HTL Leonding
next level

HÖHERE TECHNISCHE
BUNDESLEHRANSTALT
LEONDING

Elektroauto Lademanager für die Firma Flexsolution

DIPLOMARBEIT

verfasst im Rahmen der

Reife- und Diplomprüfung

an der

Höheren Abteilung für Medientechnik

Eingereicht von:
Teresa Holzer
Marcel Pouget

Betreuer:
Professor Martin Huemer

Projektpartner:
Alfred Pimminger, Flexsolution GMBH

Leonding, April 2023

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die Arbeit wurde bisher in gleicher oder ähnlicher Weise keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Die vorliegende Diplomarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Leonding, April 2022 S. Schwammal & S. Schwammal

Abstract

This is the description of the journey to succeed this project. It took way longer than I thought, it was a lot of work and struggled, but in the end we did it. And if you are reading this right now it means, that everything went exactly as planned. And this is great, because it means that I have my A-Levels done, and don't have to go to this shitty school. Don't worry, I will correct this summary, but right now I don't know what's important, so I write just what comes in my mind. Enjoy it, and pls write me, if you find some mistakes: smalldickenergy@grethat.tu"

Zusammenfassung

Zusammenfassung unserer genialen Arbeit. Auf Deutsch. Das ist das einzige Mal, dass eine Grafik in den Textfluss eingebunden wird. Die gewählte Grafik soll irgendwie eure Arbeit repräsentieren. Das ist ungewöhnlich für eine wissenschaftliche Arbeit aber eine Anforderung der Obrigkeit. *Bitte auf keinen Fall mit der Zusammenfassung verwechseln, die den Abschluss der Arbeit bildet!*

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Herangehensweise 1

1.2 Zeitplan 1

1.3 Verwendete Tools 1

2 Umfeldanalyse **2**

2.1 Algemeiner Technologie-Part 2

2.2 Firmenstruktur 2

3 Wallbox **3**

3.1 Untersuchungsanliegen 3

3.2 Ist-Stand 3

3.3 Aufbau des Projektes 3

3.4 FlexTasks in der Flexcloud 14

3.5 Datenpunkte 17

3.6 Modbus 21

3.7 FlexHMI 21

3.8 Energiemanagement 21

3.9 Wallboxen 21

4 Flexlogger **22**

4.1 Untersuchungsanliegen 22

4.2 Aufbau des Projektes 22

4.3 Threads 22

4.4 Performance 22

4.5 Datenbanken 22

4.6 Visuelle Darestellung 22

4.7 Abspeicherung von Daten 22

4.8 Quarkus 22

5 test **23**

6 Zusammenfassung **25**

Literaturverzeichnis **VI**

Abbildungsverzeichnis **VII**

Tabellenverzeichnis **VIII**

Quellcodeverzeichnis **IX**

Anhang **X**

1 Einleitung

1.1 Herangehensweise

1.2 Zeitplan

1.3 Verwendete Tools

1.3.1 Latex

1.3.2 Intellij

1.3.3 Webstorm

1.3.4 Filezilla

1.3.5 Linux Terminal

1.3.6 Discord

1.3.7 Google drive

1.3.8 vs code

2 Umfeldanalyse

2.1 Algemeiner Technologie-Part

2.1.1 Java

2.1.2 JSON

2.1.3 HTML/CSS + Javascript

2.1.4 Typescript

2.1.5 Beckhoff

2.1.6 Raspberry

2.2 Firmenstruktur

2.2.1 Beschreibung der Tätigkeiten der Firma

2.2.2 Flex Tasks

2.2.3 Datenpunkte

3 Wallbox

3.1 Untersuchungsanliegen

Das ist ein Test, um zu schaun, ob es so funktioniert, wie ich mir das vorstelle

3.2 Ist-Stand

Zu der Zeit vor dem Projekt ist es der Firma nicht möglich, Elektro-Fahrzeuge aufzuladen. Um das erreichen zu können, soll auf dem Dach des Firmengeländes eine Photovoltaik Anlage installiert werden, welche mit bis zu 170 kWh die Firma und die neuen E-Autos mit Strom versorgen kann. Die eigens dafür angeschafften Elektroautos sollen mit 5 Wallboxen der Marke Ï-CHARGE CION"beladen werden. Da es bis zu dem Zeitpunkt des Startes des Projekts keine Möglichkeit gab, jene Wallboxen anzusteuern und überwachen zu können, beschäftigt sich dieser Teil der Diplomarbeit mit diesen Themen.

3.3 Aufbau des Projektes

3.3.1 Beschreibung der Funktionen

Das Projekt besteht hauptsächlich aus drei Teilen. Einer Website, auf welcher die Statusanzeigen der einzelnen Wallboxen und andere Funktionen angezeigt werden, einen Charge Controller, welcher dafür verantwortlich ist, die Befehle der GUI entgegenzunehmen und auszuwerten, und aus einem Gateway, welches zwischen den Controllern und dem Modbus-Adapter liegt. Wenn also jemand eine Ladestation einschaltet, schickt die Website (HMI (Human Machine Interface)) über die Flexcloud einen Befehl zu dem Charge Controller. Siehe Aufbau des Projektes: 1.

3.3.2 Beschreibung der Wallboxen

Die Wallboxen sind 5 Ladestationen der Marke Ï-Charge". Sie wurden im August 2021 an einer Außenwand der Firma montiert. Jede einzelne ist mit 400 Volt an das Stromnetz der Firma gebunden, und kann mit bis zu 32 A bzw. 22 kWh Leistung das Auto laden.

