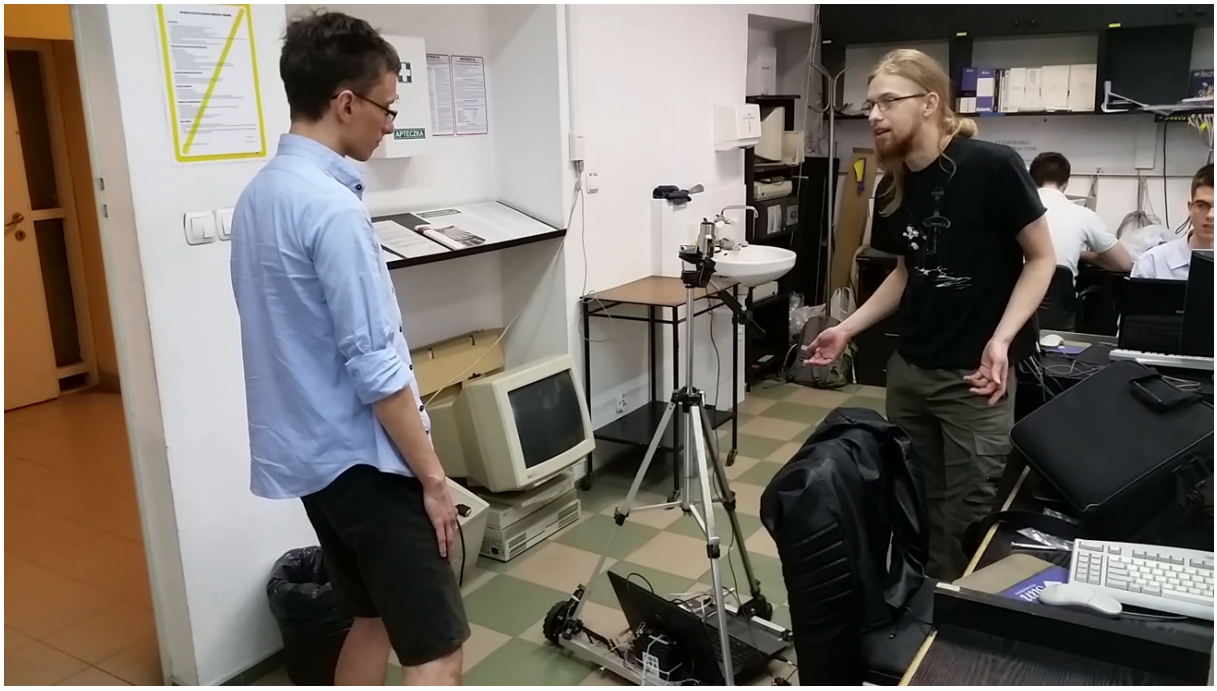


Figure 1.5: FaceFollower by Michał Kowalski and Adam Ćwik

1.4.3 “Followers”

Last category of robots are “Followers”, designed to follow something. Most popular ones are rather simple line-followers, which are built for tournaments, where they have to finish the route, marked with a single line, in shortest time. Another example, much more complicated, could be following face (e.g. during video conference) - it can’t be realized with simple color detection. It requires a camera and more computational power, than MCUs can offer. One of such robots was the one designed for one of previous projects (fig. 1.5).

It was extremely big one, because it was using a laptop as controller and external camera on a tripod, and all of that needed to be placed on a robot. It was also using one on Freescale’s microprocessors.

Camera was capturing video, application developed for laptop was responsible for detecting faces on it, and then was sending message with command to MCU over USB cable, which was responsible for translating it into steering for wheels and so on. MCU also echoed back commands and was informing about current position and speed of wheels, read from encoders on them. All of that required sending several characters long strings in both directions.

Main idea behind this thesis, is to check, if smartphones can be used to miniaturize such robot, by replacing laptop and external camera with much smaller, but not much worse, device. Therefore, only checking possibility and performance of face detection and communication with MCU needs to be done.

Chapter 2

Platforms

2.1 Android

Today Android is most popular mobile operating system. Almost everyone has one or even more phones with it, so they can easily be temporarily used as controllers. For this thesis two smartphones were used: currently exploited Motorola Moto G LTE (img. 2.1), and older Sony Ericsson Xperia Neo (img. 2.2).

Their most important parameters are:

1. Xperia Neo:
    - Release date: end of 2011,
    - Android version: 4.0.4 (API 15),
    - CPU: 1x1.00GHz,
    - GPU: Adreno 205,
    - RAM: 512MB
    - Camera: 0.3Mpx (front), 8.1Mpx (rear),
    - Screen: 3,7” 480x854,
2. Moto G (LTE):
    - Release date: middle 2014,
    - Android version: 5.1 (API 22),
    - CPU: 4x1.20GHz,
    - GPU: Adreno 305,
    - RAM: 1GB
    - Camera: 1.3Mpx (front), 5Mpx (rear),
    - Screen: 4,5” 720x1280,

Applications for Android can be written in Java (not only), which allows for high-level programming. Earlier, development ws done using ADT (Android Development Tools) plugin for Eclipse, but now it was replaced by Android Studio (img. 2.3) based on IntelliJ IDEA. It is a great tools, which makes writing code, refactoring, designing layout and building application (using Gradle) really easy. Unfortunately, it’s still new, and many tutorials and are still available

Figure 2.1: Motorola Moto G LTE

Figure 2.2: Sony Ericsson Xperia Neo

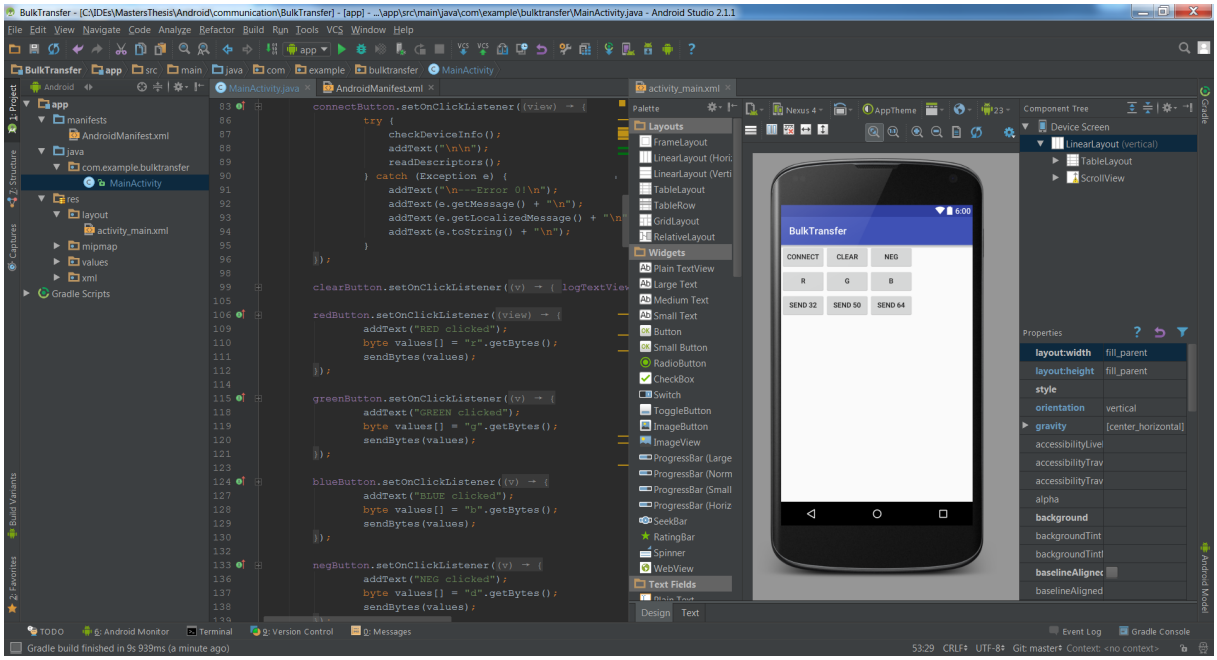


Figure 2.3: Android Studio

only for ADT. It's especially big problem for development with NDK (Native Development Kit), which allows for writing code in C++, and is offering better performance, but is still not 100% supported by Android Studio.

However, it's a problem only with setting up project (which can often be done automatically by Android Studio), and all code remains the same, so it's possible to easily find answers to most problems. Great place for looking for them is StackOverflow, but there's also a lot of blogs and sites dedicated to development on Android.

Next advantage of Android is that usually the same application can be used without any changes on different smartphones (as long as they are new enough).

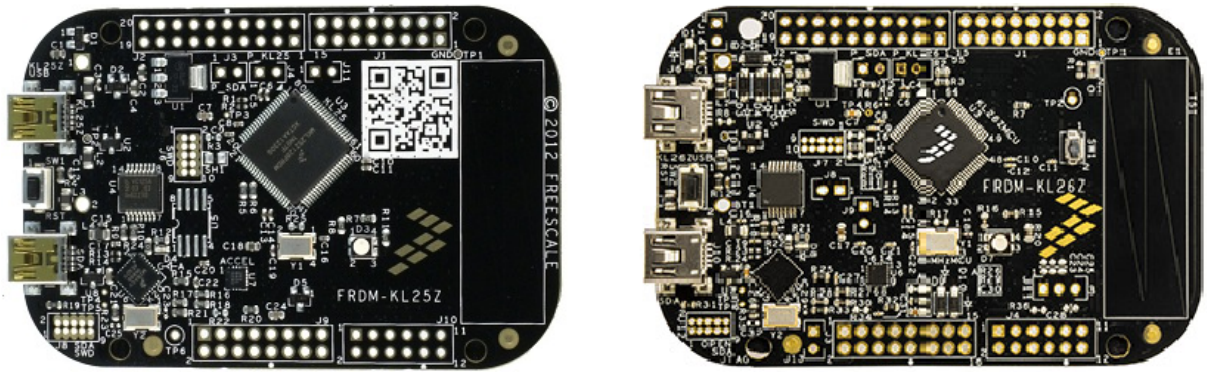


Figure 2.4: Used microcontrollers
Freescale's FRDM KL25Z (left) and KL26Z (right)

2.2 MCU

2.2.1 Introduction

There's a lot of microcontroller manufacturers, and even MCUs from the same one can be completely different, so applications need to be designed for specific one. Freescale's FRDM KL25Z (and when it has broken, almost identical KL26Z, both shown on img. 2.4) was chosen, because it is quite popular, [1] is a great blog focused on Freescale's microcontroller and KL25Z in particular, and development, deployment and debugging can be done using Eclipse-based CodeWarrior (img. 2.5).

CodeWarrior is a great tool, which simplifies a lot configuration of MCU's peripherals and functions - each is presented as a "bean"/component configurable through GUI. Beans with new functionality can be created or imported - lot of them are available on [1], with tutorials how to use them. It's possible to auto-generate methods for beans, which allows for pseudo high-level programming.

Most important functions of both MCUs are:

- IO pins,
- RGB LED diode,
- PWM (Pulse Width Modulation),
- button (only KL26Z),
- UART,
- capacity slider,
- two USB CDC ports (one with openSDA for debugging or virtual UART),
- pins position compatible with Arduino platform.

Easiest method to connect them with smartphone is through USB cable using CDC or UART.

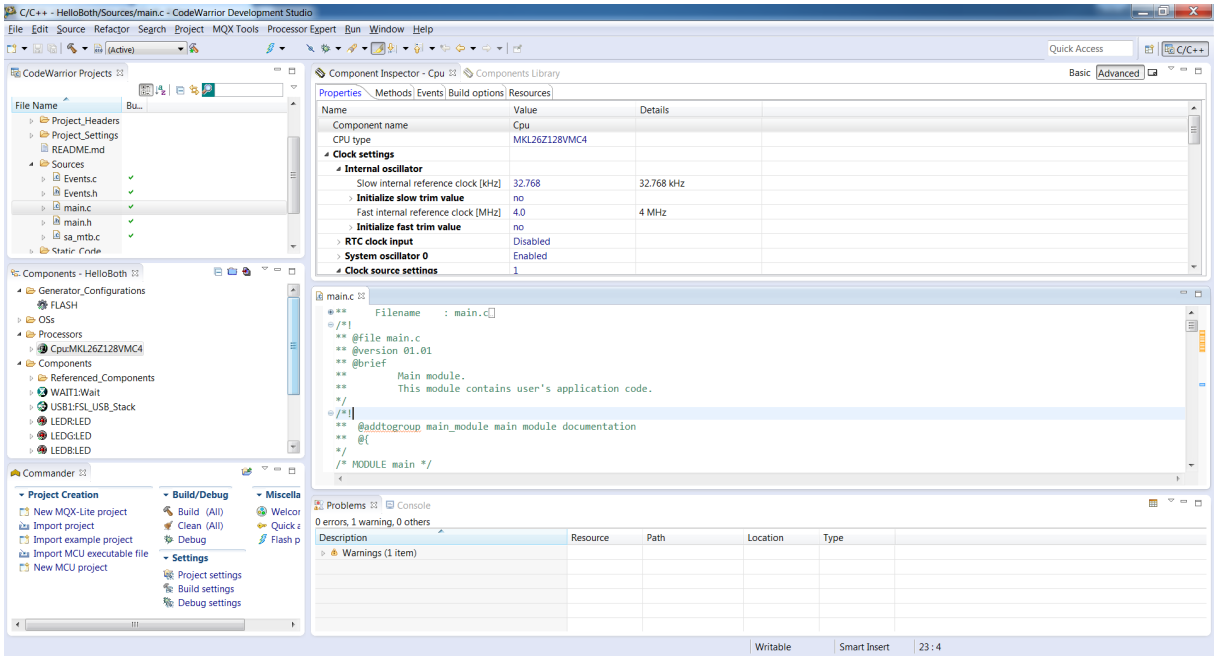


Figure 2.5: CodeWarrior

2.2.2 Communication through UART

UART (which stands for Universal Asynchronous Receiver/Transmitter) is actually a hardware device used for sequential transmission of bits, but it's quite popular to use its name as a type of transmission. It was implemented based on article *"Tutorial: printf() and "Hello World!" with the Freedom KL25Z Board"* available on [1]. MCU receives data and sends it back as long as there is something in its read buffer (which is actually a queue).

UART is rather popular among microcontrollers, and KL25/26Z even have it virtualized in their USB openSDA port, so it requires only a popular mini USB cable. Unfortunately this blocks openSDA port for debugger.

2.2.3 Communication through CDC

USB CDC stands for USB Communications Device Class, and is type of device, which can communicate using USB. There is also an article on [1], titled *"Tutorial: USB CDC with the KL25Z Freedom Board"*, how to make MCU a representative of CDC.

Implementation is rather complex (easy with using beans, but generates a lot of code), but big advantage is fact, that it can use "normal" USB port, and leaves openSDA one for a debugger.

Chapter 3

Communication

3.1 Introduction

The most important thing in extending robot's capabilities with a smartphone, is to connect smartphone with MCU. Because chosen MCU supports USB CDC and UART through it's OpenSDA USB port, one of possible ways is to do it with USB cable connected to smartphone's micro USB port. Next thing is to find a way, which will allow for access to a port, and to send a text message through it. After research, three ways were found:

- USB Host API [5],
- usb-serial-for-android library by mik3y [8],
- UsbSerial by felHR85 [9].

Because of similar names of projects, they will be referenced as Host API, mik3y and felHR85.

Each method will be implemented, and tested on both smartphones with both solutions on MCU's side. Few test messages with different size will be send, and received echo will be analyzed. Selected sizes and messages are:

1. 1 character:
   any single letter,
2. 32 characters:
   "String with 32 charsBA987654321!",
3. 50 characters:
   "String with 50 chars..RQPONMLKJIHGFEDCBA987654321!",
4. 64 characters:
   "String with 64 chars.....RQPONMLKJIHGFEDCBA987654321!",
5. 70 characters (optionally, if previous works):
   "String with 70 chars.....RQPONMLKJIHGFEDCBA987654321!"

Because buffer in MCU is 64 elements long, it should allow for good analysis of correctness of result and performance. MCU uses C-strings, where last element of string is "\0", so it can actually fit only 63 characters, therefore 64 letters long string should be too long for it. Also, echo is in format "*protocol*: *text*\n", where *protocol* is "uart" or "cdc", and *text* is content of protocol's buffer in MCU. All methods are transmitting bytes, and not letters, but it's easy to transform it in both directions.

Listing 3.1: USB Host API

```
UsbManager manager = (UsbManager)
   getSystemService(Context.USB_SERVICE);
UsbDeviceConnection connection = manager.openDevice(device)
/* lot of code to find endpoints */
connection.controlTransfer(/* lot of hard to deduct parameters */);
// write
connection.bulkTransfer(usbEndpointOut, writeBuffer,
   writeBuffer.length, TIMEOUT);
// read
connection.bulkTransfer(usbEndpointIn, readBuffer,
   readBuffer.length, TIMEOUT);
```

Tests for each combination were performed, and are documented (with conclusions in captions) on screenshots (img. 3.1-3.23) in section dedicated to each method. However, screenshots were made only if such combination was working, and only for Moto G (results for Xperia Neo were generally the same, if not stated otherwise).

