Entwicklungsprojekt
Dokumentation

Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

1. Fachsemester Effiziente Mobilität in der Fahrzeugtechnologie – Master

Thema

WebSockets ESP32 – SmartDevice

Betreuer

Prof. Jürgen Walter

Koordinator

Prof. Dr. Eckhard Martens

Teilnehmer

Johannes Marquart (B.Ing.)
Matr.Nr.: 65353

Inhaltsverzeichnis

1 Eigenständigkeitserklärung

Ich versichere hiermit wahrheitsgemäß, die Abschlussarbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles einzeln kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Karlsruhe, den 13. März 2019

Johannes Marquart

Unterschrift

2 Einleitung und Motivation

Einen leichten Einstieg für Schüler in die Roboterwelt bietet der FT32 – Microcomputer. Es handelt sich hierbei um einen ESP32 – Microcontroller, welcher auf einer passenden Platine in der Lage ist, Gleichstrommotoren anzusteuern und Sensoren wie Taster abzuprüfen. Die Programmierung erfolgt über eine speziell für diesen Controller entwickelte Umgebung, welche ohne Installation im Browser eines beliebigen Smart-Devices bedienbar ist. In einem Laborprojekt mit 4 Teilnehmern wurde der Vorgänger des FT32 sowie die Programmieroberfläche aufgebaut. In Folgeprojekten, welche die Entwicklung einer eigenen Platine und mehreren Anschlussmöglichkeiten enthält, wurde festgestellt, dass immer mehr Aufwand in die eigene Einarbeitungsphase gesteckt werden muss. Insbesondere die Einarbeitung in und Entwicklung der Kommunikation zwischen dem ESP32 und dem Smart Device erfordert von allen daran Arbeitenden Teilnehmern viel Zeit.

Um diesen Aufwand und die Zeit zu verringern, bietet die vorliegende Arbeit eine schrittweise Anleitung, um den Einstieg in die Kommunikation über Websocket zu erleichtern.

Das Websocket – Protokoll wurde unter anderem ausgewählt, da es eine bidirektionale Verbindung bietet, es können Nachrichten jederzeit sowohl vom Smart-Device zum Server als auch umgekehrt gesendet werden. Außerdem sind mit diesem Protokoll bisher einige Projekte umgesetzt, welche in einem lokalen Funknetzwerk (WLAN) kommunizieren.

Mit der Anleitung dieser Arbeit soll prinzipiell ein ferngesteuertes Fahrzeug aufgebaut werden. Dafür wird der FT32 – Microcomputer des Fahrzeugs als sogenannten Websocket – Server programmiert, welcher sich mit einem vorhandenem WLAN verbindet. Der FT32 übernimmt auch die Steuerung der Motoren und das Auslesen der Sensoren. Das eigene Smart-Device, wie ein Smartphone oder Laptop befindet sich im gleichen Netzwerk und wird als Fernsteuerung genutzt. Sehr gerne kann auch ein vollständiges Fahrzeug aufgebaut werden.

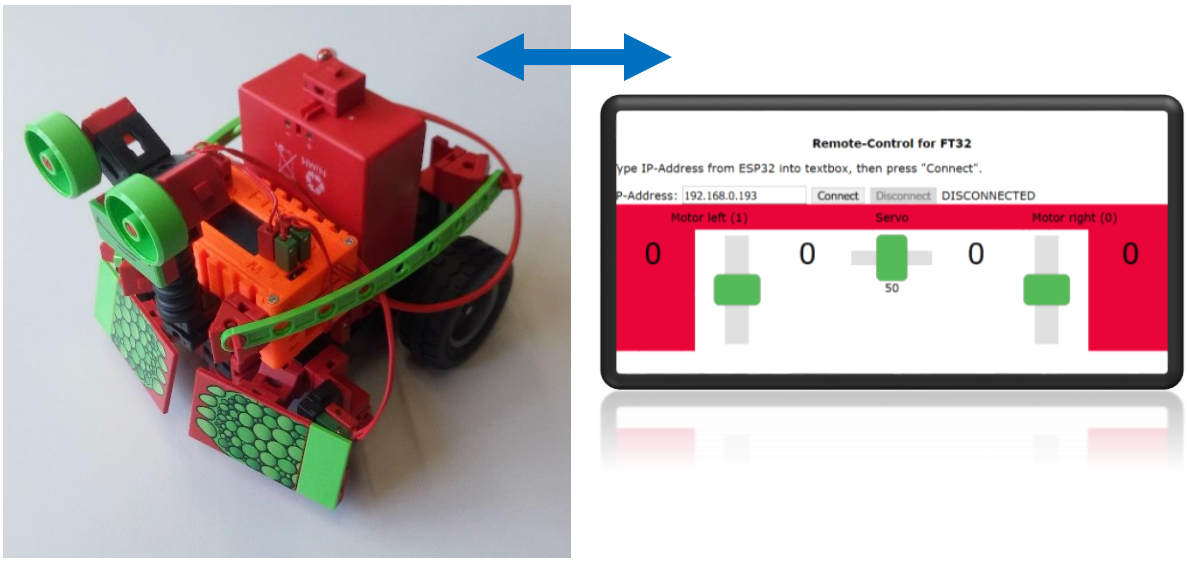

Abbildung 1: Fischertechnik-Fahrzeug mit FT32 und Fernsteuerung

In dieser Arbeit wird zunächst im Kapitel „Stand der Technik“ das Websocketprotokoll erklärt und einige Anwendungsbeispiele aufgezeigt. Anschließend folgen in „Vorkenntnisse und Vorbereitung Hardware/Software “ einige Hinweise auf Vorkenntnisse, welche für einen eigenen Aufbau erforderlich und/oder hilfreich sind. Dies enthält auch Hinweise auf einen vollständigen Aufbau eines ferngesteuerten Fahrzeugs. Das darauffolgende Kapitel „Umsetzung und Aufbauanleitung“ entspricht der softwareseitigen Umsetzung des Websocket – Protokolls auf der Serverseite wie auf der Clientseite.

Die beiden Kapitel 4 und 5 können als eigenständige ausführliche Anleitung zur Umsetzung des Websockets gesehen werden. Die anderen Kapitel bieten weitere Hintergrundinformationen und Ergänzungen sowie Anregungen zu weiteren Projekten.

Im Anschluss folgen in „Weitere Punkte“ Erklärungen zu besonderen Elementen in der Anleitung, welche dort nicht näher behandelt wurden, da sie nicht zwingend für die Verwendung des Protokolls notwendig sind. Im Kapitel 7 sind Hinweise und Anregungen auf weitere Entwicklungen in diesem Rahmen zu finden.

Die mit diesem Projekt erstellten Quellcodes und weitere Dateien können von

hit.hs-karlsruhe.de/hit-info/FT32/

heruntergeladen werden.

3 Stand der Technik

Dieses Kapitel stellt das Websocket-Protokoll vor und gibt einen kleinen Überblick über die Vorgänger und den Weg zum Websocket.

Das bisherige HTTP-Protokoll ist für ein klassisches Client-Server – Modell gedacht. Dabei sendet der Client, also indirekt der Benutzer, über den Browser eine Anfrage und erhält vom Server die entsprechende Antwort. Für jede weitere Information, die der Client zusätzlich benötigt muss eine neue Anfrage an den Server gestellt werden.

Das HTTP sieht nicht vor, den Zustand zu speichern: Für eine Clientanfrage wird eine Verbindung geöffnet und vom Server direkt beantwortet. Nach Erhalt der Antwort wird die Verbindung vom Client wieder geschlossen. Der Vorteil ist, der Server muss keine Daten diesbezüglich speichern und spart somit Speicherplatz. Dagegen muss der Client bei jeder Anfrage alle sich selbst betreffenden Informationen, also den Header mitsenden. Dieser beinhaltet verschiedene Informationen zum Client-Gerät, wie Browser, Betriebssystem, Sprache, etc. Der Server antwortet ebenfalls mit einem eigenen Header.

Dies ergibt einen Datensatz von mehreren hundert Bytes (in [1] ist ein Beispiel mit 871Bytes gegeben), der bei jeder Anfrage versendet wird, die tatsächliche Nutzdaten sind darin nicht enthalten.

Client-Server – Netzwerke mit HTTP sind als halbduplex aufgebaut. Das heißt, nur ein Teilnehmer kann senden, der andere muss warten (Vergleich: Walkie-Talkie, Flugfunk, Militärfunk).

HTTP Polling

Bei zeitkritischen Anwendungen (Börsenkurse,) kann es sein, dass die z.B. in einem Browser angezeigten Informationen bereits veraltet sind. Eine Möglichkeit, dieses Problem zu beheben ist es, die Internetseite durchgehend neu zu laden. Auch wenn die durchgehende manuelle Aktualisierung nicht besonders praktisch erscheint, wurde dies technisch so durchgeführt: In regelmäßigen Abständen sendet der Client automatisiert Anfragen und der Server antwortet. Insbesondere wenn neue Informationen beim Server in der gleichen Frequenz zur Verfügung stehen, wie abgefragt wird funktioniert diese Kommunikationsart sehr gut. In der Regel stehen die Nachrichten aber nicht synchron zur Verfügung.

HTTP Long Polling

Eine weitere weit verbreitete Technik ist „Long Polling“. Dabei antwortet der Server auf eine Client-Anfrage erst, wenn neue entsprechende Informationen vorliegen. Solange bleibt die Verbindung geöffnet. Nach Erhalt einer Nachricht wird die Verbindung wieder geschlossen und der Client muss die Verbindung durch eine erneute Anfrage wieder öffnen. Abhängig von der Anwendung ist die Anzahl der Anfragen und die Datenmenge durch die Header vergleichbar mit durchgehendem Polling.

HTTP Streaming

Bei dieser Technik sendet der Server kein Ende der Nachricht. Somit kann dieser sofort neue Informationen senden, ohne auf eine Anfrage des Clients zu warten. Kritisch ist diese Übertragungsart bei Proxys und Firewalls, da diese häufig HTML-Nachrichten puffern. Somit kann es zu größeren Verzögerungen kommen, bevor die Nachricht am Client ankommt.

Lösung: Websocket

Um die Beschriebenen Nachteile zu beseitigen wurde von der „Internet Engineering Task Force“ (IETF) [2] das Websocket-Protokoll entwickelt.

Das Websocket-Protokoll erweitert das bisherige HTTP so, dass ein bidirektionaler Kanal zwischen zwei Teilnehmern entsteht. Bei einer HTTP-Anfrage des Clients kann ein Handshake zwischen Client und Server durchgeführt werden. Nach dem Handshake können beide Teilnehmer jederzeit Nachrichten versenden. Dadurch verringert sich die Datenmenge, die als Header bisher bei jeder Nachricht versendet werden musste. Zusätzlich ist insbesondere eine gewisse Echtzeitfähigkeit gegeben, da der Server neue Informationen ohne Anfragen des Clients weitergeben kann.

4 Vorkenntnisse und Vorbereitung Hardware/Software

Um die Fernsteuerung des Fahrzeugs und die Fahrzeugsteuerung zu bauen sind neben der notwendigen Hardware weitere Vorbereitungen in am PC notwendig.

Die Voraussetzungen in Hard- und Software wie auch in Vorkenntnisse werden aufgeteilt in:

Notwendig
Diese Voraussetzungen müssen erfüllt sein um den Websocket-Server und eine Client-Webseite aufzubauen und zu testen.

Empfohlen
Um das Fahrzeug vollständig aufzubauen müssen diese Voraussetzungen erfüllt werden. Wahlweise und je nach Kenntnisstand können hier auch einzelne Module weggelassen oder verändert werden. Einige empfohlene Punkte sind auch (lediglich) zur vereinfachten Entwicklung aufgelistet und ändern nichts an der abschließenden Funktionalität des Fahrzeugs oder des Websockets.

4.1 Vorkenntnisse

Erforderlich sind Grundkenntnisse in C++, idealerweise einschließlich Objektorientierung.

Grundkenntnisse in HTML (mit CSS) und JavaScript sind von Vorteil.