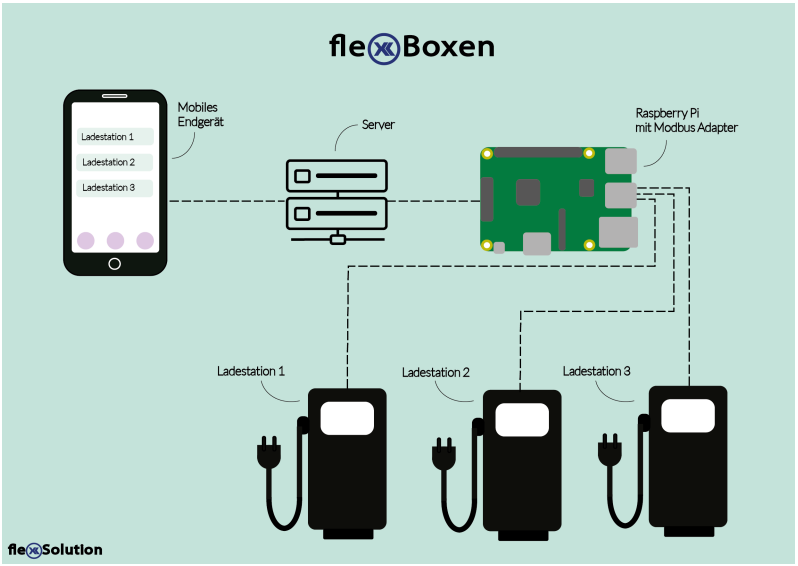

Abbildung 1: Darstellung des Aufbaues

Um den Status der Wallbox überwachen zu können, gibt es an der Vorderseite des Panels eine LED-Anzeige mit 5 Punkten, welche in verschiedenen Farben leuchten können. grün durchgehend: Station ist frei blau blinkend: Station ist besetzt, das Auto lädt aber nicht, weil es entweder voll oder die Wallbox ausgeschalten ist blau durchgehend: Station ist besetzt, das Auto lädt rot blinkend: Station ist defekt Weitere wichtige Anschlüsse der Stationen sind eine Modbus485 Schnittstelle, und einen 5 Volt Pin. Erstere ist dazu da, um der Box Befehle zu geben, und Register, also laufende Werte auszulesen. Mit Zweiterem kann man jede einzelne Wallbox ausschalten, indem man auf den 5 Volt Pin eine Spannung anlegt, und einschalten, wenn die Spannung wieder weggenommen wird. Die Wallboxen sind alle seriell miteinander verbunden, um die Modbus-Kommunikation zu ermöglichen. Dafür wurden jeweils zwei Drähte genutzt, die in eine Wallbox reingehen, an das Modbus-Interface angesteckt wurden, und dann wieder aus der einen Wallbox in die nächste Ladestation gelegt wurden. Am Ende der seriellen Leitung befindet sich ein Abschlusswiderstand, um ein möglichst genaues und sauberes Signal zu ermöglichen. Die Kabel der 5 Volt Leitungen sind alle parallel geschalten, und gehen am anderen Ende in eine Beckhoff Steuerung. Jene ist dafür ausgelegt, um die Spannung in den Drähten zu verändern. Das Kabel, welches für die Modbus Kommunikation zuständig ist, ist am anderen Ende mit einem Modbus zu USB-Adapter verlötet. Dieser Adapter steckt in einem Raspberry PI 4, welcher in demselben Schaltschrank montiert ist, wie die Beckhoff Steuerung.

3.3.3 Beschreibung des Gateways

Das Gateway ist ein sogenannter "FlexTask"(später dazu mehr, siehe 1.1), welcher auf dem Raspberry mit dem USB-Adapter läuft. Der Task ist in drei Abschnitte unterteilt. Wenn er gestartet wird, initialisiert dieser zuerst mehrere Arrays, welche die benötigten Datenpunkte beinhalten. Dafür wurden in der Tabelle für jeden Befehl jeweils 5 Datenpunkte gespeichert. Dies ist notwendig, da jeder Befehl auch an jede Wallbox geschickt werden kann, das Gateway aber die einzelnen IDs zuordnen muss. Der Datenpunkt an der Stelle "3ïn einem Befehlsarray repräsentiert den Befehl, welcher an die dritte Wallbox gesendet werden soll. Sobald alle Datenpunkte ohne Fehler angelegt und angemeldet wurden, beginnt der Task damit, eine serielle Verbindung zu dem Adapter aufzubauen. Es werden die Parameter für die Modbus-Verbindung gesetzt, und danach wird die Verbindung geöffnet. Für diese Verbindung wird die Java Library “jlibmodbus” verwendet, bei welcher folgende Parameter gesetzt werden müssen.

Wichtige Parameter

- setDevice(/dev/ttyUSB0") -> hier wird angegeben, an welchem Port der Adapter liegt.
- setBaudRate(SerialPort.BaudRate.BaudRate57600) -> Die Baut-Rate ist ein von der Wallbox vorgegebener Parameter, welcher beschreibt, wie hoch die Baut-Rate ist. Hier wird der Wert 57600 gesetzt
- setDataBits(8) -> Die Databits müssen 8 Bits betragen, da auch hier der Wallboxhersteller sich für diese Konfiguration entschieden hat
- setParity (SerialPort.Parity.NONE) -> Das bedeutet so viel wie, dass es keinen signed, als kein Überwachungsbit gibt
- setStopBits(2) -> Auch dieser Parameter wurde vom Hersteller vorgeschrieben. Die Stopbits werden in einem späteren Kapitel (1.2) noch genauer erklärt

Der erste Teil des Tasks besteht aus einer for-Schleife, und einem Code Block, welcher, auf Veränderungen bei den Datenpunkten hört. Als erster Schritt geht die for-schleife alle vorhandenen "Devices"(siehe 1.3) durch. In diesem Fall sind es 5, jedes Gerätsteht für eine Wallbox. Danach werden für jedes Device die Datenpunkte rausgesucht, und wenn der in der Datenbank gesetzte SSpecificDataType"(siehe 1.4) nicht "nullïst, wird ein sogenannter "DatapointCommandängehängt. Dieser Command ist dafür zuständig, dass Änderungen bei den Werten erkannt, und dann in dem oben beschriebenen Codeblock ausgeführt werden. Dort werden dann über die Deviceid, die Specificaddress und der

Wert jene Daten entnommen, welche man für das Beschreiben der Modbus Register benötigt. Dadurch wird ermöglicht, dass der Task, sobald die Value eines Datenpunktes sich ändert, dieser Wertdirekt an die Wallbox mit der ID des Geräts gesendet wird.

Der Zweite Teil des Tasks ist ein Timer, welcher alle 300 Millisekunden drei verschiedene Register ausliest.

Dazu wird bei jedem Durchlauf an alle Wallboxen ein read-Command geschickt, um folgende Adressen auszulesen:

- 153 -> ist das Register, in welchem die Ansteckdauer gespeichert wird.
- 151 -> ist das Register, in welchem die Ladedauer gespeichert wird
- 126 -> ist das Register, in welchem der aktuelle Ladestromwert in Ampere gespeichert wird.

Die Werte, welche das Modbus- Protokoll zurückliefert, werden an Datenpunkte übergeben, welche dafür zuständig sind, Werte vom Gateway weiterzuschicken (im Gegensatz zu den vorhererwähnten Datenpunkte, denn diese sind dafür da, um Values von anderen Tasks zu bekommen).

Der Dritte Part benützt, nicht so wie die ersten zwei Abschnitten, Modbus TCP statt Modbus RTU. Dieser ist wiederum in weitere 2 Teile aufgeteilt. Der eine Part baut eine Verbindung zu dem Controller der PV-Anlage auf, der zweite Part baut auf dieselbe Arte eine Verbindung zu den Janitza Messklemmen auf (siehe 1.6). Folgende Beschreibung gilt für beide Verbindungen, es ändern sich nur die Parameter, und die Art, mit den Rückgabewerten umzugehen.

Folgende Parameter sind für die Fronius-Verbindung einzustellen:

- tcpParameters.setHost(InetAddress.getByName("10.50.30.200")) -> Hier wird die IP des Modbus-Masters eingegeben
- tcpParameters.setKeepAlive(true) -> Hier wird ein Signal gesetzt, um die Verbindung aufrecht zu erhalten
- tcpParameters.setPort(Modbus.TCP.PORT) -> Standardparameter für den TCP-Port des Modbus (502)
- int slaveId = 1 -> SlaveID des Modbus-Masters
- int offset = 499; -> die Startadresse des Registers, in welchem der aktuelle Stromwert in Watt gespeichert wird
- int quantity = 2; die Anzahl der Register, welche ausgelesen werden sollen. Hier sind es zwei, da in jedem register nur bis zu 65536 gespeichert werden kann. Da

aber die PV-Anlage mehr als 65 kW erzeugen kann, müssen hierfür 2 Register zum Speichern verwendet werden. Das Erste Register zeigt dabei an, wie oft das zweite Register schon befüllt wurde. Ist also in dem ersten Register eine 1, und im zweiten Register 30000, bedeutet das, das einmal 65536 + 30000 W produziert werden.