Before testing own implementation, tests with terminals available on Google Play were performed, and showed, that Xperia Neo works only with CDC, and not UART, while Moto G works with both. It should be tested, if this is true also for own applications.

3.2 USB Host API

First method is USB Host API, which is built-in Android. It's a low-level method, and because of that, it's hard to configure. At first, it required finding endpoints which can be used for writing and reading, which is realized by many nested-loops. Then, usage of controlTransfer(...) method is required, to configure. It's not documented, what actually should be passed to it. Fortunately, example working with another Freescale microcontroller was found, and was also working with FRDM KL26Z, so it was possible to test it, however it was working only with CDC, and not UART.

Sample code for sending and receiving message is shown on listing 3.1. It's synchronous method, so programmer has to care for regular calling of reading function. In case of echo, it can be done only after sending something, but it can't be used, if MCU sends something unexpected, like message about error, or current state of a robot. Which is even worse, with this type of communication, write causes freeze of sender, until other device reads it. On the other hand, it means, that message can't be missed.

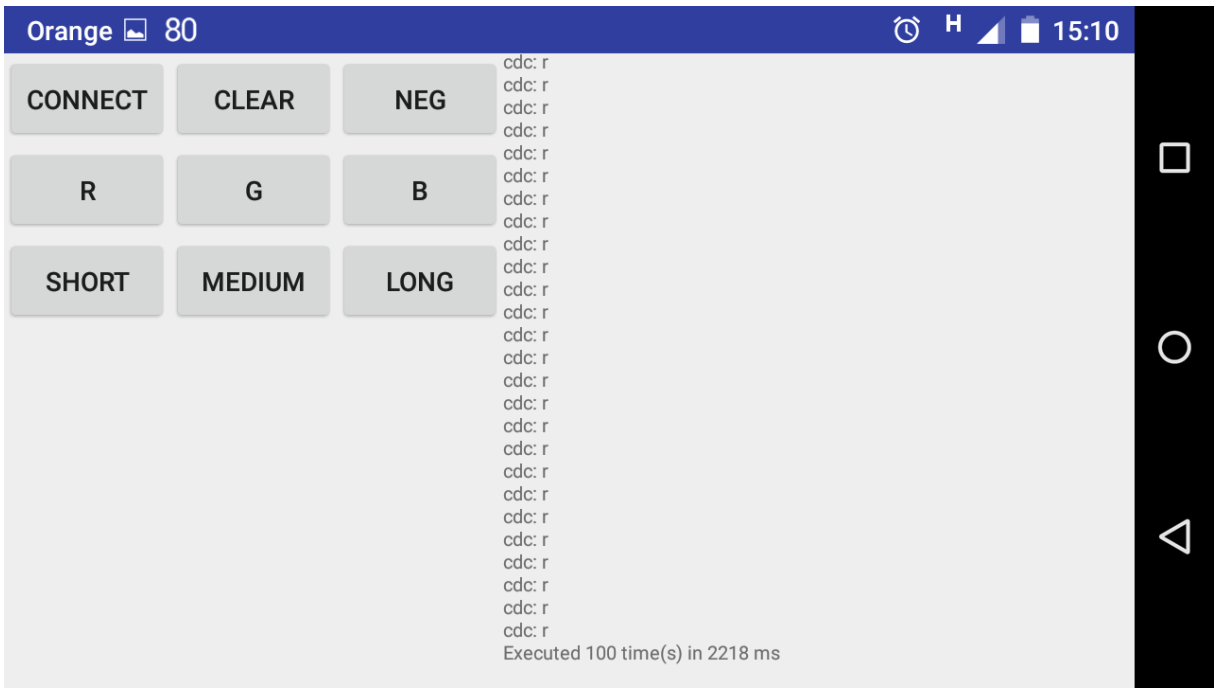


Figure 3.1: USB Host API - sending 1 letter
Received text is correct. As it can be seen, in this test it takes about 22 ms to get an echo of each sent message. Although, during other (not documented with screenshot) tests, sometimes (but less often) average duration was about 12 ms.

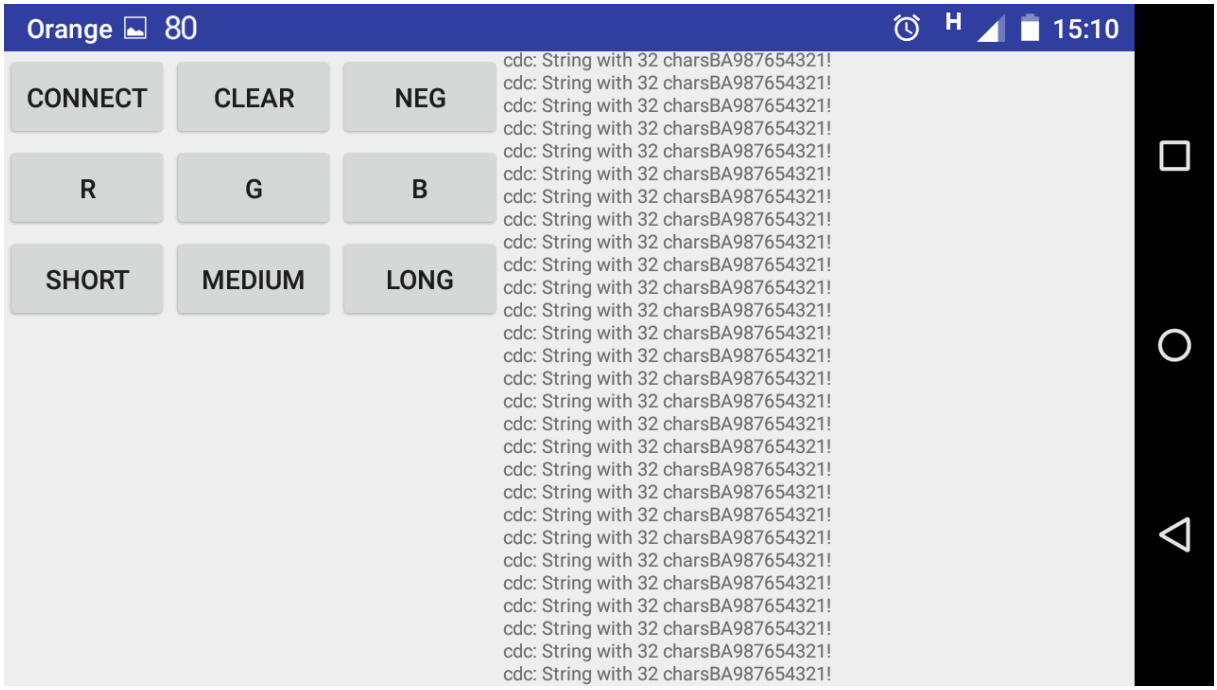
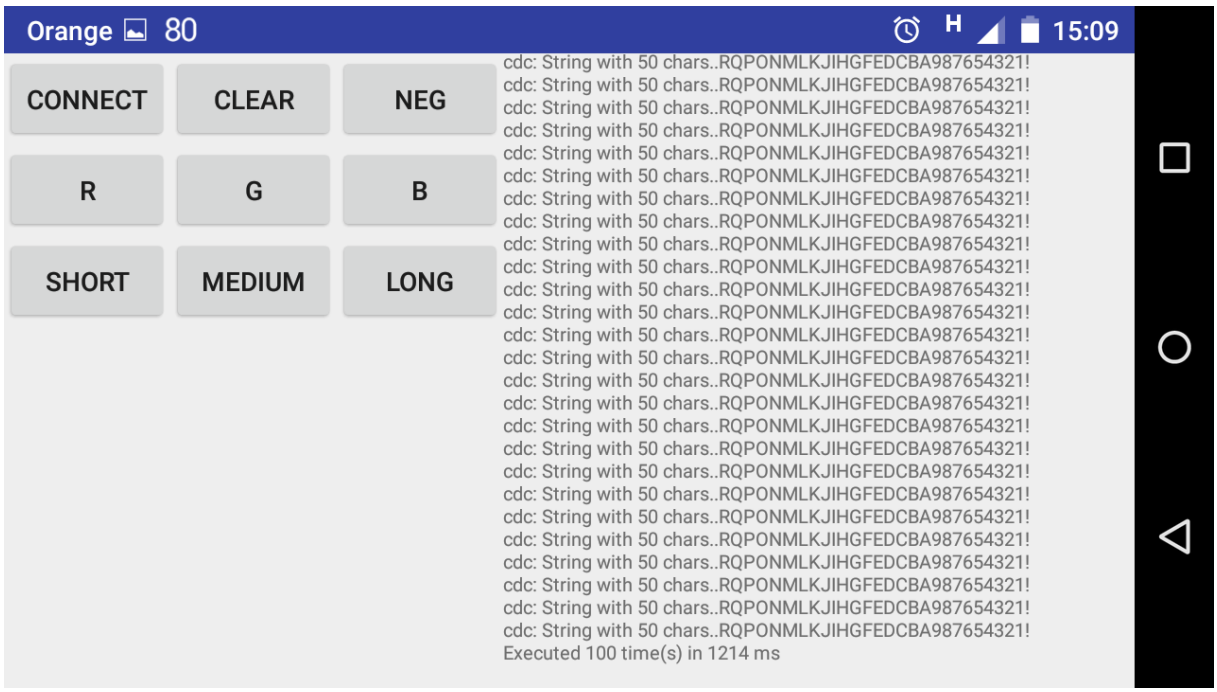
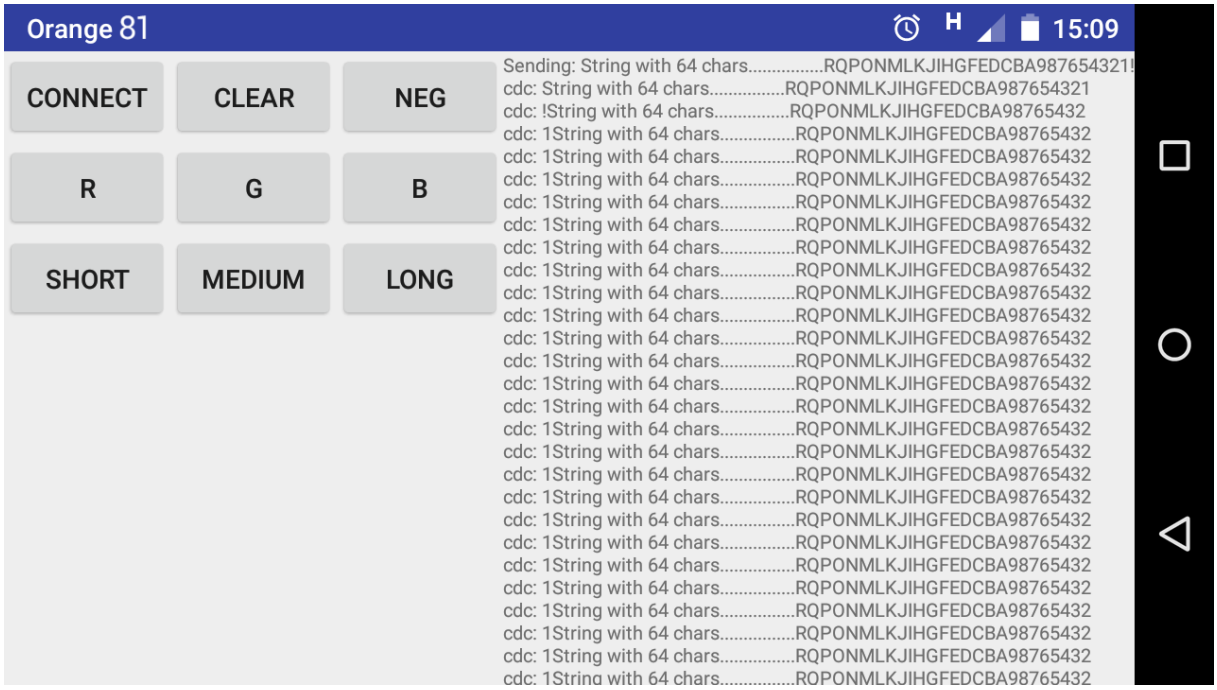


Figure 3.2: USB Host API - sending 32 letters
Received text is correct. Average time of execution was 12 ms, but in other tests, it also happened (but less often) to be 22 ms.



```
cdc: String with 50 chars..RQPONMLKJIHGFEDCBA987654321!
cdc: String with 50 chars..RQPONMLKJIHGFEDCBA987654321!
cdc: String with 50 chars..RQPONMLKJIHGFEDCBA987654321!
cdc: String with 50 chars..RQPONMLKJIHGFEDCBA987654321!
cdc: String with 50 chars..RQPONMLKJIHGFEDCBA987654321!
cdc: String with 50 chars..RQPONMLKJIHGFEDCBA987654321!
cdc: String with 50 chars..RQPONMLKJIHGFEDCBA987654321!
cdc: String with 50 chars..RQPONMLKJIHGFEDCBA987654321!
cdc: String with 50 chars..RQPONMLKJIHGFEDCBA987654321!
cdc: String with 50 chars..RQPONMLKJIHGFEDCBA987654321!
cdc: String with 50 chars..RQPONMLKJIHGFEDCBA987654321!
cdc: String with 50 chars..RQPONMLKJIHGFEDCBA987654321!
cdc: String with 50 chars..RQPONMLKJIHGFEDCBA987654321!
cdc: String with 50 chars..RQPONMLKJIHGFEDCBA987654321!
cdc: String with 50 chars..RQPONMLKJIHGFEDCBA987654321!
cdc: String with 50 chars..RQPONMLKJIHGFEDCBA987654321!
cdc: String with 50 chars..RQPONMLKJIHGFEDCBA987654321!
cdc: String with 50 chars..RQPONMLKJIHGFEDCBA987654321!
cdc: String with 50 chars..RQPONMLKJIHGFEDCBA987654321!
cdc: String with 50 chars..RQPONMLKJIHGFEDCBA987654321!
cdc: String with 50 chars..RQPONMLKJIHGFEDCBA987654321!
cdc: String with 50 chars..RQPONMLKJIHGFEDCBA987654321!
cdc: String with 50 chars..RQPONMLKJIHGFEDCBA987654321!
cdc: String with 50 chars..RQPONMLKJIHGFEDCBA987654321!
cdc: String with 50 chars..RQPONMLKJIHGFEDCBA987654321!
cdc: String with 50 chars..RQPONMLKJIHGFEDCBA987654321!
Executed 100 time(s) in 1214 ms
```

Figure 3.3: USB Host API - sending 50 letters
Received text is correct. Average time of execution was 12 ms, but in opposition to sending 1 or 32 letters, other values were not observed.



```
Sending: String with 64 chars................RQPONMLKJIHGFEDCBA987654321!
cdc: String with 64 chars................RQPONMLKJIHGFEDCBA987654321
cdc: !String with 64 chars................RQPONMLKJIHGFEDCBA98765432
cdc: 1String with 64 chars................RQPONMLKJIHGFEDCBA98765432
cdc: 1String with 64 chars................RQPONMLKJIHGFEDCBA98765432
cdc: 1String with 64 chars................RQPONMLKJIHGFEDCBA98765432
cdc: 1String with 64 chars................RQPONMLKJIHGFEDCBA98765432
cdc: 1String with 64 chars................RQPONMLKJIHGFEDCBA98765432
cdc: 1String with 64 chars................RQPONMLKJIHGFEDCBA98765432
cdc: 1String with 64 chars................RQPONMLKJIHGFEDCBA98765432
cdc: 1String with 64 chars................RQPONMLKJIHGFEDCBA98765432
cdc: 1String with 64 chars................RQPONMLKJIHGFEDCBA98765432
cdc: 1String with 64 chars................RQPONMLKJIHGFEDCBA98765432
cdc: 1String with 64 chars................RQPONMLKJIHGFEDCBA98765432
cdc: 1String with 64 chars................RQPONMLKJIHGFEDCBA98765432
cdc: 1String with 64 chars................RQPONMLKJIHGFEDCBA98765432
cdc: 1String with 64 chars................RQPONMLKJIHGFEDCBA98765432
cdc: 1String with 64 chars................RQPONMLKJIHGFEDCBA98765432
cdc: 1String with 64 chars................RQPONMLKJIHGFEDCBA98765432
cdc: 1String with 64 chars................RQPONMLKJIHGFEDCBA98765432
cdc: 1String with 64 chars................RQPONMLKJIHGFEDCBA98765432
cdc: 1String with 64 chars................RQPONMLKJIHGFEDCBA98765432
cdc: 1String with 64 chars................RQPONMLKJIHGFEDCBA98765432
cdc: 1String with 64 chars................RQPONMLKJIHGFEDCBA98765432
cdc: 1String with 64 chars................RQPONMLKJIHGFEDCBA98765432
cdc: 1String with 64 chars................RQPONMLKJIHGFEDCBA98765432
cdc: 1String with 64 chars................RQPONMLKJIHGFEDCBA98765432
cdc: 1String with 64 chars................RQPONMLKJIHGFEDCBA98765432
```

Figure 3.4: USB Host API - sending 64 letters
Received text is NOT correct. Only first message is received completely, but sent from MCU as two. Next ones are missing last character, and have last non-missed character from previous one at the beginning.

