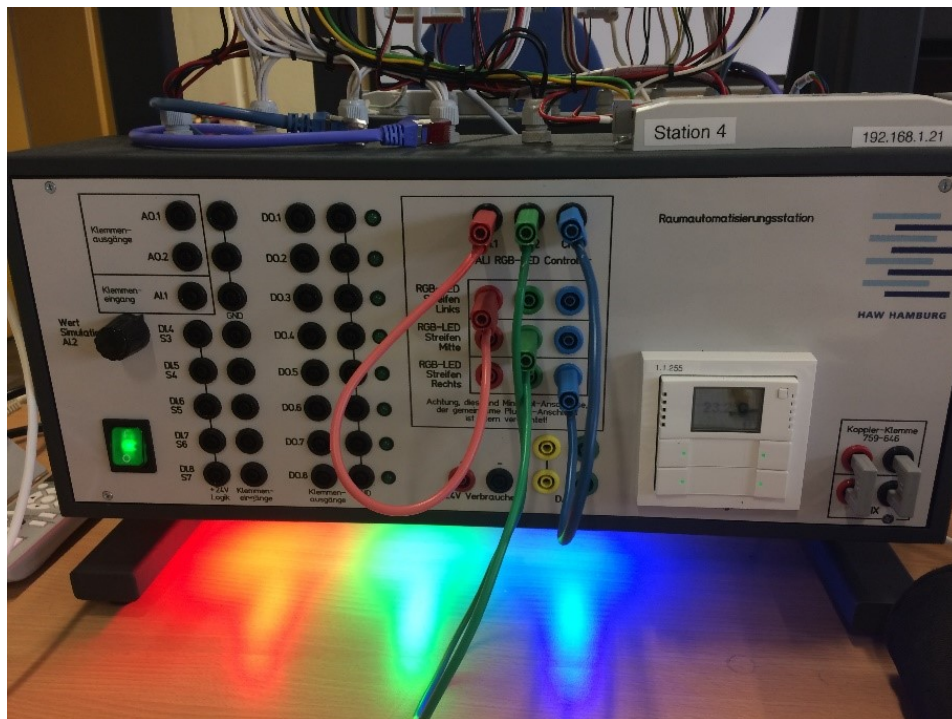


HAW Hamburg
Wintersemester 2016/2017

Bachelorprojekt

Dynamische Lichtsteuerung per Activity-App



Dennis Claren

Tobias Latta

Petrik Krajnovic

Betreuer: Herr Gräßner

Inhaltsverzeichnis

1	Einleitung	1
2	Aufgabenbeschreibung und Zielsetzung	2
3	Teammitglieder und Organisation	3
4	Codesys	4
4.1	vorbereitende Maßnahmen und Aufbau des Codesys-Projekts	4
4.2	Baustein „PLC_PRG“	4
4.3	Baustein „MODBUS_Kommunikation“	5
4.4	Baustein „Zuweisung_Array_to_Bit“	5
4.5	Baustein „Farbverknuepfung“	6
4.6	Baustein „Farbdimmung“	6
4.7	Baustein „DALI_Kommunikation“	6
4.8	Baustein „Aufwachphase“	7
5	Aufbau einer Android-App	8
5.1	Java Code (Brain)	9
5.1.1	Imports	9
5.1.2	Klassen	10
5.1.3	Methoden	11
5.1.4	Activities	12
5.2	XML Code (Body)	13
6	Benutzte Programme	15
6.1	Android Studio	15
6.2	Modbus Slave	16
6.3	Modbus Poll	16
7	Benutzte Bibliotheken	17
7.1	Jamod	17
7.2	BLE Android	18
8	Programmierung	18
8.1	Programmklassen	18
8.1.1	BluetoothGattExecutor; DeviceServicesActivity; DeviceScanActivity;BleService	18
8.1.2	ModbusService	19
8.1.3	setupModbus	19
8.1.4	setupActivity	20
8.2	net.wimpi.Modbus	20
8.3	Allgemeiner Programmablauf	20
9	Weiterführende Aufgaben	22

Abbildungsverzeichnis

1	Schematische Darstellung des Gesamtsystems	1
2	Graphische Oberfläche von Android Studio mit geöffnetem Projekt sowie Ordnerstruktur des Projekts im Windows Explorer	8
3	Den unterschiedlichen Bildschirmen in der "DynLiCo"-App lassen sich Activities zuordnen, die für ihre Ausprogrammierung zuständig sind.	12
4	Grafische Oberfläche von Android Studio 2.2 mit markierten Schaltflächen .	15
5	Grafische Oberfläche von Modbus Slave mit geöffnetem Verbindungsfenster in der rechten- und manipulierbarer Übertragungsbitmaske in der linken Bildschirmhälfte	16
6	Grafische Oberfläche von Modbus Poll mit geöffnetem Definitionsfenster für die Lese- und Schreibbefehle. In der linken Bildschirmhälfte werden Werte des Slaves empfangen und angezeigt	17

Tabellenverzeichnis

1	Teammitglieder	3
---	--------------------------	---

1 Einleitung

Es handelt sich um den letzten Statusbericht zu dem Bachelor-Projekt „Dynamische Lichtsteuerung per Activity-App“. Hierbei wird ein Bluetooth-fähiges Pulsmessgerät mit einem Smartphone gekoppelt, welches wiederum per MODBUS eine Gebäudeautomationseinheit steuert. Diese Einheit regelt dann RGB-LEDs und stellt so verschiedene Lichtszenarien ein. Im Zusammenhang mit der GPM-Vorlesung bei Frau Heinemann wurden verschiedene Techniken zur Aufgabenstellung, Verteilung und Strukturierung des Projekts angewandt. Zu Beginn findet eine Teamdefinition mit Mitgliedern und Umgangsregeln statt. Darauf folgend werden Methoden zur Risikoanalyse und Projektplanung angewandt. Grundsätzlich wird dieses Projekt gegliedert in: Informationsgewinnung aus dem Vorgängerprojekt, Ausprogrammierung von Pulsuhr- und MODBUS-Kommunikation mit Smartphone sowie dem Abschluss in Form von Funktionstests und Präsentation.

