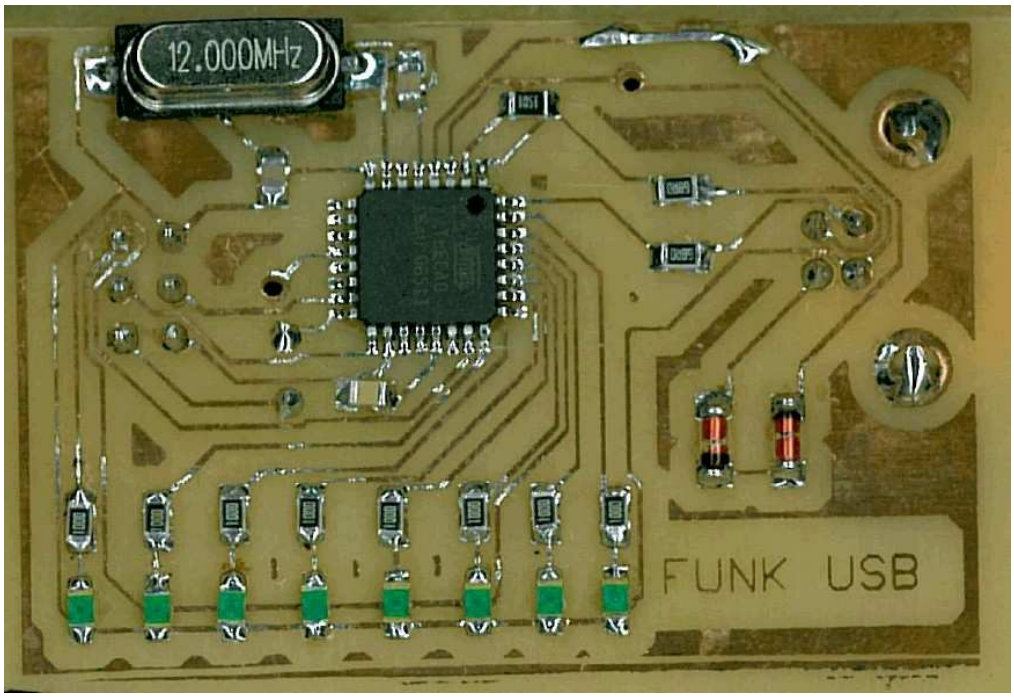
UWPDC
(USB Wireless Power Device Control)
Dokumentation



Teammitglieder:

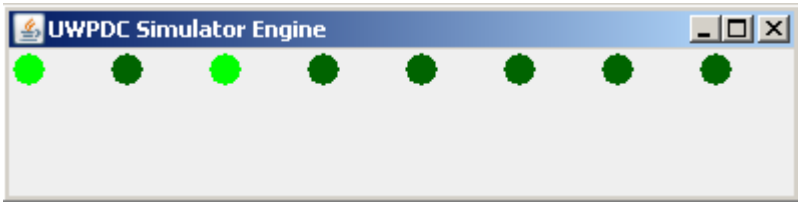
Widmann, Daniel (PL)
Appenmaier, Daniel
Sättele, Ralf
Müller, Daniel

Inhalt

1. Funktionalität

USB Wireless Power Device Control, nachfolgend UWPDC genannt, ist eine Software die es dem Benutzer ermöglicht, ein USB-Gerät zum Schalten von handelsüblichen Funksteckdosen zu steuern.
Dies kann auf drei verschiedene Arten gemacht werden:

- Simulator



Der Simulator ermöglicht das Simulieren des Gerätes, die 8 Geräte werden mit LED's symbolisiert. Hier ist Beispielsweise Leitung 1 und 3 aktiv.

- Direkt

Hier wird das USB Gerät, welches lokal am Rechner angeschlossen ist, von der Software angesteuert. Den Status geben die 8 LED's auf dem Gerät aus.

- Über Netzwerk

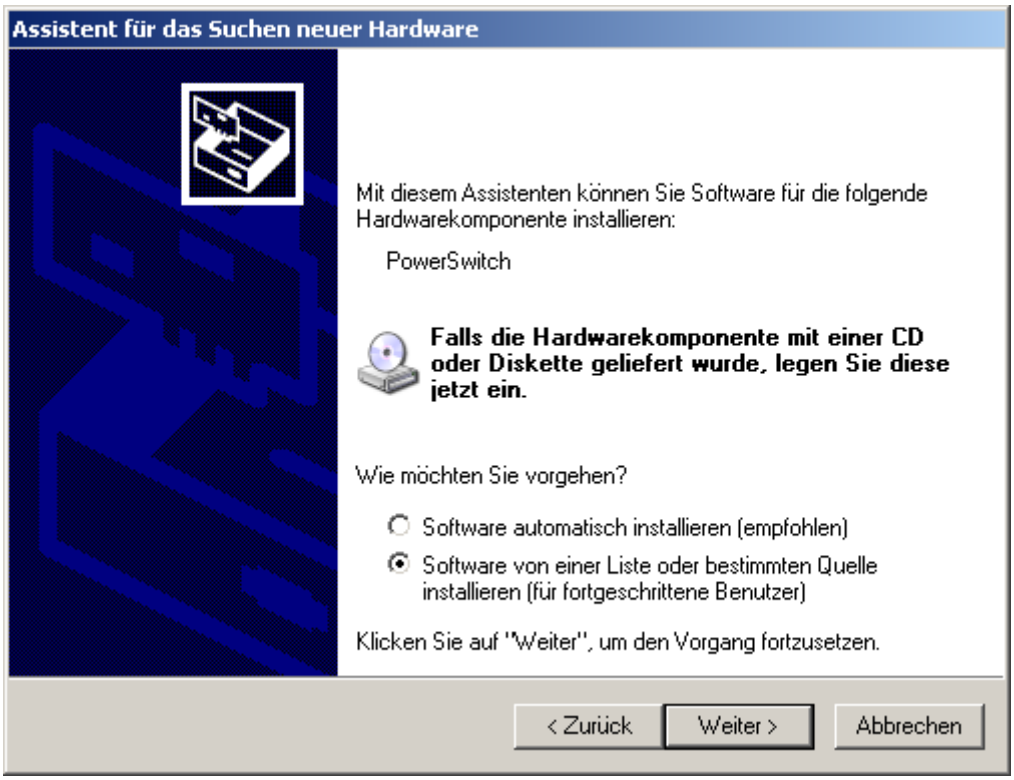
Die Software steuert das USB Gerät an, welches an einem anderen PC angeschlossen ist.
So können beispielsweise von der Arbeit aus Geräte zu Hause eingeschaltet werden (PC, Heizung, ...)

2. Installation des USB Gerätes

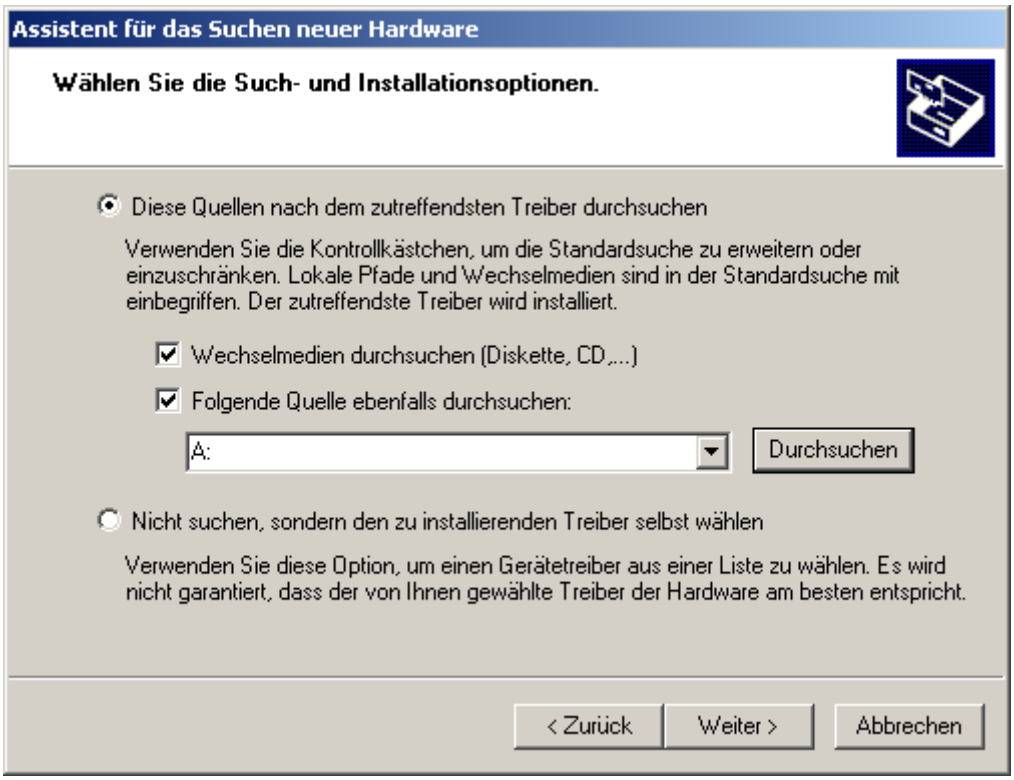
Die Installation des USB-Funkmoduls ist kinderleicht. Nach dem Einstecken öffnet sich im laufenden Windows-Betrieb folgendes Fenster:

Der Anwender hat „Nein, diesmal nicht“ auszuwählen, da der Treiber vorhanden ist und nicht über WindowsUpdate bezogen werden kann.