Da die Janitza Klemmen eine andere Art der Persistierung nutzen, musste in dieser Klasse anders mit den zurückgelieferten Werten umgegangen werden. Die Werte der Janitza Klemmen sind als Float abgespeichert, und können aus diesem Grund nicht einfach abgelesen werden. Um trotzdem die richtigen Daten zu bekommen, wurde der zurückgegebenen Value mithilfe der "Integer.toBinaryString(value)" in binäre Zeichen umgewandelt. Mit der Hilfe eines Stringbuilders wurden dann die 2 Register aneinandergehängt. "str.append(Integer.toBinaryString(value));". Um nun den Binären Wert in eine echte Zahl zurück zu verwandeln, wurde die Funktion "Float.intBitsToFloat" verwendet. Der Wert aus dem String aus dem Stringbuilder wird zu einem UnsignedInt geparsed, und der Wert dann einer temporären Variablen zugewiesen. "tmp = Float.intBitsToFloat(Integer.parseUnsignedInt(str.toString(), 2));". Da der zurückgegebene Wert aufgrund der Übertragung Fehlern anfällig ist, wird noch überprüft, ob der Wert nicht 0 ist, und kleiner als 200.000, da bei manchen Abfragen die Value nicht stimmt. Die Werte der zwei Modbus TCP-Verbindungen werden alle 300 Millisekunden ausgelesen, und in die Flexcloud gepushed.

(Link zu den Fehlern. Beschreiben, wie durch Forum die Art der Umrechnung gefunden wurde. PV und Janitza Klemmen!!!)

Controller

Der sogenannte Chargecontroller ist die Verbindung zwischen dem Graphical User Interface und dem Gateway. Dieser ist für den Logikteil der Anwendung zuständig. In ihm werden die Inputs des Benutzers auf Richtigkeit überprüft, Einheiten umgewandelt, States von Buttons geändert, und User-Feedback generiert. Der Task läuft, genau wie das Gateway, in einem sogenannten Flextask und wird auf einem weiteren Raspberry initialisiert. Wenn der Task gestartet wird, werden Arrays für die einzelnen Daten angelegt. Jedes Array hat dabei die Länge 0-5, um damit die Wallbox ID zu simulieren. Die Daten der ersten Wallbox, ist somit an der Stelle [1], und nicht [0], so wie es normalerweise der Fall ist. Das ist notwendig, um im späteren Verlauf der Applikation das Ansprechen der Datenpunkte zu vereinfachen.

- Ladedauer -> ist ein Zwischenspeicher, um die Veränderung der Ladedauer zu überprüfen.
- Ansteckdauer -> ist ein Zwischenspeicher, um die Veränderung der Ansteckdauer zu überprüfen.
- wallboxTurnOff -> ermöglicht das Ein- und Ausschalten der Wallboxen.
- Wallboxpriority -> ermöglicht das Wechseln zwischen priorisiertem und normalem Laden.
- pressdWB -> Ein Array aus booleans, welche alle auf false gesetzt werden. Sobald eine Wallbox eingeschaltet wurde, wird der Wert an der richtigen Stelle auf true gesetzt
- ChangePriority -> Array, in welchem die Variable an der Stelle [x] auf true gesetzt wird, sobald eine Wallbox auf automatisches Laden gestellt wird

Nachdem der Task erfolgreich gestartet ist, werden im Main Thread die Datenpunkte zur Datenübertragung angelegt. Diese bestehen wieder aus einem Array von jeweils 5 Datenpunkte, da man für jede einzelne Wallbox jeden Datenpunkt braucht.

- dpsetChargingActive -> ist für das Userfeedback zuständig. Wenn jemand auf einen Button drückt, und sich der Status der Wallbox erfolgreich geändert hat, wird mit diesem Datenpunkt in der HMI die Farbe des Buttons geändert.
- DpChangePriorityOfCharching -> Dieser Datenpunkt macht genau dasselbe, nur ist er für die Farbe des Buttons zuständig, welcher das Priorisierte Laden aktiviert.
- dpChargingSlider -> Wenn die Wallbox eingeschalten ist, und jemand den Slider für die Stromvorgabe ändert, wird auf diesem Datenpunkt der vom User eingestellte Wert gepublished. Das ist notwendig, um dem Benutzer ein direktes Feedback in der UI zu geben. Er kann dadurch sehen, wie viel Strom er der Wallbox vorgegeben hat.
- wallboxStatus -> Dieser Datenpunkt ist für die Farbe des Status-Symbols da. Es gibt drei verschiedene Status:
  - Rot: -> Ladesäule ist besetzt, aber das Auto lädt nicht
  - Blau: -> Ladesäule ist besetzt, und das Auto lädt mit den Vorgegebenen Ampere
  - Grau: -> Ladesäule ist frei und kann jederzeit benutzt werden

  Diese werden, je nach Status der Wallbox aktualisiert. Achtung: Vor allem der Wechsel zwischen “Blau” und “Rot” (in genau dieser Reihenfolge) kann etwas länger dauern, da die Wallbox selbst erst nach einigen Sekunden den Stromfluss zum Auto unterbricht, und somit das Laden stoppt.

- wallboxLädtMitKW -> Dieses Array speichert den aktuellen Wert, mit welchem die Wallbox gerade lädt. Da bei den E-Autos meist die Kapazität des Akkus in kW/h angegeben sind, wird im Chargecontroller der Wert der Wallbox in Ampere umgerechnet, und mit diesem Datenpunkt zur HMI geschickt. Dient dazu, um dem Benutzer eine Möglichkeit zur Kontrolle zu geben. Achtung! Es ist nicht derselbe wert wie im "dpChargingSlider" Array, da bei letzterem der Wert direkt wieder zur UI geschickt wird, während "wallboxLädtMitKW" der Tatsächliche Wert der Wallbox ist!
- wallboxLadestromvorgabe -> Dieser Datenpunkt ist auch wieder für die Vorgabe des Ladestroms zuständig. Jedoch wird mit diesem Array die Position des Sliders angepasst, sodass selbst nach Verlassen der Oberfläche der Slider immer die zuletzt eingestellte Position besitzt, und der Wert wird an das Gateway weitergeleitet. Dafür wird der Wert des Sliders von kW/h in Ampere umgerechnet und gepubilshed
- AktuellerStromWertVonJanitzaForHMI -> Dieser Datenpunkt ist nur zur Kontrolle da. Er dient der Veranschaulichung des Stromverbrauchs in der Firma. Ist der Wert unter 50.000 W, kann das priorisierte Laden nicht aktiviert werden.