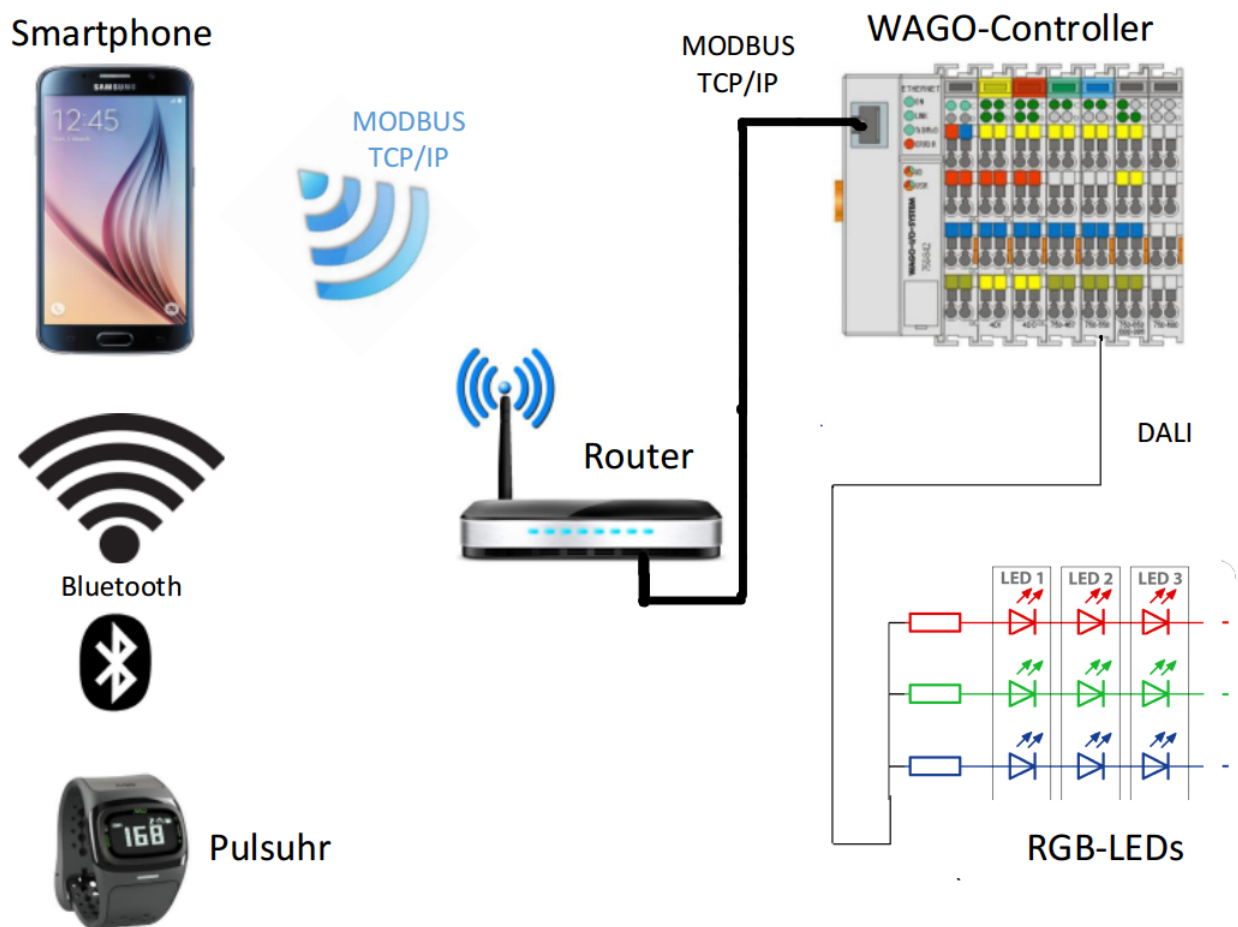


Abbildung 1: Schematische Darstellung des Gesamtsystems

2 Aufgabenbeschreibung und Zielsetzung

Das Ziel dieses Bachelorprojekts ist es Daten von einem Activity-Tracker (Uhr, Brustgurt oder Kombination aus beidem) auszulesen und über eine Bluetooth-Verbindung an ein Smartphone zu übertragen. Eine Android-App auf dem Smartphone dient als Steuerungsmöglichkeit und über MODBUS wird eine Verbindung zum WAGO-Controller hergestellt. Über den WAGO-Controller lassen sich nun mittels DALI/KNX-Bus LED-Lichtleisten ansteuern. Als Komponenten werden der WAGO-Controller 750-831 und die RGB-LED-Lichtleisten aus dem Inventar des Gebäudeeffizienz-Labors verwendet. Es wird an ein Bachelor-Projekt des SS 2016 angeknüpft. Ein besonderes Augenmerk liegt in der Verknüpfung verschiedener Übertragungsprotokolle.

Optimierungsbedarf besteht dabei insbesondere bei folgenden Punkten:

- Kommunikation zwischen Uhr und App über ANT+/Bluetooth
- Kommunikation zwischen App und WAGO-Controller über MODBUS
- Kommunikation zwischen WAGO-Controller und RGB-Leisten über DALI/KNX-Bus
- Funktionstest mit weiteren optionalen Szenarien (siehe unten)

Um die vollständige Kommunikation zu testen werden verschiedene Szenarien verwendet:

- der Puls steigt auf über 120 bpm -> warme Lichtfarbe
- der Puls liegt unter 120 bpm-> kalte Lichtfarbe

optional sind weitere Szenarien möglich, die als Zusatzaufgabe angesehen werden können:

- Puls längere Zeit bei ca. 50 bpm (Schlafphase)-> Licht aus
- Puls steigt nach Schlafphase wieder an (Aufwachphase)-> Lichtintensität steigt und kalte Lichtfarbe
- Musik wird wahrgenommen/Partymodus auf App einstellen-> Licht blinkt bunt

3 Teammitglieder und Organisation

Für die Organisation wird zum einen die Owncloud verwendet. Owncloud ist eine freie Software, welche das Speichern und Verwalten von Daten auf dem HAW-Server ermöglicht. Zum anderen wird die Kommunikation durch eine WhatsApp-Gruppe gewährleistet. Ein wöchentliches Treffen mit allen Teammitgliedern soll eventuelle Komplikationen verhindern und die Kommunikation innerhalb des Teams verbessern. Zusätzlich werden Termine mit dem Auftraggeber vereinbart und Statusberichte angelegt, welche zu festgelegten Zeitpunkten abgegeben werden müssen, damit der Fortschritt der Gruppe innerhalb des Projekts bekannt gegeben wird. Das Team besteht aus den wie folgt dargestellten Studenten:




Petrik Krajinovic	Tobias Latta	Dennis Claren
		
5. Semester Mechatronik		
petrik.krajinovic @haw-hamburg.de	tobias.latta1 @haw-hamburg.de	dennis.claren @haw-hamburg.de

Tabelle 1: *Teammitglieder*

4 Codesys

4.1 vorbereitende Maßnahmen und Aufbau des Codesys-Projekts

In dem Projekt wird das Protokoll MODBUS TCP/IP verwendet. Dazu ist es notwendig, dass sich alle Teilnehmer im gleichen Netzwerk befinden und so über ihre jeweiligen IP-Adressen kommunizieren(TCP/IPv4).

Es werden folgende Geräte verwendet:

- Fritzbox (Router): IP-Adresse 192.168.1.1
- Automatisierungsstation (WAGO-Controller): IP-Adresse 192.168.1.21
- Laptop: IP-Adresse 192.168.1.121
- Smartphone: IP-Adresse 192.168.1.125

Die Subnetzmaske wird auf 255.255.255.0 festgelegt.

Das Codesys-Projekt besteht aus den folgenden sieben Bausteinen:

- 1 „PLC_PRG“
- 2 „DALI_Kommunikation“
- 3 „MODBUS_Kommunikation“
- 4 „Lichtfarbenverknuepfung“
- 5 „Lichtdimmung“
- 6 „Aufwachphase“
- 7 „Array_to_Bit“

Zudem werden folgende Libraries benötigt, um die verwendeten Bausteine zu benutzen:

- „WAGOLibModbus_IP_01.lib“
- „DALI-647-02.lib“
- „SysLibTime.lib“
- „Scheduler_03.lib“
- „Visu-Scheduler03.lib“.

4.2 Baustein „PLC_PRG“

Der Baustein „PLC_PRG“ wird zu Beginn der Ausführung aufgerufen und enthält die sechs „Unterbausteine“.

4.3 Baustein „MODBUS_Kommunikation“

Der Baustein „MODBUS_Kommunikation“ enthält die Konfiguration des MODBUS-Masters. In unserer Anwendung fungiert der WAGO-Controller 750-831 als MODBUS-Master und das Smartphone als MODBUS-Slave. Der Begriff „Client“ ist hierbei gleichbedeutend mit dem Begriff „Master“ und der Begriff „Slave“ ist gleichbedeutend mit dem Begriff „Server“, da es sich um die Übertragung per MODBUS TCP/IP handelt. Zur Konfiguration wurde der Funktionsbaustein „ETHERNET_MODBUSMASTER_TCP“ aus der Library „WAGOLib-Modbus_IP_01.lib“ verwendet.

Dieser Baustein erhält alle nötigen Konfigurationsparameter. Besonders wichtig ist dabei beispielsweise die IP-Adresse des Servers(Slave). Zudem muss ein gemeinsamer Port festgelegt werden. Dabei ist zu beachten, dass normalerweise der Port 502 für die MODBUS-Kommunikation festgelegt ist. Allerdings ist es in unserem Anwendungsfall kaum möglich diesen Port zu verwenden, weil er durch das Smartphone-Betriebssystem aus Sicherheitsgründen für Apps gesperrt ist. Daher ist es nötig einen anderen freien Port zu verwenden. Die Ports über 1024 sind dabei nicht beschränkt und so wählen wir für unsere Anwendung den freien Port 1030 für die Kommunikation aus.