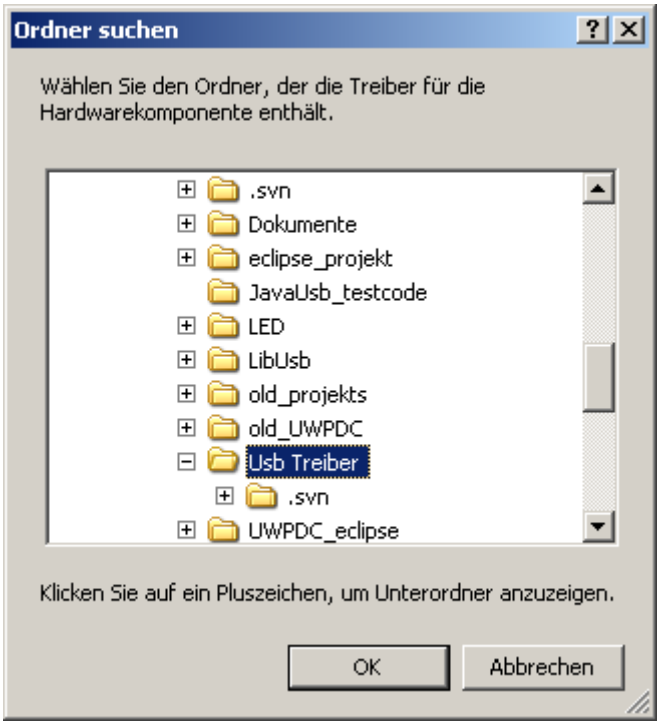
Im nächsten Fenster ist der untere Punkt auszuwählen, damit der Treiber direkt ausgewählt werden kann.



Nach dem Klick auf „Weiter“ erscheint nun:

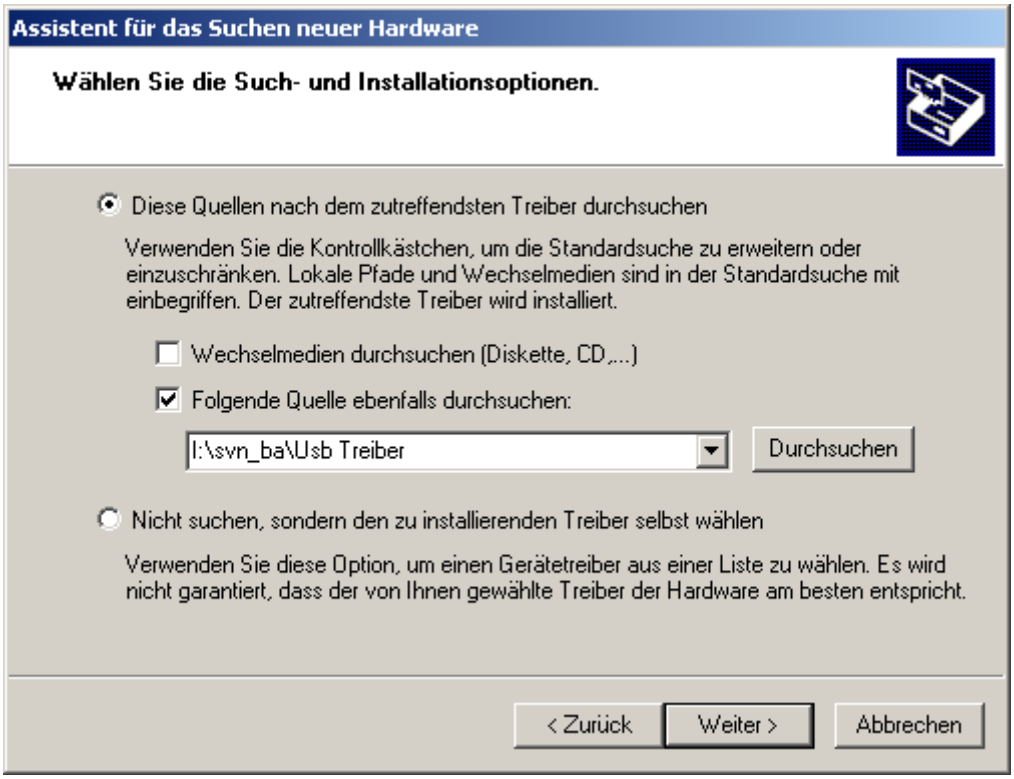


Durch einen Klick auf „Durchsuchen“ kann der Speicherort des Treibers direkt ausgewählt werden.



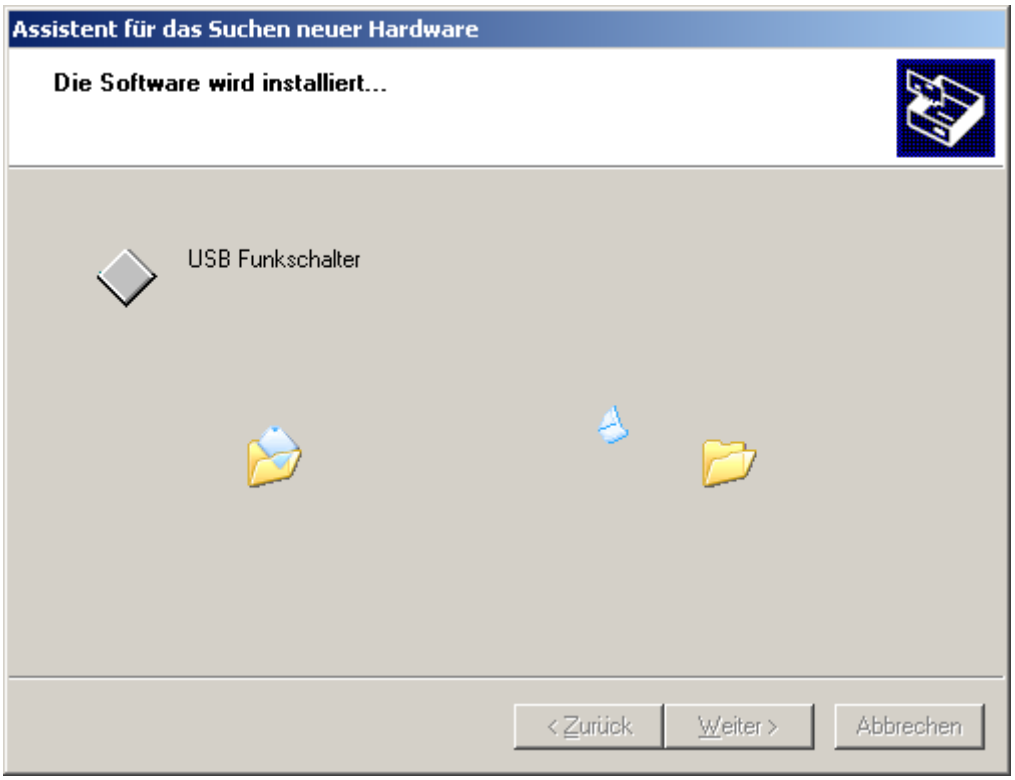
Im Projektordner ist dieser unter svn_ba/Usb Treiber zu finden, sollte der Treiber anderswo liegen so ist selbstverständlich die richtige Quelle auszuwählen und durch einen Klick auf „OK“ zu bestätigen.

Der entsprechende Pfad wird ins Fenster übernommen.



Anschliessend mit „Weiter“ bestätigen.