Listing 3.2: mik3y

```
UsbManager manager = (UsbManager)
   getSystemService(Context.USB_SERVICE);
ProbeTable customTable = new ProbeTable();
customTable.addProduct(VENDOR_ID1, PRODUCT_ID1,
   CdcAcmSerialDriver.class);
customTable.addProduct(VENDOR_ID2, PRODUCT_ID2,
   CdcAcmSerialDriver.class);
UsbSerialProber prober = new UsbSerialProber(customTable);
List<UsbSerialDriver> availableDrivers =
   prober.findAllDrivers(manager);
UsbSerialDriver driver = availableDrivers.get(0);
UsbDeviceConnection connection =
   manager.openDevice(driver.getDevice());
if (connection == null) {
    // probably lacks permission
    UsbManager.requestPermission(driver.getDevice(), ...)
    return;
}
UsbSerialPort port = driver.getPorts.get(0);
port.open(connection);
try {
    port.setParameters(baudRate, dataBits, stopBits, parity);
    byte writeBuffer[] = "text".getBytes();
    byte readBuffer[] = new byte[16];
    int numBytesWritten = port.write(writeBuffer, TIMEOUT)
    int numBytesRead = port.read(readBuffer, TIMEOUT);
} catch (IOException e) {
    // Deal with error.
} finally {
    port.close();
}
```

3.3 USB Serial by mik3y

Next method is USB Serial library available on Github of user named mik3y. It's seems to be quite popular - it's often mentioned on StackOverflow, and on project's page, there is a list of many project using it. Example usage of library is shown on listing 3.2. Analyzing implementation of methods shows, that it makes usage of USB Host API, but in more user-friendly, high-level form.

Unfortunately, it seems to be not compatible with chosen MCU, on any phone, and neither on CDC, nor UART protocol. Actually, during many tries, it was possible to send something using it, but received text was not the one, which was send. Even more interesting is fact, that MCU received some data, when reading method was called.

Therefore, this library couldn't be tested further, and also any sensible screenshot wasn't made. Nonetheless, it should work fine with other microcontrollers, and be easier to use than USB Host API.

Listing 3.3: felHR85 async

```
usbManager = (UsbManager) getSystemService(Context.USB_SERVICE);
HashMap<String, UsbDevice> usbDevices = usbManager.getDeviceList();
for (Map.Entry<String, UsbDevice> entry : usbDevices.entrySet()) {
    device = entry.getValue();
}
UsbDeviceConnection connection = usbManager.openDevice(device);
UsbSerialDevice serial =
   UsbSerialDevice.createUsbSerialDevice(device, connection);
serial.open();
serial.setBaudRate(115200);
serial.setDataBits(UsbSerialInterface.DATA_BITS_8);
serial.setParity(UsbSerialInterface.PARITY_ODD);
serial.setFlowControl(UsbSerialInterface.FLOW_CONTROL_OFF);
serial.read(new UsbSerialInterface.UsbReadCallback() {
    @Override
    public void onReceivedData(final byte[] bytes) {
        // to domething with bytes, like:
        String str = new String(bytes, "UTF-8");
    }
});
serial.write("TEXT".getBytes());
serial.close();
```

3.4 USB Serial by felHR85

Last found library is USB Serial, also available on Github, on account called felHR85. It also has some popularity, but smaller that USB Serial by mik3y. Library allows for asynchronous transmission, with optional synchronous mode. Another advantage is fact, that library's page mentions using jipack.io for easier adding it to project, and also example project works with KL26Z out-of-the-box.

3.4.1 Asynchronous mode

Asynchronous mode (listing 3.3) offers one big advantage: it reads using callbacks. Programmer doesn't have to worry about continuous reading, because it's done automatically, and only have to implement function, which operates on received bytes. Also, implementation of writing and reading is not using USB Host API, or at least it much "deeper".

First tests showed, that it works both with CDC and UART, although as it was expected, the latter was working only on Moto G.

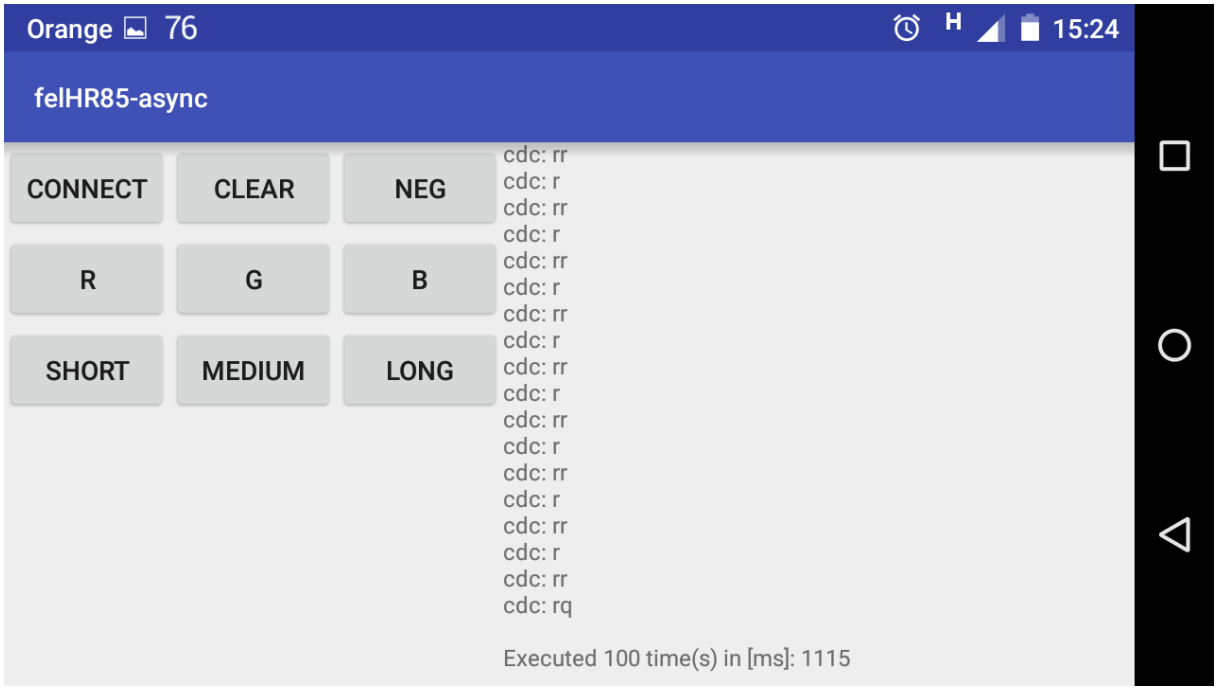


Figure 3.5: felHR85, async-CDC, 1 letter
Received text is correct, but in most cases, two messages were sent at once. Average execution time was 11 ms.

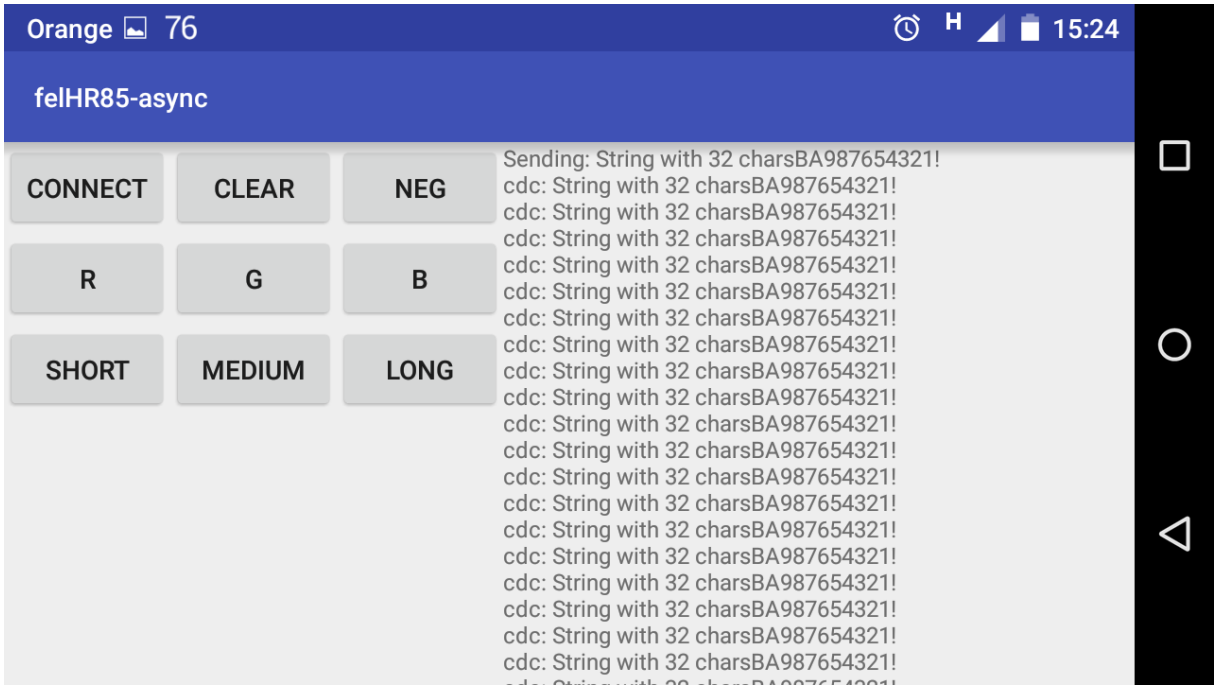


Figure 3.6: felHR85, async-CDC, 32 letters
Received text is correct. In this case, all messages were sent separately. Average execution time was also 11 ms.

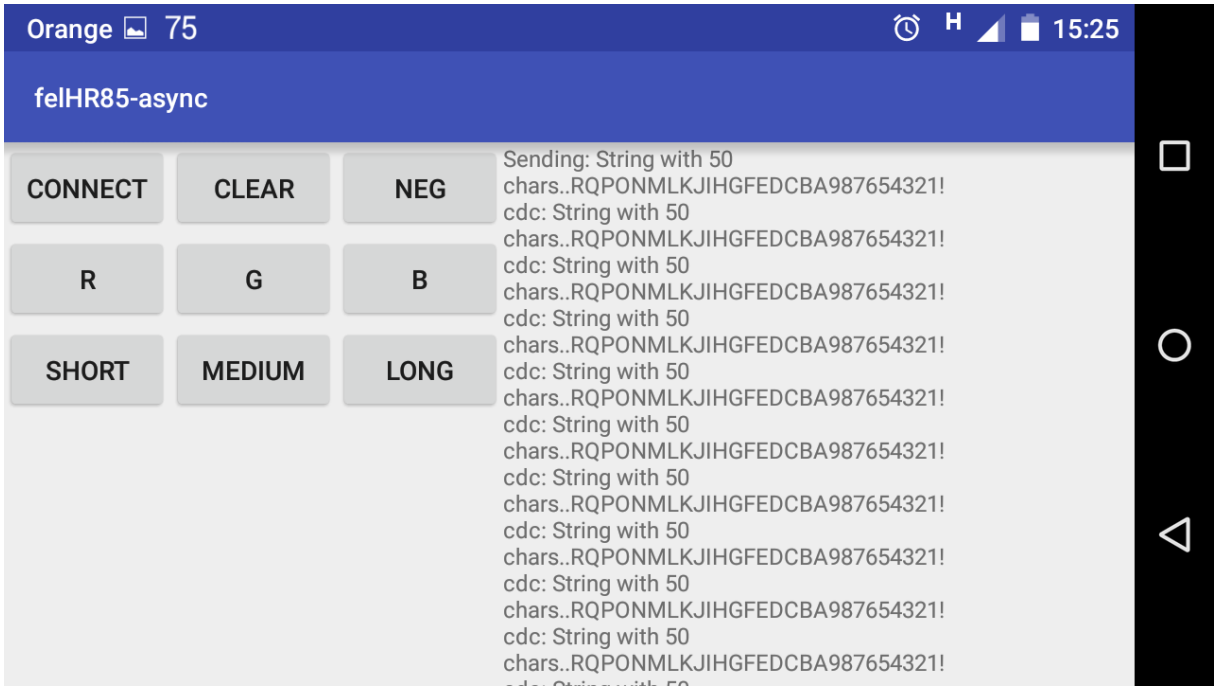


Figure 3.7: felHR85, async-CDC, 50 letters
Received text is correct, and all messages were sent separately. Each text is divided into two lines, because it doesn't fit the window, and Android does that automatically. Average execution time was 12 ms.

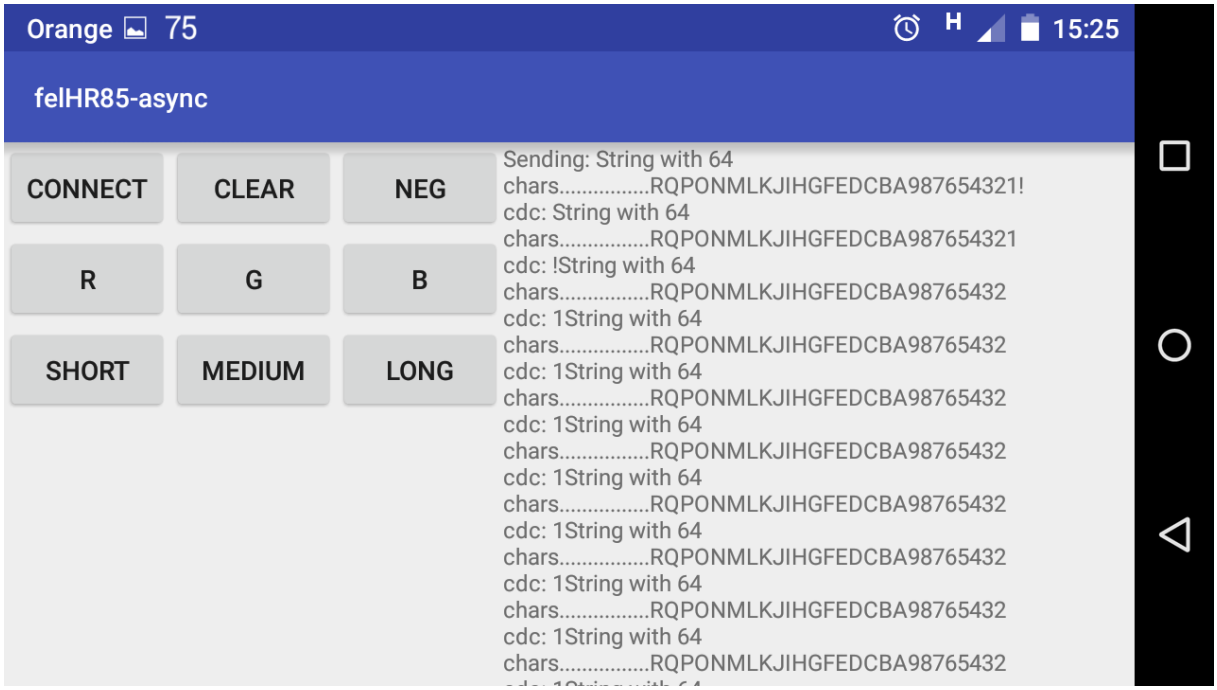


Figure 3.8: felHR85, async-CDC, 64 letters
Received text is NOT correct. Result looks the same as for using USB Host API.

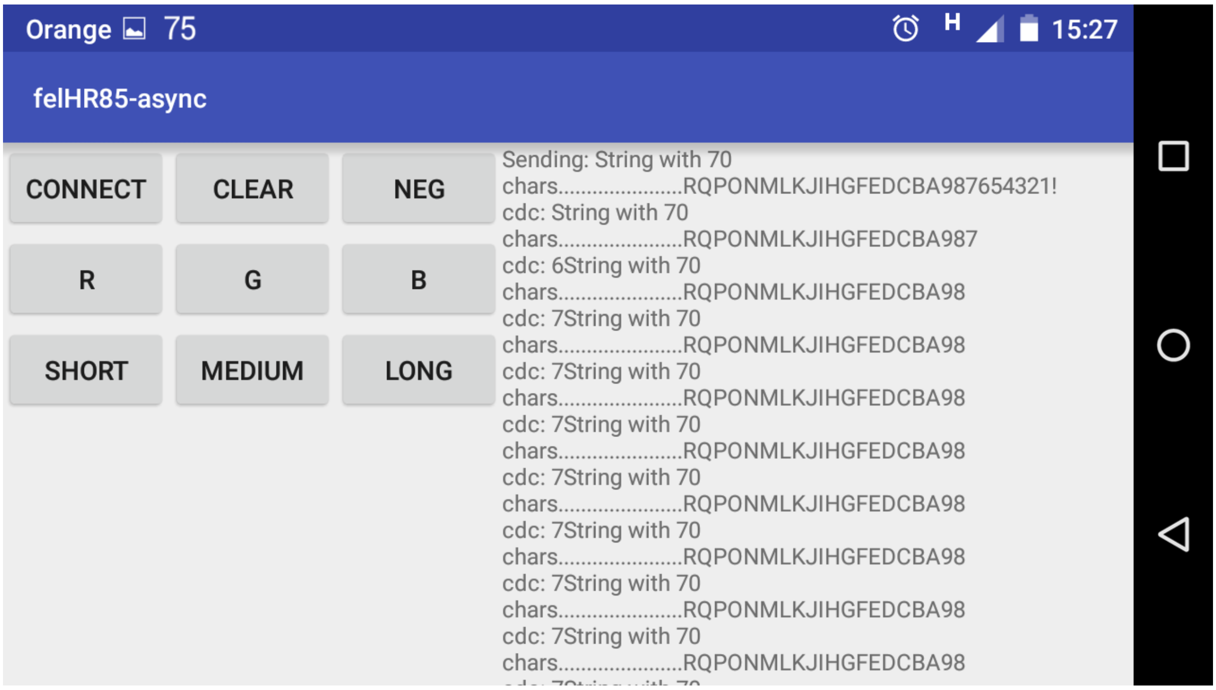


Figure 3.9: felHR85, async-CDC, 70 letters
Received text is also not correct. It can be also seen, that the same as for 64 chars long message, ending is missing, and next message starts with first of missing letters.