Der Function-Code wird auf FC03 festgelegt, weil wir mehrere Eingangsregister des Slaves auslesen wollen. Zudem muss festgelegt werden, welcher Adressbereich ausgelesen werden soll. Wir möchten zunächst nur ein Register (16Bit) auslesen, daher genügt es die Quantity auf „0x0001“ zu stellen. Achtung: Stellt man eine höhere Quantity ein, muss auch der Slave diese Adressbereiche beschreiben. Ist dies nicht der Fall kommt es in dem MODBUS-Telegramm zu dem Fehlercode 02. Ist der Slave aber auf den Master eingestellt, wird der Inhalt der ausgelesenen Adressen dann über einen ADR-Baustein in den Auslesepuffer geschrieben. Optional wäre es natürlich auch möglich mit diesem Baustein nicht nur auszulesen, sondern auch zu schreiben, wofür weitere Verknüpfungen am Baustein vorhanden sind. Über eine Einschaltverzögerung wird der Anforderungszyklus festgelegt. Da bei unserer Anwendung keine extrem schnelle Datenaktualisierung nötig ist, wählten wir eine Zeit von 100mS aus. Zur Kontrolle und simplen Visualisierung wird dieser Ausgang mit einer LED(%QX26.1) der Ausgangskarte verknüpft. Zudem sollte bei erfolgreicher Verbindung eine weitere LED(%QX26.0) dauerhaft leuchten. Tritt ein Fehler auf, kann der Fehlercode ausgelesen werden und die Bedeutung im WAGO-Handbuch zur eingesetzten Library nachgelesen werden.

4.4 Baustein „Zuweisung_Array_to_Bit“

In dem Baustein „Zuweisung_Array_to_Bit“ wird das ausgelesene Datenarray „RECEIVE_BUFFER[]“ des Slaves in einzelne Bits übertragen. Durch die Festlegung auf den Function-Code 03 lesen wir mehrere Eingangsregister aus. Ein Eingangsregister ist dabei 16 Bit breit. Daraus resultiert, dass das Array „RECEIVE_BUFFER[]“ ein Array aus Daten vom Typ WORD ist. Bei unserer Übertragung wollen wir nun die 16 Bits des ersten WORDS auslesen und benennen diese aufsteigend mit „RECEIVE_BITX“.

4.5 Baustein „Farbverknuepfung“

In dem Baustein „Farbverknuepfung“ wird dem jeweiligen ausgelesenen Bit eine Farbe zugeordnet. Es wird der Funktionsbaustein „FbDaliSwitchOnOff“ aus der Library „DALI_647_02.lib“ verwendet. Die anzusprechenden Betriebsgeräte erhält man durch Verwendung des DALI-Konfigurators und anschließendem Import der „.exp-Datei“. Der Dimmwert für den ausgeschalteten Zustand wird auf null eingestellt. Durch das Ansprechen des Broadcasts der Betriebsgeräte werden alle Farben angesprochen, wodurch ein weißes Licht einstellbar ist. Die Ansteuerung ist dabei etwas vereinfacht, weil nur ein Bit für die weiße Farbgebung gesetzt werden muss.

4.6 Baustein „Farbdimmung“

In dem Baustein „Farbdimmung“ wird, ähnlich wie beim Baustein „Farbverknuepfung“, ein empfangenes Bit einer Lichtfarbe zugeordnet. Allerdings wird die jeweilige Farbe nun auf einen einstellbaren Wert gedimmt.

4.7 Baustein „DALI_Kommunikation“

In dem Baustein „DALI_Kommunikation“ wird die Kommunikation des Controllers mit der DALI-Busklemme sichergestellt. Der Baustein „FbMaster753_647“ stammt aus der Library „DALI_647_02.lib“. Die Konfiguration findet mit dem Programm „WAGO DALI Configurator“ statt und ist in der Laboreinführung, sowie in den WAGO-Dokumentationen beschrieben. Bei erfolgreicher Konfiguration und Importierung sollte beim Feedbackausgang eine „0“ stehen!

4.8 Baustein „Aufwachphase“

Der Baustein „Aufwachphase“ kann als kleines Extrafeature unseres Projekts gewertet werden. Mit diesem Baustein ist es möglich zu einem gewissen Zeitpunkt/Zeitpunkten eine Aktion auszuführen. Um die darin enthaltenen Funktionsbausteine verwenden zu können, müssen zunächst die Bibliotheken „SysLibTime.lib“ und „Scheduler_03.lib“ von WAGO eingebunden werden. Über den Funktionsbaustein „SysRtcGetTime“ erhält man die aktuelle Uhrzeit. Diese muss zuvor einmalig im Webbrowser auf der IP-Seite des WAGO-Controllers eingestellt werden. Die aktuelle Uhrzeit wird dann in die Variable „actualTime“ vom Datentyp „dT“ gespeichert. Um möglichst viele Möglichkeiten der Ein- bzw Abschaltung zu haben, wird der Funktionsbaustein „FbScheduleSpecialPeriod“ aus der „Scheduler_03.lib“ verwendet. Für diesen Baustein wird logischerweise die aktuelle Zeit gebraucht. Zudem gibt es einen Enable-Eingang, der mit einem SR-Baustein verknüpft ist. Das achte Bit, das vom Smartphone gesetzt werden kann, setzt die Aufwachfunktion. Durch das neunte Bit kann diese Funktion wieder ausgeschaltet werden. Zudem muss eine Struktur vom Typ „typScheduleSpecialPeriod“ an den Baustein anknüpfen. Über diese Struktur werden alle einstellbaren Werte festgelegt. Darunter zählen in diesem Fall der Anfangszeitpunkt und das Anfangsdatum, der Endzeitpunkt und das Enddatum und die Wochentage, an denen die jeweilige Aktion ausgeführt werden soll. Eingestellt werden diese Parameter über eine Visualisierung, die zuvor hergestellt werden muss. Dazu benötigt man die Bibliothek „Visu_Scheduler03.lib“. Zum genaueren Vorgehen ist hierbei die WAGO-Dokumentation zu den verwendeten Bibliotheken sehr hilfreich.

In diesem Baustein wird nun der Ausgang beschaltet, falls der zuvor eingestellte Zeitpunkt erreicht wurde und der Benutzer über das Smartphone dieses Feature „freigeschaltet“ hat. In den nachfolgenden Netzwerken wird eine lichtfarbenbezogene Aufwachphase realisiert, indem zunächst nur das rote Licht eingeschaltet ist und nach weiterer Zeit die Farben Grün und Blau dazukommen. So soll ein beruhigteres Aufwachen ermöglicht werden.

Ausblick: Dies ist nur eine mögliche Anwendung, die sicherlich optimiert und auch erweitert werden kann. Die verwendete Bibliothek bietet dabei sogar die Funktion, dass die Aufweckzeit nicht über eine Visualisierung eingestellt werden muss, sondern ebenfalls über MODBUS eingestellt wird. So wäre eine Aufwachfunktion, eine Einschlafphasenfunktion oder aber auch die Einstellung des gewünschten Lichts zu gewissen Zeiträumen bequem vom Smartphone aus möglich.

5 Aufbau einer Android-App

Grundsätzlich lässt sich eine Android-App in zwei Gruppen aufgliedern.

Auf der einen Seite befindet sich das klassische Programm, welches alle logischen Prozesse verarbeitet und mit Bibliotheken, Variablen sowie allen weiteren bekannten Programmierwerkzeugen ausgestattet ist.

Was nun im Vergleich zu ANSI-C oder C++ hinzu kommt, ist die Darstellung des Programms.

Die von Programmieren 1 und 2 bekannte Konsole wird nun durch selbst definierte Fenster und Buttons ausgetauscht, die eine Visualisierung ermöglichen.

Die Datenstruktur ist zudem etwas verschachtelt, was am Anfang zu Verwirrung führen kann, wenn man normalerweise nur mit Quell- und Header-Dateien arbeitet.