Der Treiber wird nun installiert:

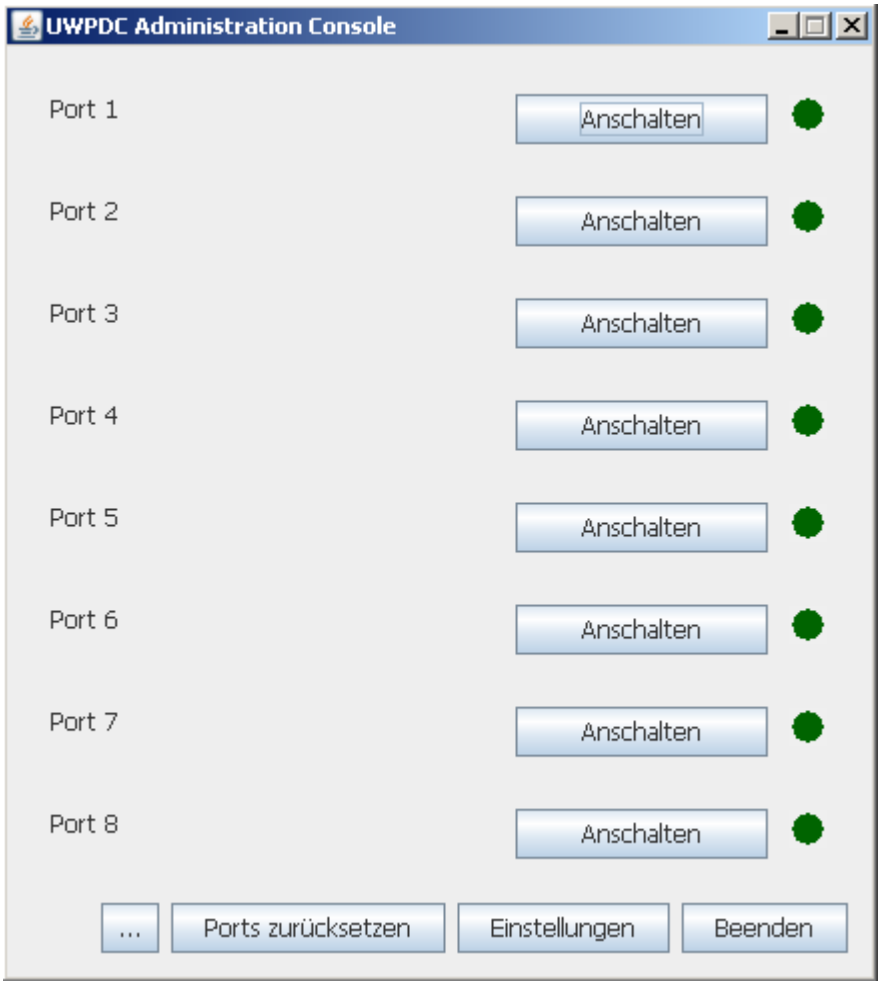


Nach Fertigstellung erscheint das Bestätigungsfenster:

Durch einen Klick auf „Fertig stellen“ ist die Installation abgeschlossen, das USB-Modul müsste nun funktionieren.

3. Bedienung der Software

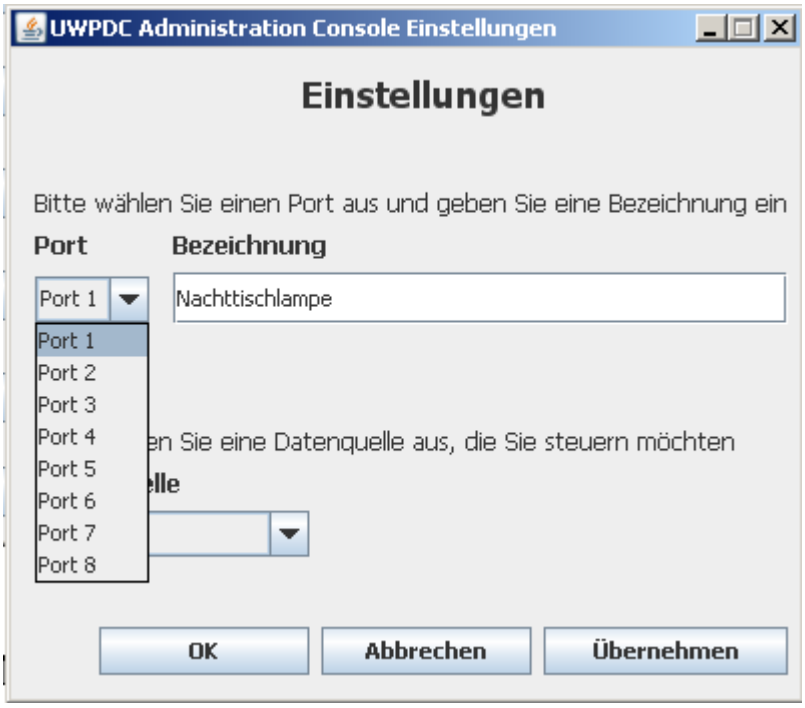
Die Software wird über die .jar Datei UWPDC.jar gestartet. Es erscheint die UWPDC Administration Console:



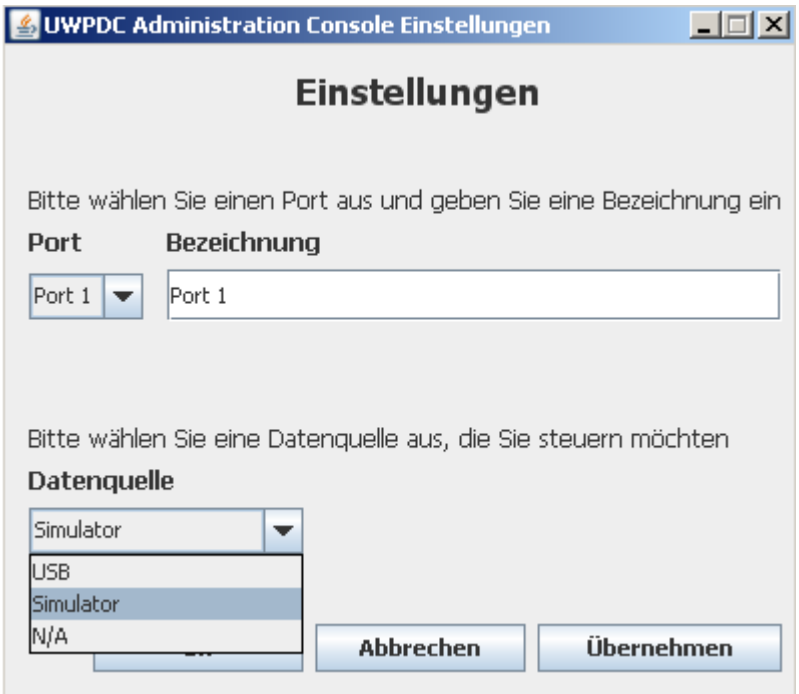
Alle Kanäle sind defaultmäßig auf AUS geschaltet, der Benutzer kann einzelne Kanäle durch Klick auf den „Anschalten“-Button einschalten. Die Kontrollleuchte des Ports wird dann aktiv, bzw. die Leitung erhält Strom. Beispielsweise geht die Nachtischlampe an.

Über dem Button 

kann der Benutzer die Namen der einzelnen Ports ändern:



Hierfür muss er links das Menü aufklappen und bei „Bezeichnung“ den Namen des Gerätes, welches er an Port 1 angeschlossen hat, eingeben. Diesen Vorgang muss er für alle Ports die er umbenennen möchte wiederholen.
Außerdem kann er hier die Datenquelle auswählen. Hier kann er den Simulator, das USB-Gerät oder keine Quelle (N/A) auswählen. Wird N/A gesetzt so kann kein Gerät gesteuert werden. Dabei werden die Ports zurückgesetzt, der Proxy hängt das gegenwärtige Gerät ab.
Die Fernsteuerung im Netzwerk ist mit einem Applet und mit einer eigenständigen Java-Applikation möglich. Die Einschränkung: Das Applet kann nur mit dem Server Verbinden, von dem auch das Applet geladen wurde (ist eine Sicherheitseinschränkung der Java VM).