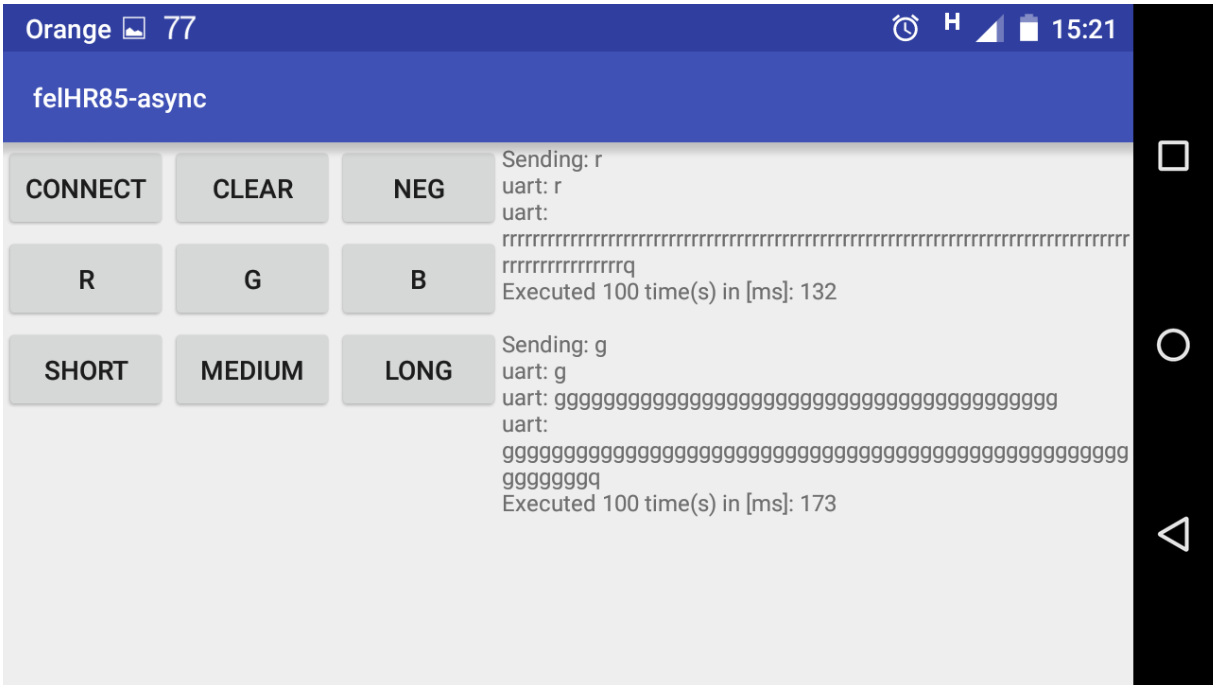
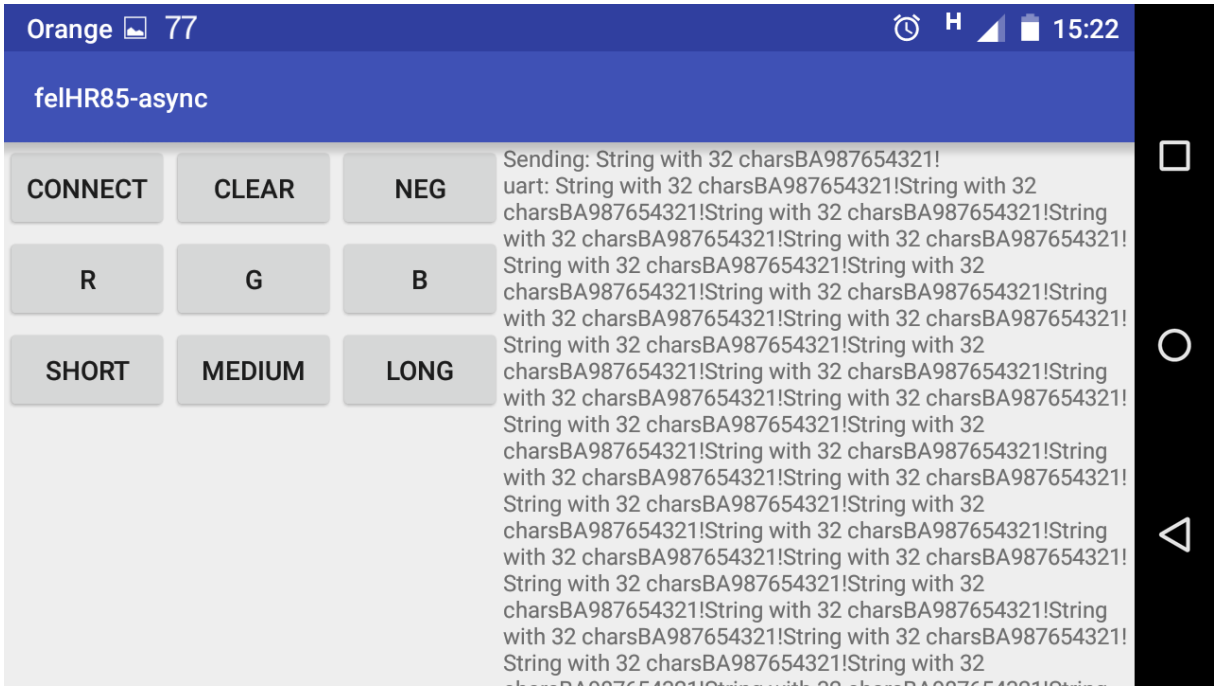
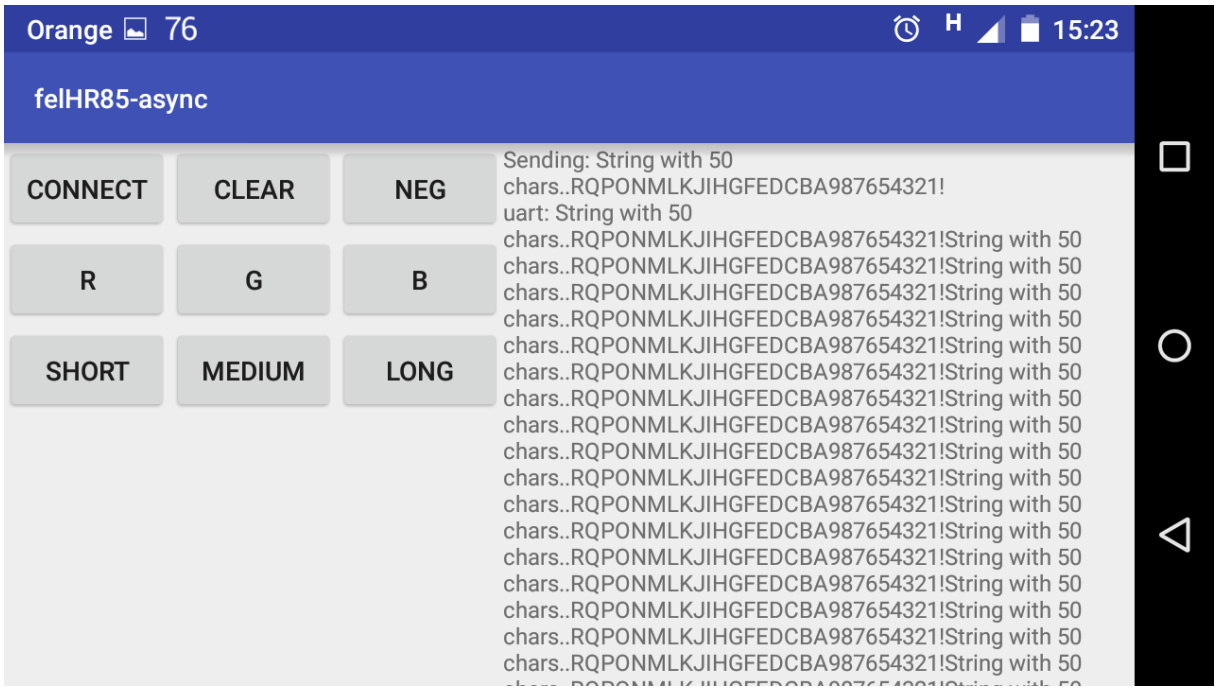


Figure 3.10: felHR85, async-UART, 1 letter
Received text was correct, but send as only a few messages. In async-CDC it was only 2 messages at once, in this case all is send in 2-3. Average execution time was about 1.5 ms, which is really faster than using CDC.



```
Sending: String with 32 charsBA987654321!
uart: String with 32 charsBA987654321!String with 32
charsBA987654321!String with 32 charsBA987654321!String
with 32 charsBA987654321!String with 32 charsBA987654321!
String with 32 charsBA987654321!String with 32
charsBA987654321!String with 32 charsBA987654321!String
with 32 charsBA987654321!String with 32 charsBA987654321!
String with 32 charsBA987654321!String with 32
charsBA987654321!String with 32 charsBA987654321!String
with 32 charsBA987654321!String with 32 charsBA987654321!
String with 32 charsBA987654321!String with 32
charsBA987654321!String with 32 charsBA987654321!String
with 32 charsBA987654321!String with 32 charsBA987654321!
String with 32 charsBA987654321!String with 32
charsBA987654321!String with 32 charsBA987654321!String
with 32 charsBA987654321!String with 32 charsBA987654321!
String with 32 charsBA987654321!String with 32
charsBA987654321!String with 32 charsBA987654321!String
with 32 charsBA987654321!String with 32 charsBA987654321!
String with 32 charsBA987654321!String with 32
```

Figure 3.11: felHR85, async-UART, 32 letters
Received text looks messy, but is also regular, so it still can be seen, that is correct. “uart“ can be seen only once, so it means, that MCU can continuously get text from buffer, before escaping loop. Average execution time was about 34 ms, which is longer than using CDC, but seems to be correlated to length of message.



```
Sending: String with 50
chars..RQPONMLKJIHGFEDCBA987654321!
uart: String with 50
chars..RQPONMLKJIHGFEDCBA987654321!String with 50
chars..RQPONMLKJIHGFEDCBA987654321!String with 50
chars..RQPONMLKJIHGFEDCBA987654321!String with 50
chars..RQPONMLKJIHGFEDCBA987654321!String with 50
chars..RQPONMLKJIHGFEDCBA987654321!String with 50
chars..RQPONMLKJIHGFEDCBA987654321!String with 50
chars..RQPONMLKJIHGFEDCBA987654321!String with 50
chars..RQPONMLKJIHGFEDCBA987654321!String with 50
chars..RQPONMLKJIHGFEDCBA987654321!String with 50
chars..RQPONMLKJIHGFEDCBA987654321!String with 50
chars..RQPONMLKJIHGFEDCBA987654321!String with 50
chars..RQPONMLKJIHGFEDCBA987654321!String with 50
chars..RQPONMLKJIHGFEDCBA987654321!String with 50
chars..RQPONMLKJIHGFEDCBA987654321!String with 50
chars..RQPONMLKJIHGFEDCBA987654321!String with 50
chars..RQPONMLKJIHGFEDCBA987654321!String with 50
chars..RQPONMLKJIHGFEDCBA987654321!String with 50
```

Figure 3.12: felHR85, async-UART, 50 letters
Received text is correct, and looks neater than the one for 32 chars, but it’s only because of displaying it. As for 32 chars, “uart: “ can be seen only from time to time (next one was off the screen). Average execution time was about 52 ms.

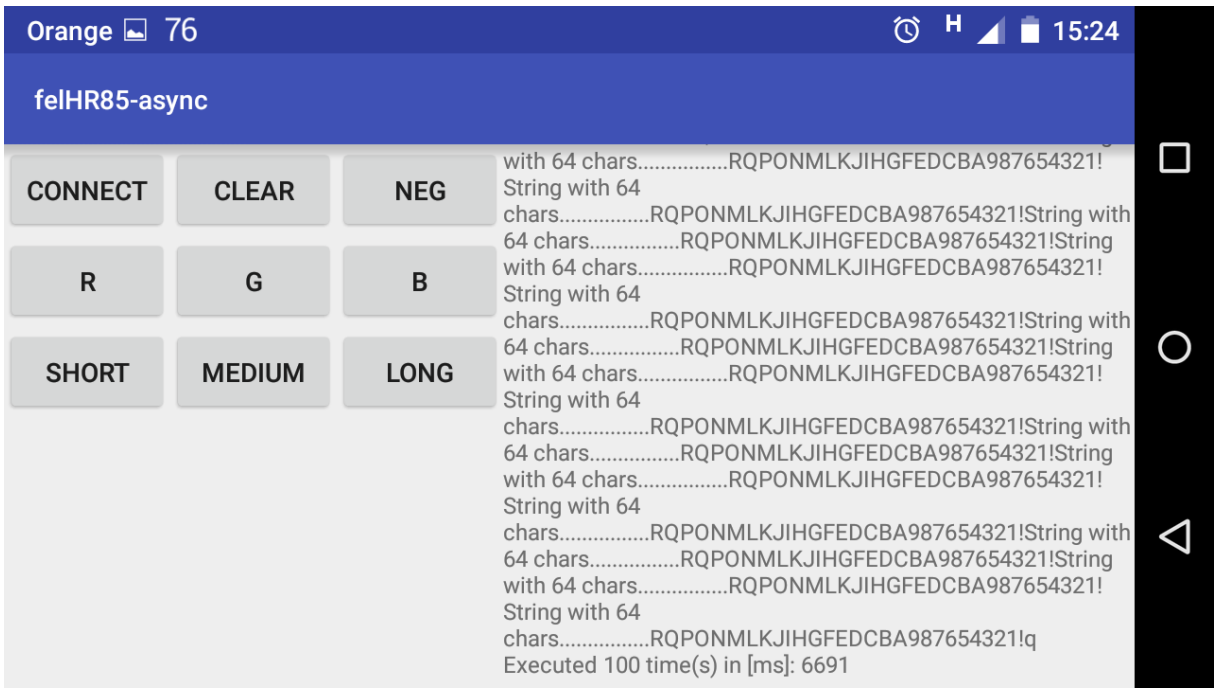


Figure 3.13: felHR85, async-UART, 64 letters
It's the first time, that 64 long message was transmitted completely without any errors. Average execution time was about 67 ms.

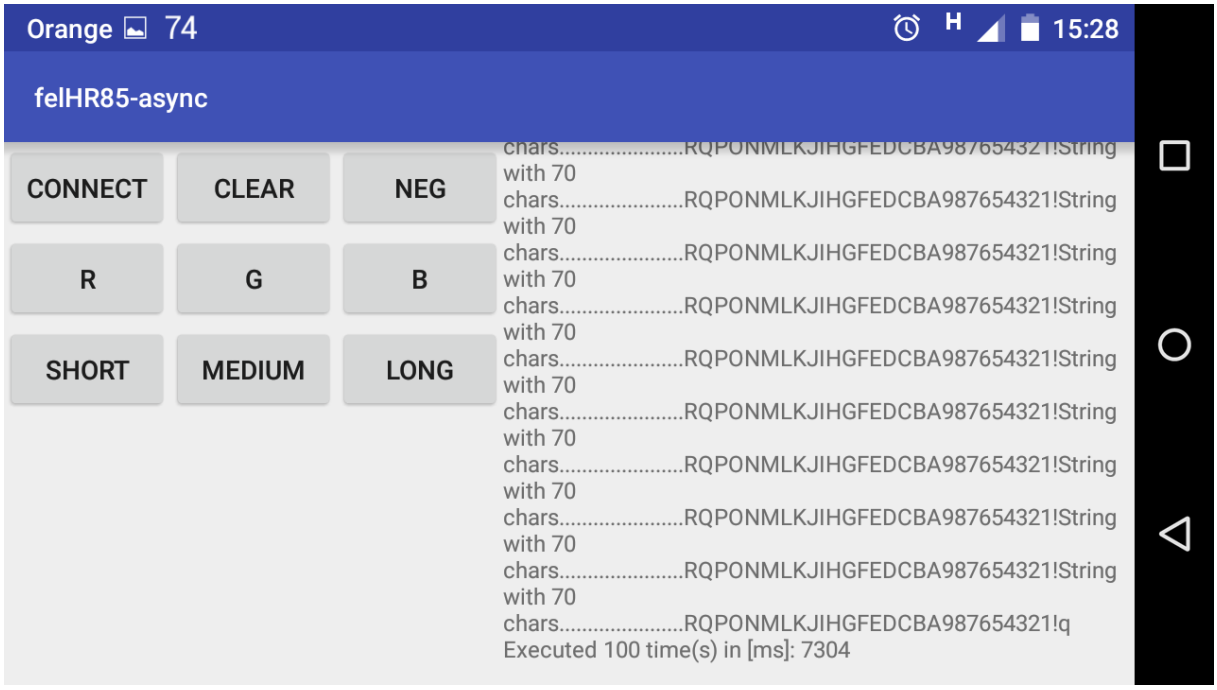


Figure 3.14: felHR85, async-UART, 70 letters
Because 64 chars long message was transmitted without errors, it was worth checking, if even longer messages could be send. And indeed, with UART it's possible to do it without errors. Average execution time was about 73 ms, so it actually increases with length.