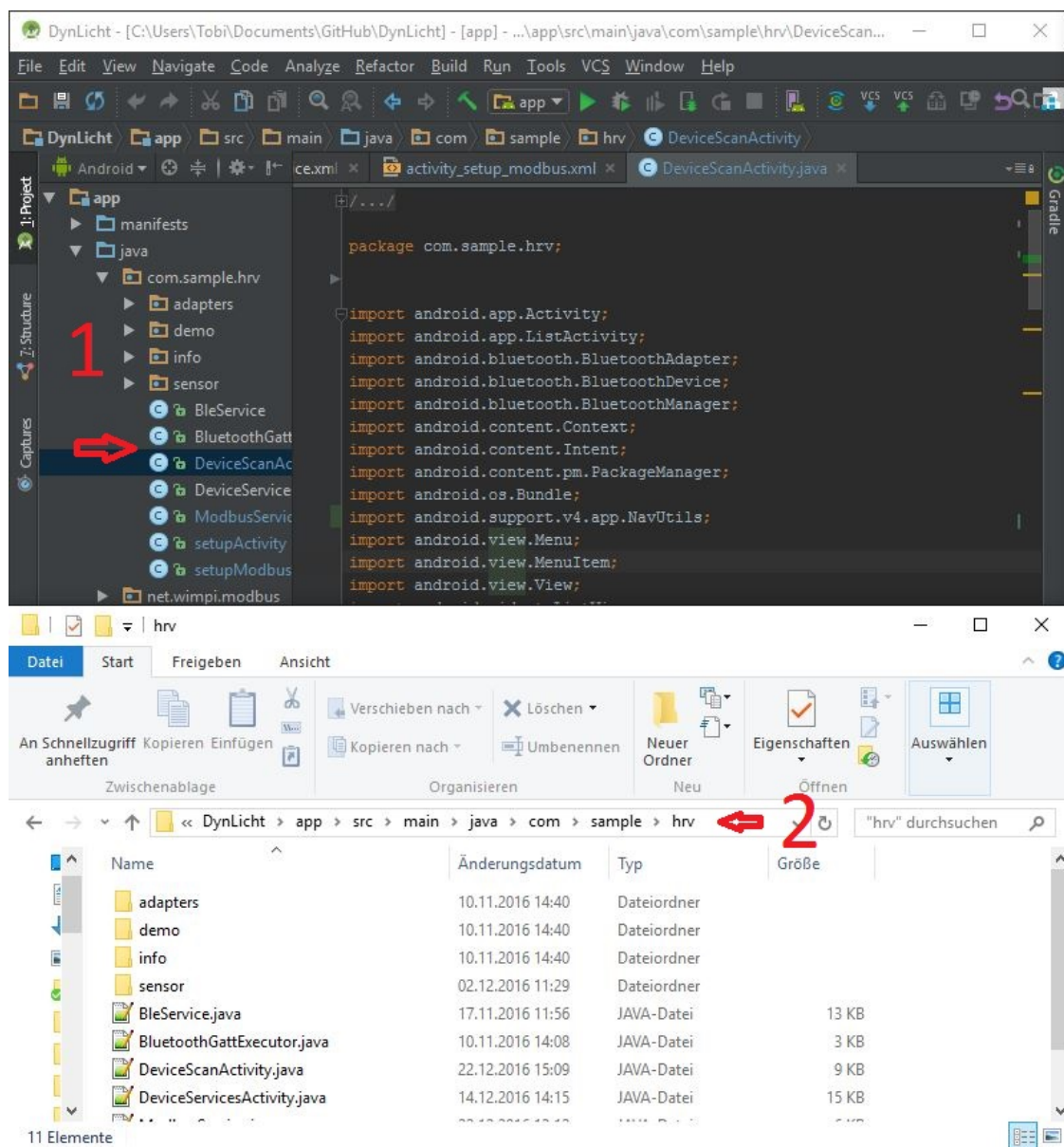


Abbildung 2: Graphische Oberfläche von Android Studio mit geöffnetem Projekt sowie Ordnerstruktur des Projekts im Windows Explorer

Hier lässt sich nun die Ordnerstruktur von Android-Projekten in der Entwicklungsumgebung(IDE) sowie in dem Datei-Explorer von Windows erkennen. Im linken Bereich der oberen Bildschirmhälfte (1) lässt sich der ausgeklappte Pfad zu den Java-Klassen (Pfeil nach rechts) ablesen. Hier wird hauptsächlich gearbeitet, wenn die Java-Programmierung stattfindet.

Der gleiche Ordner ist in der unteren Bildschirmhälfte im typischen Windows-Dateipfad zu erkennen.(2)

5.1 Java Code (Brain)

Im Folgenden werden die wichtigsten Namens- und Funktionsunterschiede von Java in Bezug auf ANSI-C erklärt. Für eine bessere Veranschaulichung findet dies am Beispiel eines "Hauses" statt. Weiterführende Erklärungen folgen.

5.1.1 Imports

Bei C sind es Bibliotheken und in Java nennen sie sich Imports:

Von Extern programmierte Teile, die Funktionalitäten bereitstellen, welche für den eigenen Code in der jeweiligen Klasse wiederverwendet werden können. Die Quellen für diese Imports können sowohl lokal als auch aus dem Internet stammen. Möchte man auf Methoden oder Variablen von fremden Klassen zugreifen muss man ebenfalls einen Import aufstellen. Wo sich die Ursprungs-Klasse befindet, lässt sich anhand des kompletten Ausdrucks hinter dem "import" ablesen.

In unserem Projekt wird beispielsweise in der DeviceScan-Klasse auf eine lokal abgespeicherte Klasse mit folgender Zeile zugegriffen:

```
import com.sample.hrv.adapters.BleDevicesAdapter;
```

Die ersten drei Begriffe hinter dem *import* bezeichnen das Ursprungs-Package. Ein Package fasst mehrere Klassen nach einem Themengebiet, oder einer Anwendungen zusammen. Innerhalb eines Packages müssen die Klassennamen eindeutig vergeben sein, jedoch kann es in zwei unterschiedlichen (bsp. Bluetooth- und WLAN-)Packages Klassen geben, die beispielsweise die Datenübertragung realisieren.

Mit Bezug auf das Beispiel handelt es sich somit um den Katalog eines jeden Architekten mit unterschiedlichen Haus-Typen, die gebaut werden könnten. Jeder Architekt hat beispielsweise ein *HausGross*, ein *HausMittel* und ein *HausKlein*. Der Architekt muss keinen exklusiveren Namen für seine Baupläne haben, da diese aus seiner Sicht eindeutig bestimmt sind. Trifft er sich nun mit einem Kollegen, der zufälligerweise auch drei Häuser in seinem Dossier hat und sollen beide Architekten nun für einen Auftraggeber zwei *HausKlein* bauen, muss explizit definiert werden, um wessen *HausKlein* es sich jeweils handelt.

Mit dem Update der Android-Studio IDE wird ebenfalls das API ("application programming interface" -> Programmierschnittstelle) aktualisiert. Diese Schnittstelle liefert weitere Bibliotheken, mit denen die gewünschten Funktionen der Klasse realisiert werden.

Als Beispiel hierfür dienen folgende Importe, die für eine Bluetooth-Funktionalität der Klasse DeviceScanActivity benötigt werden und allesamt von Google stammen:

```
import android.bluetooth.BluetoothAdapter;  
import android.bluetooth.BluetoothDevice;  
import android.bluetooth.BluetoothManager;
```

5.1.2 Klassen

Im klassischen Sinn der Objektprogrammierung handelt es sich bei den Methoden um die Funktionen der einzelnen Objekte, die ausgeführt werden können, nachdem von einer bestimmten Klasse eine Instanz erstellt wurde.

In Bezug auf das Beispiel lässt sich sagen, dass es sich bei der Klasse um den Bauplan eines Hauses handelt. Mit diesem Plan können mehrere Häuser (Instanzen beziehungsweise Objekte) gebaut werden.

Durch die Kapselung der Methoden in unterschiedliche Objekte findet auch eine Abgrenzung der Parameter oder Variablen statt. Dies hat bei größeren Programmen den Vorteil, dass Zugriffe auf Systemvariablen und sogar Methoden begrenzt werden können.