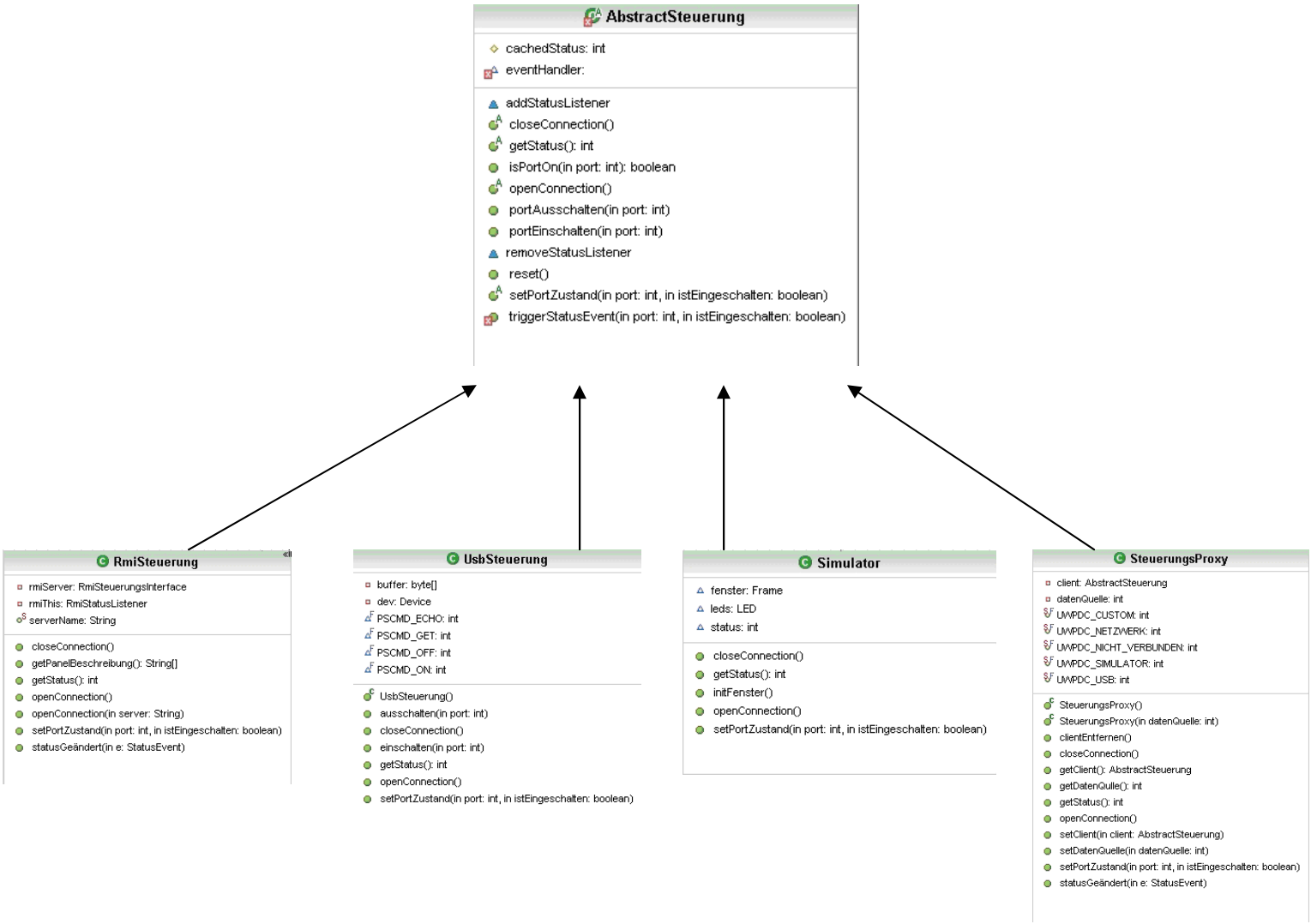
Mit dem Button 

kann der Benutzer alle Ports auf Null zurücksetzen, d.h. alle Steckdosen werden vom Strom getrennt.

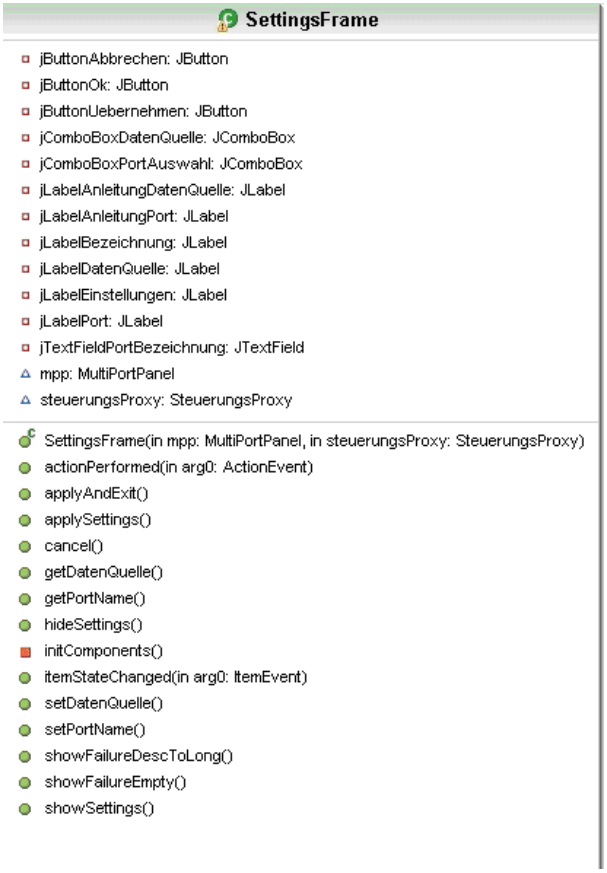
Die Taste 

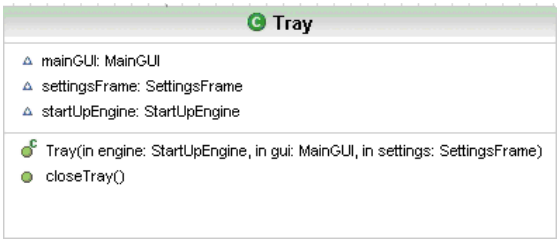
ist der sogenannte „KnightRider-Modus“. Mit dem Drücken der Taste werden die Ports wie ein Lauflicht ein- und ausgeschaltet. Der KnightRider-Modus schaltet sich nach gut 12 Sekunden automatisch ab.

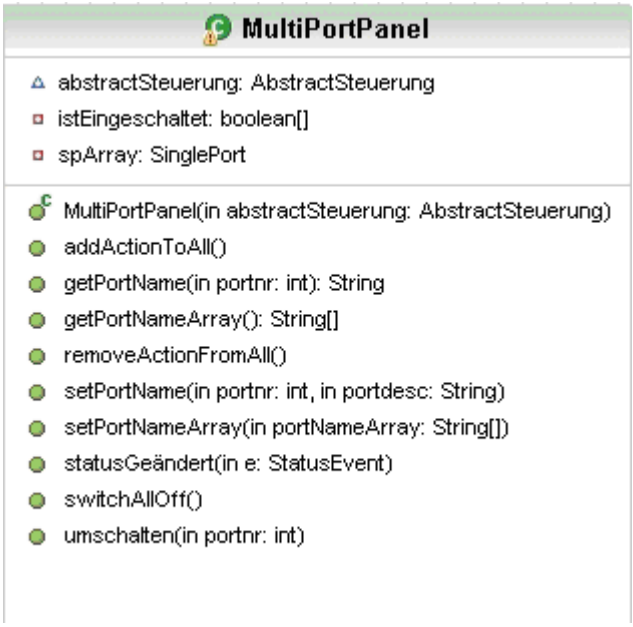
4. Programmtechnischer Aufbau




MainGUI
jButtonBeenden: JButton
jButtonEE: JButton
jButtonEinstellungen: JButton
jButtonReset: JButton
mpp: MultiPortPanel
settingsFrame: SettingsFrame
startUpEngine: StartUpEngine
steuerungsProxy: SteuerungsProxy
MainGUI(in startUpEngine: StartUpEngine, in mpp: MultiPortPanel, in settingsFrame: SettingsFrame, in steuerungsProxy: SteuerungsProxy)
actionPerformed(in arg0: ActionEvent)
hideMainGui()
initComponents()
knightRiderModuleEasterEgg()
showMainGUI()
showSettings()
shutDown()


SettingsFrame
jButtonAbbrechen: JButton
jButtonOk: JButton
jButtonUebernehmen: JButton
jComboBoxDatenQuelle: JComboBox
jComboBoxPortAuswahl: JComboBox
jLabelAnleitungDatenQuelle: JLabel
jLabelAnleitungPort: JLabel
jLabelBezeichnung: JLabel
jLabelDatenQuelle: JLabel
jLabelEinstellungen: JLabel
jLabelPort: JLabel
jTextFieldPortBezeichnung: JTextField
mpp: MultiPortPanel
steuerungsProxy: SteuerungsProxy
SettingsFrame(in mpp: MultiPortPanel, in steuerungsProxy: SteuerungsProxy)
actionPerformed(in arg0: ActionEvent)
applyAndExit()
applySettings()
cancel()
getDatenQuelle()
getPortName()
hideSettings()
initComponents()
itemStateChanged(in arg0: ItemEvent)
setDatenQuelle()
setPortName()
showFailureDescToLong()
showFailureEmpty()
showSettings()


Tray
mainGUI: MainGUI
settingsFrame: SettingsFrame
startUpEngine: StartUpEngine
Tray(in engine: StartUpEngine, in gui: MainGUI, in settings: SettingsFrame)
closeTray()