Listing 3.4: felHR85 sync

```
UsbSerialDevice serial = /* same as in async */

serial.syncOpen();
serial.syncOpen();
serial.syncWrite(/*byte[] writeBuffer, int timeout*/)
serial.syncRead(/*byte[] readBuffer, int timeout*/)
serial.syncClose();
```

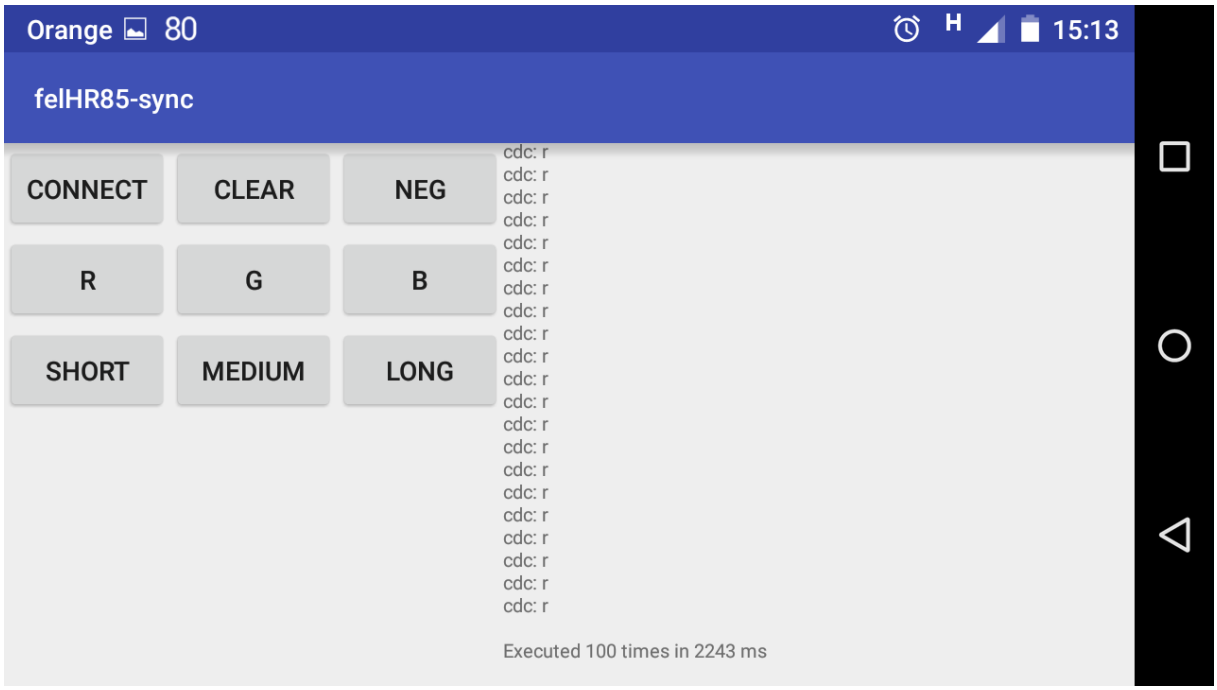


Figure 3.15: felHR85, sync-CDC, 1 letter
Received text is correct. Average execution time was about 22 ms, and in opposition to USB Host API, much shorter time was not observed.

3.4.2 Synchronous mode

USB Serial by felHR85 supports also synchronous way, where programmer has to care for reading. In this approach, writing and reading is actually only wrapping USB Host API, but looks much more nicely (listing 3.4).

What is interesting, is that it worked not only with CDC, but also with UART (although only on Moto G).

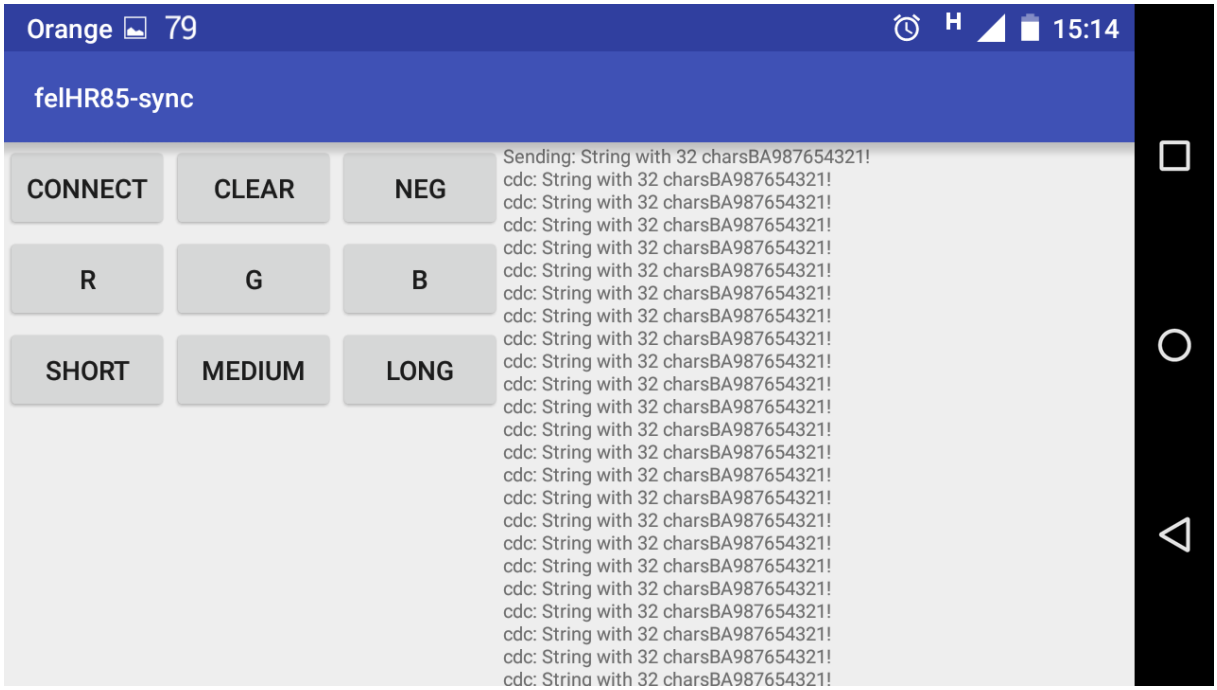


Figure 3.16: felHR85, sync-CDC, 32 letters
Same as earlier, result is correct. What is different to sending 1 char or to USB Host API, average execution time was always 12 ms.

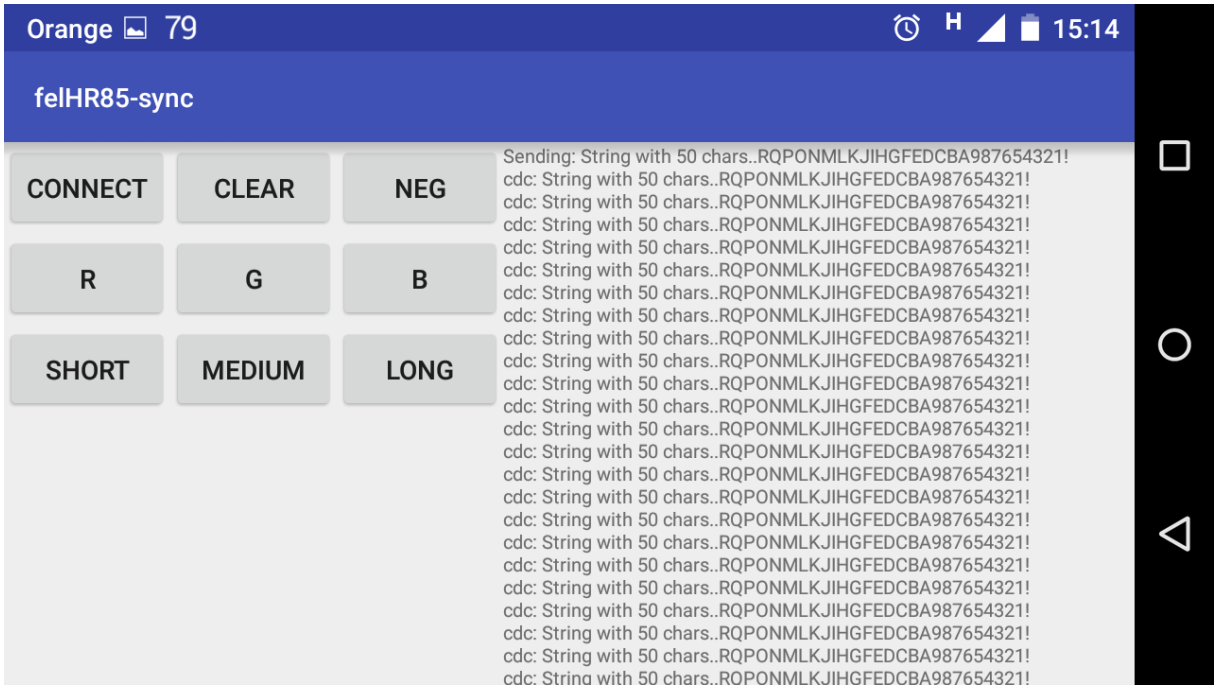


Figure 3.17: felHR85, sync-CDC, 50 letters
In this case, results were exactly the same as for USB Host API: text was correct and average execution time was about 12 ms.

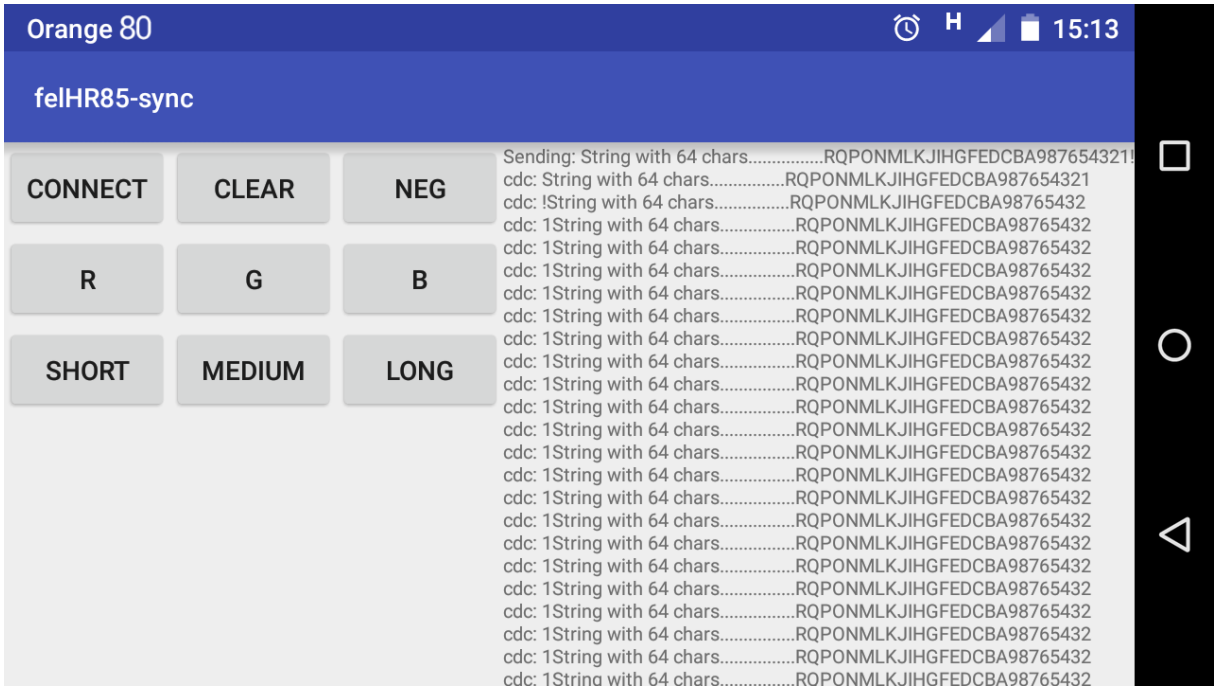


Figure 3.18: felHR85, sync-CDC, 64 letters
Also in case of exceeded buffer, result is the same as with USB Host API: chars exceeding buffer are missing, and one char from previous message appears in next one.

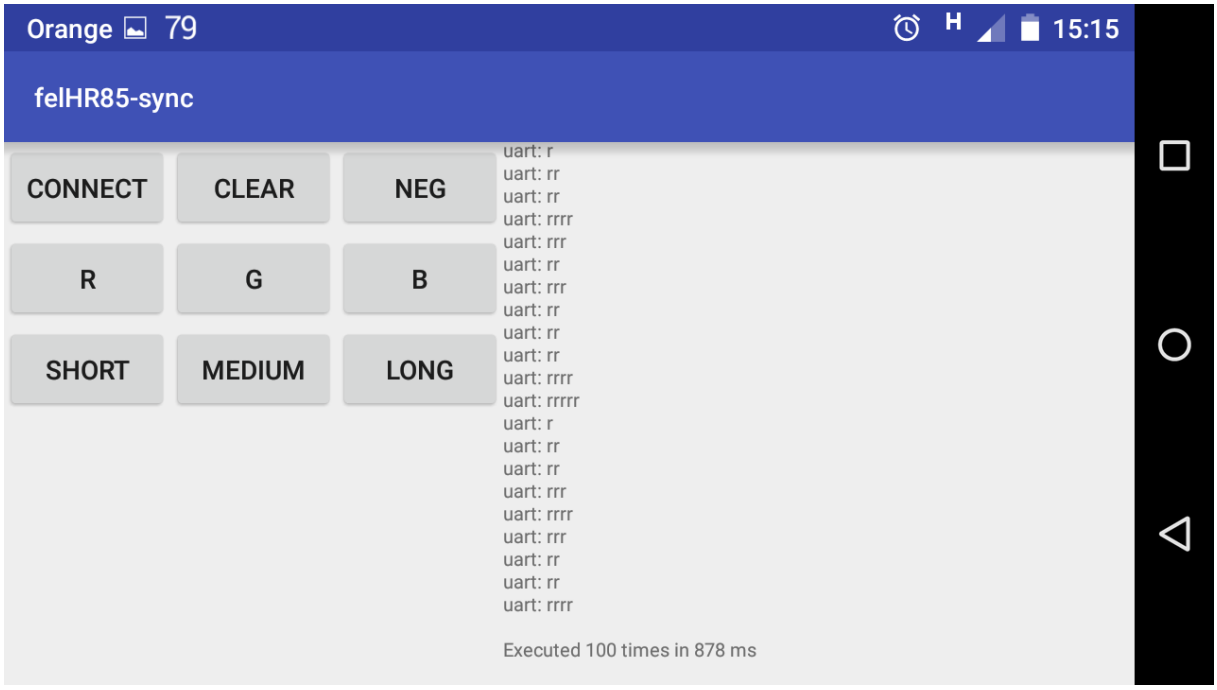


Figure 3.19: felHR85, sync-UART, 1 letter
Result looks similar to one for async-CDC (so it differs on both sides) but here more than two messages were sent at once. Observed average time is about 9 ms.