Empfohlen sind Grundkenntnisse in: Arduino IDE, Einrichtung des ESP32 in die IDE.
Auf der GitHub-Seite von espressif befindet sich eine Anleitung zur Installation des ESP32 unter Arduino:
github.com/espressif/arduino-esp32/blob/master/docs/arduino-ide/boards_manager.md

4.2 Hardware

Um nur den Websocket auszutesten wird nur ein

- ESP32 (Wroom – Board) und Micro-USB Kabel zum Anschluss an den PC (*)

benötigt. Wenn man das gesamte Fahrzeug oder Teile der Aktorik und ein Display aufbauen möchte, werden weitere Hardwareteile gebraucht:

- 5V Modellbauservo (z.B. HITEC HS-53) mit Verbindungskabel. Der Servo kann beim Fahrzeug als Malroboter auch zum Anheben eines Stifts genutzt werden.
- Display 128x64px OLED mit I²C Anschluss (z.B. Adafruit SSD1306). Zur Anzeige insbesondere für die IP-Adresse nicht zwingend notwendig aber sehr empfohlen. (*)
- Motortreiber (z.B. Pololu A4990) (*)
- Spannungsregler (z.B. Pololu 5V, 2.5A Step-Down D24V22F5) (*)
- Steckbrett + Verbindungskabel (*)
- 9V Spannungsversorgung (z.B. Blockbatterie oder Fischertechnik-Akku 8,4V)
- Fischertechnik – Mini Bots Baukasten (oder Vergleichbar). Dieser sollte 2x 9V DC Motoren und Taster enthalten.

(*) Die gekennzeichneten Bauteile sind im FT32 – Mini Mikrocontroller (Abbildung 2) vorhanden

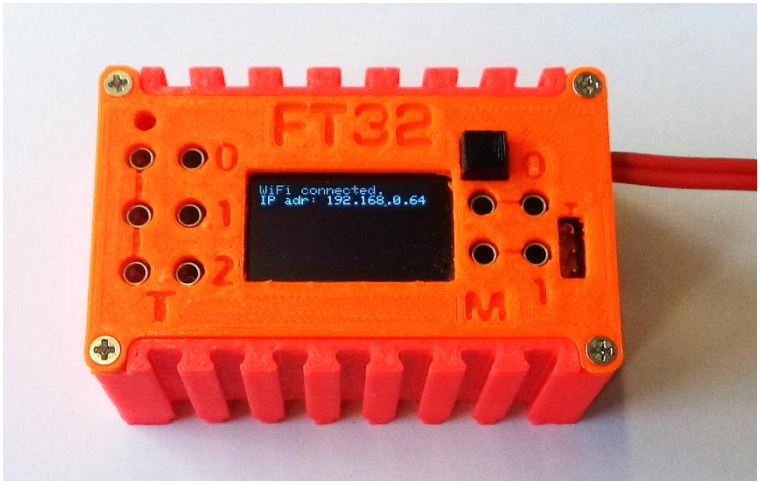

Abbildung 2: FT32 mit Fernsteuerungssoftware

4.3 Notwendige Software

Folgende Software wird benötigt:

- Arduino – IDE (www.arduino.cc) mit ESP32 – Framework (s.o.)
- Editor für HTML/CSS/JavaScript
  Ein (Windows-) Standardeditor ist als Minimum ausreichend, empfohlen wird aber mindestens ein Texteditor wie notepad++ oder atom für Syntaxhighlighting etc.:
  notepad-plus-plus.org
  atom.io
- Websocket-fähiger Browser
  Die meisten aktuellen Browser (geprüft wurde Firefox, Internet-Explorer/Edge und Chrome) unterstützen Websocket. Um die Unterstützung durch den eigenen Browser gegeben ist, kann durch Aufrufen der Webseite websocket.org/echo.html geprüft werden.

Empfohlen wird zusätzlich ein Editor oder eine IDE zur einfacheren Programmierung wie z.B. vMicro mit VisualStudio (visualmicro.com/page/How-to-install-Esp32-for-Arduino-and-Visual-Micro.aspx) oder Eclipse (instructables.com/id/ESP32-With-Eclipse-IDE/). Es handelt sich hier um eine etwas umfangreichere Anleitungen und ist auch nur zu empfehlen, wenn man größere Projekte unter der Arduino-Umgebung umsetzen möchte.

Ein eigener Server, der die Webseite für die Fernsteuerung zur Verfügung stellen kann, lässt sich mit HFS (Http File Server) einfach aufsetzen. Eine Anleitung dafür befindet sich in Kapitel 5.4.

4.4 Einrichtung des Arduino – Projekts

Nach der Installation der obigen Programme muss das Projekt eingerichtet werden.

Die Arduino-Umgebung wird für die Programmierung des Websocketservers verwendet. Um den ESP32 zu flashen muss im Boardverwalter das ESP32 Dev Module ausgewählt werden (Abbildung 3).

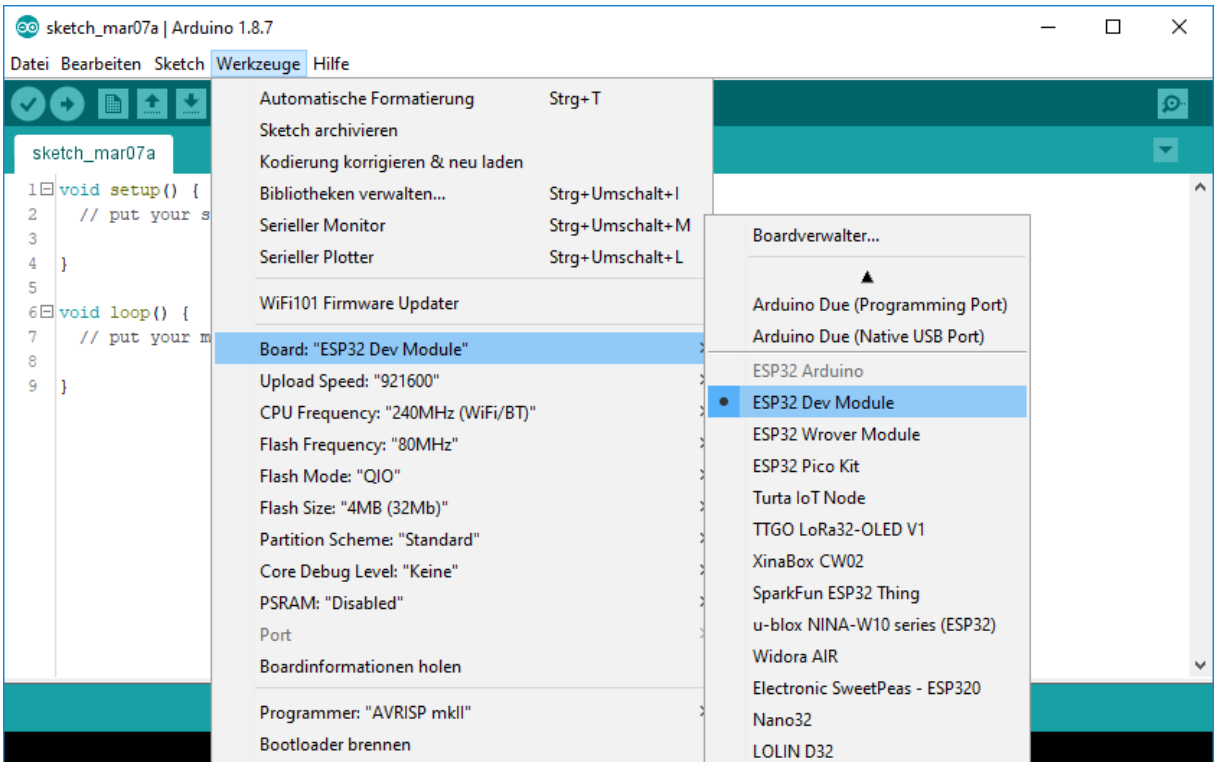

Abbildung 3: Auswahl Mikrocontroller

Für das Adafruit-Display braucht man die passenden Treiber, diese können mit dem Bibliotheksverwalter geholt werden (unter Werkzeuge → Bibliotheken verwalten…)

Benötigt werden

- Adafruit GFX Library (Version 1.3.4)
- Adafruit SSD1306 (Version 1.1.2)

Neuere Versionen der GFX-Library wurden noch nicht getestet, neuere Versionen der SSD1306 – Bibliothek funktionieren nicht!

Der Websocketserver selbst basiert auf der Bibliothek Arduino-Websocket. Diese kann von GitHub unter github.com/brandenhall/Arduino-Websocket heruntergeladen werden. Folgende Dateien müssen hierführ in den Projektordner (neben die .ino-Datei) abgelegt werden:

- WebSocketServer.h + WebSocketServer.cpp
- Base64.h + Base64.cpp
- global.h
- sha1.h + sha1.cpp

Die Ansteuerung der Motoren und das Auslesen der Taster erfolgt mit der IOobjects – Bibliothek, die Dateien können von hit-karlsruhe.de heruntergeladen und ebenfalls im Projektordner eingefügt werden.

- ft_ESP32_IOobjects.h + ft_ESP32_IOobjects.cpp

Für die Fernsteuerung kann ein Projektordner im Dateisystem angelegt werden (hier „remote_control“ genannt). Dieser enthält je einen Ordner für CSS und JavaScript (siehe Abbildung 4). Außerdem befindet sich hier die index.html. Damit wird die Oberfläche der Fernsteuerung entwickelt.

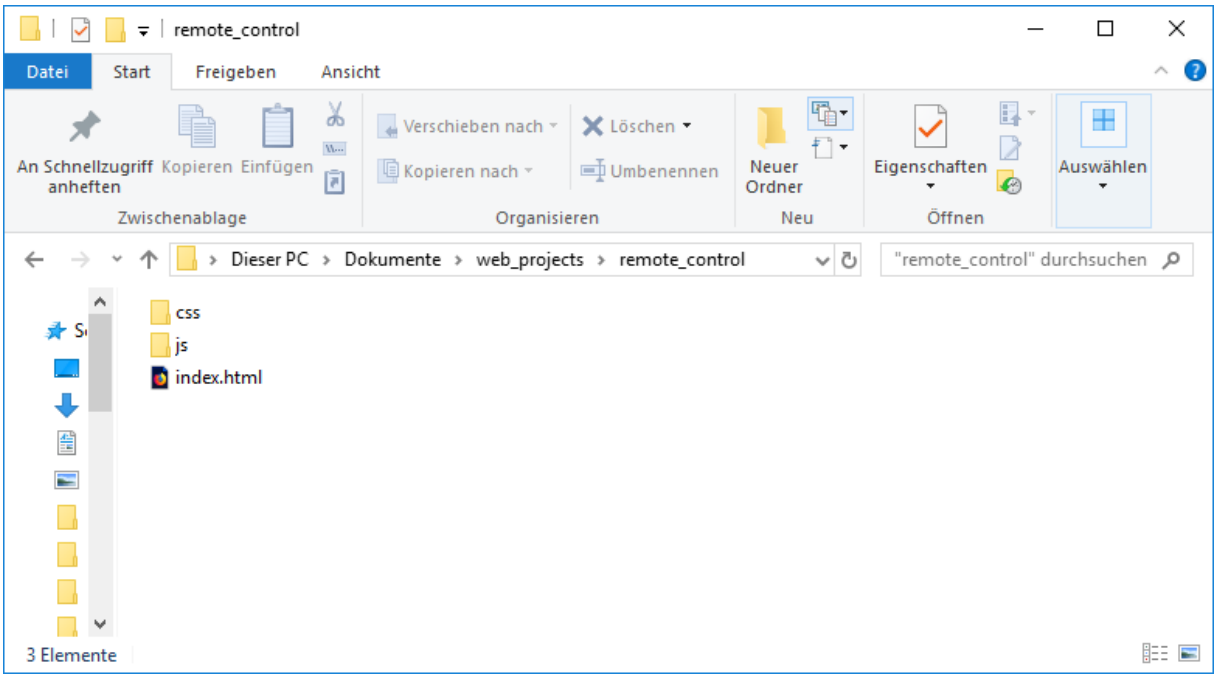

Abbildung 4: Projektordner Fernsteuerung (HTML)

In dem Ordner css befindet sich die Datei „**default.css**“, im Ordner js die „**main.js**“.

5 Umsetzung und Aufbauanleitung

Dieses Kapitel enthält die eigentliche (softwareseitige) Aufbauanleitung der Fernsteuerung. Aufgeteilt wird dieses in die Bereiche Websocket – Server (ESP32) und Websocket – Client (Browser).