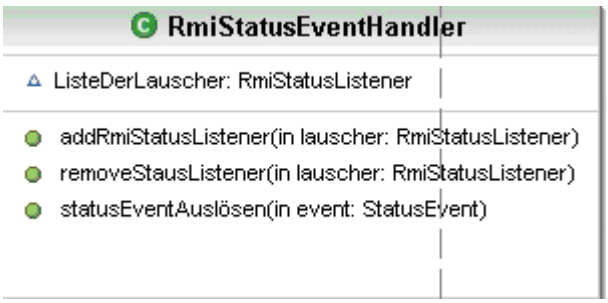

MultiPortPanel
abstractSteuerung: AbstractSteuerung
istEingeschaltet: boolean[]
spArray: SinglePort
MultiPortPanel(in abstractSteuerung: AbstractSteuerung)
addActionToAll()
getPortName(in portnr: int): String
getPortNameArray(): String[]
removeActionFromAll()
setPortName(in portnr: int, in portdesc: String)
setPortNameArray(in portNameArray: String[])
statusGeändert(in e: StatusEvent)
switchAllOff()
umschalten(in portnr: int)


SinglePort
jButtonUmschaltButton: JButton
jLabelBezeichnung: JLabel
led1: LED
mpp: MultiPortPanel
portNr: int
SinglePort(in portNr: int, in mpp: MultiPortPanel)
actionPerformed(in arg0: ActionEvent)
addActionToSP()
getPortName(): String
initComponents(in portNr: int)
ledAusschalten()
ledEinschalten()
removeActionFromSP()
setButtonLabel(in desc: String)
setPortName(in desc: String)


RmiServer
eventHandler: RmiStatusEventHandler
panel: MultiPortPanel
registry: Registry
steuerung: AbstractSteuerung
RmiServer(in steuerung: AbstractSteuerung)
addRmiStatusListener(in listener: RmiStatusListener)
getPanelBeschreibung(): String[]
getStatus(): int
removeRmiStatusListener(in listener: RmiStatusListener)
setMultiPortPanel(in panel: MultiPortPanel)
setPortZustand(in port: int, in istEingeschalten: boolean)
statusGeändert(in e: StatusEvent)
stopServer()

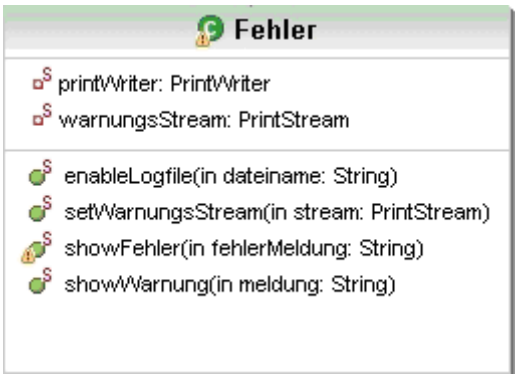

«interface»
RmiSteuerungsInterface
addRmiStatusListener(in listener: RmiStatusListener)
getPanelBeschreibung(): String[]
getStatus(): int
removeRmiStatusListener(in listener: RmiStatusListener)
setPortZustand(in port: int, in istEingeschalten: boolean)


«interface»
RmiStatusListener
statusGeändert(in e: StatusEvent)


RmiStatusEventHandler
ListeDerLauscher: RmiStatusListener
addRmiStatusListener(in lauscher: RmiStatusListener)
removeStausListener(in lauscher: RmiStatusListener)
statusEventAuslösen(in event: StatusEvent)


StartUpEngine
mainGUI: MainGUI
mpp: MultiPortPanel
rmiServer: RmiServer
settingsFrame: SettingsFrame
steuerungsProxy: SteuerungsProxy
storageEngine: StorageEngine
tray: Tray
main(in args: String[])
shutDown()
startUp()


StorageEngine
dateiName: String
StorageEngine(in dateiName: String)
loadSettings(in mpp: MultiPortPanel, in steuerungsProxy: SteuerungsProxy)
saveSettings(in mpp: MultiPortPanel, in prox1: SteuerungsProxy)


Fehler
printWriter: PrintWriter
warnungsStream: PrintStream
enableLogfile(in dateiname: String)
setWarnungsStream(in stream: PrintStream)
showFehler(in fehlerMeldung: String)
showWarnung(in meldung: String)

5. Implementierungskonzepte und Einschränkungen

5.1. die GUI

Nach den anfänglichen Versuchen mit einem GridLayout, welches die Verfügbare Fensterbreite zu gleichen Teilen auf die Bereiche Labe, Button und LED aufteilt wurde den GUI-Ingenieuren schnell klar, dass die derzeitige Oberfläche nur als Alpha-Version betrachtet werden kann und einer Portierung in ein professionelles Layout bedarf. Die Entscheidung ist auf Swing gefallen, welches sich mit NetBeans relativ einfach erstellen lässt.

Im ersten Schritt wurde der Einstellungsdialog im Rahmen der Migration auf Swing in eine eigene Klasse geschrieben. Es stellte sich jedoch heraus, dass bei der Implementierung der GUI auf Swing-Basis viele Eigenschaften vom Betriebssystem geerbt werden, wie z.B. die Wahl der Schriftgröße und Art. Dies wurde dann im zweiten Build der Klasse SettingsFrame berücksichtigt. Die Schriftarten wurden diesmal hart implementiert. Das Layout kann sich so auf den unterschiedlichen Plattformen nicht mehr verändern.

Im zweiten Schritt wurde dann die MainGui portiert. Da sie ebenfalls eine eigene Klasse darstellt, mussten die zuvor integrierten Funktionen ausgelagert werden.

Das Layout der MainGui verursachte durch die Fülle der 28 Objekte unvorhersehbare Probleme. Es kam zu Veränderungen des Layouts während der Laufzeit durch die veränderbaren Labels / Buttons. Daher musste eine einfach zu administrierende, jedoch ebenso leistungsfähig wie auch portable Oberfläche geschaffen werden.

Das Design wurde wie folgt verabschiedet:

- SinglePort Panel – Eine Bezeichnung, ein Button, eine LED
- Acht SinglePorts werden zu einem MultiPortPanel zusammengefasst, welches Funktionalität nach außen anbietet. Dabei wurde darauf geachtet, dass das MPP auch ohne StorageEngine funktioniert.
- Die Gui besteht aus dem Hauptfenster und dem Einstellungsfenster
- Das Hauptfenster besteht aus einem Panel und den Buttons.
- Das Panel beinhaltet ein Array mit 8 Single-Ports, sie sind quasi eine Zeile, die haben jeweils eine Beschriftung, einen Umschalte-Button und eine LED, welche den Zustand des Ports anzeigt.

Als zugrunde liegendes Layout wurde beim SinglePort auf ein javax-Layout zurückgegriffen, welches mit NetBeans 5.5.1 erzeugt wurde. Das Panel implementiert einen StatusListener, der aufgrund von Status-Events die einzelnen Ports umschaltet. Einen direkten Zugriff auf die SinglePorts ist nicht möglich. Ein Zugriff auf die SinglePorts ist nur über das MultiPortPanel möglich. Die Bezeichnungen der Labels können unter dem Einstellungsmenü

geändert werden. Dazu muss der passende Port ausgewählt werden. Dann muss im nebenstehenden Textfeld die gewünschte Bezeichnung angegeben werden. Ungültige Bezeichnungen sind leere Strings, oder Strings mit mehr als 30 Buchstaben. Der Benutzer wird mit einer Fehlermeldung hierüber unterrichtet. Zudem hat der Benutzer unter dem Einstellungsmenü die Möglichkeit die zu steuernde Datenquelle während der Laufzeit zu verändern. Im Hintergrund wird dabei eine Methode auf dem SteuerungsProxy aufgerufen, welcher dementsprechend die Datenquelle ändert.

Als Datenquellen stehen zur Verfügung:

- Simulator (Default-Einstellung, falls Einstellungen nicht geladen werden konnten)
- USB – der USB-Funkschalter wird damit gesteuert
- N/A (Not available) – kein Gerät kann gesteuert werden. Dabei werden die Ports zurückgesetzt, der Proxy hängt das gegenwärtige Gerät ab.

Der Benutzer hat zudem die Möglichkeit die Ports im Hauptmenü zurückzusetzen. Dabei wird eine Reset-Methode auf dem SteuerungsProxy ausgeführt. Das Endgerät liefert durch Statusevents das erfolgreiche Durchführen des Resets. Erst dann werden die Buttons und LEDs zurückgesetzt.