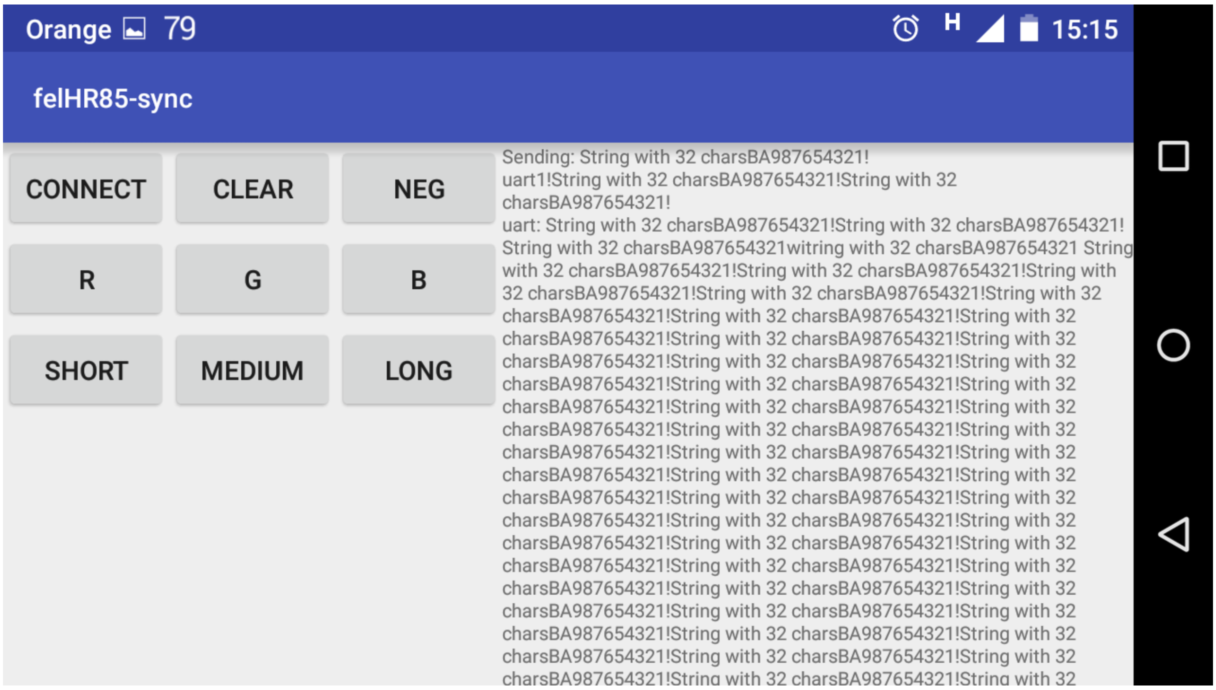


Figure 3.20: felHR85, sync-UART, 32 letters

Observed result is most similar to one for async-UART. Result is correct, and also average time of execution was about 33 ms.

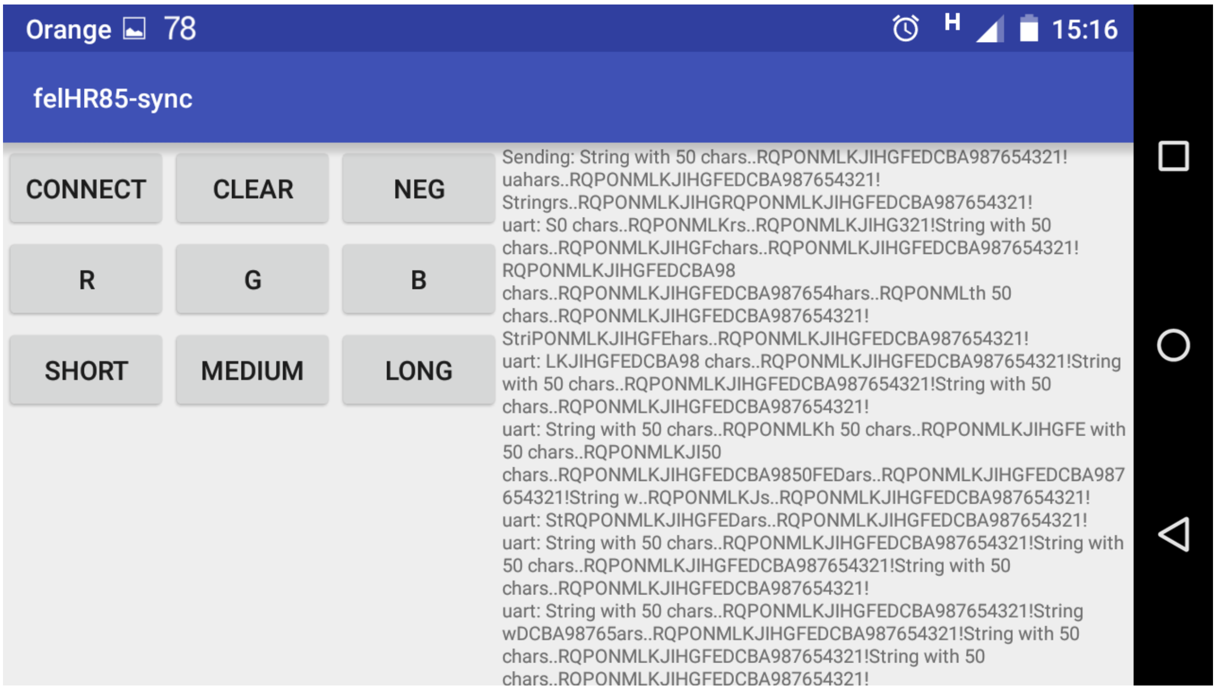
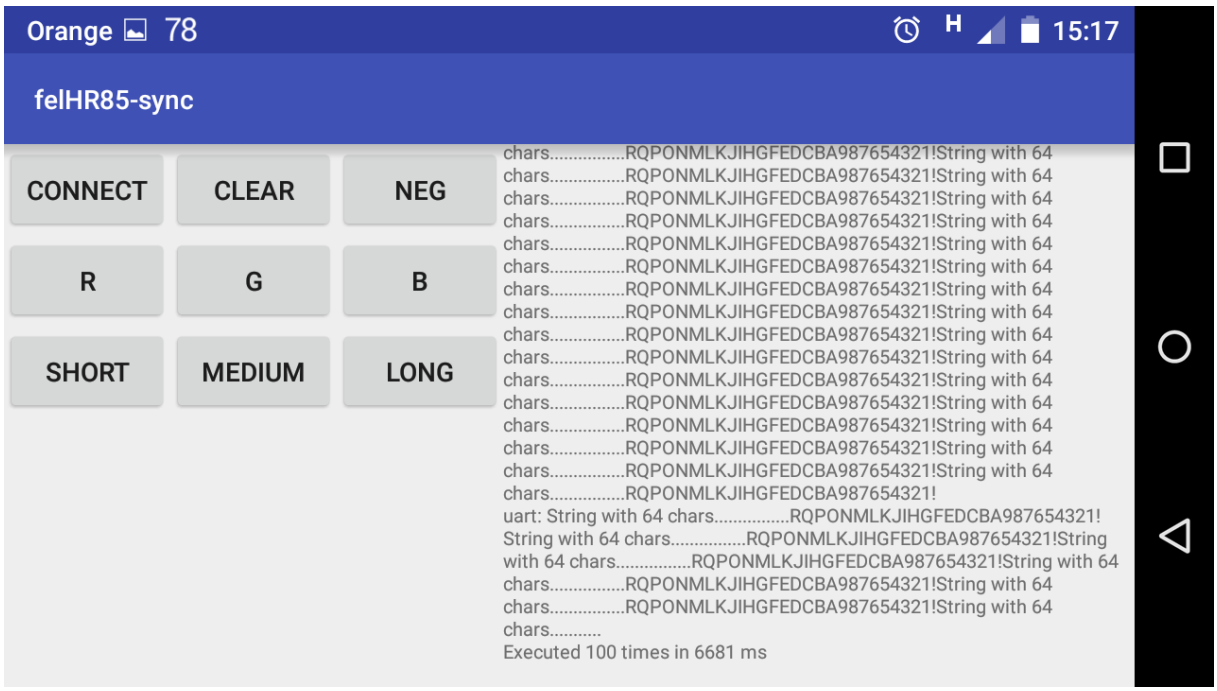


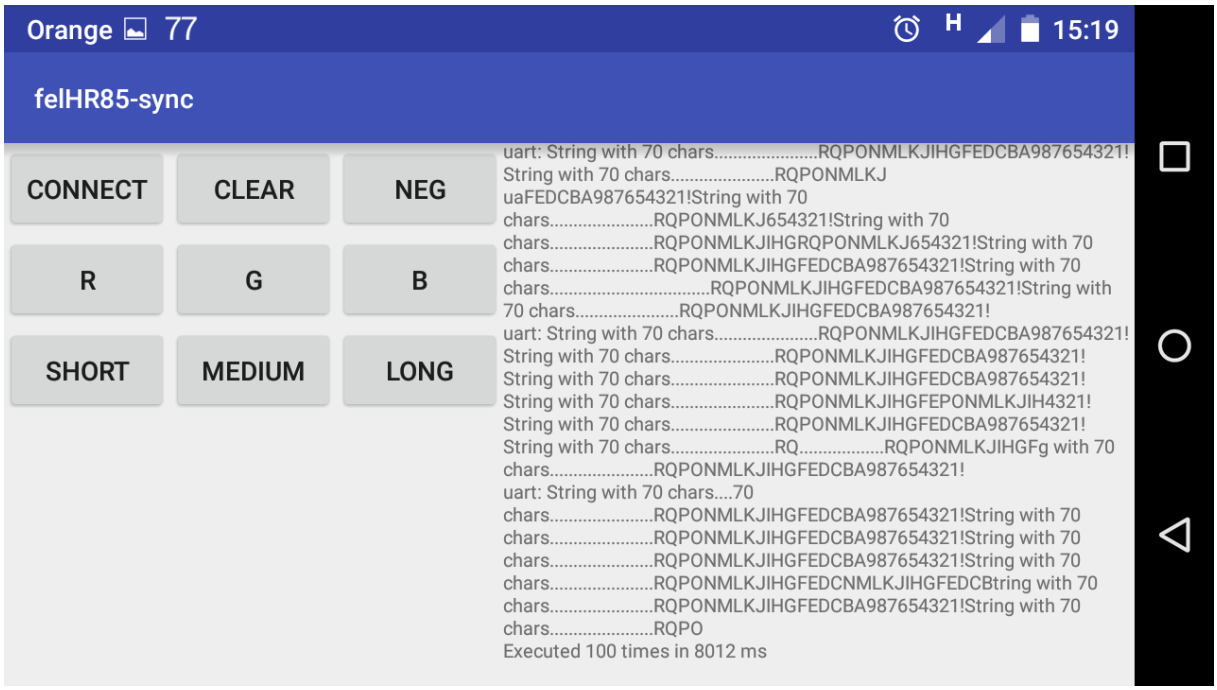
Figure 3.21: felHR85, sync-UART, 50 letters

This example is interesting, because text is shorter than buffer, and result is wrong. Worth noticing is fact, that “uart” text can be observed really often, and also that correct message appeared few times. Average execution time was still related to length of message: about 57 ms.



```
chars.................RQPONMLKJIHGFEDCBA987654321!String with 64
chars.................RQPONMLKJIHGFEDCBA987654321!String with 64
chars.................RQPONMLKJIHGFEDCBA987654321!String with 64
chars.................RQPONMLKJIHGFEDCBA987654321!String with 64
chars.................RQPONMLKJIHGFEDCBA987654321!String with 64
chars.................RQPONMLKJIHGFEDCBA987654321!String with 64
chars.................RQPONMLKJIHGFEDCBA987654321!String with 64
chars.................RQPONMLKJIHGFEDCBA987654321!String with 64
chars.................RQPONMLKJIHGFEDCBA987654321!String with 64
chars.................RQPONMLKJIHGFEDCBA987654321!String with 64
chars.................RQPONMLKJIHGFEDCBA987654321!String with 64
chars.................RQPONMLKJIHGFEDCBA987654321!String with 64
chars.................RQPONMLKJIHGFEDCBA987654321!String with 64
chars.................RQPONMLKJIHGFEDCBA987654321!String with 64
chars.................RQPONMLKJIHGFEDCBA987654321!String with 64
chars.................RQPONMLKJIHGFEDCBA987654321!
uart: String with 64 chars.................RQPONMLKJIHGFEDCBA987654321!
String with 64 chars.................RQPONMLKJIHGFEDCBA987654321!String
with 64 chars.................RQPONMLKJIHGFEDCBA987654321!String with 64
chars.................RQPONMLKJIHGFEDCBA987654321!String with 64
chars.................RQPONMLKJIHGFEDCBA987654321!String with 64
chars...........
Executed 100 times in 6681 ms
```

Figure 3.22: felHR85, sync-UART, 64 letters
Errors for 50 chars suggest, that longer messages should also have them, but... result is correct! Average execution time wasn't surprising: about 67 ms.



```
uart: String with 70 chars.......................RQPONMLKJIHGFEDCBA987654321!
String with 70 chars.......................RQPONMLKJ
uaFEDCBA987654321!String with 70
chars.......................RQPONMLKJ654321!String with 70
chars.......................RQPONMLKJIHGRQPONMLKJ654321!String with 70
chars.......................RQPONMLKJIHGFEDCBA987654321!String with 70
chars.................................RQPONMLKJIHGFEDCBA987654321!String with
70 chars.......................RQPONMLKJIHGFEDCBA987654321!
uart: String with 70 chars.......................RQPONMLKJIHGFEDCBA987654321!
String with 70 chars.......................RQPONMLKJIHGFEDCBA987654321!
String with 70 chars.......................RQPONMLKJIHGFEDCBA987654321!
String with 70 chars.......................RQPONMLKJIHGFEPONMLKJIH4321!
String with 70 chars.......................RQPONMLKJIHGFEDCBA987654321!
String with 70 chars.......................RQ..................RQPONMLKJIHGFg with 70
chars.......................RQPONMLKJIHGFEDCBA987654321!
uart: String with 70 chars....70
chars.......................RQPONMLKJIHGFEDCBA987654321!String with 70
chars.......................RQPONMLKJIHGFEDCBA987654321!String with 70
chars.......................RQPONMLKJIHGFEDCBA987654321!String with 70
chars.......................RQPONMLKJIHGFEDCNMLKJIHGFEDCBtring with 70
chars.......................RQPONMLKJIHGFEDCBA987654321!String with 70
chars.......................RQPO
Executed 100 times in 8012 ms
```

Figure 3.23: felHR85, sync-UART, 70 letters
This result is also surprising: errors started appearing again, but most of messages are correct. Actually, only 2 of errors are visible on first glance (it even looks better than badly formatted, but correct previous ones), but there's more of them. However, result is still better than for 50 chars, but of course it takes longer to send it: about 80 ms.

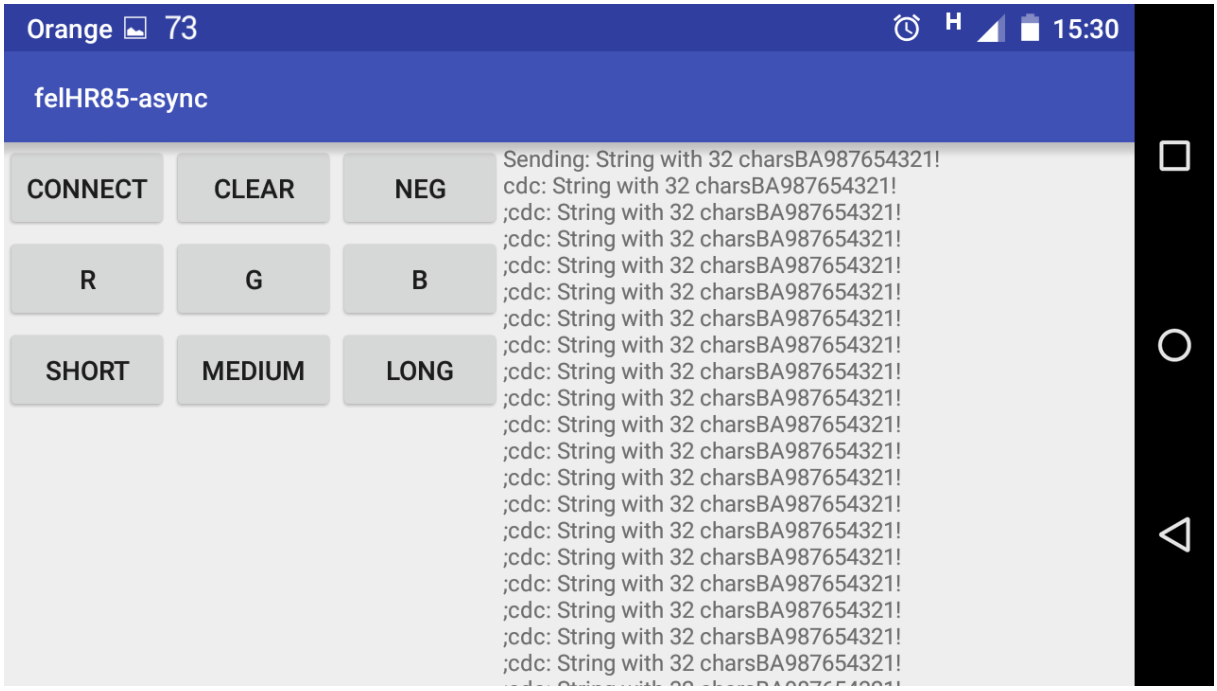


Figure 3.24: Division of message: async-CDC
Text is received in one message. Semicolon is visible on line with next message, because MCU sends new line character (“\n”), and semicolor is added on Android side after that.

3.4.3 Division of messages

As it was seen, MCU not always receives one message at once: in async-CDC example, few messages were sent together, and in (a)sync-CDC exceeding buffer, message was broken into two parts.