5.1 Ablaufprinzip

Der Server auf dem ESP32 wie der Client auf dem Smart Device durchlaufen prinzipiell folgenden Ablauf:

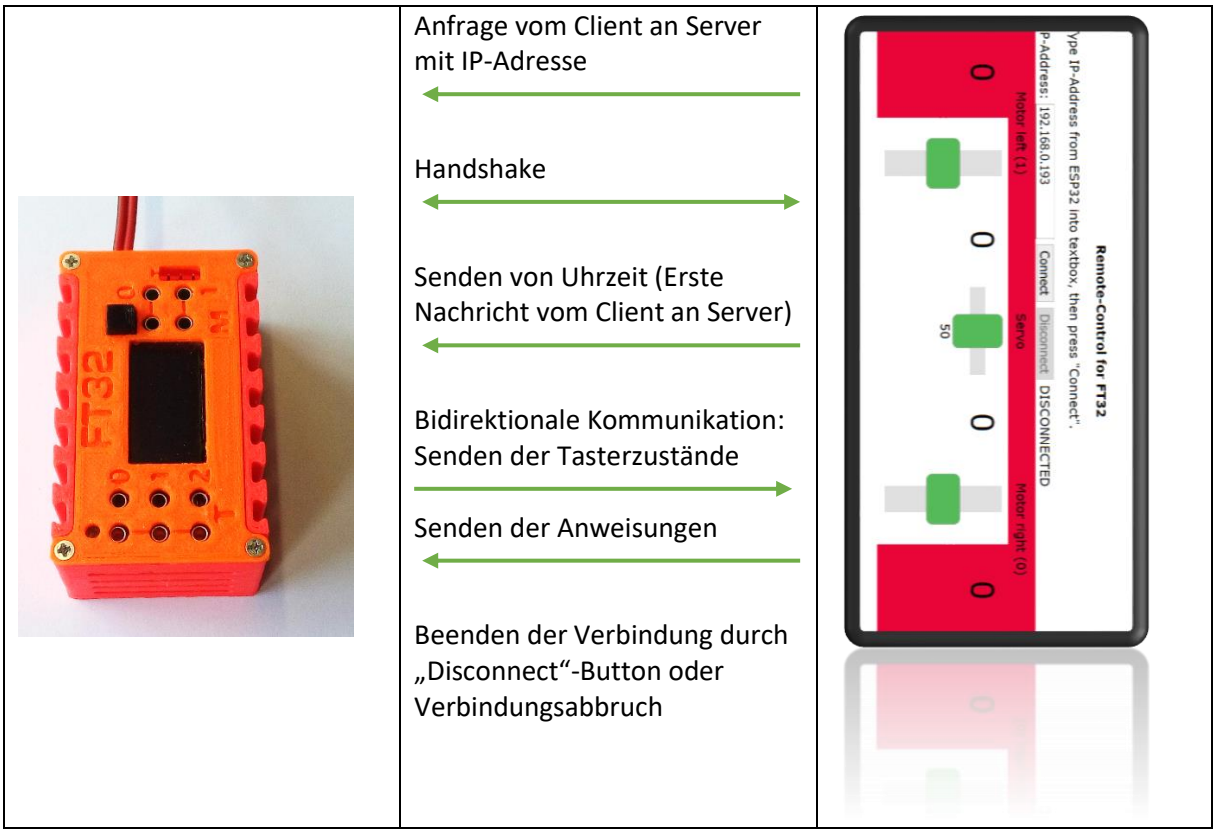

Abbildung 5: Ablauf Websocketverbindung

5.2 ESP32 – Server

Das Programm auf dem ESP32 lässt sich aufteilen in

- Bibliotheken und Deklarationen
- Setup
- Loop

5.2.1 Notwendige Bibliotheken / Deklarationen

Zu Beginn des Programmcodes werden die notwendigen Bibliotheken eingebunden.

Die WiFi – Bibliothek (Version 1.2.7) ist eine Standardbibliothek von Arduino und fasst verschiedene Wifi-Funktionen zusammen. Darunter sind die Verwaltung der IP-Adresse, die Verwendung des WiFi-Moduls als Server oder Client, etc.

```
1  #include <WiFi.h>
```

Die Verwendung des Websocket-Protokolls setzt die Websocket-Bibliothek voraus (siehe Kapitel 4.4). Diese bietet grundlegende Funktionen wie den Handshake, Senden und Empfangen von Daten, Ping-Pong und das Beenden der Websocket-Verbindung. Die Bibliothek ist kein Arduino-Standard und muss zusammen mit der zugehörigen Codedatei (WebSocketServer.cpp) im Projektordner neben die .ino – Datei abgelegt werden.

```
1  #include "WebSocketServer.h"
```

Für das OLED – Display (SSD1306 mit 128x64 Pixel) werden zwei Bibliotheken vom Hersteller Adafruit benötigt. Diese können in der Arduino IDE geladen werden Die GFX – Bibliothek (Version 1.3.4) bietet Text-, Formatierungs- und Grafikmethoden, die SSD1306 – Bibliothek (Version 1.1.2, neuere haben noch Fehler) enthält den eigentlichen Treiber zur Kommunikation zwischen dem ESP32 und dem Display.

```
1  #include <Adafruit_GFX.h>
2  #include <Adafruit_SSD1306.h>
```

Um die Hardware anzusteuern bietet sich die „Ein-/Ausgangsbibliothek“ für den FT32 – Controller an. Sie bietet Funktionen an zur vereinfachten Ansteuerung der Fahrzeughardware wie Pololu-9V-Motortreiber, Servomotor (z.B. und das Einlesen von Pegelwerten an Pins. Auch diese Bibliothek ist kein Arduino-Standard und muss manuell in den Projektordner eingefügt werden.

```
1  #include "ft_ESP32_IOobjects.h"
```

- Bibliotheken
    - Ft_ESP32_IOobjects.h → bietet Funktionen an zur vereinfachten Ansteuerung von Hardware (). Insbesondere in Verbindung mit den FT32-Boards und Cody++.

Vor der Setup-Funktion müssen global Objekte und Variablen angelegt werden, welche von allen folgenden Funktionen genutzt werden. Eine detailliertere Beschreibung zu den Klassen und Objekten folgt bei der Verwendung dieser.

Display – Objekt (optional), zur Verwendung des Adafruit 128x64px – Displays:

Um das Display mit der Treiberversion 1.1.2 in der Auflösung 128x64px nutzen zu können, muss in der Adafruit_SSD1306.h

Der einfachste Konstruktor des display – Objektes benötigt laut Bibliothek eine Pinnummer um das Display zurücksetzen zu können (obwohl das Display selbst keinen Resetpin hat). In diesem Beispiel wird der Pin 4 zugewiesen, dieser wird im weiteren Programm auch nicht verwendet.

```
1  #define OLED_RESET 4
2  Adafruit_SSD1306 display(OLED_RESET);
```

Server – Objekt, erlaubt die Kommunikation von Clients mit dem ESP32 und dient als Basis für den Websocket – Server:

Übergeben wird die Port-Nummer des Servers… Standard ist die 80, wird aber trotzdem übergeben.

```
1  WiFiServer server(80);
```

WebsocketServer – Objekt, bietet Methoden zum Senden und Empfangen von Nachrichten über das Websocket – Protokoll:

```
1  WebSocketServer webSocketServer;
```

Ein- und Ausgangsobjekte zur Ansteuerung der Fahrzeughardware:

Damit der ESP32 den Servomotor ansteuern kann benötigt das mServo – Objekt eine Nummer (hier 2), mit welcher der PWM-Generator zugewiesen wird und die Startposition des Servos in Prozent (hier 50%). Zusätzlich kann auch der Ansteuerungspin festgelegt werden, wenn er nicht dem Standard der Bibliothek entsprechen soll. Hier wird der Pin 24 benutzt, da dieser beim FT32 in unserem Fahrzeug nicht verwendet wird.

```
1  CServoMotor mServo(2, 50, 25);
```

Zur Ansteuerung der beiden Motoren werden zwei Motorobjekte angelegt. Auch diesen wird je eine Nummer (0 und 1) zugewiesen. Die Nummern aller Objekte zur Ansteuerung der Aktorik müssen sich voneinander unterscheiden um interne Kollisionen zu vermeiden.

```
1  CMotor mMotor0(0);
2  CMotor mMotor1(1);
```

Um die beiden Taster vorne abfragen zu können werden ebenfalls zwei Objekte angelegt.

```
1  DigitalAnalogIn mButton0(3);
2  DigitalAnalogIn mButton1(4);
```

Zuletzt legen wir eine globale Variable an, mit welcher eine Unterbrechung der Netzwerkverbindung zwischen dem ESP32 und dem WLAN erkannt werden kann.

```
1  bool connectedToNetwork = false;
```

Der gesamte Code sieht dann wie folgt aus:

```
1  #include <WiFi.h>
2  #include "WebSocketServer.h"
3  #include <Adafruit_GFX.h>
4  #include <Adafruit_SSD1306.h>
5  #include "ft_ESP32_IOobjects.h"
6
7  #define OLED_RESET 4
8  Adafruit_SSD1306 display(OLED_RESET);
9
10 WiFiServer server(80);
11 WebSocketServer webSocketServer;
12
13 CServoMotor mServo(2, 50, 25);
14 CMotor mMotor0(0);
15 CMotor mMotor1(1);
16 DigitalAnalogIn mButton0(3);
17 DigitalAnalogIn mButton1(4);
18
19 bool connectedToNetwork = false;
```

5.2.2 Setup

In der Setup-Funktion werden alle Programmteile ausgeführt, die zur Vorbereitung notwendig sind. Beispielsweise wird hier die Netzwerkverbindung und das Display eingerichtet und der Websocketserver gestartet.

Zunächst wird eine serielle Verbindung eingerichtet. Das Serial-Objekt erlaubt eine direkte Kommunikation über UART z.B. mit dem eigenen Entwicklungsrechner. Das Objekt gehört zum Arduino – Standard und muss somit nicht selbst angelegt werden. Die Kommunikation wird mit der Methode .begin() gestartet, übergeben wird dabei die Baudrate, welche entsprechend Standards frei gewählt werden kann. Wir verwenden hier 115200 Baud:

```
1  Serial.begin(115200);
```

Folgend wird die das Display mit der Angabe der Spannungsversorgung und der I²C – Adresse eingerichtet. Die Adresse muss je nach verwendeter Auflösung geändert werden, hierfür möchte ich auf eine Videoanleitung von Prof. Walter verweisen.

```
1  display.begin(SSD1306_SWITCHCAPVCC, 0x3C);
```

Generell ist für die Verwendung des Displays wichtig zu wissen, dass alle Anweisungen für Zeichnungen und Texte „im Hintergrund“ das Bild aufbauen. Erst mit dem Befehl **`display()`** wird das Bild tatsächlich auf das Display gebracht und sichtbar.