Der Nächste Abschnitt ist vom Aufbau her ähnlich wie das Gateway. Zuerst wird mit einer For-Schleife über alle Datenpunkte, deren Spezifischer Datentyp nicht leer ist, iteriert. An jeden gefundenen Datenpunkt wird dann wieder der "DatapointChangedCommand" gehängt. Das ist dafür da, um auf Änderungen der Werte zu hören. Der nächste Abschnitt der App besteht aus einem "DatapointCommand", welches Ausgeführt wird, sollte die Flexlib eine Änderung der Werte erkennen. Darin ist ein switch Statement, welches auf die ID der einzelnen Datenpunkte hört. Sollte also der Datenpunkt mit der ID "ladedauerWb" einen neuen Wert zugewiesen bekommen, wird der darunterliegende Codeblock ausgeführt. Mit Hilde der Device Id, welche 1-5 betragen kann, bekommt man die ID der Wallbox, für welche die Änderungen bestimmt sind. Es gibt folgende Datenpunkte IDs, auf welche gehört werden:

- getDataFromModbuswb -> Dieser Wert verändert sich, wenn das Gateway eine Veränderung der Ansteckdauer feststellt, und diese dem Controller sendet. Wenn sich die Ansteckdauer nun verändert, wird der Wert in das Array "Ansteckdauer" an der Stelle [x] (Device ID) gespeichert.
- ladedauerWb -> Sollte sich der Wert des Datenpunktes mit der Ladedauer verändern, wird der Wert, genau wie im oberen Datenpunkt, in das Array für die "Ladedauer" gespeichert.

- aktuellerLadestromWertWb -> Das ist der Wert, der sich ändert, wenn das Gateway den aktuellen Ladestrom misst und dem Controller sendet. Dieser wird wieder in W/h umgerechnet, und an den Datenpunkt “wallboxLädtMitKW” gepublished.
- cctGetLadestromvorgabeFromHmi -> Sobald die Wallbox eingeschalten ist (die Valie des Icons ist 2) bekommt der Task die Vorgabe des Sliders auf von HMI. Da der Slider auf kW eingestellt ist (1-22), wird der Wert des Datenpunktes in Ampere umgerechnet, und dann sowohl zur HMI (zur Kontrolle) und an den Modbus Task geschickt. Der Wert wird außerdem in einem Array zwischengespeichert, um ihn beim Einschalten direkt an den Modbus-task mitzuschicken.
- cctGetStartChargingcommandWb -> Dieser Datenpunkt wird dann ausgelöst, wenn ein User den Button zum Starten des Ladevorganges drückt. Bei jedem zweiten Event (ein Klick löst zwei Events aus) wird der Status in dem Array “pressedwb” überprüft. Ist die Variable auf “false”, wird das Icon des Buttons auf grün (Value 2) gesetzt, und der aktuelle Wert des Sliders wird “wallboxLadestromvorgabe” mitgegeben. Außerdem wird die Variable “pressedwb” auf true gesetzt, da die Wallbox nun lädt. Sollte das Icon schon grün sein und somit die Wallbox schon eingeschaltet, wird bei einem weiteren Button-press das Icon auf Rot (aus) gestellt, und “wallboxLadestromvorgabe” wird auf 0 gesetzt. Das Boolean wird auf false gesetzt
- ChangePriorityOfCharching -> Hier wird das automatische Laden geregelt. Zuerst wird geschaut, ob das Icon ein bzw ausgeschaltet ist, dann, ob die Wallbox eingeschaltet ist. Und wenn dann auch noch das Array von auf true ist, welches überprüft, ob die Janitza klemmen mehr als 10 kW zurückgeben, dann wird das Auto automatisch geladen. In meinem Fall wird einfach an den Modbus-task der Wert 32 (A) geschickt, was die maximale Ladegeschwindigkeit ist. Wenn er nicht priorisiert lädt, bekommt der Modbus-task den Wert 10 (A)
- AktuellerStromWertVonPv -> Dieser Datenpunkt ist nur dafür da, um den Wert von dem Gateway, welches den aktuellen Wert der PV schickt, an die HMI weiterzuleiten
- AktuellerStromWertVonJanitza -> Macht genau dasselbe, nur mit dem Wert der Janitza Messklemmen

Der Controller besteht neben dem Main Thread noch aus einem Timer, welcher alle 1,2 Sekunden einen Codeblock ausführt. In Diesem wird geschaut, ob sich die Ladedauer oder die Ansteckdauer im Vergleich zum letzten Durchgang verändert hat. Sollte dies der Fall sein, werden die Farben der Icons angepasst. Außerdem wird überprüft, ob das Priorisierte Laden aktiviert werden darf oder nicht. Sollte der Wert der Janitza

Klemmen innerhalb von 30 Wiederholungen nicht unter 10.000 gefallen sein, darf man das Auto priorisiert laden.

HMI / GUI

Die letzte und für den User die Wichtigste Komponente ist das Graphical user interface (kurz: GUI). Um den Nutzern in der Firma das Verwalten der Oberfläche so einfach wie möglich zu machen, hab wurde die Oberfläche mit den Firmeneigenen Systemen entwickelt. Die von der den Mitarbeitern entwickelte HMI (HMI steht für Human Machine Interface) funktioniert nur auf den internen Servern. Das Besondere ist, dass alle Oberflächen der Firma via Sockets verbunden sind, und somit alle Anzeigen zu jeder Zeit gleich sind. Der große Vorteil ist dadurch, dass sich alles Systeme miteinander verbinden lassen. Dies geschieht mit dem Flextask "HMI-Machine", der einzig dafür verantwortlich ist, sich auf neue Oberflächen zu subscriben, und ihr alle für sie bestimmten Daten zu schicken. Dies geschieht auch wieder über Datenpunkte, die auf die einzelnen Elemente der Oberfläche gemappt sind. Diese Elemente werden in einem großen JSON-Objekt zusammengestellt, konfiguriert, benannt und ausgerichtet.

Listing 1: Example Element

```
1    Test_DynButtonRGBSliderOnOff: {
2      type: "dynButton",
3      items: {
4        label: { id: "name", textContent: "DynButton RGB Slider Ein/Aus"},
5        color: { id: "farbe" },
6        slider: { id: "range" },
7        icon: { id: "symbol" },
8      },
9    },
```

Man kann nun für diesen Button einen Datenpunkt anlegen, welcher in die Flexcloud published, wann jener Button gedrückt wird. Man kann dabei verschiedene Attribute hinzufügen bzw verändern. Beispiele sind:

- Type: gibt an, welchen Typ das Element besitzt. In diesem Fall ist es ein Button, der sich drücken lässt
- Label: gibt an, welche ID und welchen Content der Button haben soll. Man kann den Content über den Namen des Elements und dem Label ansprechen und verändern.
- Color: Dadurch lässt sich die Farbe des Buttons / Elements verändern.
- Slider: erstellt einen Slider mit einer gewissen Range und mit einstellbaren "Sprüngen"
- Icon: dort kann man Icons von librarys wie zum Beispiel Font-Awesome anbinden. Die Icons können mit setzten von verschiedenen Werten die Farbe ändern.