Die Bezeichnungen der Ports werden beim Start durch die StorageEngine geladen. Dabei wird dem Panel ein String Array mit den Bezeichnungen übergeben, das MPP wertet dieses dann selbstständig aus. Beim Speichern übergibt das MPP der StorageEngine ein String Array, die StorageEngine wertet dieses dann aus. Die Auswahl der Datenquelle wir in Form eines Integers in der neunten Zeile der Konfigurationsdatei abgelegt und wieder ausgelesen. Dazu kommuniziert die StorageEngine mit dem Steuerungsproxy

Die Klassen:

- mainGUI
- MultiPortPanel
- settingsFrame
- StorageEngine
- SteuerungsProxy
- RmiServer
- und Tray werden von der StartUpEngine geladen

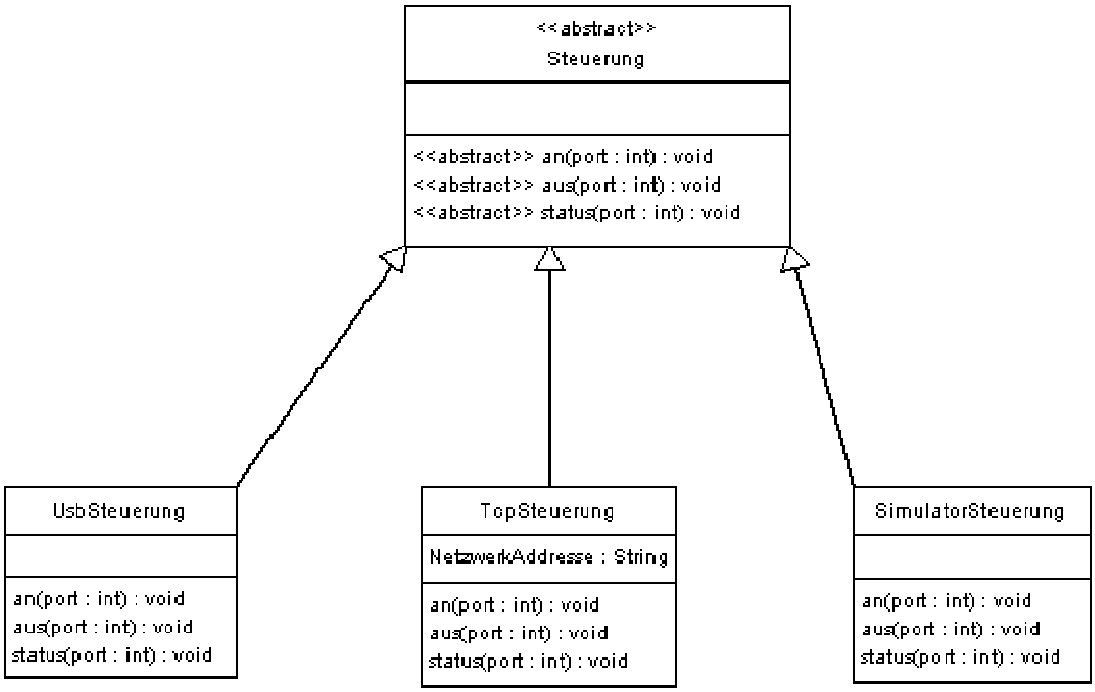
Die StartUpEngine hat eine Main-Methode und startet zu Beginn alle Prozesse und beendet diese auch wieder durch Aufruf der ShutDown() Methode. Die Klasse Fehler enthält statische Methoden, auf die jede andere Klasse zugreifen kann. Mithilfe der Klasse Fehler können dem Benutzer Fehlermeldungen angezeigt werden. Diese erscheinen dann als MessageDialog der Klasse JOptionPane auf der jeweiligen Benutzeroberfläche. Zudem bietet Fehler ein Logging mit Zeitstempel der Form YYYY-MM-DD HH:MM:SS an, womit gezielt Debugging-Informationen entnommen werden können. Um das Logging zu aktivieren

muss die Methode enableLogfile(Dateiname) aufgerufen werden. Fortan wird mitgeloggt. Dabei wird stets an die bereits vorhanden Logfile weitergeschreiben.

5.2. Die Steuerung

Die Ansteuerung des physikalisch UsbGeräts und des Simulators, direkt angesteuert und übers Netzwerk angesteuert sollte möglichst transparent ablaufen. Es sollte für Grafische Oberfläche also keine Rolle spielen was für ein Gerät tatsächlich verwendet wird. Hier gibt es grundsätzlich 2 Möglichkeiten diese Polymorphie zu erreichen.

5.2.1. Variante: abstrakte Basisklasse



Es gibt eine Abstrakte Basisklasse die den Rahmen festlegt und die Kind-Klassen füllen diesen dann mit unterschiedlichen Funktionalitäten.

5.2.2. Variante: Streams

Eine andere Möglichkeit ist diese Transparenz über Streams zu erreichen. Das es also eine Steuerungs Klasse gibt der man je nach dem was für ein Gerät man verwendet einen anderen Stream zuweißt. Hierbei müssen die verschiedenen Streams dann das gleiche Protokoll verwenden.

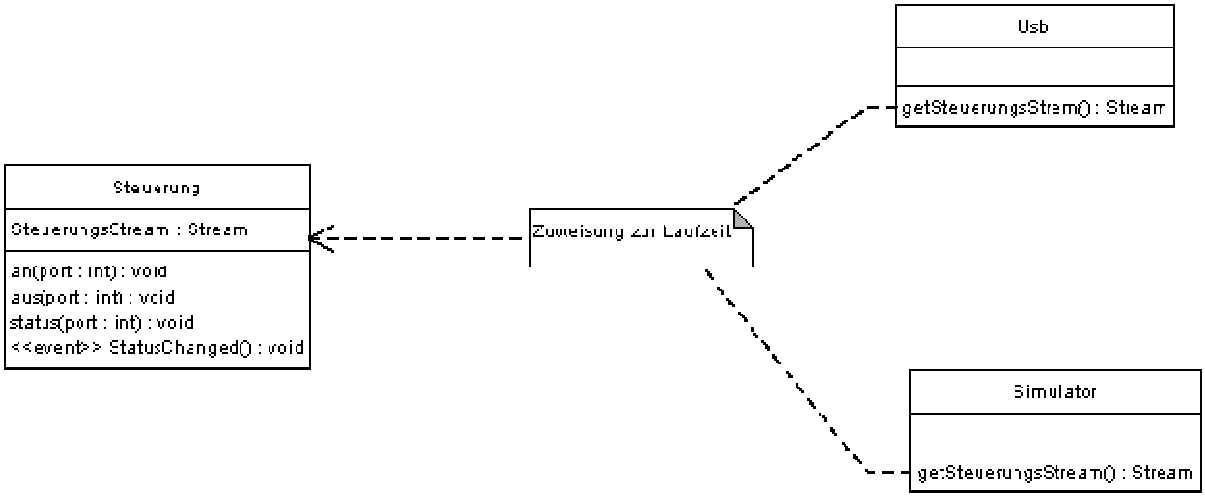


Abbildung 1: Steuerung über Streams

5.2.3. Vor- und Nachteile

Variante 2 hätte den Vorteil, dass sich Streams realtiv einfach übers Netzwerk verwenden lassen, denn Netzwerk-Sockets arbeiten in Java auch mit Sreams. Wir haben uns aber für Variante 1 entschieden, da man so die vorteile der Objektorientierten Programmierung nutzen kann. Es ist deutlich einfacher, auch wenn das Projekt schon weit fortgeschritten ist, Änderungen vorzunehmen. Ein weiterer Problem der Streams ist das Debugging. In Streams kann man schlechter reinschauen als in Klassen (Dank des Debuggers ☺). Ein weiterer Nachteil ist hier das der Simulator auch einen Stream mit diesem Protokoll emulieren muss.

5.3. Das USB-Gerät

5.3.1. USB Grundlagen

Der USB Bus sollte den Meisten Personen geläufig sein, die mit Computern zu tun haben. Zum Entwickeln von USB Geräten und der dazugehörigen Software ist jedoch etwas Hintergrundwissen über USB nötig. Da die USB Spezifikationen sehr umfangreich sind, wird hier überwiegend auf die Dinge eingegangen, die für diese konkrete Anwendung Wichtig sind.

5.3.1.1. USB Geschwindigkeit

Der USB Funkschalter verwendet den langsamsten, verfügbaren Übertragungsmodus, den es für USB gibt, Low-Speed USB mit 1,5 MBit/s bzw. 187,5 KByte/s. Höhere Datenraten sind mit dem verwendeten Mikrocontroller und der Software Implementierung nicht möglich. Die theoretisch möglichen Datenraten können auch bei weitem nicht erreicht werden, für diese einfachen Steuerungsaufgaben ist dies jedoch kein Problem.

5.3.1.2. USB Endpoints

Die Kommunikation läuft beim USB über Endpoints ab. Diese Endpoints stellen unabhängige Kommunikationskanäle dar, mit denen entweder Daten gesendet oder empfangen werden können. Die Enpoints sind durchnummeriert, dabei sind die Endpoints in feste Bereiche zum Lesen und Schreiben eingeteilt. Eine besondere Rolle nimmt der Enpoint mit der Nummer 0 ein. Er ermöglicht die Kommunikation mit ControlMessages, jedes Gerät muss diesen Control Endoint Implementieren, denn Windows ließt Geräteeigenschaften über ihn aus. Dadurch ist es erst möglich, dass Windows den Richtigen Treiber für ein USB-Gerät installiert.