Mit obigem display.begin(…) wurde initial ein Bild (Adafruit – Logo) im Hintergrund aufgebaut, dieses kann jetzt angezeigt werden:

```
1  display.display();
```

Der angezeigte Inhalt des Displays kann mit **`clearDisplay()`** gelöscht werden. Folgend wird die Textfarbe auf Weiß umgestellt und eine kurze Zeit von 10ms gewartet, da Displayanweisungen gewöhnlich nicht beliebig schnell sind:

```
1  display.clearDisplay();
2  display.setTextColor(WHITE);
3  delay(10);
```

Für die folgende Verbindung des WIFI – Moduls mit einem WLAN werden die Verbindungsdaten eingetragen. Benötigt wird der Name/ bzw. die SSID des Netzwerks und das zugehörige Passwort:

```
1  char ssid[] = "HIT-FRITZBOX-7490";
2  char password[] = "6......2";
```

Zur Überprüfung, ob das Programm bis hierhin fehlerfrei funktioniert hat, werden Textausgaben über die serielle Schnittstelle an den PC gegeben und auf dem Display angezeigt. Hier ist die Ausgabe der Name des Netzwerks, mit dem verbunden werden soll.

```
1  display.clearDisplay();
2  Serial.print("Connecting to ");
3  Serial.println(ssid);
4
5  display.print("Connecting to ");
6  display.println(ssid);
7  display.display();
```

Mit der Bibliothek Wifi.h wird ein (globales) Wifi-Objekt „WiFi“ erstellt, welches das Wifi – Modul des ESP32 und somit alle Netzwerkverbindungen verwaltet. Zunächst wird mit der Methode .mode() mitgeteilt, ob das Wifi-Modul ein Client in einem bestehendem Netzwerk oder ein eigener Accesspoint (AP) sein soll (oder beides, der ESP32 kann in begrenztem Umfang auch als Repeater verwendet werden). Für unsere Anwendung soll das Wifi-Modul als Client verwendet werden, also wird der Methode .mode() WIFI_STA (=1, „stationary mode“) übergeben:

```
1  WiFi.mode(WIFI_STA);
```

Der Verbindungsaufbau wird mit der Übergabe von SSID und dem Passwort gestartet:

```
1  WiFi.begin(ssid, password);
```

Bei dem Verbindungsaufbau treten häufig Fehler auf, welche dafür sorgen, dass das Wifi-Modul unendlich lange für eine Verbindung benötigt. Es ist daher sinnvoll zu messen, wie lange der Aufbau bisher benötigt hat und bei zu langer Wartezeit den Aufbau neu zu starten. Aus Erfahrung kann man für dieses Projekt sagen, dass ein Verbindungsaufbau, der länger als 5 Sekunden dauert, nicht zu einer funktionierenden Verbindung führt. Wir fragen dafür den Status des Wifi-Moduls regelmäßig ab, ist eine Verbindung vorhanden, kann mit dem Programm fortgefahren werden, ansonsten wird nach 5 Sekunden der Aufbau neu gestartet. In der Regel ist nach einem Neustart spätestens nach 2 bis 3 Sekunden eine Verbindung hergestellt.

```
1  int i = 0;  //counts connection time
2  while (WiFi.status() != WL_CONNECTED)    //while wifi isn’t connected
3  {
4        delay(500);
5        Serial.print(".");
6        if (10 == i)      //if no connection after some time (i*delay)
7        {
8              Serial.println("Connection Failed! Restart Wifi...");
9              display.println("Connection Failed! Restart Wifi...");
10             display.display();
11             delay(10);
12             WiFi.reconnect(); //restart wifi
13             delay(1000);      //wait for short time
14             i = 0;      //reset time-counter
15       }
16       else  //if counter not 10: increase and wait a little longer
17       {
18             i++;
19       }
20 }
```

Nach erfolgreichem Verbindungsaufbau setzen wir das vor der Setup-Funktion festgelegte Statusflag **`connectedToNetwork`** auf true und zeigen über die serielle Schnittstelle und das Display die eigene IP – Adresse an. Diese kann mit **`Wifi.localIP()`** abgefragt werden. Die IP – Adresse wird später auf der Fernsteuerungsseite benötigt um eine Websocketverbindung herzustellen.

```
connectedToNetwork = true;

Serial.println("");
Serial.println("WiFi connected.");
Serial.print("IP address: ");
Serial.println(WiFi.localIP());

display.clearDisplay();
display.setCursor(0, 0);
display.println("WiFi connected.");
display.print("IP adr: ");
display.println(WiFi.localIP());
display.display();
```

Am Ende der Setupfunktion wird der eigene Server gestartet.

```
server.begin();
delay(100);
```

5.2.3 Loop

In der **`void loop()`** – Funktion läuft das Programm in einer Endlosschleife. Alle Aufgaben die mit dem Mikroctontroller zu brearbeiten sind befindenden sich in dieser Funktion. In unserem Beispiel wird hier auf einen Client gewartet. Bei einer Anfrage eines Clients wird eine Websocketverbindung aufgebaut und die vom Client gesendete Websocket-Daten, also Fahranweisungen und Servopositionen, zyklisch eingelesen und verarbeitet. Gleichzeitig sendet der Websocketserver zyklisch die Tasterzustände des Fahrzeugs an den Client zurück.

Zunächst wird ein Client – Objekt initialisiert. Dieses Objekt erlaubt unter Anderem den Aufbau und das Handling der Websocketverbindung auf der Seite des ESP32. Die Initialisierung erfolgt mit einer Abfrage des Serverobjektes ob sich eine Webseite als Client verbinden möchte.

```
WiFiClient client = server.available();
```

Folgend führt das webSocketServer – Objekt einen sogenannten Handshake mit dem Client durch. Dieser Funktionsaufruf baut die tatsächliche bidirektionale Websocketverbindung auf.

Wenn sich ein Client verbunden hat und der Handshake erfolgreich war, kann der Datenaustausch begonnen werden. In unserem Programm sind diese beiden Abfragen direkt als Bedingung in die Verzweigung eingefügt.

```
if (client.connected() && webSocketServer.handshake(client))
```

Anschließend werden notwendige Variablen angelegt: eine Zeichenkette (**`String data`**) in dem die per Websocket empfangenen Daten gespeichert werden, ein Flag zur Auslastung des Websocketservers (**`bool serverIdle`**) und ein bool – Array (**`bool buttonPressed[2]`**) um die Tasterzustände zu speichern und zu verarbeiten.

```
String data;
bool serverIdle = false;
bool buttonPressed[2] = { false, false };
```

Der folgende Programmteil wird in einer Schleife solange ausgeführt, wie sich ein Client mit dem Server verbunden hat.

```
1  while (client.connected())
```

5.2.3.1 Handling der Websocket – Nachricht

Das Objekt **webSocketServer** erhält asynchron alle Daten über die Websocketverbindung. Diese werden als Zeichenkette gespeichert und können in einen String kopiert werden. Damit die Daten korrekt empfangen und gespeichert werden können, wird eine kleine Wartezeit von 10ms benötigt.

```
1  data = webSocketServer.getData();
2  delay(10);
```

In **data** befindet sich jetzt die Nachricht des Websockets. Da von der Fernsteuerung die Daten unregelmäßig gesendet werden (je nach Benutzereingabe), kann es vorkommen, dass die „empfangene“ Nachricht leer ist.

```
1  if (data.length() > 0)
```

Bei vorhandenen Daten wird das Flag zur Auslastung gesetzt und die Nachricht seriell ausgegeben.

```
1  serverIdle = false;
2  Serial.println(data);
```

Für die Interpretation der Nachricht wird die Protokollierung aus Kapitel 6.2 verwendet. Die Nachricht kann aus mehreren Nachrichtenblöcken aufgebaut sein und wird blockweise verarbeitet. Nach Bearbeitung eines Blocks wird dieser aus **data** gelöscht und die Verarbeitung mit dem nächsten Block wiederholt.

```
1  while (data.length() > 0)
2  {
3        (*)
4        if (0 <= data.indexOf(';'))  //if ';' is not the last symbol
5        {
6              data.remove(0, data.indexOf(';') + 1);   //remove handled
                                                        data-block up to next ";"
7        }
8        else
9        {
10             data = "";  //empty string
11       }
12 }
```

Folgende Anweisungen befinden sich im mit **(*)** markierten Bereich. Ein einzelner Nachrichtenblock beginnt immer mit einem Großbuchstaben, abhängig von diesem wird der weitere Teil der Nachricht untersucht und umgesetzt: D – Datum, S – Servomotor, M – Motor. Verwendet wird für die unterschiedlichen Aktionen eine **switch case** Anweisung.

```
1  switch (data.charAt(0))
2  {
3  case 'D':   //Handshake- and echo-message
4        { ... }
5        break;
6  case 'S':   //Servo-message, sets servomotor to new value
7        { ... }
8        break;
9  case 'M':   //Motor-message, sets motor-speed to new value
10       { ... }
11       break;
12 default:    //in case of unknown data
13       { ... }
14       break;
15 }
```

`Case 'D':` (in Zeile 4)

Das Datum (bei uns nur die aktuelle Uhrzeit) wird vom Client (der Fernsteuerung) beim Aufbau der Websocketverbindung als erste Nachricht versendet. Die Uhrzeit wird auf dem Display angezeigt und mit **`sendData(data)`** zurückgesendet. Somit ist auch der Client informiert, dass die Daten korrekt über den Websocket verschickt werden.

```
1  display.setCursor(0, 16);
2  display.fillRect(0, 16, 128, 32, BLACK); //clear required text-area
3  display.println(data.substring(2));      //print timestamp (without
   identifier "D:")
4  display.display();      //print everything on display
5  webSocketServer.sendData(data);    //send data-block back to client
6  delay(10);
```

`Case 'S':` (in Zeile 7)

Nach dem Anfangsbuchstaben (und dem folgenden Doppelpunkt) der Nachricht für den Servomotor steht der Wert, den der Servo einnehmen soll, zwischen 0 und 100 (in %). Dieser muss aus dem **`data`** – String herausgezogen und als Zahl interpretiert werden. Die Verkettung der dafür benötigten Befehle lässt sich grafisch an einem Beispiel zeigen.

Index	0	1	2	3	4	5	
Nachricht (**data**)	**S**	**:**	**1**	**0**	**0**	**;**	**...**

Befehl	Beschreibung	Rückgabewert (in diesem Beispiel)
`data.indexOf(':')`	gibt den Index des ersten Zeichens **`':'`** zurück	**`1`**
`data.indexOf(':')+1`	Erhöhung des Index um die Position der ersten Zahl zu erhalten	**`2`**
`data.substring(2)`	Erstellt einen Substring ab oben festgestelltem Index (2)	**`'100'`**
`substring.toInt(...)`	Wandelt den obigen Substring in einen Integer bis zum ersten Zeichen, das keine Ziffer ist.	**`100`**

Die Befehle werden zusammengefasst und in einem Integer gespeichert.

```
1  int servoVal = data.substring(data.indexOf(':') + 1).toInt();
```

Folgend wird der ausgelesene Wert auf den gültigen Bereich zwischen 0% und 100% limitiert

```
1  if (servoVal > 100)     //limit maximum position of servo
2  {
3        servoVal = 100;
4  }
5  if (servoVal < 0)       //limit minimum position of servo
6  {
7        servoVal = 0;
8  }
```

Abschließend wird mit dem in Kapitel 5.2.1 erstellten Servoobjekt dem Servomotor angewiesen, die neue Position anzufahren.

```
1  mServo.setValue(servoVal);
```

Achtung: alle Anweisungen im Bereich **`Case 'S':`** müssen in geschweifte Klammern **`{...}`** gepackt werden, da hier Variablen deklariert werden.

`Case 'M':` (in Zeile 10)

Für den Motor wird wie beim Servo oben die Nachricht ausgelesen und als Zahl gespeichert.

```
1  int motorVal = data.substring(data.indexOf(':') + 1).toInt();
```

Das Vorzeichen der erhaltenen Zahl entspricht der Drehrichtung des Motors: Größer 0 = „true“ = Rechtslauf, kleiner 0 = „false“ = Linkslauf.

```
1  bool motorDir = (motorVal >= 0);
```

Das Motorobjekt erwartet für die Drehgeschwindigkeit nur positive Ganzzahlen. Mit einer kompakten If-Abfrage wird der Betrag der Zahl gebildet.

```
1  motorVal = motorVal >= 0 ? motorVal : -motorVal;
```

Die Zahl nach dem Buchstaben M entspricht dem Motor, der mit dieser Nachricht angesteuert wird.

```
1  if ('0' == data.charAt(1))   //Motor 0 is addressed
2  {
3        mMotor0.setValues(motorDir, motorVal);
4  }
5  if ('1' == data.charAt(1))   // Motor 1 is addressed
6  {
7        mMotor1.setValues(motorDir, motorVal);
8  }
```

Achtung: alle Anweisungen im Bereich **`Case 'M':`** müssen in geschweifte Klammern **`{...}`** gepackt werden, da hier Variablen deklariert werden.

`default:` (in Zeile 10)

Falls im unwahrscheinlichen Fall der Identifier des Nachrichtenblocks unbekannt ist, wird dies seriell ausgegeben und der weitere Nachrichtenblock ignoriert

```
1  Serial.println("Wrong data-string");
```

Wenn keine Nachricht empfangen wurde oder data leer geworden ist, setzt der Server einmalig den Zustand auf „idle“ und sendet dem Client, dass weitere Nachrichten empfangen werden können.

```
1  if (!serverIdle) //server-idle message not sent yet
2  {
3        serverIdle = true;
4        webSocketServer.sendData("--serverIdle--");
5  }
```

Eine kleine Wartezeit an dieser Stelle reduziert die CPU-Last.