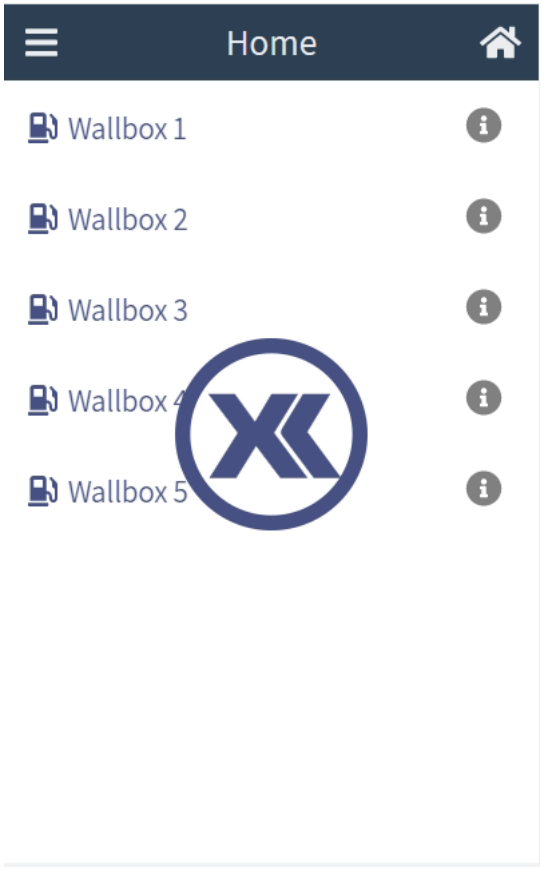

Abbildung 2: Übersicht über die fünf Wallboxen auf der HMI:

Die Oberfläche der App besteht aus einer Hauptseite, wo man eine Übersicht die verschiedenen Ladestationen bekommt. Die Startseite besteht aus einem Icon als Zeichen für die Wallbox, einem Label mit dem Namen der Wallbox, und einem zweiten, dynamischen Icon. Letzteres ist dafür da, um über den aktuellen Status der Wallbox zu informieren.

- Grau -> Station ist frei
- Rot -> Station ist besetzt, aber das Auto lädt nicht
- Blau -> Station ist besetzt, und das Auto lädt

Siehe Übersicht über die einzelnen Wallboxen: 2.

Jedes dieser 5 Label ist ein Link zu der Unterseite der jeweiligen Wallbox. Jede der Unterseiten besteht dabei aus einem Dynamischen Header mit der ID der ausgewählten Wallbox. Das Nächste Element besteht aus einem Label, einem Dynamischen Slider und einem Icon, welches zum Ausschalten der jeweiligen Wallbox verwendet wird. Der Slider hat eine Range von 1 – 22, was die möglichen kW veranschaulichen soll. Darunter ist noch ein Status Element, welches mit dem Icon auf der Startseite gekoppelt ist. Die Soll-Vorgabe, welche sich darunter befindet, ist der Wert, den der User mit dem Slider

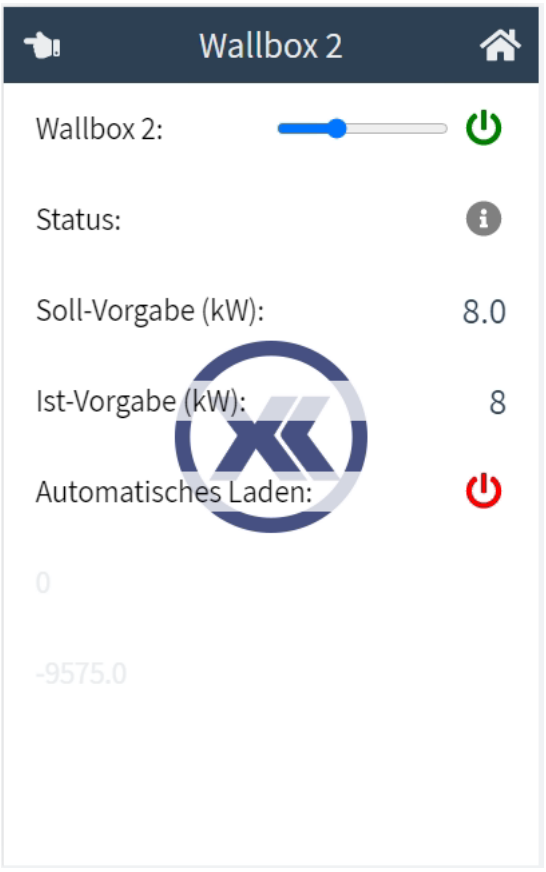

Abbildung 3: Detailansicht der Oberfläche für die Einzelnen Wallboxen

eingestellt hat. Er ändert sich dynamisch, und gibt dem Benutzer die Möglichkeit, die gewünschten kW/h genau einzustellen. Bei Verlassen bzw. Neu laden der Seite wird der Wert natürlich gespeichert, und auch der Slider behält seine Position. Die Ist-Vorgabe ist der Wert, welcher bei der Wallbox eingestellt ist, und mit welcher das Auto dann wirklich aufgeladen wird. Dieser ändert sich meist mit einer kleinen Verzögerung, da die Wallbox den neuen Wert erst nach wenigen Augenblicke übernimmt. Das Element mit dem Namen “Automatisches Laden” ist dafür da, um zwischen Solar und Netzstrom zu schalten. Die Funktion “priorisiertes Laden”, welche in den Vorherigen Kapiteln beschrieben wurde, kommt hier zum Einsatz. Die Oberflächen sind für jede Wallbox gleich, nur wird auf jeder Unterseite eine andere Ladestation angesteuert. Die GUI überreicht man nur im Firmen internen Netzwerk über eine URL. Mit der Raute Raute walli kommt man auf das mobile Template der Anwendung. Die Buttons lassen sich nur mit einem Touch screen oder den Entwicklungs-Tools von Chrome betätigen.

Siehe Deteilansicht der Wallbox: 3.

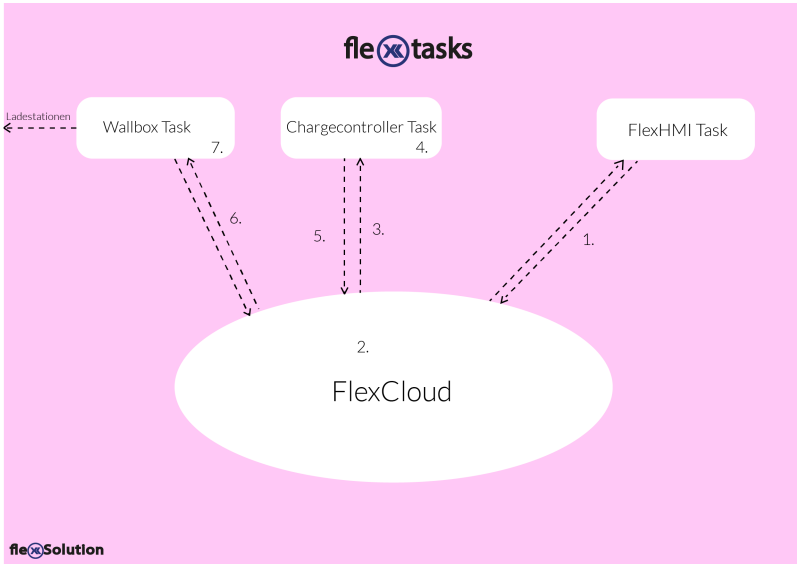

Abbildung 4: Darstellung des Aufbaues

3.4 FlexTasks in der Flexcloud

3.4.1 Was ist die Flexcloud?

Siehe Aufbau der Flexcloud: 4.