Das blaue *protected* gewährt den Zugriff für die Klasse selbst sowie weitere Klassen aus dem Package. Auch Subklassen können in diesem Fall auf die Methode zugreifen. Allgemein wird somit die Sicherheit sowie Stabilität gesteigert.

Ein Zugriff findet grundsätzlich nicht direkt auf Variable XY statt, sondern meist wird eine kleine Methode in der Mutter-Klasse geschrieben, welche einen einzigen Zweck hat. Den Wert der Variable XY über *return* zurückgeben. Dadurch kann an benötigter Stelle ein versehentliches Überschreiben vermieden werden, wenn nur die Methode aufgerufen wird. Ein geeigneter Methoden-Name wäre dann *getVarXY*. Auch für das gewollte Setzen von Variablen gibt es eine Art von Methoden(*setVarXY*), genannt *Setter*.

5.1.3 Methoden

Am Beispiel: Jedes Haus erfüllt mit seiner Existenz mehrere Funktionen. Zum Beispiel *SchutzBieten*, *MöbelVerwahren* oder auch *FinanzielleAbsicherung*. Diese Funktionen, welche zu der Klasse *Haus* gehören, werden Methoden genannt und können nur ausgeführt werden, wenn es von *Haus* auch eine Instanz, also ein tatsächlich errichtetes Objekt gibt. Die Namensgebungs-Konvention ist darüber hinaus bei Methoden in Java sehr angenehm. Es wird sehr stark darauf geachtet Methodennamen auszuwählen, die schon über die Funktionalität beziehungsweise den Auslöser der Methode Aufschluss liefern.

Hier wird auf eine Methoden-Programmierung der Klasse *DeviceScanActivity* verwiesen:

```
@Override
protected void onResume() {
    super.onResume();

    // Ensures Bluetooth is enabled on the device. If Bluetooth is not currently enabled,
    // fire an intent to display a dialog asking the user to grant permission to enable it.
    if (!bluetoothAdapter.isEnabled()) {
        final Intent enableBtIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
        startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);
        return;
    }
    init();
}
```

Die Methode *onResume* wird bei Rückkehr in die Klasse ausgeführt. Die blau markierten Stichwörter sind programmiersprachen-spezifische Kommandos.

Inwiefern bei eine Klasse geöffnet und geschlossen werden kann und was für eine Bedeutung die blau markierten Stichwörter haben, wird nachfolgend erläutert.

5.1.4 Activities

Eventuell ist in Abbildung 2 schon aufgefallen, dass die Klassen-Namen auf [...]Activity enden. Grundsätzlich können diese *Activities* auch als Bildschirme der Android-App verstanden werden.



Abbildung 3: Den unterschiedlichen Bildschirmen in der "DynLiCo"-App lassen sich Activities zuordnen, die für ihre Ausprogrammierung zuständig sind.

Die setupActivity fungiert als Art Hauptmenü, aus welcher die Bluetooth-Activity und die setupModbus(Activity) aufgerufen werden kann. Um das System in Betrieb zu nehmen müssen diese beiden Verbindungen aufgebaut werden.

Es lässt sich jetzt in dem Code der setupActivity beispielsweise die Reaktion auf das Drücken des Buttons *BluetoothSetup* erkennen:

```
final Button blesetup = (Button) findViewById(R.id.but_menu_ble);
blesetup.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        final Intent intent = new Intent(setupActivity.this, DeviceScanActivity.class);
        startActivity(intent);
    }
});
```


Es wird ein Objekt der Klasse `Button` mit dem Namen *blesetup* erzeugt. Dieses Objekt verfügt über die Methode *setOnClickListener*, welche ihren Code ausführt, sobald der Button gedrückt wird. In diesem Falle wird ein Wechsel der vorliegenden Activity über ein neues Objekt der Klasse *Intent* ausgelöst. *newIntent(setupActivity.this, DeviceScanActivity.class)* definiert die zu verlassende und die zu öffnende Activity. Anschließend wird die Aktion mit *startActivity(intent)*; ausgelöst.

5.2 XML Code (Body)

Mit dem Button wurde im letzten Kapitel schon ein grafisches Objekt eingeführt, welches visuell repräsentiert wird. Die Verbindung zwischen dem "Brain" sowie dem "Body" findet sich in dem Code-Auszug in der ersten Zeile. Der "Code-Button" wird mit dem "Body-Button" über die ID (*findViewById(R.id.but_menu_ble)*) verbunden. Der Button nennt sich *but_menu_ble* und gehört zu dem Programm-Package *R.id*.

Innerhalb der "Body"-Datei, also dem XML-File (*eXtensibleMarkupLanguage*) wird dann das Aussehen des Buttons, beziehungsweise der kompletten Activity beschrieben.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/activity_setup"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:orientation="vertical"
    tools:context="com.sample.hrv.setupActivity">

    <LinearLayout
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"

        android:weightSum="1">

        <Button
            android:text="@string/menu_ble"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:id="@+id/but_menu_ble" />

        [...]