It’s also worth testing, how smartphone receives messages from MCU. It was visualized by putting a semicolon on end of each received block.

Results with conclusions can be seen on images 3.24-3.27. As it can be seen, CDC transmits data in blocks (with size not bigger than buffer), and UART does this letter by letter (and buffer is used for queuing them).

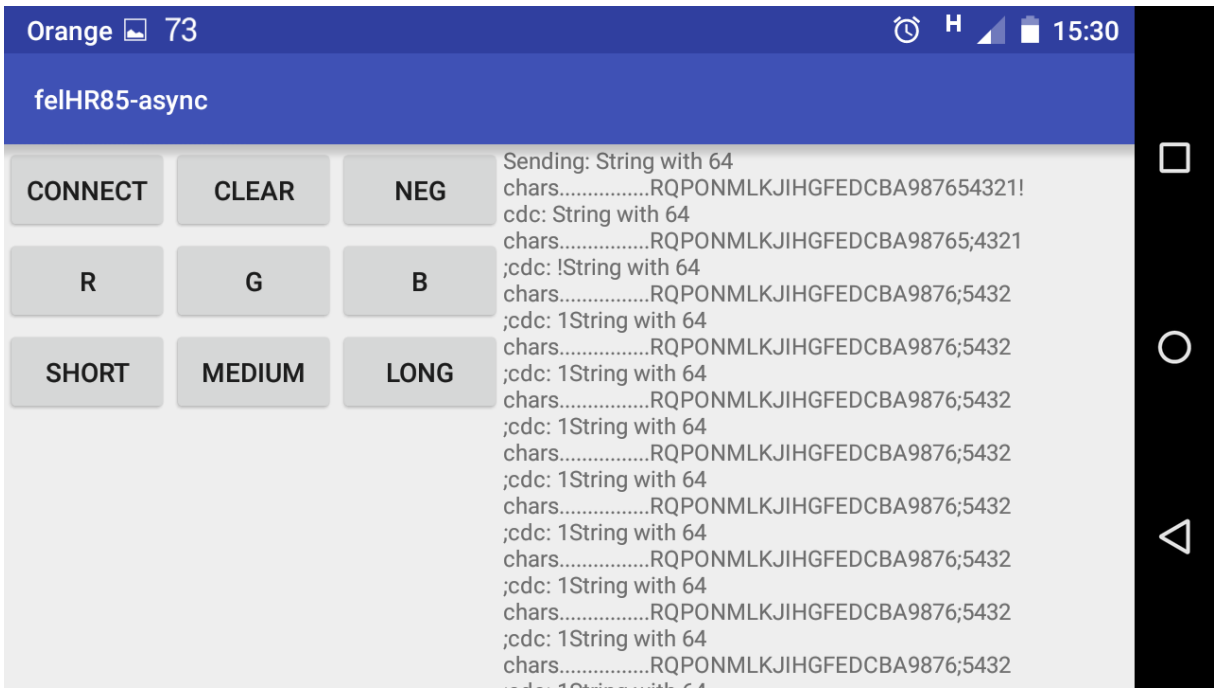


Figure 3.25: Division of message: async-CDC with too long message
It can be seen, what happens when message exceeds buffer. Each message is received as two: 64 chars long, and 5 chars long ("5432" + " \n").

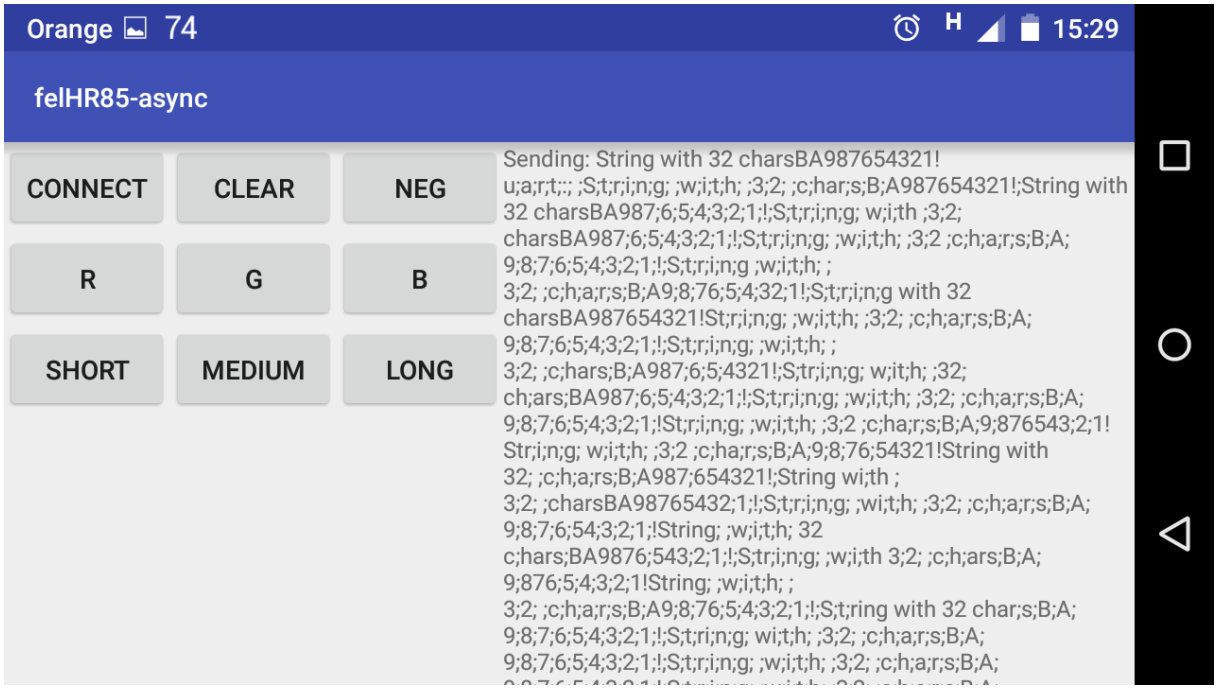


Figure 3.26: Division of message: async-UART
Message is usually send letter by letter, with few exceptions. As it was noted earlier, "uart" occurs only from time to time, which means, that MCU has something to read in buffer all the time.

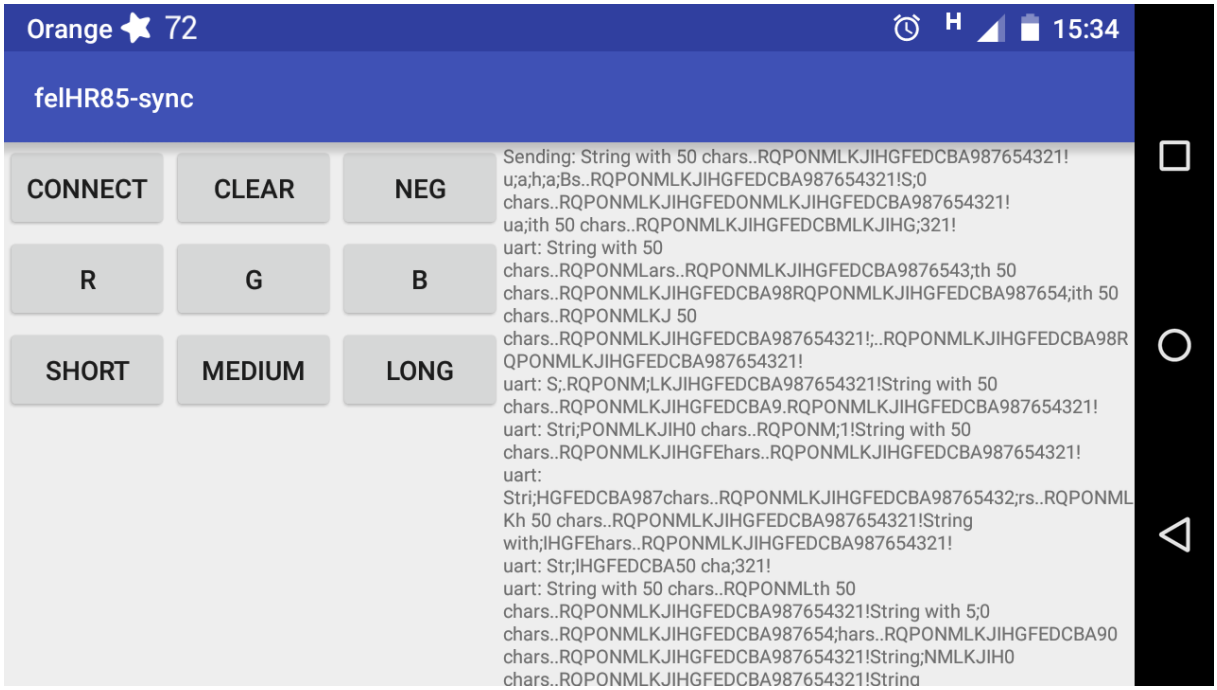


Figure 3.27: Division of message: sync-UART

MCU still reads and sends message char-by-char, but smartphone reads them in blocks. It can also be seen, that all blocks are correct, and errors can be seen only between them. Also in this case, “uart“ can be seen only from time to time, but more frequently than in “async” version.

3.5 Summary

Tests have shown, that it's possible to communicate between chosen smartphones and MCU using two libraries: USB Host API and USB Serial by felHR85.

Consolidated performance results are shown in table below:

	Xperia Neo - CDC			Moto G - CDC			Moto G - UART				
message length	1	32	50	1	32	50	1	32	50	64	70
USB Host API	11/22	12/22	12	11/22	12/22	12	not working				
felHR85 - async	11	11	12	11	11	12	2	34	52	67	73
felHR85 - sync	25	21	24	22	12	12	9	33	57	66	80

As it can be seen CDC offers the same performance for all libraries, smartphones, and message length, because it transmits whole buffer at once. Interesting is fact, that sometimes shorter message needed more time than longer. It limits message length to size of a buffer, but message is always correct. Therefore, if there's a need to send something longer, it should be divided into shorter messages, and then merged on the other device.

In case of using UART, longer messages can be sent, because they are send byte-by-byte anyway, and buffer is using for queueing them. Because of that, in UART there's a need to get received message from smaller blocks, so using some marker of message end could be a good solution. Another good solution could be sending some checksums, because errors were observed during tests (but only for using sync mode with it).

Big disadvantage of UART is also lack of compatibility with older smartphones, which is a problem especially because CDC on MCU is a much more complicated solution, and not every MCU supports it, or even have USB port. Advantage is fact, that UART is much more popular, and USB is only one of form, it appears on microcontrollers. However, doing it though USB openSDA port blocks possibility to use debugger on the same port.

Each found library has some advantages and disadvantages, which can be presented as a list:

- USB Host API:
  - \+ works with good results,
  - \+ no external libraries,
  - \+ low-level, many options for configuration,
  - – low-level, hard configuration,
  - – doesn't work with UART,
  - – only synchronous approach
- USB Serial by mik3y:
  - \+ popularity,
  - \+ should work with UART,
  - \+ high-level,
  - – only synchronous approach,
  - – external library,
  - – not compatible with tested MCU,

- USB Serial by felHR85:
  - + the same code works both CDC and UART,
  - + easy to use,
  - + reading using callbacks in async mode,
  - − external library (but can be easily added using jitpack.io),
  - − was harder to find than USB Serial by mik3y
  - − UART doesn't work on Xperia Neo (but problem is probably on phone's side).

Advantages and disadvantages of both protocols:

- CDC:
  - + compatibility with both smartphones,
  - + in KL26Z, openSDA port is still available for debugging,
  - + whole message sent at once,
  - + waits, until seconds device reads message,
  - − freezes, until seconds device reads message,
  - − limited length of message,
  - − requires USB in microcontroller,
  - − complex implementation in microcontroller,
- UART:
  - + popularity in microcontrollers,
  - + simple implementation in microcontroller,
  - + length of message is not limited,
  - + doesn't freeze, when message is not received,
  - − limited compatibility with smartphones
  - − in KL26Z, needs openSDA port for connection through USB cable, so debugger can't be connected using USB

Summing up, there is several ways to communicate between smartphone and microcontroller, which allows for using smartphone as high-level controller. Next step is to check, if this connection can actually allow for extending robot's functionality by something not available (or working worse) for microcontrollers.

Chapter 4

Sensors

4.1 Introduction

Modern smartphones have many sensors, and most of them can extend robot's functionality. Sensors differ between phones, and new (or more advanced) ones can be connected using possible connections (mostly USB and Bluetooth). Most popular ones are:

- touch screen,
- accelerometer,
- gyroscope,
- microphone(s),
- front and rear camera(s),
- position sensors:
  - GPS,
  - multilateration based on GSM and/or WiFi,
- magnetometer,
- light sensor,
- proximity sensor.

Some (mostly high-end, or specialized ones) smartphones have also sensors like electronic compass, humidity/temperature sensors, fingerprint scanner, or even thermal camera.

One of possible application can be using one of cameras for face detection, and then turning robot around to make it directly at front. Such functionality was implemented for "Face Follower" (img. 1.5) developed for one of previous projects. It was using "full" version of OpenCV in C++. Because Android is a different platform, the same method couldn't be used for it. Fortunately, Android offers few different ways for face detection, including:

- FaceDetector API,
- Camera API,
- OpenCV for Android (Java and NDK),

Figure 4.1: Image used for testing face detection

All methods are described in their own sections, and results are aggregated in table in summary sections. During development, search results for “face” on Google were used, and final results were obtained on image from [10], shown on img. 4.1.

4.2 FaceDetector API

First method is FaceDetector API, which is part of Android environment. Usage is simple (listing 4.1) but the problem is, that it requires image as a Bitmap, which is hard to get from a camera. Obtaining frame as a Bitmap actually takes more time than detecting faces on it, so it was tested on image already in memory. Result is shown on img. 4.2 - it can be seen, that it’s possible to detect many faces (on this image: all 20). Unfortunately, Android Reference [5] doesn’t describe, how face detection is done.

Summing up, this API is rather for working on photos, not video, also because it has rather weak performance.

4.3 Camera API

Next method is CameraAPI, which is also part of Android, and fortunately allows for easy working with camera preview (but still, it’s not described, how detection actually works). Actually, it’s an API to access camera, and face detecting is only part of it. This API is now also deprecated and replaced with Camera2 API, but because newer one doesn’t work with older Android versions (like version on Xperia Neo), older one was used. Configuration of preview is a little complicated, but after that, face detection requires only starting it, and defining a method, which should be invoked on each detection. Example is shown in listing 4.2.

Listing 4.1: Face Detector API

```
void detectFaces(Bitmap image) {
    // detection
    FaceDetector face_detector = new FaceDetector(image.getWidth(),
       image.getHeight(), MAX_FACES);
    FaceDetector.Face[] faces = new FaceDetector.Face[MAX_FACES];
    int faceCount = face_detector.findFaces(image, faces);
    // drawing
    Canvas canvas = new Canvas(image);
    canvas.drawBitmap(image, 0, 0, null);
    PointF tmp_point = new PointF();
    Paint tmp_paint = new Paint();
    tmp_paint.setColor(COLOR);
    tmp_paint.setAlpha(ALPHA);
    for (int i = 0; i < facesCount; i++) {
        FaceDetector.Face face = faces[i];
        face.getMidPoint(tmp_point);
        canvas.drawCircle(tmp_point.x, tmp_point.y,
           face.eyesDistance(), tmp_paint);
    }
}
```

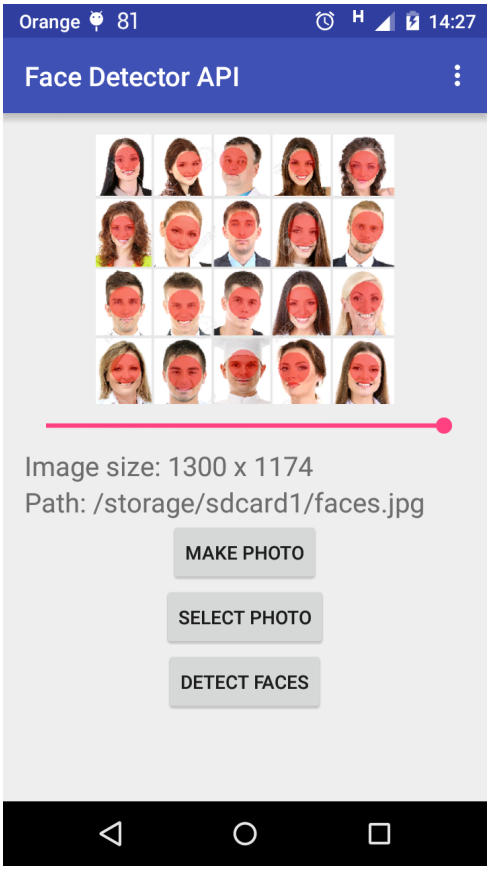


Figure 4.2: Face Detector API

Listing 4.2: Camera API

```
mCamera.startPreview();
mCamera.setFaceDetectionListener(new Camera.FaceDetectionListener() {
    @Override
    public void onFaceDetection(Camera.Face[] faces, Camera camera) {
        // no easy way to display it on image
        for(Camera.Face face : faces) {
            Log.i(FACE, face.rect.flattenToString());
        }
    }
});
mCamera.startFaceDetection();
```

Big plus is a fact, that detection automatically happens in background, so it doesn't freeze application, and that it runs fast (real-time on Moto G). On the other hand, it is hard to visualize effects, so during tests it only logged result (position, number of detected faces) to console. In opposition to FaceDetector API, Camera API is able to found only up to 5 faces.