Die Kommunikation auf den anderen Endpoints kann auf verschiede Arten Ablaufen, da aber Low-Speed-Usb verwendet wird, ist nur der Interrupt Transfer erlaubt.

5.3.1.3. USB Treiber

Zur Kommunikation mit dem USB-Gerät wird LibUsb verwendet, eine frei verfügbare Bibliothek, die den Zugriff auf die Endpoints ermöglicht, ohne das ein Anwendungsspezifischer Kernel-Treiber programmiert werden muss. Dazu muss für das Usb-Gerät ein generischer LibUsb Treiber installiert werden. In der späteren Anwendung greift man dann mit Hilfe einer LibUsb-dll auf den LibUsb Treiber zu und kann mit dem Usb-Gerät kommunizieren. Damit Windows weiß, dass es den LibUsb Treiber für den Usb Funkschalter verwenden soll, muss eine Inf-Datei Erzeugt werden. In dieser Datei wird festgehalten, welche Treiber-Dateien für eine Bestimmte PID/VID Kombination verwendet werden soll.

5.3.1.4. Verwenden der LibUsb in Java

LibUsb ist eigentlich für c / c++ programmiert, es stellt seine Funktionen über Dlls bereit. Um diese Funktionen komfortabel in Java nutzen zu können wird eine LibUsb Java Wrapper verwendet (http://libusbjava.sourceforge.net/wp/). Die ganzen Funktionen und Daten werden hier über Klassen zur Verfügung gestellt und ermöglichen so Objektorientierten Zugriff auf USB. Da die LibUsb und der LibUsb Java Wrapper auch Linux kompatibel bzw. sogar dort entstanden sind sollte ein Betrieb des ganzen Systems unter Linux relativ unproblematisch möglich sein.

Alle Dlls, die zum Betreiben des Libusb-Wrapper benötigt werden, werden automatisch bei der Treiberinstallation ins Windows Systemverzeichnis kopiert.

5.3.2. Hardware

Die Hardware Besteht aus den folgenden Bausteinen:

8 bit AVR Mikrocontroller, er ist Quasi das Gehirn des USB Funkschalters.

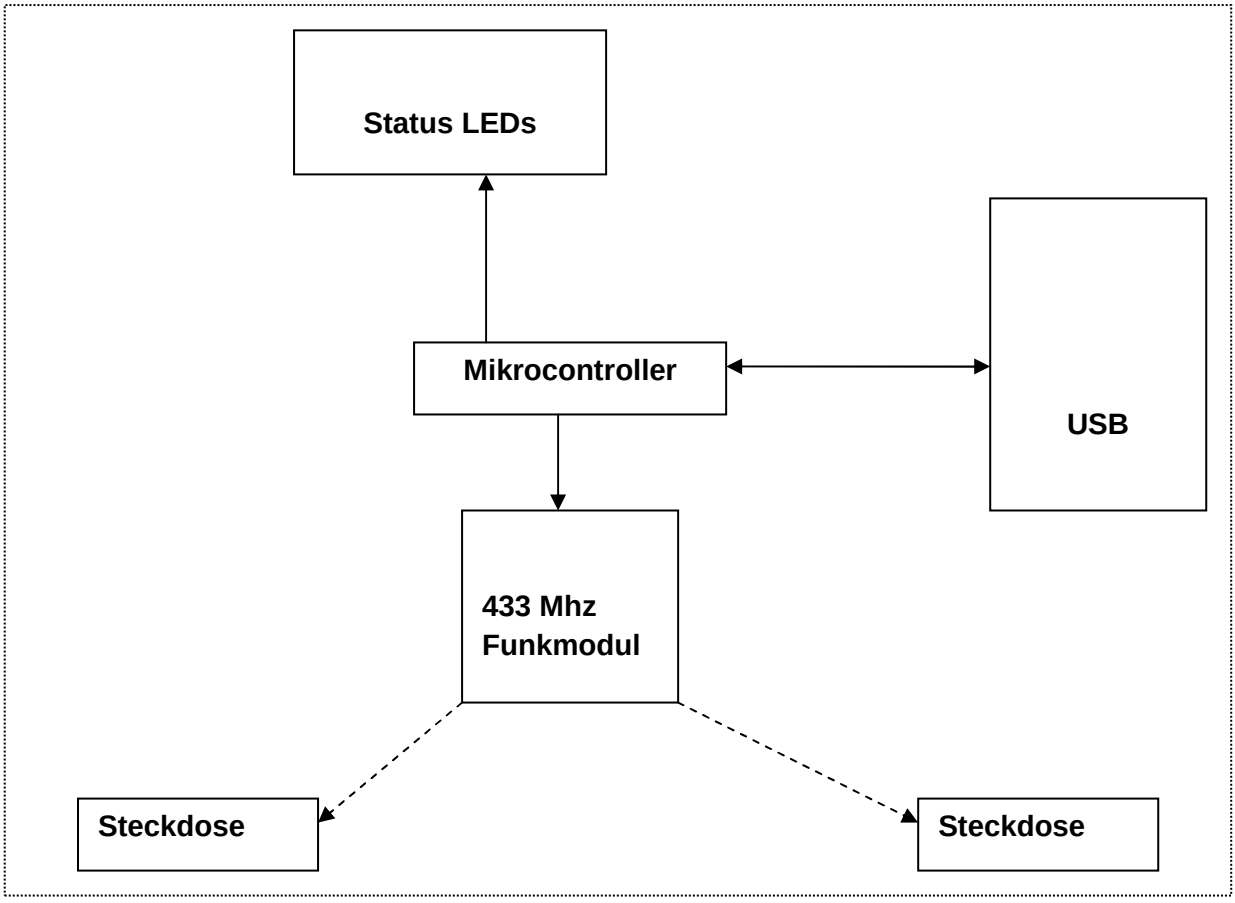
Quarz: Da die USB Kommunikation komplett in der Firmware des Mikrokontrollers implementiert ist, wird ein 12 MHz Quarz Benötig, um die USB-Timing-Spezifikationen einhalten zu können.

Spannungsregler: Um das Gerät über den USB Anschluss versorgen zu können, müssen die 5V von der USB Buchse auf ca. 3,3 V gewandelt werden. Hierzu werden einfach 2 Si-Dioden in Durchlassrichtung in Reiche geschalten, durch den Spannungsabfall von ca. 0,7 V pro Diode werden dann 3,6V erreicht. Dies liegt innerhalb den Spezifikationen für die USB Übertragungspegel.

Status LEDs: Des weiteren befinden sich 8 LEDs auf der Platiene, die den Zustand der jeweiligen Funksteckdosen anzeigen.

433 MHz Funkmodul: Zum Steuern der Funksteckdosen wird dann noch ein 433,92 Mhz AM Funkmodul benötigt. Der Mikrocontroller Emuliert hierbei das Signal eines Handsenders, mit dem die Funksteckdosen ursprünglich geschalten wurden. Das Funkmodul wird für den Betrieb des USB Geräts nicht zwingend benötigt.

Schematische Ansicht:



5.3.3. Firmware

Die Firmware basiert auf einem unter GPL verfügbaren Software von Objective Development(http://www.obdev.at/products/avrusb/index.html). Es wurde eine Beispielanwendung erweitert, die ursprünglich nur zum schalten von LEDs verwendet wurde. Es wurde dann der Code zum Emulieren der Funkfernsteuererung hinzugefügt. Die Einzelnen Funksteckdosen verhalten sich dabei gleich wie die LEDs. Außerdem wurde die Firmware um je einen Interrupt Endpoint zum Lesen und Schreiben erweitert, da die Eingesetzte JAVA USB Library Probleme mit Geräten ohne zusätzliche USB Endpoints hat. Das ursprüngliche Kommunikationsprotokoll wurde zusatzlich auf diesen Endpoints implementiert, so dass man das USB-Gerät sowohl über ControlMessages als auch über die Interrupt Endpoints steuern kann.

5.3.4. Das Protokoll

Das Protokoll wurde von dem Ursprünglichen Anwendungsbeispiel übernommen, man kann das USB-Gerät jedoch sowohl über Controlmessages als auch über die Interrupt Endpoints steuern. Das Protokoll ist relativ einfach aufgebaut, ein Frame besteht immer aus einem Kommando und einem Parameter, eventuell antwortet das Gerät mit einem Rückgabewert. Kommando und Parameter sind jeweils ein Byte groß und werden einfach hintereinander übertragen.