    </LinearLayout>
```

Der letzte Block definiert das Objekt `Button` mit den wichtigen Charakteristika: Die Beschriftung `BluetoothSetup`, wird nicht direkt beschrieben. Es findet eine Verknüpfung zu der `string.xml`-Datei statt, in welcher der Wert unter `menu_ble` abgerufen wird. Diese vermeintlich komplizierte Auslagerung der String-Werte hat den Vorteil, dass man bei weiteren Knöpfen mit gleicher Beschriftung einfach den gleich Pfad angeben kann und somit nicht jedes Mal den Wert neu definieren braucht. Möchte man seine Applikation beispielsweise für einen anderen Kommunikations-Standard umschreiben, braucht man in dem Fall nur die String-Werte in der `string.xml` ändern und schon wird jede Activity/ jedes Layout die aktuelle Beschriftungen haben.

Die Größe des Buttons ist nicht festgesetzt und kann für jedes Objekt variabel festgelegt werden. Es bietet sich an neben absoluten Höhen- und Weite-Werten, eine Größe zu wählen, die Bezug auf andere Layouts nimmt. Der Ausdruck `android : layoutheight = "wrap_content"` legt fest, dass die Höhe so hoch gewählt wird, wie auch der Inhalt ist. Somit ändert sich die Button-Höhe falls die Größe der Beschriftung geändert wird. Bei der Weite wird sich auf "elterliche" Werte bezogen. Es ist wichtig zu wissen, dass die Reihenfolge der Programmierung an dieser Stelle maßgeblich ist.

Betrachtet man die zweite Zeile des Programm-Ausschnitts fällt auf, dass ein `LinearLayout` definiert ist. Dieses Layout bildet die oberste Ebene. In Zeile 14 wird ein weiteres `LinearLayout` definiert, dass dem ersten untergeordnet ist. Werden hier Charakteristika festgelegt, gelten sie für weitere `Kinder`, jedoch nicht für das `Parent – Layout` aus Zeile 2.

Eventuell wird hier deutlich, was es mit der `layout_width = "match_parent"` auf sich hat. Die Weite-Eigenschaft wird von dem übergeordneten Layout übernommen. Dies hat den Vorteil, dass weitere Buttons mit der gleichen Eigenschaft definiert werden können und alle eine einheitliche Breite haben.

Grundsätzlich werden innerhalb eines `LinearLayouts` die einzelnen Inhalte entweder vertikal oder horizontal angeordnet. Die Fläche des Layouts wird im oberen Bereich definiert. Die Objekte verfügen dann über eine Gewichtung, mit der sie die Befugnis bekommen bestimmte Flächen einzunehmen. In unserem Beispiel findet die Knopfausrichtung vertikal statt und die Höhe, welche dann über Gewichtung geregelt werden könnte, ist mit dem `"wrap_content"` begrenzt. Wäre diese Zeile auskommentiert, würde eine Aufteilung über die Gewichtung der einzelnen Objekte vorliegen. Sofern nicht genauer festgelegt, haben die Objekte die gleiche Priorität und der Platz wird gleichmäßig aufgeteilt.

Neben dieser Art von Layout kann man seine Objekte auch relativ zueinander in einem `RelativeLayout` ausrichten. Dies bietet sich an wenn Text-, Grafik- oder Eingabefelder in konkretem Bezug zueinander stehen sollen oder allgemein eine unabhängigere Positionierung notwendig ist. Hierbei wird keine konkrete Gewichtung, also Priorisierung vorgenommen.

6 Benutzte Programme

6.1 Android Studio

Um eine Android-App zu programmieren bietet es sich an die kostenfreie Entwicklungsumgebung von Google zu verwenden, die ständig aktualisiert wird und über einen exzellenten Support verfügt. Nachfolgend werden die wichtigsten Schritte zur IDE-Installation besprochen und eigene Kniffe weitergegeben, die sich bei Inbetriebnahme als sinnvoll herausgestellt haben.

Von der offiziellen Entwickler-Website lässt sich die aktuelle Version der IDE mit notwendigen SDK-Dateien herunterladen. Sollte dieser Link nicht mehr aktuell sein, wird man mit etwas Recherche über Google die Installer-Datei finden können. Ist die Datei komplett heruntergeladen, sollte man die .exe-Datei ausführen und favorisierte Speicher-Orte etc. festlegen. Treten Probleme auf, ist die Dokumentation im Internet sehr gut. Man bekommt auf fast alle Fragen eine Antwort und ist sicherlich nicht der Einzige, der jeweilige Probleme hat.

Beim ersten Start der Entwicklungsumgebung muss noch das *VirtualDeviceImage* heruntergeladen werden, welches die App auf einem emulierten Android-Device abspielen lässt. Verfügt man über ein reales Android-Device, kann dieses ebenfalls zur Ausführung der Applikation verwendet werden. Mit der Modellbezeichnung lassen sich im Internet die benötigten Gerätetreiber finden. Entwicklereigenschaften lassen sich auf dem Endgerät derzeit freischalten, indem man über den "Über das Telefon"-Reiter seine *Build – Nummer* mehrmals anklickt. Nach etwa fünf Klicks wird jeweils eine Meldung im unteren Bildschirmbereich eingeblendet, die Informationen darüber gibt, wie oft noch gedrückt werden muss bis die Optionen freigeschaltet sind.

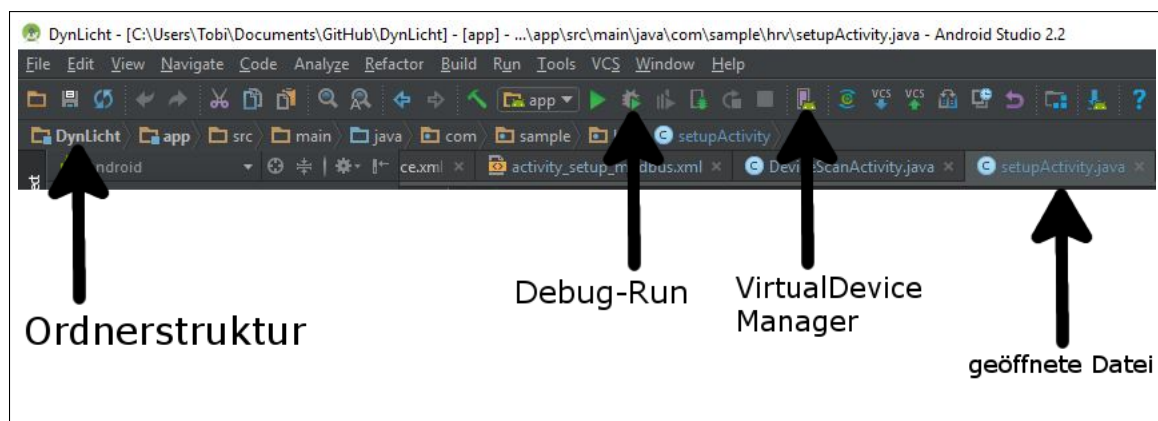


Abbildung 4: Grafische Oberfläche von Android Studio 2.2 mit markierten Schaltflächen

Als weiteren Tipp kann die Online-Universität UDACITY empfohlen werden, die unter anderem von Google selbst ins Leben gerufen wurde. Es werden kostenpflichtige- aber auch kostenfreie Kurse in unterschiedlichen Bereichen angeboten.

Von VR und KI-Entwicklung über Autonomes Fahren bis zur Vermittlung von klassischen Programmierfähigkeiten werden einige Kurse angeboten. Sehr sinnvoll ist es in den kostenfreien Android-Basic-Kurs zu schnuppern, da man bei der Installation der IDE an die Hand genommen wird und erste Übungen in Bezug auf App-Layouts erledigen kann.

Ein Mindestmaß an Englischfähigkeiten wird jedoch vorausgesetzt.

Das Projekt "DynLiCo" kann, nachdem es entpackt wurde, über "File -> Open" geöffnet werden. Weitere benötigte SDKs oder Gradle-Updates werden durchgeführt. Bei Gradle handelt es sich quasi um das Verarbeiten der "makefile". Weiterführende Optionen beim Erstellen beziehungsweise Kompilieren des Projekts sind hier festgelegt. Diese Datei muss für eine Inbetriebnahme zum jetzigen Wissensstand nicht verändert werden.

6.2 Modbus Slave

Da innerhalb dieses Projektes sowohl ein Modbus-Master über den WAGO-Controller sowie auch ein Slave mit dem Smartphone realisiert wird, bietet es sich an Dritt-Software zu verwenden. Bei der Entwicklung des Masters kann mit Modbus Slave somit ein einwandfreier Slave-Betrieb gewährleistet werden, der vordefinierte Werte erzeugt. Würde man versuchen die selbst-programmierten Busteilnehmer direkt miteinander zu koppeln, wüsste man bei auftretenden Fehlern nicht, wem diese zuzuordnen sind. Modbus Slave spielt bei der Entwicklung eine sehr große Rolle, da überprüft werden kann, wie das Telegramm aufgebaut werden muss um bestimmte Bits im Bitfeld des Masters zu manipulieren.