4.4 OpenCV for Android

Original robot was using OpenCV library on Windows, and it's also available for Android, written in Java (actually, only interface is in Java, and there is C++ below it). Similarly to Windows, it's also hard to make it work, but when it's done, it offers a lot of capabilities.

It's possible to override method invoked on each frame (listing 4.3), and then do e.g. face detection on it. Such approach was not possible in Camera API - it was possible to invoke method on each frame, but its face detection couldn't be used in such case. OpenCV offers more control, and allows for doing more things with images, because it's a whole library created for that matter. It has methods to look for patterns on image using, e.g. Haar cascades. It's possible to create own ones, but library also provides several pre-defined ones - few different for faces, for eyes, even for face of a cat. In sample code, detection freezes application, but because it's possible to control, when and how it's invoked, it's possible to do it in separate thread (which was actually done) or even work on few frames simultaneously.

It's also easy to visualize effects on preview for camera. Results for high and low resolution can be found on img. 4.3 and 4.4. As it can be seen, even for really small resolution, quality of detection is still quite good, and it was almost 15 times faster (~1500 ms vs 105 ms). There is also no (or very high) limit for maximum number of detected faces.

4.5 OpenCV NDK

OpenCV on Android has also NDK (Native Development Kit) version. Using NDK instead of normal Android's SDK offers better performance, but working with it is much more complicated.

Unfortunately, Android Studio still have problems with development of NDK, and most tutorials to it focuses on older Eclipse ADT. OpenCV provides some example projects using NDK (and face detector is one of them), but they are created also for Eclipse ADT.

Therefore, it wasn't implemented and tested. Fortunately, there is already built example of

Listing 4.3: OpenCV

```
@Override // from CameraBridgeViewBase.CvCameraViewListener
public Mat onCameraFrame(final Mat aInputFrame) {
    // convert to grayscale
    Mat grayscaleImage;
    Imgproc.cvtColor(aInputFrame, grayscaleImage,
       Imgproc.COLOR_RGBA2RGB);
    MatOfRect faces = new MatOfRect();
    // detect faces
    cascadeClassifier.detectMultiScale(grayscaleImage, faces);
    // mark faces on frame from camera
    for (Rect faceRect : faces.toArray) {
        Imgproc.rectangle(aInputFrame, faceRect.tl(), faceRect.br(),
           COLOR, THICKNESS);
    }
    return aInputFrame;
}
```

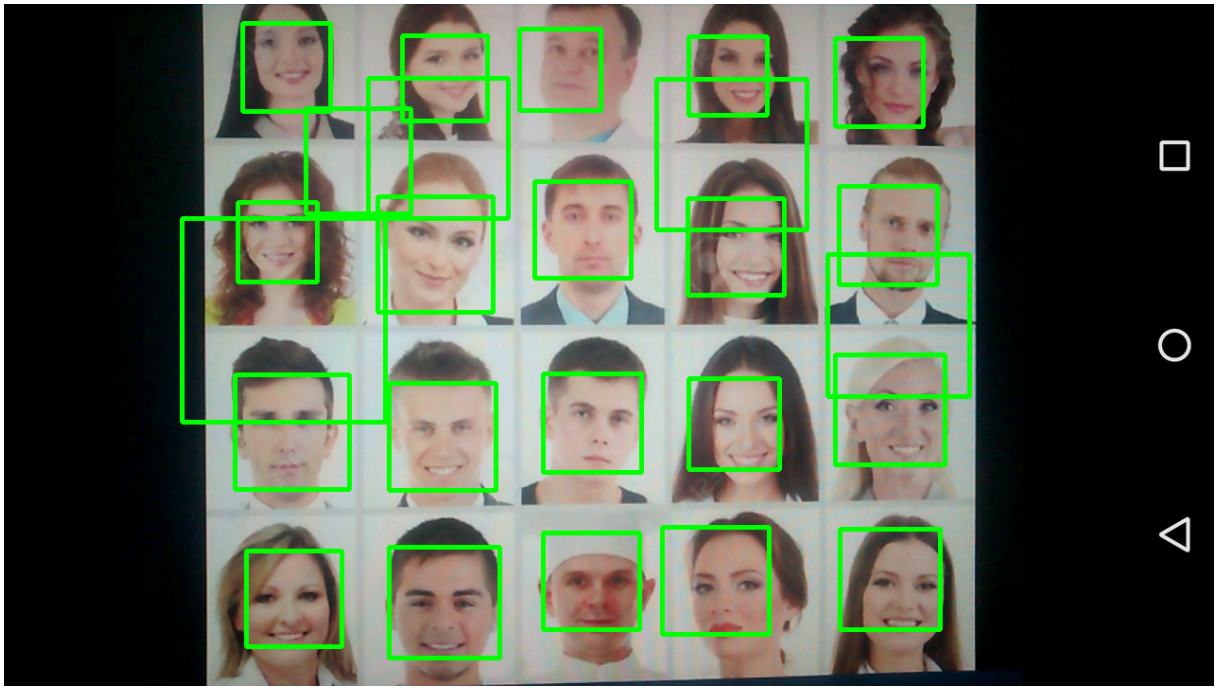


Figure 4.3: openCV - high resolution
All faces were detected, but also false matches can be seen. Faces were always found, and false matches occurred only from time to time.

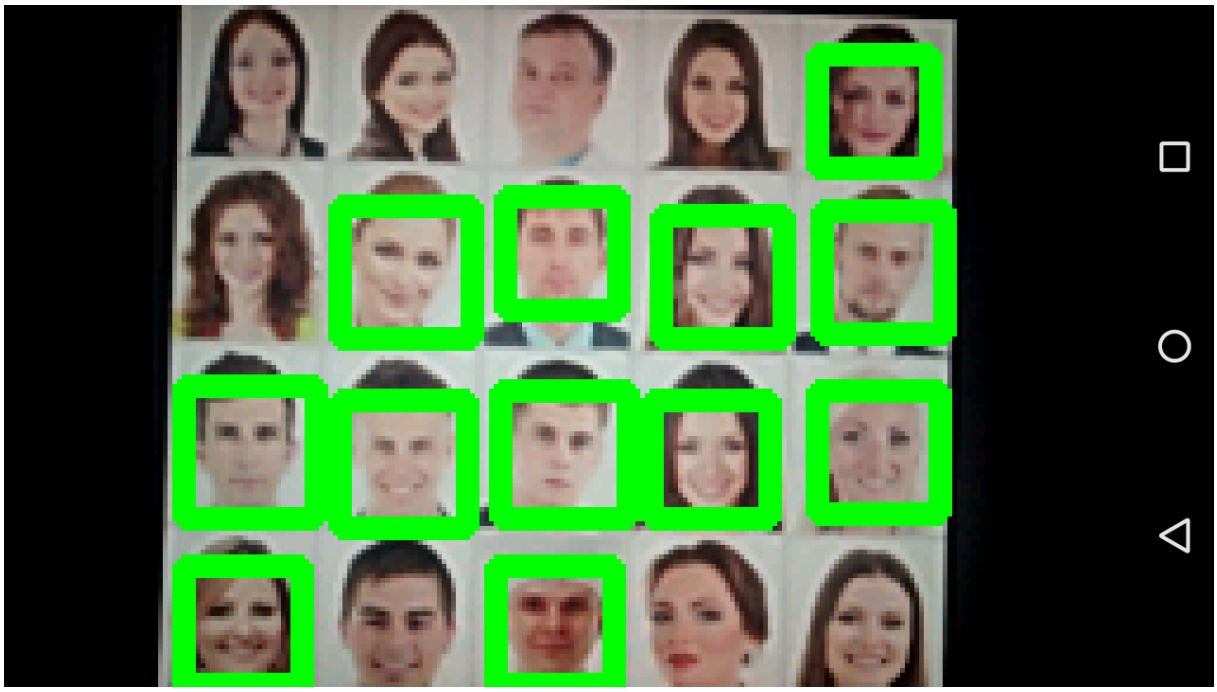


Figure 4.4: openCV - low resolution
Not all faces were detected, but there's also no false matches. Frame around face has the same width (3px) as on previous image, so resolution is really small.

Face Detector on Google Play, so it was possible to see, how it works (img. 4.5). It was impossible to measure time needed for detection, but it seemed to be (almost) smooth, so probably about 50 ms for frame with quite high resolution (also unknown).

4.6 Summary

As it was checked, it's possible to track faces using Android, and there's several ways to do it. It's also possible to do it in several ways, so they should be compared. Methods differ in functionality and ease of usage, but the most important is to compare performance.

To have meaningful results, they should be tested in the same conditions. Therefore, the same image (4.1) was used for all of them. Then, they were tested for different resolutions. During configuration of Camera API example, list of available preview size for both smartphones was obtained:

Moto G	Xperia Neo
1280 x 960	1600 x 1200
1280 x 720	1280 x 720
960 x 720	864 x 480
864 x 480	640 x 480
768 x 432	480 x 320
720 x 480	352 x 288
640 x 480	320 x 240
320 x 240	176 x 144
176 x 144	

As it can be seen, some resolutions are available for both smartphones, so they were the ones used during testing of Camera API. However, they couldn't be used for other methods. Face

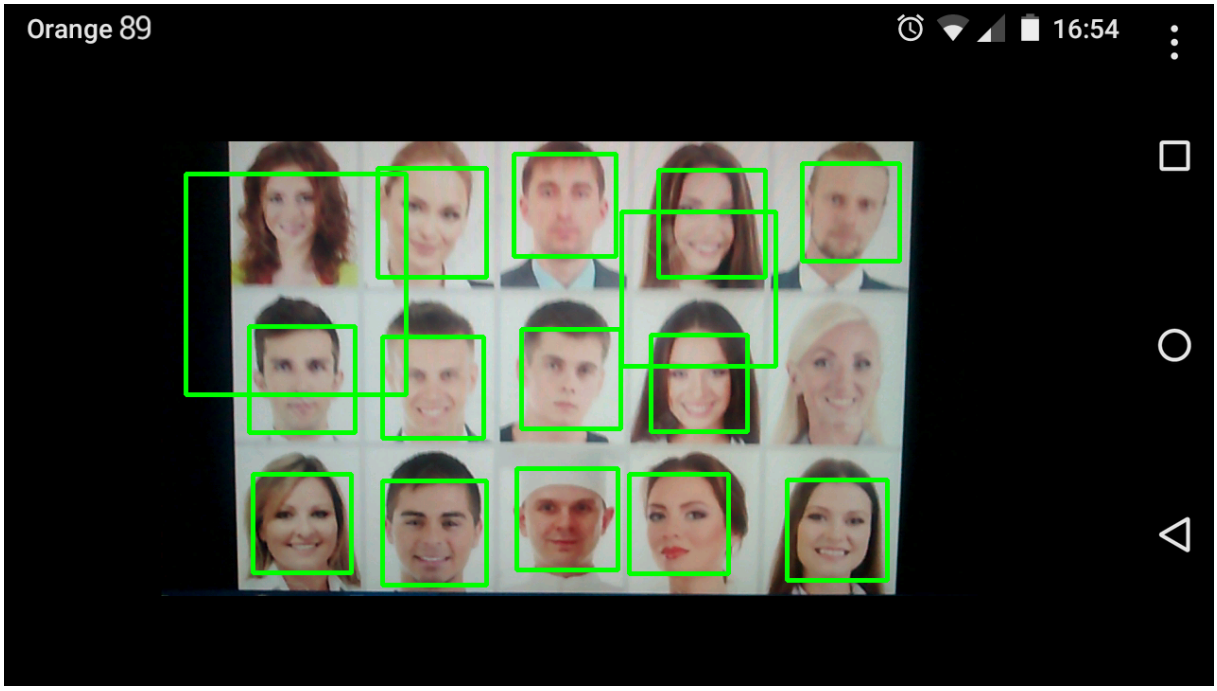


Figure 4.5: openCV - NDK
Application has limited screen area, so only 3 rows fitted. False matches also occurred, and not all faces were discovered correctly, but detection was working in almost real-time, and quality is still high.

Detector API doesn't work on camera preview, so file with the same faces was used, and scaled to have the same height as resolutions available to Camera API. On the other hand, in OpenCV only maximum width and height could be set, so it was set to the same as in Camera API, but actual resolution could be a little smaller. Methods working on camera preview were tested with image displayed on a FullHD computer display.

Average time needed for calculating single frame looks as follows (all values are in milliseconds):

	Xperia Neo			Moto G		
	FD	Camera	OpenCV	FD	Camera	OpenCV
1280 x 720	420	180	-	380	33	1500
864 x 480	390	110	-	320	33	1200
640 x 480	390	123	5000	320	33	1000
320 x 240	-	128	2500	-	-	275
176 x 144	-	101	500	-	-	105

As it can be seen, Camera API was the fastest. FaceDetector API was also fast, because its time was measured only for face detection, and obtaining it required much more time. OpenCV was slow, but in lower resolutions offered best quality. It can also be seen, that Moto G is executing exactly the same code about 5 times faster.

Each method has its advantages and disadvantages, and different method can be used, depending on requirements:

- FaceDetector API:
  - + easy to use,
  - + can work on image other than camera preview,

  + no face limit,
  + no external libraries,
  – poor performance,
  – hard to make it work with camera,

- Camera API:
  + easy to use,
  + doesn't freeze application,
  + no external libraries,
  + really fast,
  – max 5 faces
  – hard to visualize result on preview,

- OpenCV (Java):
  + allows for much more than face detection,
  + acceptable performance, especially for lower resolution,
  + no face limit,
  + easy to visualize results,
  + more control in developer's code,
  – more control in developer's code,
  – external library, hard to make it work,
  – slower than Camera API,

- OpenCV (NDK):
  + same as in Java, but with better performance and more control,
  – same as in Java, but much harder to work with it,
  – no Android Studio support.

Summing up, the best method of doing face detection is using Camera API, because it's fastest and simplest. FaceDetector API is inappropriate, because it's slow, and useful only for working with already captured picture, when Camera API can't be used.

OpenCV is also a good library for face detection, but much more complex (especially NDK), which is both huge advantage and disadvantage. It starts being really useful, when there is a need to make something different with an image, than face detection, because the rest offers only that.

Chapter 5

Summary

As it was shown in previous chapters, Android smartphone can gather some useful data for robot using sensors, and send it to it over USB cable. It offers also quite good performance, and is easy to implement. Also compatibility is great: application implemented for older phone was working without any changes on newer, completely different one, and boost in performance was visible. In MCU's case, moving from KL25Z to almost identical KL26Z required few changes in configuration.

However, using smartphone as a controller has a few disadvantages, especially if they would be used in "professional" area:

1. Communication seems to be immediately (<30ms), but it can be too slow for some usages.
2. Lot of performance is used by Android itself, and lot of applications and services running in background.
3. Android is a rather insecure system, with lot of viruses.
4. Modern smartphones are all-in-one devices, so more specialized devices will easily outperform them.

Therefore, it seems to be a bad idea to e.g. send a probe controlled by a smartphone to Mars, with viruses and music playing in background, and battery drained by continuous trying to connect with Facebook.

But all of listed problems doesn't actually matter, if they will be used for fun. It would be really nice to see already pre-built robots/drones/RC cars with USB port and API to extend their functionality. Some of them offers remotes in form of smartphone's application, so even an API for them could be useful.

Currently, if someone wants to control ones available in shops, can do it only with a remote, and controlling it programatically requires building own one almost from scratch. It is hard and time consuming, while it could be done much simpler and faster, if there would be available pre-built ones.

Bibliography

[1] Erich Styger, MCU on Eclipse, mcuoneclipse.com/, 29.04.2016

[2] Ganeev Singh, Mobile Controlled Robot, engineersgarage.com/contribution/mobile-controlled-robot, 29.05.2016

[3] Robotics Bible, Mobile Controlled Robot, roboticsbible.com/project-mobile-controlled-robot-without-microcontroller.html, 29.05.2016

[4] Mayoogh Girish, Mobile Controlled Robot, diyhacking.com/mobile-controlled-robot, 29.05.2016

[5] Google Inc., Android Reference, developer.android.com

[6] Itseez Inc., OpenCV Reference, http://opencv.org/

[7] Stack Overflow, http://stackoverflow.com/

[8] Mike Wakerly, usb-serial-for-android, github.com/mik3y/usb-serial-for-android, 20.05.2016

[9] Felipe Herranz, UsbSerial, github.com/felHR85/UsbSerial, 20.05.2016

[10] 123rf.com, Stock Photo - Collage of many different human faces, http://www.123rf.com/photo_24176012_collage-of-many-different-human-faces.html, 15.06.2016