5.2.3.2 Handling der Taster

Quasi parallel zum Handling der Websocket – Nachrichten werden mit dem ESP32/FT32 die beiden Taster abgefragt und über den Websocket an den Client gesendet. Der Server soll, um den Websocket – Kanal nicht zu stark zu belasten, nur die Änderungen/Flanken der Tasterzustände versenden.

Hier wird je ein kleiner Zustandsautomat für die Taster verwendet, die Zustände werden im oben deklarierten Array **`buttonPressed[]`** gespeichert (gelbe Kreise in Abb…). Der Zustand wechselt, sobald sich der gespeicherte Zustand vom „gemessenen" Tasterzustand unterscheidet. Der Wechsel wird per Websocket an den Client gesendet.

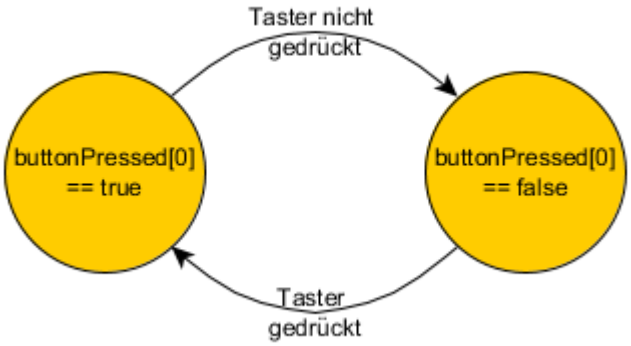

Abbildung 6: Zustandsautomat Taster 0

```
1  //button 0 released
2  if ((0 == mButton0.getValueDigital()) && buttonPressed[0])
3  {
4        webSocketServer.sendData("B0:0");
5        buttonPressed[0] = false;
6        Serial.println("Button 0 released");
7  }
8  //button 0 pressed
9  if ((0 < mButton0.getValueDigital()) && !buttonPressed[0])
10 {
11       webSocketServer.sendData("B0:1");
12       buttonPressed[0] = true;
13       Serial.println("Button 0 pressed");
14 }
```

(Der Zustandsautomat und die obigen Programmzeilen gelten genauso für Taster 1.)

Sobald die Verbindung zwischen der Fernsteuerung und des FT32 getrennt wird, verlässt das Programm die obige Schleife und zeigt die Trennung auf dem Display und seriell an. Auch werden die beiden Fahrmotoren gestoppt.

```
1  Serial.println("Client disconnected");
2  display.println("Client disconnected");
3  display.display();
4  mMotor0.setValues(true, 0);
5  mMotor1.setValues(true, 0);
```

5.3 Client – Fernsteuerung

Die Fernsteuerung, welche als Webseite im eigenen Browser dargestellt wird, besteht aus dem HTML – Rahmen mit CSS – Anpassungen und mehreren JavaScript – Funktionen zur Kommunikation über Websocket.

5.3.1 Ablaufprinzip – Programmablauf Client

Für den Benutzer der Fernsteuerung sieht der Programmablauf wie folgt aus:

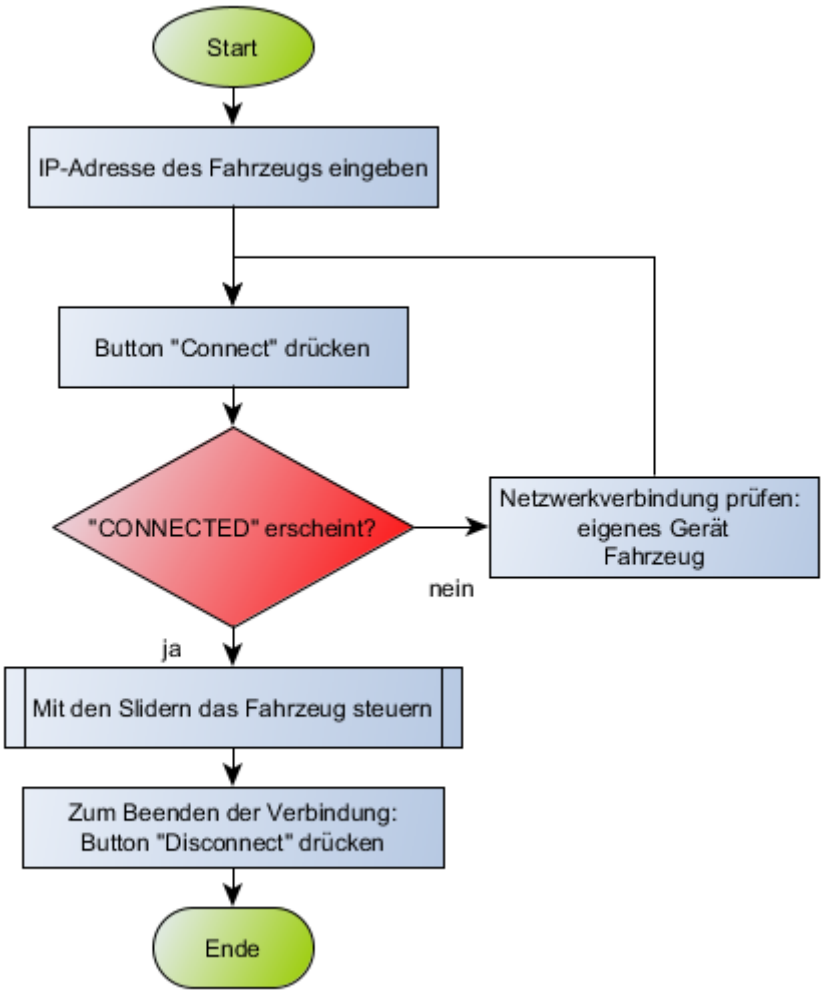

Abbildung 7: Prinzipieller Programmablauf des Clients

Technisch lässt sich dieser Vorgang nicht 1:1 abbilden, da die Funktionen unter JavaScript parallel laufen.

5.3.2 HTML - Rahmen, Vorstellung der Oberfläche, Design

Auf den genauen HTML-Aufbau und -Syntax der Fernsteuerung wird hier nicht im Detail eingegangen. Zu empfehlen sind dafür Online – Nachschlagwerke wie selfhtml oder w3schools.

Der Hauptteil **<body>** teilt sich auf in (siehe auch Abbildung 8):

A einfacher HTML – Teil für die Überschrift und kleiner Einleitungssatz
B Steuerbereich für die Websocketverbindung
C Fernsteuerung mit Slider für die Motoren und Anzeigen für die Buttons

Nicht sichtbar in Abbildung 8 ist eine nur zum Testen aktivierte Ausgabekonsole unterhalb der Fernsteuerung, im Folgenden mit **output** bezeichnet.

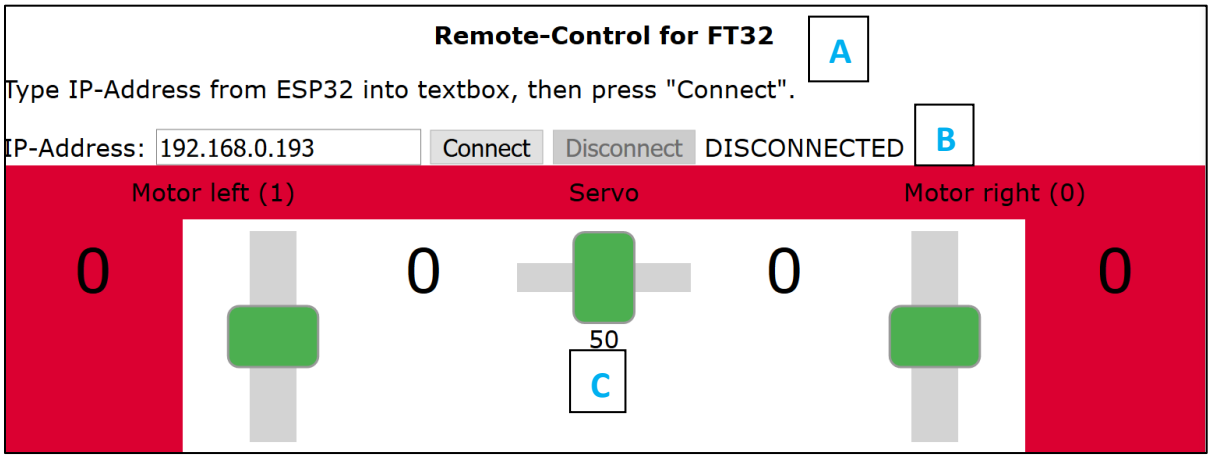

Abbildung 8: Fernsteuerung Übersicht

Wichtig ist im HTML – Quelldokument, dass alle Elemente, auf die mit JavaScript zugegriffen werden soll mit einem eindeutigen Identifier versehen werden. Zum Beispiel erhält der Connect – Button (Teil B) die Bezeichnung `id="btnConnect"` im Eröffnungstag. Außerdem benötigt das Dokument einen Verweis auf die JavaScript-Datei (bzw. auf den Dateipfad):

```
<script src="js/main.js"></script>
```

5.3.3 JavaScript zur Steuerung (Websocket) / Intelligenz der Webseite

In diesem Unterkapitel werden alle verwendeten JavaScript – Funktionen beschrieben. Grob Aufgeteilt gibt es Funktionen für die Websocket – Verwendung und Funktionen für die „Intelligenz“ der Webseite.

Vorbereitend werden einige globale Variablen angelegt.

```
1  var wsUri = "";   //stores URL (IP-Address) of the server: ws://"IP"/
2  var output;       //for testing, stores HTML-position for simple out-
   put messages on screen
3  var websocket;    //ws-variable, becomes ws-object later
4  var outputDataStream = "";          //message, will be sent, if web-
   socket-server is idle
5  var connectedToServer = false;      //flag, shows if client is con-
   nected to server
6  var serverIdle = false;             //flag, shows if server is idle.
```

Das Objekt **window** kann mit der Methode **addEventListener** verschiedene Ereignisse der Webseite überwachen. Unser Programm nutzt **"load"** um mit dem vollständigen Laden der Webseite im Browser die später definierte **init** – Funktion aufzurufen.

```
1  window.addEventListener("load", init, false);
```

init() ein HTML-Objekts anhand des Identifiers festgestellt und in der Variable **output** gespeichert. Weitere Funktionen können damit dieses HTML-Objekt verändern. Diese Funktion wird nur zum Testen benötigt.

```
1  function init() {
2        output = document.getElementById("output");
3  }
```

Folgende Funktionen bis einschließlich **doSend()** und **doSend_buffered()** betreffen direkt die Websocketverbindung.

mainWebSocket() erstellt ein neues Websocketobjekt und verbindet sich mit dem Server hinter der übergebenen URL. Auch werden weitere Funktionen verschiedenen Websocket-Methoden zugewiesen:

- **Websocket.onopen** – Aufruf bei Aufbau einer Websocketverbindung
- **Websocket.onclose** – Aufruf bei Schließen einer Verbindung
- **Websocket.onmessage** – Aufruf beim Erhalt einer Nachricht über Websocket
- **Websocket.onerror** – Aufruf bei Fehler (z.B. Aufbau einer Verbindung nicht möglich)

```
1  function mainWebSocket() {
2        websocket = new WebSocket(wsUri);
3        websocket.onopen = function () { onOpen(); };
4        websocket.onclose = function () { onClose(); };
5        websocket.onmessage = function (evt) { onMessage(evt); };
6        websocket.onerror = function (evt) { onError(evt); };
7  }
```

onOpen() bei erfolgreichem Verbindungsaufbau gibt die Funktion über **writeToScreen()** (s.u.) eine Meldung über die HTML-Seite unten aus und ändert die Verbindungsstatusanzeige in B (Abb.) auf „CONNECTED“. Für den Benutzer wird das Eingabefeld deaktiviert (**disabled = true**) und der Disconnect – Button aktiviert (**disabled = false**). Auch sendet diese Funktion die aktuelle Uhrzeit mit **doSend()** als erste Websocketnachricht.

```
1  function onOpen() {
2        writeToScreen("CONNECTED");
3        document.getElementById("showConnectionState").innerHTML =
   'CONNECTED';
4        document.getElementById("btnDisconnect").disabled = false;
5        document.getElementById("ipTextField").disabled = true;
6        connectedToServer = true;
7        doSend("D:" + Date().substr(16,8));
8  }
```

onClose() bei Beendigung einer Verbindung wird mit **writeToScreen()** eine Meldung unten an die HTML-Seite angehängt. Die beiden Buttons und das Eingabefeld werden entsprechend aktiviert/deaktiviert und in die Verbindungsstatusanzeige in B (Abb.) „DISCONNECTED“ geschrieben.

```
1  function onClose() {
2        writeToScreen("DISCONNECTED");
3        document.getElementById("btnConnect").disabled = false;
4        document.getElementById("btnDisconnect").disabled = true;
5        document.getElementById("ipTextField").disabled = false
6        document.getElementById("showConnectionState").innerHTML =
   'DISCONNECTED';
7        connectedToServer = false;
8  }
```

onMessage() bei Eintreffen einer neuen Websocketnachricht wird der Nachrichtentext mit dieser Funktion interpretiert. Die gesamte Nachricht befindet sich in der Variablen **evt**. Die Methode **data** gibt den Inhalt der Nachricht wieder.