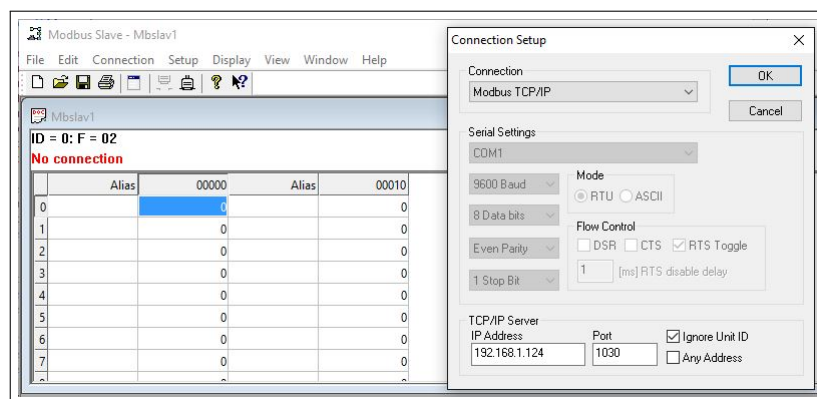


Abbildung 5: Grafische Oberfläche von Modbus Slave mit geöffnetem Verbindungsfenster in der rechten- und manipulierbarer Übertragungsbitmaske in der linken Bildschirmhälfte

6.3 Modbus Poll

Eine ähnliche Rolle hat dieses Programm bei der Entwicklung der App gespielt. Es konnte ermittelt werden, wie das vom Smartphone gesendete Telegramm tatsächlich aussieht und in welchem Bereich der Master dann die gesendeten Bits auszulesen hat. Im Vornherein war nicht ersichtlich, wie sich die Methoden der Java-Modbusbibliothek auf das versendete Bitfeld auswirken. Somit hat es sich zu Beginn schon sehr bewährt, als belegt werden konnte dass es überhaupt zu einer Kommunikation der Busteilnehmer kommt.

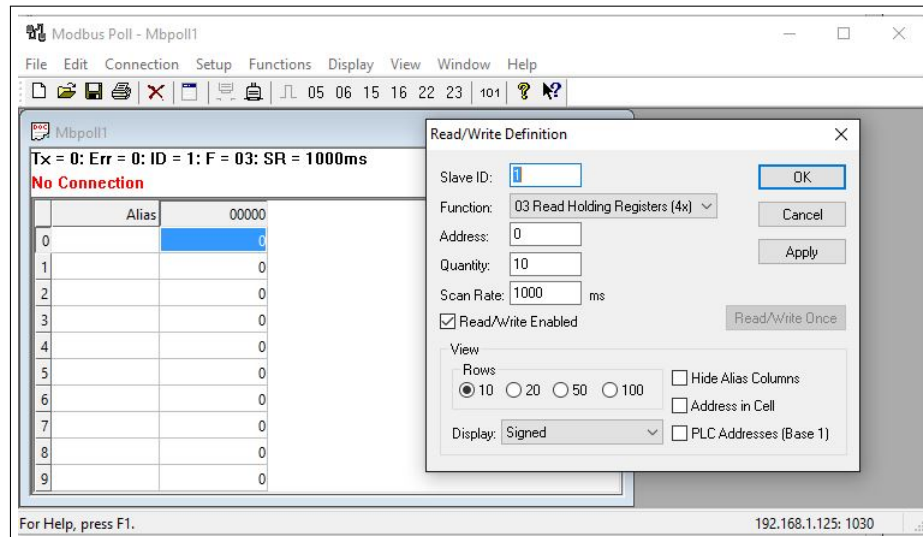


Abbildung 6: Grafische Oberfläche von Modbus Poll mit geöffnetem Definitionsfenster für die Lese- und Schreibbefehle. In der linken Bildschirmhälfte werden Werte des Slaves empfangen und angezeigt

7 Benutzte Bibliotheken

7.1 Jamod

Die Standard-Bibliotheken von Android sind in Bezug auf herkömmliche Kommunikationsstandards sehr umfangreich und im Internet auch sehr gut dokumentiert. Liegt nun jedoch ein ungewöhnlicheres Kommunikationsprotokoll vor hat man entweder die Möglichkeit eine Implementierung selbst vorzunehmen, oder aber nach Bibliotheken zu suchen, die benötigte Methoden und Funktionalitäten schon bereitstellen. Auch wenn Modbus zwar in der Industrie keine starke Konkurrenz zu EtherCat oder CAN ist, hat es sich in der Gebäudeautomation jedoch aufgrund seiner offenen Protokollierung etabliert und eine Nische gefüllt. Offene Protokollierung bedeutet in dem Fall, dass keine großen Nutzungsgebühren zur Verwendung des Protokolls anfallen und man die Möglichkeit hat auf Programme zuzugreifen, deren Source-Code im Internet frei verfügbar ist. Allgemein gesprochen geht es der *ModbusOrganization* vorerst um die Verbreitung von Modbus-fähigen Geräten weshalb auch eine Bibliothek für Java im Internet verfügbar ist.

Unter diesem Link lassen sich Beispiele zur Implementierung finden sowie auch der Download-Link zum aktuellen Source-Code.

Es ist unter folgender Adresse auch ein *HowToimplementTCPSlave* zu finden, welches letztendlich jedoch nicht sofort alle Fragen beantworten konnte und gerade im Bezug auf den letztendlichen Telegramm-Aufbau Fragen unbeantwortet lässt.

Um nachfolgenden Gruppen den Umgang zu erleichtern, findet im Quellcode eine dementsprechende Kommentierung statt.

7.2 BLE Android

In Anbetracht der zur Verfügung stehenden Zeit wurde bei der Bluetooth-Implementierung kein neuer Ansatz gewählt. Über GitHub konnte Quellcode zu einer App ausfindig gemacht werden, die eine Verbindung zu einer Pulsuhr aufbaut und die Herzfrequenz anzeigt. Auf dieser Applikation wurde das eigene Programm aufgebaut. So konnte sich auf die wesentliche Aufgabenstellung der korrekten Kommunikation zwischen Pulsuhr und Lichtsteuerung konzentriert werden.

GitHub ist eine Online-Plattform zur Versionsverwaltung. Man lädt sein eigenes Projekt hoch und andere Teilnehmer haben Zugriff. Nun können diese Teilnehmer den Code bearbeiten und hoffentlich verbessern. Übernommen wird der Code jedoch nicht automatisch, die Änderungen dem Admin vorgeschlagen werden. Er kann anschließend etwaige Bugfixes oder Zusatz-Funktionalitäten freischalten.

Die Basis-Version des Bluetooth-Projekts ist unter folgendem Link zu finden. Es bietet sich an den GitHub-Desktop-Client herunterzuladen, da hiermit die Aktualisierungsvorgänge leichter vorgenommen werden können. Nach der Installation befindet sich ein Ordner auf der Festplatte, der stetig nach Änderungen durchsucht wird und diese dann in dem Klienten anzeigt. Man sollte also sein Projekt dorthin abspeichern und nach jedem größeren Arbeitsschritt eine Aktualisierung der Online-Version vornehmen.

Auch gibt es die Möglichkeit unterschiedliche "Speicherstände" zu laden, wodurch immer zurück zu einer stabilen Version gewechselt werden kann.

Unter diesem Link ist die AKTUELLE Version des Projekts zu beziehen.

8 Programmierung

Im Nachfolgenden Kapitel findet eine Erklärung der unterschiedlichen Klassen statt. Es wird dabei auf eine hohe Detaillierung verzichtet, da im Anschluss der Programmablauf erklärt wird. Abgesehen davon geht die genaue Bedeutung der einzelnen Variablen und Methoden aus dem kommentierten Quellcode besser hervor.

8.1 Programmklassen

8.1.1 BluetoothGattExecutor; DeviceServicesActivity; DeviceScanActivity; BleService

Mit der *BLEHeartRateDemo* (also dem Basis-Projekt) sind die ersten vier Klassen vordefiniert. Sie ermöglichen den Bluetooth-Scan(*DeviceScanActivity*), die übertragenen Informationen des verbundenen Bluetooth-Geräts

(*DeviceServicesActivity*) sowie die grundlegende Klassen zur Verbindung zu Bluetooth-Geräten(*BleService* und *BluetoothGattExecutor*). Von besonderem Interesse sind für diese Anwendung die erstgenannten Klassen, da sie ein Beispiel dafür geben, wie die Daten des Bluetooth-Geräts ausgelesen werden können.

Grundsätzlich funktioniert der Datenempfang nach dem Herstellen der Verbindung nach dem Prinzip von *Broadcast* und *Receiver*. Man kann sich die Pulsuhr dabei als Funkgerät vorstellen, welches die Informationen einfach an jeden interessierten (aber auch verbundenen) Teilnehmer versendet. Jeder Teilnehmer muss dafür einen Empfänger (Receiver) besitzen, der die relevanten Daten zum Auslesen aufnimmt.

In *DeviceScanActivity* ist diese Funktionalität mit der Methode *displayData* und *BroadcastReceiver*

realisiert. Der Receiver schließt sich an das BLE-Gerät und `displayData` wird als "getter" an der Stelle aufgerufen, an der die Herzfrequenz abgerufen werden soll.

8.1.2 ModbusService

Im Vergleich zu Anwendungen auf dem Computer wird ein großer Prozentsatz von Apps auf Smartphones im Hintergrund ausgeführt. Der Datenempfang und die Steuerung der LEDs soll natürlich auch funktionieren, wenn das Handy im "Standby-Modus" ist. Dafür müssen Activities, die normalerweise nur im Vordergrund ausgeführt werden, als sogenannter *Service* ausgelagert werden. Sie besitzen eine Methode, welche sich *workRoutine* nennt. Bei der Klasse *ModbusService* wird sie aufgerufen, sobald der Service gestartet wird. Hier befindet sich die Kategorisierung der Herzfrequenzen über drei *if-else-Verzweigungen*. Anschließend wird die jeweilige Bitmaske (hier *SimpleProcessImage*) aufgebaut und über eine Methode der *jamod-Bibliothek* gesetzt.