Die Flexcloud, auch FlexcommunicationCoud genannt, ist eine Erfindung der Firma FlexSolution. Die Anwendung findet im Firmeneigenen Netzwerk statt, und ermöglicht es, viele kleine Microservices miteinander kommunizieren zu lassen. Die Hauptkomponente, der Sogenannte FlexCore, wurde von den Mitarbeitern in der Sprache Java entwickelt. Mit der sogenannten Flexlib lassen sich die Anwendungen mit vielen weiteren kleinen Dependencies erweitern. Das wird vor allem dann benötigt, wenn z.B. eine Datenbankanbindung notwendig ist. Die Flexcloud besteht also aus lauter sogenannten Flextasks, welche verschiedene Aufgaben ausführen. Es zum Beispiel einen Task, der für die Ansteuerung der Lampen und Rollos zuständig ist. Ein Weiterer Task, der für die HMI zuständig ist, verbindet die Flexcloud mit dem Browser. Die FlexcommunicationCoud ist auch für viele kleinere Anwendungen nützlich, da sich solche Microservices sehr schnell aufsetzten lassen. Man kann einzelne Tasks aufsetzten, oder mehrere miteinander verbinden und kommunizieren lassen. Das hat den Vorteil, dass man wiederverwendbare Anwendungen entwickeln kann, welche nur grundliegende Funktionen erfüllen, aber von vielen Tasks gebraucht werden. So ist es z.B. in Planung, einen eigenen Modbus-task zu implementieren, welcher nur dafür da ist, Befehle von anderen Tasks zu übernehmen und an eine USB-Schnittstelle weiterzuleiten. Damit die Tasks untereinander kommunizieren können, hat die Firma die sogenannten Datenpunkte (Datapoints) entwickelt. Sie dienen dazu, um zwischen einen oder mehreren Tasks Informationen oder Werte auszutauschen.

Auch Funktionen-Aufrufe können durch solche Datapoints ermöglicht werden. Um durch Datenpunkte miteinander zu kommunizieren, müssen besagte FlexTasks mit den ProjectID´s verbunden werden. Wie man die Datenpunkte anlegen kann, was dabei wichtig ist und wie die Daten gemapped werden, wird im Kapitel (Datapoints) beschrieben.

3.4.2 Grundaufbau der Tasks

1. Pom.xml des Tasks editieren: Als erstes sollte die MainClass gesetzt werden. Dies folgt immer einem gewissen Schema. Die Klasse sollte im Package “eu.flexsolution.task.” zu finden sein. Die GroupID ist dieselbe wie das Package der MainClass. In dem Fall ist es wieder “eu.flexsolution.task”. In der “artifactId” steht der Namen des Tasks. Bei den Dependencies wird der “Flexsolution core”, “log4j”, “org.json”. Eine weiter wichtige Sache ist das Einbinden des “Internal Snapshot Repository”. Das ist der Ort, an dem der Flexsolution core und die Flexlib gepublished werden. Maven lädt sich dann beim Starten der Applikation die Dependencies von dem von der Firma bereitgestellten Server herunter, und fügt sie zur Applikation hinzu.
2. Das Anlegen der Main Klasse. Der erste Schritt ist, die Klasse “FlexTask” zu erweitern.

Listing 2: Example Element

```
1          public class FlexTaskModbuswallbox extends FlexTask
```

Dadurch werden einige Funktionen überschrieben, bei denen die meisten jedoch nicht verwendet werden. In der “public static void main” wird die ein neues Objekt der Main Klasse erstellt.

Listing 3: Example Element

```
1          new FlexTaskModbuswallbox();
```

In der “initTask()” Methode kann dann die Hauptfunktion des Tasks implementiert werden. In den meisten Fällen wird hierfür Datenpunkte und andere Kommunikations-Methoden eingebunden. Mach Fertigstellung der App muss der Code mithilfe von Maven zu einem JAR-File kompiliert werden.

3. Linux-Maschine vorbereiten: Als erster Schritt ist es wichtig, ein Verzeichnis für den Task zu erstellen. In diesen wird dann das .jar File kopiert. Da in der Firma mit “log4j” gearbeitet wird, muss hierfür noch ein Konfigurations-file angelegt werden. (siehe log4j File) Der nächste Schritt ist im Verzeichnis “ /.local/share/systemd/user/”

ein File mit dem Namen des Services anzulegen: ‘gateway.service’. Dieses File muss wie folgt aufgebaut sein

Listing 4: Example Element

```
[Service]
Type=simple
ExecStart=/usr/bin/java -Xmx32m -Dlog4j.configurationFile=./log4j2.xml
    -jar /srv/tasks/CURRENT/modbuswallbox/modbuswallbox.jar
Environment="TASKNAME=modbuswallbox"
WorkingDirectory=/srv/tasks/CURRENT/modbuswallbox
TimeoutStopSec=3
Restart=always
RestartSec=10
Slice=flexTasks.slice
[Install]
WantedBy=default.target
```

4. Da jeder Flextask eine Datenbank mit dem für ihn zugehörigen Datenpunkten besitzt, muss dieses File auch erst angelegt und konfiguriert werden: im Verzeichnis ‘/var/flex/tasks/’ wird ein neuer Ordner mit dem Namen des Tasks angelegt. In diesem wird die Datei conf.db erstellt, und folgende Tables hinzugefügt:
   a. Device: besteht aus einer “ID” und einem Label. Im Falle des Gateways sind es 5 Devices, für jede Wallbox eines. Das ist dafür gut, um Datenpunkte mit denselben Namen nutzen zu können, da man sie mithilfe der Deviceid unterschieden kann
   b. DeviceParameter: Hier können wichtige Parameter gesetzt werden, die dann beim Starten eines Tasks mit der Methode “getNeededDeviceParameters(Map<String, ValueCheck> map)” ausgelesen werden können. Die Tabelle besteht aus einer ID, einer DeviceID, einer Value und einem Label.
   c. taskParameter: hier können wichtige Parameter für das System gesetzt. Meistens wird hier nur der Standart-Port 8150 gesetzt. Die Tabelle besteht aus einer Spalte für die ID, für Values und einer für Labels. Auch hier können beim Starten eines Tasks die Werte der DB ausgelesen werden. Hierfür verwendet man die Funktion “getNeededTaskParameters(Map<String, ValueCheck> map)”.
   d. datapointValueMapping: Besteht aus einer Datapoint1Id und Datapoint2Id, einer Value und einer Spalte für neue Werte (newValue). Dies ist dafür da, um zwischen zwei Datenpunkte die Werte zu synchronisieren oder gegebenenfalls mit newValue zu überschreiben
   e. Datapoint: Herzstück der Datenbank. Hier werden die Datenpunkte angelegt (mehr dazu siehe 3.5). Die Tabelle besteht aus einer ID, einer deviceId, einem Label, einer “specificAddress”, einem “specificDatatype”, einem “specificStruct”, einem Datentyp (datatype), Flags, einem Intervall, einem Threshold und einer Value. Für das Starten des Tasks ist diese Tabelle zu befüllen, jedoch ist sie unumgänglich, wenn man die Hauptfunktionen der Flexlib verwenden möchte

f. datapointMap: Diese Tabelle besteht aus einer datapoint1Id, einer datapoint2Id und einer Funktion. Dieser Table wird dazu verwendet, um Zwei Datenpunkte zu mappen (Verknüpfen).