Kommando	**Parameter**	**Rückgabewert**	**Beschreibung**
0x00-Echo	*	*	Gibt die als Parameter gesendeten Daten als Rückgabewert zurück, zum Testen der USB Verbindung
0x01-Get	*	Status als Bit-Feld	Gibt den Aktuellen Status der Ports als Bit-Feld zurück(für jeden Port je ein Bit im Rückgabewert
0x02-On	Nummer des Port	-	Schält einen Port ein
0x03-Off	Nummer des Ports	-	Schält einen Port aus

5.4. Fernsteuerung über RMI

Die Fernsteuerung ermöglicht es, mit Hilfe eines Java-Fähigen Browsers, Ports ein und auszuschalten. Einstellungsfunktionen, wie das Ändern eines Portnamens oder der Datenquelle ist nicht möglich.

5.4.1. Möglichkeiten

Wir haben verschiedene Möglichkeiten in Betracht gezogen, wie sich die Fernsteuerfunktion realisieren lässt. Es hätte die Möglichkeit gegeben, das wir ein eigenes Protokoll Implementieren, das über TCP-Sockets funktioniert. Dazu muss man einen Server Programmieren, der Multithreading benötigt. Auch muss man auf beiden Seiten einen Protokoll Handler schreiben. Wir haben uns dann aber für Java RMI entschieden, da dies mit relativ wenig Code funktioniert, und da es in der Java Welt weit verbreitet ist sollte es auch stabiler funktionieren wie eine selbstprogrammierte Lösung.

5.4.2. Funktionsweise der Fernsteuerung

Bei der Fernsteuerung wurde versucht, Oberfläche und Steuerungsfunktionalität zu trennen. Deshalb wird zunächst die Steuerung und danach die Oberfläche beschrieben.

5.4.3. RmiSteuerung

Die RmiSteuerung ist von der AbstraktenSteuerung abgeleitet und kann somit wie eine Lokale Steuerungsklasse verwendet werden. Man kann die Verbindung entweder zu einem bestimmten Server aufbauen oder es werden Standarteinstellungen verwendet. Diese Standarteinstellungen werden in einer statischen Variable gespeichert. Die Standarteinstellung zum öffnen der Verbindung wird benötigt, da AbstractSteuerung keine Argumente zum öffnen einer Verbindung hat, die RmiSteuerung sich aber möglichst ähnlich verhalten verhalten soll.

5.4.4. Kommunikation mit RMI

Die Kommunikation des Fernsteuerungs-Applet mit dem Hauptprogramm findet über Java Remote Methode Invokation (RMI) statt.

RMI ermöglicht es Methoden von Objekten aufzurufen, die sich auf einem anderen Computer befinden. Dabei können Parameter übergeben werden und es können Rückgabewerte empfangen werden. Dies funktionier über Marshalling, was fast das gleiche wie Serialisierung ist. Dadurch ist es auch möglich komplexe Objekte auf einen entfernten Computer zu übertragen.

Um RMI einsetzen zu können ist nicht viel Code nötig, jedoch muss man einige Interfaces und Klassen programmieren. Für jede Klasse, die man übers Netzwerk nutzen will, muss man ein Interface erstellen, das von der Klasse *Remote* abgeleitet ist und die Methoden beinhaltet, die man übers Netzwerk nutzen will. Dann braucht man noch eine Klasse, die dieses Interface Implementiert. Wir haben uns dafür entschieden, eine eigene Klasse zu erzeugen, die RmiServer Klasse. Diese Klasse übernimmt auch verschiedene Dinge, die benötigt werden, damit man das Objekt übers Netzwerk erreichen kann.

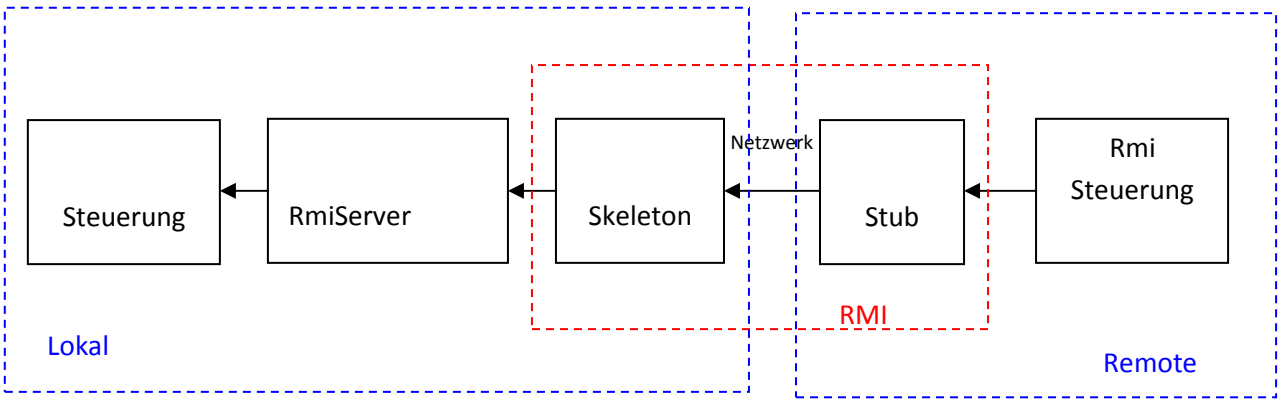


Abbildung 2: Datenfluss beim Funktionsaufruf

5.4.5. Event Callback

Damit die Fersteuerung auch mitbekommt, wenn man lokal einen Port verändert hat, muss das entfernte Objekt benachrichtigt werden. Dies geschieht über einen Callback. Dabei ist die rmiClient Selbst wieder ein Rmi-taugliches Objekt. Beim Aufbauen der Verbindung registriert sich die RmiClient beim RmiServer. Wenn der RmiServer jetzt ein StatusEvent bekommt, ruft er den statusListener des RmiClient über RMI auf und so kann ein anderer PC übers Netzwerk Events empfangen. Für diesen RmiEventListener wurde wieder ein eigenes Interface benötigt.

5.4.6. Generierung des Stubs

Damit die RMI-Kommunikation funktioniert, müssen Stubs erzeugt werden (Stubs sind Klassen, die Java intern für die Rmi Kommunikaton benötigt). Ab Java 1.5 können die Stubs automatisch zur Laufzeit erzeugt werden, hierzu muss beim starten der Java Anwendung ein Kommandozeilen Parameter angegeben werden. Deshalb wurden die Stubs mit Hilfe des Kommandozeilenprogramms rmic aus dem SDK erzeugt. Mit der Batchdatei *RmiStubsErzeugen.bat* können alle benötigten Stubs erzeugt werden. Dies ist jodoch nur nötig, falls änderungen an den Interfaces vorgenommen wurden.

5.4.7. Die Oberfläche der Fernsteuerung

Das Applet verwendet das gleiche Panel, wie die normale GUI. Der Konstruktor des Panels wird lediglich mit der RmiSteuerung aufgerufen, damit die Steuerbefehle übers Netzwerk geschickt werden. Wir wollten auch, dass im Applet die gleiche Beschriftung der Ports angezeigt wird, wie in der GUI des Server. Deshalb haben wir im RmiServer noch eine zusätzliche Methode implementiert, die es ermöglich die Beschriftungen über RMI zu übertragen.

5.4.8. Verbindung zum richtigen Server

Das Applet muss den Namen des Server kennen, auf dem die UWPDC GUI ausgeführt wird. Dazu wird in der HTML Datei, in die das Applet eingebettet ist, mit einem param-tag der Servername festgelegt.

```
<param name="serverName" value="localhost">
```

5.5. Das System-Tray-Icon

Beim Minimieren verschwinden Hauptmenü und Einstellungsmenü als TrayIcon im SystemTray. Der Anwender hat hier die Möglichkeit das Hauptmenü sowie das Einstellungsmenü separat voneinander aufzurufen und die Anwendung zu beenden. Per Doppelklick auf das TrayIcon wird das Hauptmenü wieder aufgerufen. Der Simulator bleibt jedoch weiterhin geöffnet und der Anwender hat auch keine Möglichkeit diesen über das TrayIcon aufzurufen bzw. zu beenden. Dies ist von den Entwicklern so gewünscht.

Mögliche Erweiterungen des SystemTrays sind die direkte Auswahl der Datenquelle, sowie die direkte Beschriftung der einzelnen Ausgänge. Desweiteren können problemlos unterschiedliche Status,- Informations- und Fehlermeldungen, sowie animierte Icons hinzugefügt werden.

Grundlage für den Quellcode waren die folgenden Quellen:

- http://java.sun.com/developer/technicalArticles/J2SE/Desktop/javase6/systemtray
- http://www.tutego.com/blog/javainsel/2006/03/insel-system-tray-nutzen.html

```
http://libusbjava.sourceforge.net/wp/res/doc/ch/ntb/usb/LibusbJava.html
```

http://de.wikipedia.org/wiki/Usb