```
public void workRoutine() {
while(isRunning){
tmp_heartValue = heartValue;

if (tmp_heartValue != old_heartValue) {

if (tmp_heartValue >= 100 && tmp_heartValue < 200) {
// GREEN
lightValue = 4;
} else if (tmp_heartValue < 100 && tmp_heartValue > 0) {
// RED
lightValue = 2;
} else {
// WHITE
lightValue = 1;
}
}
SimpleProcessImage spi = null;
spi = new SimpleProcessImage(); //Erstellen eines Bitfelds
spi.addRegister(new SimpleRegister(lightValue)); //Manipulieren des Bitfeldes
ModbusCoupler.getReference().setProcessImage(spi); //Setzen des Bitfeldes
old_heartValue = tmp_heartValue; //Abspeichern als alten Wert
//um
if (isRunning) {
Log.i(TAG, "Service running");
}
}
}
```

8.1.3 setupModbus

Um eine einwandfreie Verbindung per Modbus aufzubauen, muss bei jedem Komplett-Start der App eine Initialisierung stattfinden. Während dieser Initialisierung werden Eigenschaften festgelegt wie die Port-Nummer und die Definition des Smartphones als Slave. Darüber hinaus hat sich gezeigt, dass die DALI-Steuerung bei einem Reconnect nicht direkt ein gesetztes Bitfeld umsetzen kann. Daher wurde ein *newCountDownTimer* programmiert, der

die Lichter für 0,5 Sekunden in einen vordefinierten Zustand versetzt. Anschließend kann die dynamische Steuerung in Betrieb genommen werden.

8.1.4 setupActivity

Die *setupActivity* bildet, wie in Abbildung 3 dargestellt, die übergeordnete Menü-Klasse, aus welcher der Herzfrequenz-Modus oder auch der Nachtmodus gestartet werden kann. Des weiteren findet eine Neuinitialisierung bei einem Zurückkehren in die App (*resume*) statt. Hierbei muss der Receiver neu registriert werden, damit der Bluetooth-Kanal korrekt aufgebaut werden kann.

Hier sind ebenfalls die beiden Methoden zum Bluetooth-Datenempfang (*displayData* und *gattUpdateReceiver*) realisiert. Grund dafür ist die Status-Anzeige am unteren Bildschirmrand, welche in Abbildung 3 mit NoData initialisiert ist.

8.2 net.wimpi.Modbus

Da dieses Projekt auf einem Bluetooth-Beispiel aufgebaut wurde, jedoch die Modbus-Bibliothek auch mit eingebunden werden musste, existiert eine zweite Ordnerstruktur.

net.wimpi.modbus liefert dabei einige Klassen zur Implementierung unterschiedlichster Betriebsarten. Die Standard-Bibliothek wurde dabei jedoch um den Serial-Modus gekürzt. Hauptsächlich finden die Klassen *ModbusTCPListener* und *ModbusCoupler* Verwendung.

8.3 Allgemeiner Programmablauf

- Die App startet in die *setupActivity* und führt dabei den Code aus, welcher unter der Methode *onCreate* aufgeführt ist. Konkret handelt es sich dabei um das Verknüpfen von Textausgabefeldern sowie das Definieren des Bluetooth-Services.
Dazu kommen noch die beiden Buttons zur Überleitung in die Setup-Klassen für Bluetooth und Modbus sowie das Aktivieren des HR-Modus mit Übergang in die *ModbusService.class*. Als letztes kann der Nachtmodus durch Drücken des entsprechenden Knopfes aktiviert werden
(siehe *finalToggleButtonNightOnOff*)
- Es bietet sich an, zuerst die Bluetooth-Verbindung aufzubauen. Dafür wird der *finalButtonbleSetup* gedrückt und eine Überleitung in die Klasse *DeviceScanActivity* findet statt. Hier befindet sich in der *onCreate*-Methode hauptsächlich das Aktivieren von Bluetooth sofern es nicht sowieso schon eingeschaltet ist.
Verfügbare Geräte werden nun über die *private static class Scanner* angezeigt und können ausgewählt werden.
Es wird die Methode *onListItemClick* ausgeführt, welche die *EXTRAS_DEVICE_NAME* und *EXTRAS_DEVICE_ADDRESS* an einen neuen Intent anhängt, welcher an *DeviceServicesActivity* übergeben wird

- Diese Activity nimmt beim Erstellen auch wieder Verknüpfung zwischen Variablen und Text-Ausgabefeldern vor. Anschließend wird über *finalIntentgattServiceIntent* und *bindService* der *BleService* im Hintergrund gestartet.
Hier sind die Methoden *BroadcastReceiver* und *displayData* ursprünglich realisiert. Sie sind zuständig für Empfang und Anzeige der Herzfrequenz.
Nun gibt es noch weitere Methoden, welche aufgerufen werden sobald die Klasse zerstört wird (*publicbooleanonDestroy()*) oder aber auch bestimmte Optionspunkte aufgerufen werden (*onCreateOptionsMenu* und *onOptionsItemSelected*)
- Befindet man sich durch Drücken des *return – Buttons* wieder in der *setupActivity* sollte nun *setupModbus* aufgerufen werden.
In *setupModbus* findet anschließend eine Initialisierung statt, die einen *ModbusTCPListener* erstellt, sowie den gemeinsamen Kommunikationsport festlegt und ein *SimpleProcessImage* erzeugt. Ein Register wird Diesem hinzugefügt und es finden weitere Einstellungen zu *setMaster(false)* und *setUnidID(0)* statt.
Der *TCPListener* wird daraufhin gestartet. *newCountDownTimer(500,100)* setzt nach einer halben Sekunde das Register wieder auf 0.
- Zurück in der *setupActivity* wird nun der *ModbusService* aufgerufen. Innerhalb des Service wird zuerst der *gattUpdateReceiver* registriert um die Daten zu Empfangen. Die *workRoutine* wird anschließend dauerhaft ausgeführt.
Hier befindet sich der Zustandsautomat für die unterschiedlichen Herzfrequenzen. Derzeitig wird nur unterschieden zwischen einem Puls größer/kleiner 100.
Der *lightValue* bildet dabei das zu setzende Register, er ist Controller-seitig mit unterschiedlichen Lichteinstellungen verbunden und aktiviert die auskommentierten Farbtöne.
Jetzt wird das Register los geschickt und der aktuelle Herzfrequenz-Wert wird zwischengespeichert. Der Zustandsautomat soll nicht aufgerufen werden, wenn die Herzfrequenz unverändert ist.
In dieser Klasse wurde die *displayData*-Methode angepasst, da es nicht ausreicht die Daten als String anzuzeigen. Es muss eine Integer-Konvertierung stattfinden um die Fallunterscheidung in dem Automaten zu ermöglichen. *BLEStringToInt* nimmt diese Wandlung vor
- Zurück in der *setupActivity* löst der NightMode mit dem Register 256 eine Funktion an dem WAGO-Controller aus. Es reicht hierbei ein einfaches Senden da in dem Codesys-Programm ein Merker implementiert wurde. Das Gleiche gilt für das Deaktivieren des NightMode (Register 512)

9 Weiterführende Aufgaben

- Überprüfen ob setupModbus notwendig ist. Unter Umständen könnte beim erneuten Öffnen der App eine stille Initialisierung durch Kopieren des anliegenden Bitmuster und anschließendem Neusenden stattfinden
- Definierte Schließungs-Routine festlegen, wenn App aus unterschiedlichen Activities beendet wird. Bis jetzt findet die Bluetooth-Kanalschließung nur in der setupActivity statt
- Funktionalität mit weiteren Pulsuhren überprüfen
- Allgemeine Programmstruktur aufräumen und Effizienz der Programmierung steigern. Die Struktur des Programms könnte dabei übersichtlicher gestaltet werden
- Graphisches Interface der Anwendung verbessern und ansprechender gestalten
- Sinnvolle Lichtfarben für jeweilige Pulsintervalle ermitteln (in Zusammenarbeit mit Kollegen aus der Fakultät Life Science)
- Datenübermittlung optimieren. Unter Umständen könnten Controller und App so abgestimmt werden, dass tatsächlich erst bei signifikanter Puls-Änderung ein Telegramm los geschendet werden braucht