g. SystemParameter: Hier werden die wichtigsten Einstellungen für den Task selbst getroffen. Es gibt Einstellungen für den Namen des Tasks, den Port, man kann das keepAlive Signal einstellen. Außerdem gibt es die PROJECTID, welche für das Verbinden mehrerer Tasks notwendig ist. Denn es können sich nur FlexTasks finden, welche dieselbe ProjectID besitzen. Im Falle des Gateways werden die Tasks mit der ID "AUT, HMI, chargecontroller" verbunden. Das ist wichtig, um den einzelnen Tasks die benötigten Parameter zu übergeben. AUT ist in dem Falle die Haussteuerung, HMI ist der oben kurz beschriebene Flextask zur Veranschaulichung der Elemente, und chargecontroller ist der auch schon beschriebene Controller für dieses Projekt.

5. Wenn alle Pfade richtig benannt wurden, kann man im Verzeichnis einen Symbolischen Link zu dem .jar File machen. Das hilft bei der Entwicklung, da man bei einer Namensänderung nur den Link anpassen muss, und nicht die angegebenen Pfade in den Konfigurations-Dateien.
6. Starten, stoppen bzw. überwachen kann man den Task mit "systemctl –user start / stop / restart / status FlextasName.service
7. Mit dem Command "tail -f /tmp/log/ FlextaskName.log" kann man in Echtzeit mitansehen, was Log4j in das File schreibt. Die Files werden alle in dem Verzeichnis "/tmp/log/" gespeichert, wenn sie nach einiger Zeit nicht mehr benötigt werden, werden sie komprimiert und in ein anderes Verzeichnis verschoben, um Platz zu sparen.

3.5 Datenpunkte

3.5.1 Struktur eines Datenpunktes

Die Datenpunkte sind im Grunde alle gleich aufgebaut. Ein Datenpunkt (dp) besteht immer aus einer:

- ID: kann nach Belieben benannt werden. Meist ein String, um den Nutzen zu veranschaulichen, und die DPÂťs voneinander unterscheiden zu kÃűnnen. Die ID muss fÃijr jedes Device eindeutig sein.
- deviceId: gibt an, zu welchem Device der Datenpunt gehÃűrt. Ein Device kann in dem config.db File hinzugefÃijgt werden. Das dient dazu, um mehrere Datenpunkte

mit derselben ID fÃijr mehrere Devices verwenden zu kÃűnnen. Die deviceId besteht immer aus einem INT. Um Datenpunkte nutzen zu kÃűnnen, muss mindestens ein Device angelegt werden. Die ID wird automatisch immer um eins erhÃűht. Im Normalfall ist schon ein Device mit der ID 1 und dem Label dev1 angelegt.

- Label: Hier kann man den Datenpunkt kurz beschreiben, um seine Funktion zu dokumentieren. Das wird dafÃijr verwendet, um den Entwicklern das Verwalten der Datenpunkte zu vereinfachen.
- SpecificAddress: Ist meistens derselbe String wie die ID, kann jedoch auch etwas anderes sein. Wird dazu benÃűtigt, um den Datenpunkt im Task anzulegen.
- SpecificDatatype: Hier kann ein String mitgegeben werden, um in einem Task einen Datentyp fÃijr den Datenpunkt festzulegen, bzw. zwischen Gruppen von Dp zu unterscheiden. Im Task fÃijr das Gateway wird der specificDatatype dafÃijr genutzt, um zwischen Datenpunkte, die nur fÃijr Modbus zustÃďndig sind, und den anderen Datenpunkten zu unterscheiden.
- Datatype: hier kann man angeben, welchen Datentyp die Value haben soll. Man kann hier zwischen âĂIJINTâĂİ, âĂIJREALâĂİ, âĂIJBOOLâĂİ und âĂIJCMDâĂİ wÃďhlen.
- Intervall: Hier kann man ein Intervall, angeben, mit welchen der Datenpunkt in die Flexcloud gepushed werden soll. StandardmÃďÃ§ig ist âĂŞ1 eingestellt, was bedeutet, dass der Datenpunkt nur nach Anfrage des Tasks bzw. bei ÃĎnderung der Value gepublished wird.
- Value: Hier werden die Werte eines DB gespeichert. Man kann in der conf.db Datei einen Standard-Wert angeben, doch meistens werden diese Werte in einem Task gesetzt bzw. ausgelesen.

3.5.2 Anlegen eines Datenpunktes

Um einen neuen Datenpunk in einem Task anzulegen, muss man sich zuerst auf die VM (Virtual machine), auf welcher der Flextask sich befindet, verbinden. In dem Verzeichnis mit der conf.db Datei muss man mit SQLITE3 die Datei Ãűffnen. Dort kann man dann mit einem insert bzw. Update Statement einen neuen Datenpunkt hinzufÃijgen, aktualisieren oder lÃűschen.

Listing 5: Example Element

```
1       INSERT INTO datapoint VALUES
            ('Wallbox_1_startCharching_icon_state',1,'','state_[Wallbox_1_startCharching_icon]','',
```

3.5.3 Nutzung eines Datenpunktes:

Somit ist der Datenpunkt in der Datenbank angelegt und ready to use. Um nun in einem Flextask auf so einen Datenpunkt Zugriff zu haben, muss man ihn erstmal anlegen und die âĂIJSPECIFICADDRESSâĂİ angeben:

Listing 6: Example Datapoint

```
1       Datapoint datapoint =
            StaticData.datapoints.getDatapoint(Datapoints.DatapointField.SPECIFIC_ADDRESS,
            âĂSPECIFIC_ADDRESS_DES_DATENPUNKTESâĂİ);
```

Nun hat man mehrere MÃűglichkeiten, wie man Daten aus dem Datenpunkt lesen kann:

1. Man ruft die Methode âĂIJsetDatapointChangedCommand()âĂİ auf. Diese Methode Ãijberschreibt eine in der Klasse Flextask, und ist dafÃijr da, um auf ÃĎnderungen des Datenpunktes zu hÃűren.