Folgende Nachrichtentypen werden vom Server (ESP32) versendet und entsprechend verschiedene Aktionen ausgeführt:

- „**--serverIdle--**“: Server kann wieder Nachrichten empfangen
  → **serverIdle** – Flag wird true
  mit **doSend_buffered()** wird eine gepufferte Nachricht (wenn vorhanden) an den Server geschickt
- Statusänderung eines Tasters am Fahrzeug: z.B. **B0,1** = Taster rechts wurde gedrückt, **B1,1** = Taster links wurde gedrückt.
  → Textanzeige (ID = **Button0Val**) wird angepasst
  Hintergrundfarbe (ID = **Button0Box**) der o.g. Textanzeige ändert sich bei gedrücktem Taster auf Grün.
- weitere Nachrichten, z.B. Echo-Nachrichten
  → Nachricht wird (in blau zur besseren Unterscheidung) unten an die HTML-Seite angehängt

```
1   function onMessage(evt) {
2      if (evt.data === "--serverIdle--") {
3         serverIdle = true;
4         doSend_buffered("");
5      } else if ("B" === evt.data.substr(0,1)) {
6         if ("0" === evt.data.substr(1, 1)) {
7            document.getElementById("Button0Val").innerHTML = evt.data.substr(3, 1);
8            if ("1" === evt.data.substr(3, 1))
9               document.getElementById("Button0Box").style.backgroundColor = "#00ff00";
10           if ("0" === evt.data.substr(3, 1))
11              document.getElementById("Button0Box").style =
                   document.getElementsByClassName("mainBoarderBox").style;
12        }
13        if ("1" === evt.data.substr(1, 1)) {
14           document.getElementById("Button1Val").innerHTML = evt.data.substr(3, 1);
15           if ("1" === evt.data.substr(3, 1))
16              document.getElementById("Button1Box").style.backgroundColor = "#00ff00";
17           if ("0" === evt.data.substr(3, 1))
18              document.getElementById("Button1Box").style =
                   document.getElementsByClassName("mainBoarderBox").style;
19        }
20     } else {
21        writeToScreen('<span style="color: blue;">RESPONSE: ' + evt.data + '</span>');
22     }
23  }
```

onError() im Fehlerfall wir die Fehlernachricht in Rot unten an die HTML-Seite angehängt, sofern man sich im Testbetrieb befindet.

```
1   function onError(evt) {
2         writeToScreen('<span style="color: red;">ERROR:</span> ' +
          evt.data);
3   }
```

doSend() Bei Aufruf dieser Funktion wird der übergebene Text direkt über den Websocket an den Server verschickt und im Testbetrieb unten an die HTML-Seite angehängt. In unserem Programm wird diese Funktion für das Senden des Datums und im Testbetrieb verwendet.

```
function doSend(message) {
      writeToScreen("SENT: " + message);
      websocket.send(message);
}
```

doSend_buffered() Die Nachrichten werden nur dann über Websocket an den ESP32 gesendet, wenn dieser ein entsprechendes Signal (**--serverIdle--**) gegeben hat. Um keine Nachrichten zu verlieren, werden diese in einer Stringvariablen **outputDataStream** gepuffert. Als erstes prüft die Funktion, ob eine Websocketverbindung besteht (**connectedToServer**). Wenn ja, wird der übergebene Text an den Puffer angehängt. Schließlich wird die Nachricht versendet sofern möglich (**serverIdle** ist **true** und **outputDataStream** ist nicht leer).

```
function doSend_buffered(message) {
   if (connectedToServer) {
      if (message !== "") {
         outputDataStream = outputDataStream + message + ";";
      }
      if ((true === serverIdle) && ("" !== outputDataStream)) {
         websocket.send(outputDataStream);
         outputDataStream = "";
         serverIdle = false;
      }
   }
}
```

Die folgenden Funktionen verarbeiten die Betätigungen der Buttons und Slider.

mBtnFkt_takeIP() wird beim Drücken des Connect-Buttons aufgerufen. Die Funktion deaktiviert das Textfeld für die Adresse und den Connect-Button. Ein mehrmaliges Auswählen (und mehrmaliges Ausführen der Funktion) ist somit verhindert. Aus der eingegebenen IP-Adresse wird die URL (**wsUri**) erstellt, für die Websocketverbindung zum Server. Ebenso wird die Funktion **mainWebSocket()** aufgerufen welche den Websocket startet.

```
function mBtnFkt_takeIP() {
   document.getElementById("btnConnect").disabled = true;
   document.getElementById("ipTextField").disabled = true;
   wsUri = "ws://"+document.getElementById("ipTextField").value+"/";
   mainWebSocket();
}
```

mBtnFkt_disconnect() wird beim Drücken des Disconnect-Buttons aufgerufen. Hiermit wird der Websocket geschlossen.

```
function mBtnFkt_disconnect() {
      websocket.close();
}
```

Zur Verwendung der Slider werden 3 nahezu identischen Funktionen verwendet, daher folgt nur eine Erklärung für den rechten Motor (Motor 0).

Die Variable **sliderMotor0** erhält eine Referenz auf den entsprechenden Slider.

```
var sliderMotor0 = document.getElementById("myMotor0Range");
```

Wird der Slider bewegt, ruft die Methode **oninput** die an dieser Stelle definierte Funktion auf. Darin wird der neue Motorwert neben dem Slider auf der HTML-Seite angezeigt. Dieser Wert wird auch als Text der Funktion **doSend_buffered** übergeben.

```
1  var sliderMotor1 = document.getElementById("myMotor1Range");
2     sliderMotor1.oninput = function () {
3     document.getElementById("Motor1Val").innerHTML =
         sliderMotor1.value;
4     doSend_buffered("M1:" + sliderMotor1.value);
5  };
```

5.3.4 Testen des Clients/der Fernsteuerung mit Echo-Server (ohne ESP32)

Die Fernsteuerung kann getestet werden, indem im HTML-Dokument (index.html) unten die Zeile

```
1  <div id="output">Output of WebSocket-Client:</div>
```

(**<!--** und **-->** entfernen) und im JavaScript-Dokument (main.js) die Funktion writeToScreen() einkommentiert wird.

Jetzt ist die Textausgabe unten im HTML-Dokument aktiviert. Im Eingabefeld für die IP-Adresse kann die URL eines beliebigen Websocketservers mit Echofunktion eingegeben werden, z.B. echo.websocket.org. Da die Echoserver i.d.R. keine **serverIdle**-Nachricht versenden, darf das **serverIdle**-Flag in der main.js nicht auf false gesetzt werden. Wir setzten daher das Flag in der Deklaration (oben) und in der **doSend_buffered()** – Funktion (unten) auf true. Dies muss nach dem Testen wieder rückgängig gemacht werden.

Alle Nachrichten, die über Websocket verschickt werden, kommen jetzt direkt wieder zurück (Echo) und können angezeigt werden, wie im Beispiel in Abbildung 9 gezeigt.

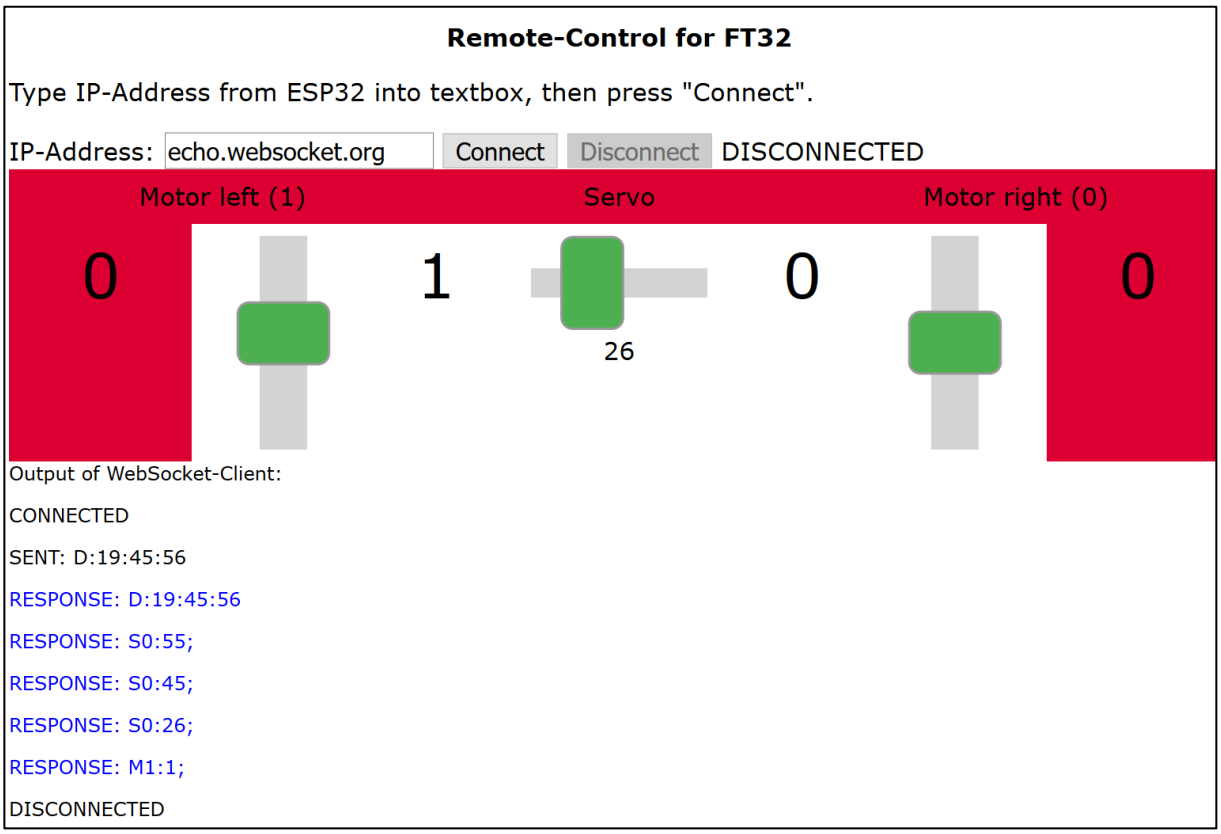

Abbildung 9: Testbeispiel mit Echo-Server

Somit sieht man, dass alle Nachrichten erfolgreich über Websocket versendet und empfangen werden können. Außerdem sind auch die Slider mitgetestet.

5.4 Inbetriebnahme / Abschlusstest

Vorgesehen ist der Betrieb dieses Websocketbeispiel mit der Fernsteuerung in einem WLAN. Idealerweise können die Dateien (index.html, default.css, main.js) auf einem (Heim-)Server abgelegt und von dort abgerufen werden.

Wenn ein eigener Server nicht zur Verfügung steht kann ein solcher leicht unter Windows mit dem Programm HFS ~ Http File Server (www.rejetto.com/hfs/) erstellt werden. Eine Installation ist nicht notwendig.

Nach dem Start des Programms wird der Ordner mit den Dateien der Fernsteuerung (index.html, default.css, main.js) dem Server hinzugefügt: Rechtsklick auf das Häuschen links → Add folder from disk… (siehe Abbildung 10).