Listing 7: Example catapoint usage

```
1           datapoint.setDatapointChangedCommand(new DatapointCommand() {
2               @Override
3               public DatapointResponse execute(DatapointResult result) {
4
5                   System.out.println(datapoint.getValue());
6                    return new
                         DatapointResponse(DatapointResponse.ResponseCode.OK);
7               }
8           });
```

   Diese Methode gibt bei jeder ÃĎnderung des Wertes den neuen Wert in die Konsole aus. Hier kann man den Wert speichern, andere Methoden ausfÃijhren, oder sonst alles machen, was man in einer Methode machen kÃűnnte.
2. Man kann Jederzeit im Code âĂIJdatapoint.getValue()âĂİ aufrufen. Mit dieser Methode bekommt man immer den jetzigen Wert des Datenpunkt zurÃijck.
3. Wie schon in den vorhergingen Kapiteln erwÃďhnt, kann man Ãijber alle Datenpunkte jedes Device iterieren, und je nach Bedingung ein DatapointCahngedCommand anhÃďngen. Dies bewirkt, dass eine Methode immer dann aufgerufen wird, wenn sich einer der Datenpunkte Ãďndert. Mit der ID der einzelnen Devices kann man dann auf die SpecificAddress zugreifen, und die einzelnen FÃďlle mit einem switch Statement abdecken.

Listing 8: Example multible datapoint usage

```
1           for (Device dev : StaticData.devices.getDevices()) {
2       for (Datapoint dp : StaticData.datapoints.getDatapoints(dev.getId())) {
3           if (!dp.getSpecificDataType().isEmpty()) {
4               dp.setDatapointChangedCommand(dpCmd);                  //!!!
5           }
6       }
7   }
```

```
8
9
10
11  DatapointCommand dpCmd = new DatapointCommand() {
12        @Override
13      public DatapointResponse execute(DatapointResult datapointResult) {
14
15          int deviceID= datapointResult.getDeviceId();
16          switch (datapointResult.getDpId()) {
17              case "datapoint_example_id":
18                  System.out.println(datapointResult.getValue() + âĂ  âĂ  +
                                    deviceID );
19                   break;
20                      }
21          return new DatapointResponse(DatapointResponse.ResponseCode.OK);
22      }
23  };
```

In diesem Beispiel werden die DeviceID und der aktuelle Wert des Datenpunktes, welcher sich geÃďndert hat, ausgegeben.

Mit âĂIJdatapoint.setValue(value)âĂİ kann man einem Datenpunkt einen neuen Wert zuweisen. Dieser wird dann automatisch in die FLexcloud gepublished, und kann von einem anderen Task abgefangen werden.

3.5.4 Datapoint mapping

Um zwei Datenpunkte von zwei verschiedenen Tasks zu mappen, und somit Daten zu Ãijbertragen, muss man einen Eintrag in der âĂIJdatapointMapâĂİ hinzufÃijgen. Um die Daten zu empfangen, muss man die conf.db des zweiten Tasks aktualisieren. Zu beachten ist, dass das Mapping nur bei dem Task gemacht werden muss, welcher die Daten bekommen soll. Man verbindet also den Datenpunkt eines anderen Tasks auf einen eigenen Datenpunkt. Wichtig dabei ist, dass man bei der âĂIJdatapoint1IdâĂİ zuerst das Label des Devices, zu welchem der Datenpunkt gehÃűrt, und anschlieÃ§end die ID des ersten Datenpunkt angibt. âĂIJdatapoint2IdâĂİ wird zuerst der Name des anderen Tasks angegeben, dann das Label des Devices, und anschlieÃ§end die ID des Datenpunktes.

Beispiel: Ich mÃűchte von dem Flextask âĂIJmodbuswallboxâĂİ, welcher in dem Projekt als das Gateway bezeichnet wurde, den Wert der Ladedauer der ersten Wallbox zum âĂIJchargecontrollerâĂİ schicken. In dem Task âĂIJmodbuswallboxâĂİ muss dafÃijr ein Device mit dem Label âĂIJwb1âĂİ und ein Datenpunkt mit der ID âĂIJladedauerWb1âĂİ angelegt werden. Wird nun im Code auf diesen Datenpunkt was gepublished, sieht man âĂIJmodbuswallbox.wb1.ladedauerWb1âĂİ. Damit die Daten jetzt auch ankommen, muss im Task fÃijr den Charge-controller ein Datenpunkt mit der ID âĂIJladedauerWbâĂİ angelegt werden. In die âĂIJdatapointMapâĂİ Tabelle muss dann bei der âĂIJdatapoint1IdâĂİ das Device und der Datenpunkt angegeben

werden, und bei der âĂIJdatapoint2IdâĂİ zuerst der Task Name, das Device und dann der Datenpunkt, der die Daten pulished.

Listing 9: Example Element

```
1       dev1.ladedauer_wb                       modbuswallbox.wb1.ladedauer_wb1
```

3.6 Modbus

3.7 FlexHMI

3.8 Energiemanagement

3.9 Wallboxen

3.9.1 Java

4 Flexlogger

4.1 Untersuchungsanliegen

4.2 Aufbau des Projektes

4.3 Threads

4.4 Performance

4.5 Datenbanken

4.6 Visuelle Darestellung

4.7 Abspeicherung von Daten

4.8 Quarkus

5 test

Siehe tolle Daten in Tab. 1.

Siehe und staune in Abb. 5.

Dann betrachte den Code in Listing 10.

Listing 10: Some code

```
# Program to find the sum of all numbers stored in a list (the not-Pythonic-way)

# List of numbers
numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]

# variable to store the sum
sum = 0

# iterate over the list
for val in numbers:
    sum = sum+val

print("The sum is", sum)
```

	Regular Customers	**Random Customers**
Age	20-40	>60
Education	university	high school

Tabelle 1: Ein paar tabellarische Daten

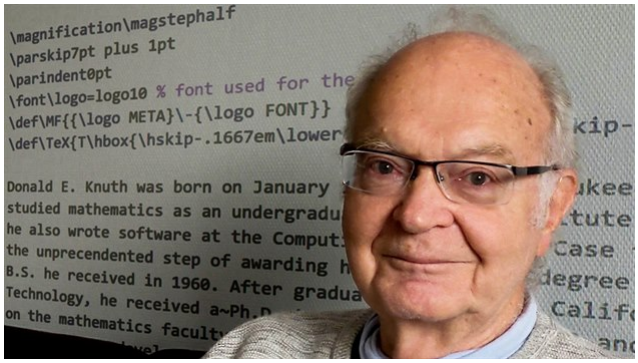

Abbildung 5: Don Knuth – CS Allfather

6 Zusammenfassung

AufzÃdhlungen:

- Itemize Level 1
  - Itemize Level 2
    - Itemize Level 3 (vermeiden)

1. Enumerate Level 1
   a. Enumerate Level 2
      i. Enumerate Level 3 (vermeiden)

Desc Level 1

Desc Level 2 (vermeiden)

Desc Level 3 (vermeiden)

Literaturverzeichnis

Abbildungsverzeichnis

1 Darstellung des Aufbaues 4
2 ÃIJbersicht Ãijber die fÃijnf Wallboxen auf der HMI: 12
3 Detailansicht der OberflÃďche fÃijr die Einzelnen Wallboxen 13
4 Darstellung des Aufbaues 14
5 Don Knuth – CS Allfather 24

Tabellenverzeichnis

1 Ein paar tabellarische Daten 23

Quellcodeverzeichnis

1 Example Element 11
2 Example Element 15
3 Example Element 15
4 Example Element 16
5 Example Element 18
6 Example Datapoint 19
7 Example catapoint usage 19
8 Example multible datapoint usage 19
9 Example Element 21
10 Some code 23

Anhang