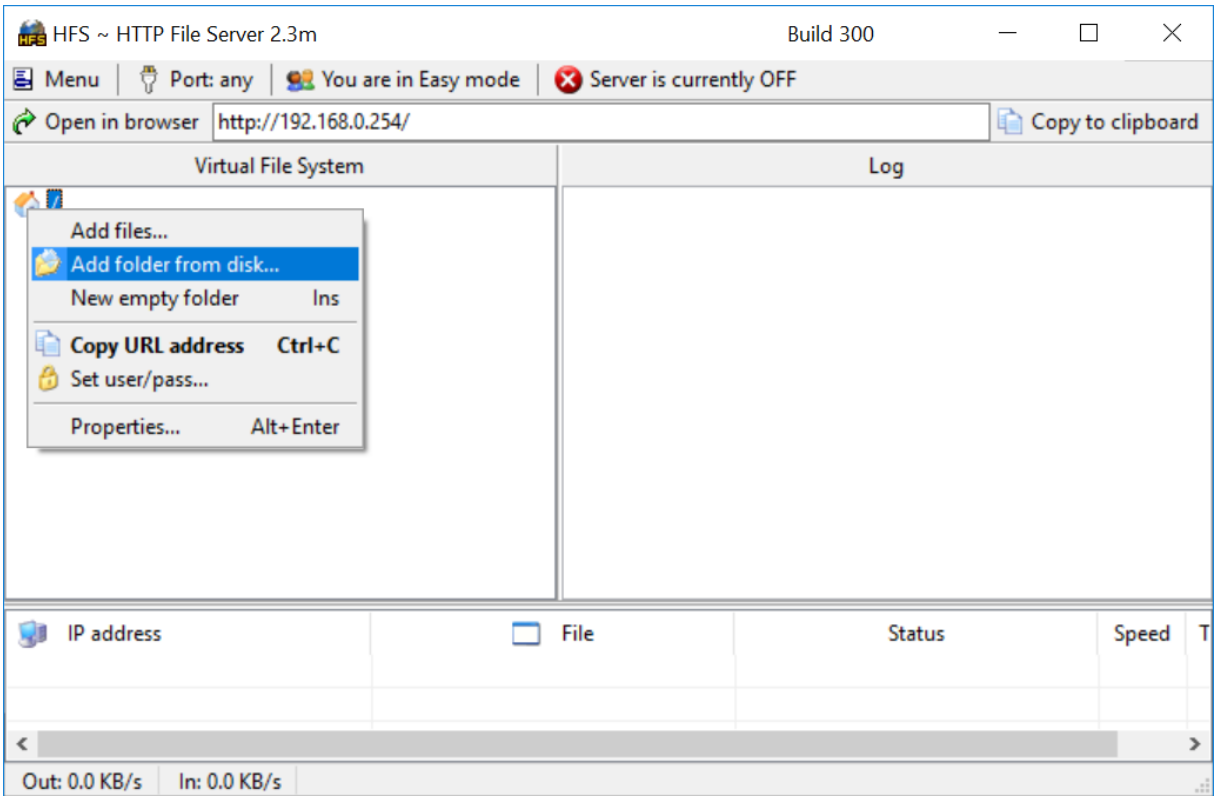

Abbildung 10: Inbetriebnahme mit HFS - Auswahl des Projektordners

Den Projektordner auswählen, in dem die oben genannten Dateien liegen, mit OK bestätigen Abbildung 11).

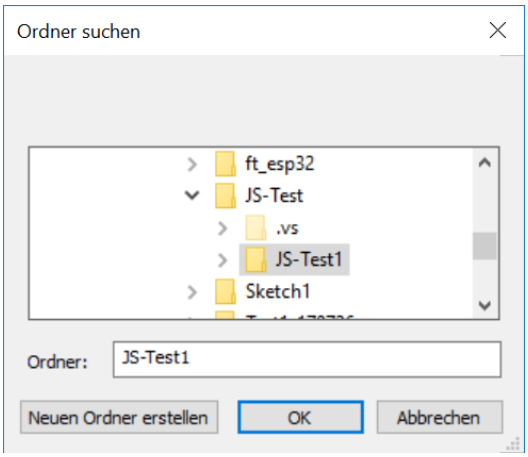

Abbildung 11: Inbetriebnahme mit HFS - Auswahl des Projektordners

Im nächsten Fenster (Abbildung 12) Virtual folder auswählen, dies ist für unsere Kurztests ausreichend.

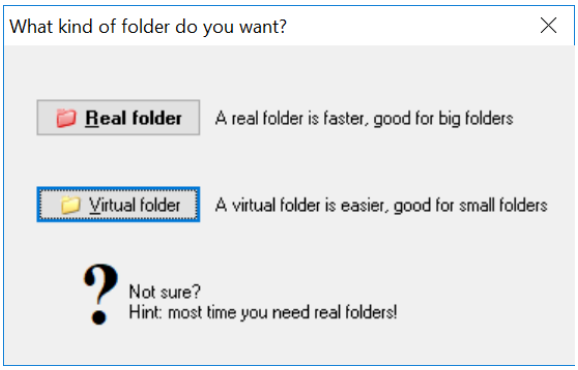

Abbildung 12: Inbetriebnahme mit HFS - Auswahl der Ordnerstruktur

Jetzt muss der Server noch eingeschalten werden. Mit Klick auf Server is currently off wird der Server aktiv und kann von allen Geräten, insbesondere Smartphones, im Heimnetz unter der angegebenen IP-Adresse über den Browser aufgerufen werden (Abbildung 13).

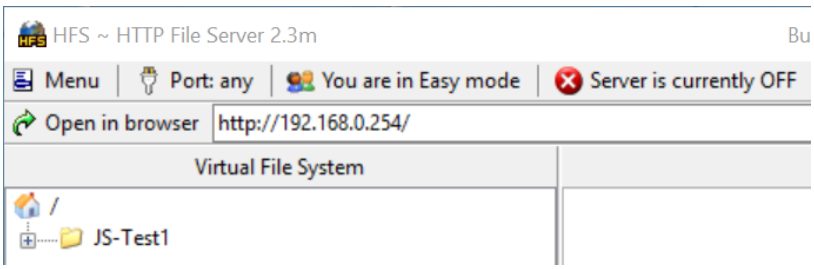

Abbildung 13: Inbetriebnahme mit HFS - Aufsetzen des Servers

Auswählen des Projektordners öffnet die die erstellte Oberfläche der Fernsteuerung (Abbildung 14).

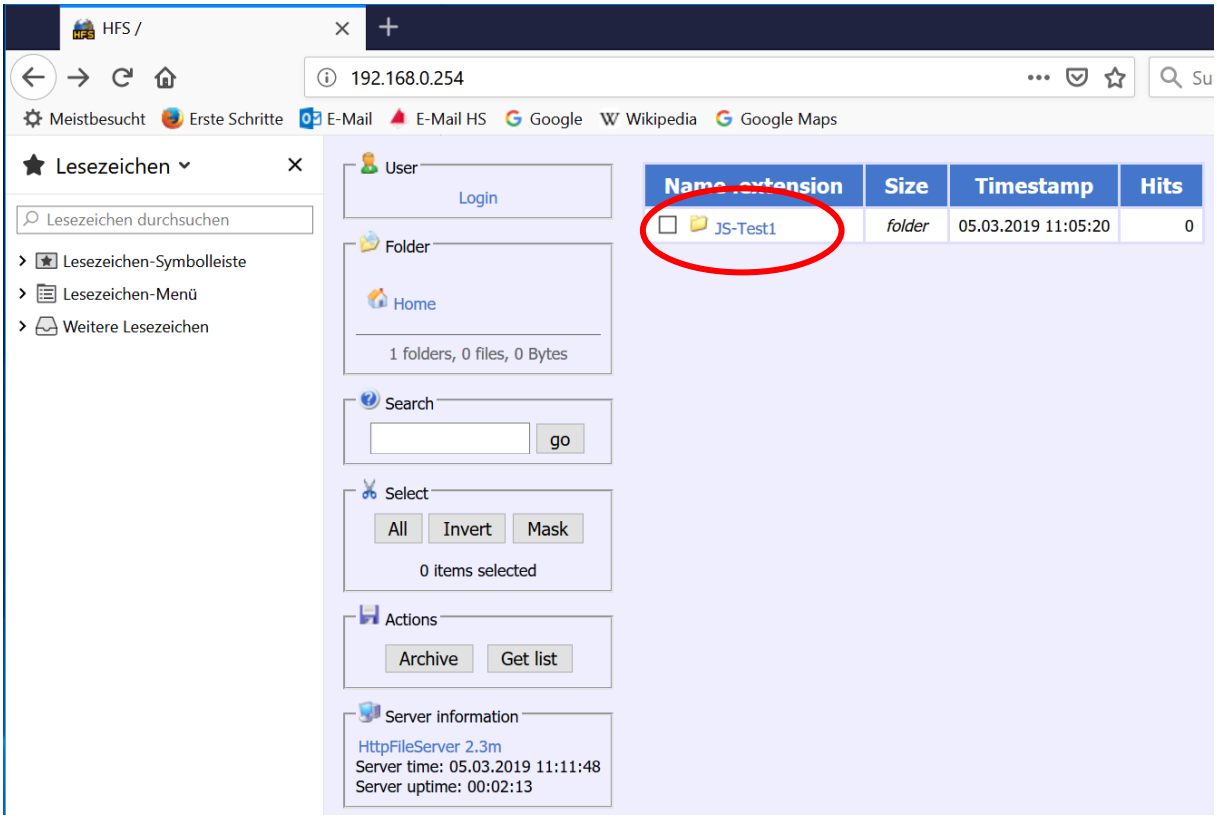

Abbildung 14: Inbetriebnahme mit HFS - Aufrufen des Servers mit Firefox-Browser

6 Weitere Punkte

In diesem Kapitel werden Elemente der umgesetzten Wifi-Fernsteuerung behandelt, welche nicht zwingend für das Websocket-Projekt notwendig sind (Optimierung Übertragungsgeschwindigkeit) oder weiterführende Informationen bieten (Übertragungsprotokoll).

6.1 Optimierung Übertragungsgeschwindigkeit

Während der Entwicklungsphase stellte sich heraus, dass beim Übertragen großer Datenmengen vom Client an den Server die Befehle nur langsam abgearbeitet werden. Wenn z.B. der Slider des Servos bewegt wurde, bewegte sich der Servomotor nur langsam und es dauerte mehrere Sekunden, bis die endgültige Position erreicht wurde.

Eine hierfür im Programm eingefügte Zeitmessung bestätigte, dass vom Auslesen einer Nachricht aus dem Websocket-Puffer bis zum Ausführen des Befehls ca. 121ms vergehen, siehe Abbildung 15.

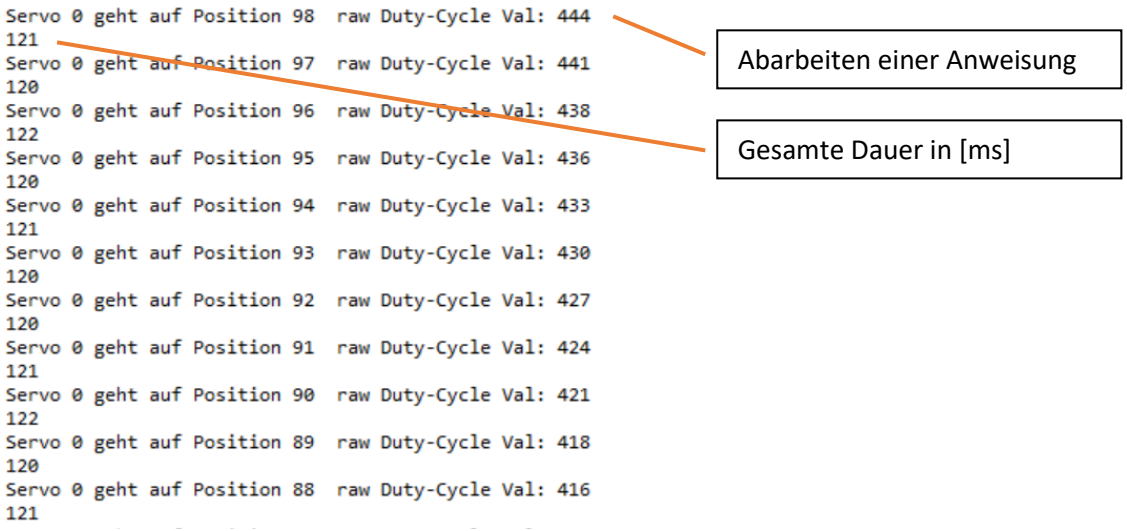

Abbildung 15: Ablaufzeit ohne Geschwindigkeitsoptimierung

Solche Zeitverzögerungen sind für einen Fernsteuerungsbetrieb nicht tragbar, daher wurde nach einer Erhöhung der Übertragungsgeschwindigkeit gesucht.

Eine erste Vermutung war, dass die Übertragung über Websocket und das Auslesen aus dem Websocket-Puffer den Großteil der Übertragungszeit verursacht. Um die Anzahl der Nachrichtensendungen zu verringern werden die Anweisungen auf Client-Seite gesammelt und nur dann versendet, sobald der Server neue Nachrichten verarbeiten kann. Dies wird mit **`--serverIdle--`** dem Client signalisiert. Dies führt zu einer Verringerung der Zykluszeit um 10ms, siehe Abbildung 16.

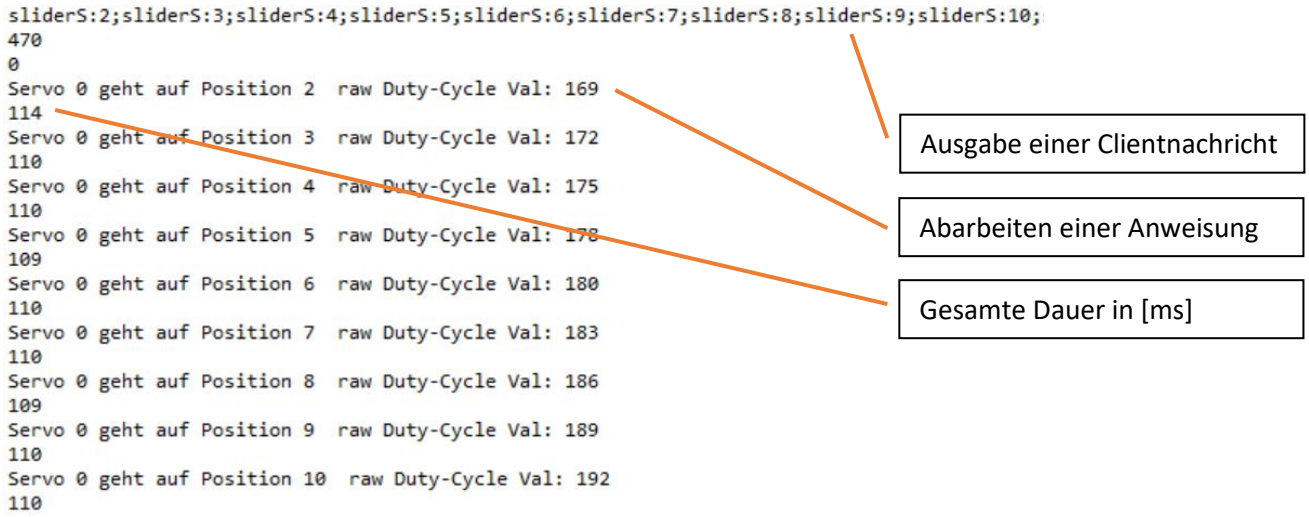

Abbildung 16: Ablaufzeit mit clientseitig gepufferten Nachrichten

An dieser Stelle wird sichtbar, dass ein Großteil der Verzögerungszeit nicht beim Versenden der Nachricht entsteht, sondern beim Abarbeiten.

Bei einer Untersuchung des Quellcodes auf dem Server ist die Ansteuerung des über I²C betriebenen Displays als Hauptverursacher für die Zeit festgestellt worden. Bis dahin wurden alle empfangenen Nachrichten zusätzlich auf dem Display ausgegeben. Werden diese Ausgaben weggelassen, sinkt die tatsächliche Verarbeitungszeit der Nachricht auf max. 6ms pro Nachrichtenblock, siehe Abbildung 17.

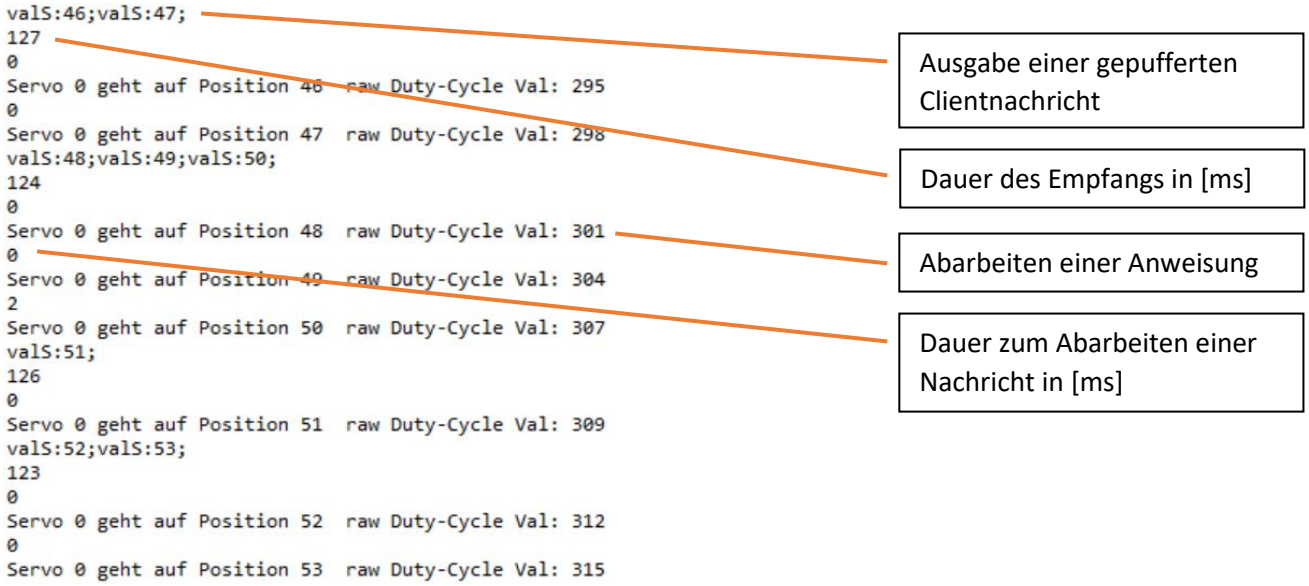

Abbildung 17: Ablaufzeit mit Puffer und ohne Display

Die gesamte Übertragungsdauer beträgt wie in Abb. weiterhin bis zu 130ms, diese schließt aber das Senden von **`--serverIdle--`** und die Verarbeitung auf Client-Seite mit ein.

Durch Anwendungsversuchen mit verschiedenen Probanden wurde die obige Verzögerungszeit als annehmbar bewertet.

6.2 Übertragungsprotokoll (innerhalb des Websocket)

In der vorliegenden Umsetzung des Websocket werden die Nachrichten zwischen dem Server und dem Client als Zeichenkette (String) übertragen. Somit ist zwischen den beiden Parteien eine

festgelegte Codierung notwendig, welche erlaubt, verschiedene Informationen eindeutig, erweiterbar und fehlerfrei zu übertragen.

Für diese Arbeit wurde die Codierung wie folgt festgelegt und an Beispielen gezeigt:

6.2.1 Nachrichten vom Client (Fernsteuerung) an den Server (ESP32):

Beispiele: **`D:10:02:36`**
`M0:5 ; M1:-2 ; S:50 ;`

- Information wird in Nachrichtenblöcken verschickt, diese sind durch Strichpunkte (ohne Leerzeichen) voneinander getrennt.
  Ausnahme: Datum (**`D`**) wird nur als einzelne Nachricht verschickt.
- Jeder Nachrichtenblock beginnt mit einem Buchstaben als Identifier:
  - **`D`** – Datum (eher Uhrzeit) für den Handshake
  - **`M`** – Motor, weitere Einteilung in **`M0`** (Motor 0) und **`M1`** (Motor1)
  - **`S`** – Servomotor
- Es folgt ein Doppelpunkt, danach beginnen die Zahlenwerte. Die Werte sind die Anweisungen an die Treiber für die Aktoren (Ausnahme beim Datum)
  - **`M`** Werte: **`-8..8`** (Radgeschwindigkeit)
  - **`S`** Werte: **`0..100`** (Servoposition)

6.2.2 Nachrichten vom Server (ESP32) an den Client (Fernsteuerung):

`--serverIdle--`	Wird vom Server versendet, wenn dieser die alte Nachricht empfangen hat du neue Nachrichten empfangen kann.
`B0:1`	Taster **`T0`** hat von nicht betätigt auf betätigt gewechselt
`B1:0`	Taster **`T1`** hat von betätigt auf nicht betätigt gewechselt

7 Zusammenfassung und Ausblick

Die vorliegende Arbeit/Anleitung bietet einen einfachen Einstieg in die Nutzung des Websocket-Protokolls auf dem ESP32 in Form einer kleinen Fahrzeugfernsteuerung. Ein Ziel war es dabei, alle Softwarekomponenten möglichst einfach zu halten. Auf dieser Basis bieten sich eine Vielzahl von Erweiterungen des Projekts an, welche gerne von Interessenten ausprobiert und umgesetzt werden können. Im Folgenden sind einige Beispiele und Anregungen für Weiterentwicklungen gegeben.

- Graphisch ansprechendere Fernsteuerung
  Der Einfachheit halber ist die Darstellung im Browser sehr schlicht gehalten. Insbesondere beim Betrieb auf Smartphones o.Ä. treten hier häufig kleinere Skalierungsprobleme auf. Mit Bibliotheken oder Frameworks, wie z.B. Bootstrap lassen sich gute Webseiten für verschiedene Endgeräte erstellen.
- Behandlung von Mehrfachzugriffen/ggf. Passwortschutz
  Bisher kann sich nur ein Client mit dem Server (ESP32) verbinden, Anfragen weiterer Clients werden nicht bearbeitet. Idealerweise werden alle Anfragen in irgendeiner Form bearbeitet, gegebenenfalls kann der Zugriff durch Passwort o.ä. kontrolliert werden.
- Ablegen der Fernsteuerungssoftware auf den internen Speicher des ESP32 (SPIFFS)
  Bisher war nur ein einfacher Betrieb in einem festen wireless – Netzwerk mit Server vorgesehen. Um das Fahrzeug und die Fernsteuerung unabhängig zu verwenden, kann der ESP32 den eigenen Accesspoint und Serverfunktionen zur Verfügung stellen.
- Flexiblerer Wechsel in verschiedene Netzwerke
  Im vorliegendem Projekt ist das Netzwerk (einschließlich Passwort) fest im Programm hinterlegt. Um den Wechsel in andere Netzwerke zu vereinfachen soll es aus dem Browser heraus eine Möglichkeit geben, z.B. durch eine geeignete Eingabemaske. Die Zugänge können dafür auf dem internen Speicher (verschlüsselt) abgelegt werden.
- Allgemeine Erweiterungen
  - Das Fahrzeug kann weitere Funktionalitäten erhalten
  - Kommunikation zu mehreren Fahrzeugen und Kommunikation zwischen Fahrzeugen
  - ESP32 bietet weitere Anschlussmöglichkeiten → Verwendung weiterer Sensorik und Aktorik

8 Literaturverzeichnis

[1] Wang, V.; Salim, F.; Moskovits P.: “The Definitive Guide to HTML5 WebSocket”, 2013, Apress Verlag

[2] Fette, I.; A. Melnikov, A.: „The WebSocket Protocol”, https://tools.ietf.org/html/rfc6455, 2011, zuletzt abgerufen am 12.03.2019

[3] Keck, M.; Tolic, T.: „Implementierung einer Kommunikationsschnittstelle zwischen dem ESP32 und einem Smart Device und einer Benutzeroberfläche für die Kommunikation zwischen den Geräten“, 2018, Bachelorarbeit Hochschule Karlsruhe – Technik und Wirtschaft

Hilfreiche Webseiten für HTML5 und JavaScript

w3schools.com

selfhtml.com