ArduSmartPilot mit WiFi-Server und MIT App Inventor

Motivation

Anfangs wurde der ArduSmartPilot über Bluetooth (BT) gesteuert. Je nach Smartphone und BT-Modul sind hier bei aber die Reichweiten der Fernsteuerung nicht komfortabel.

Daher wurde mit dem ESP8266 (Platine ESP-201) die WLAN-Kommunikation eingeführt, womit sich Reichweiten oberhalb 200 m ergeben. Die Android-App hierbei wurde wie bei BT über die IDE Processing erstellt.

Inzwischen ist das Erstellen der Android-APP mit dem „MIT App Inventor“ sehr komfortabel geworden, da diese IDE webbasiert ist, man hierfür also keine Programme auf dem Rechner installieren muss. Besonders für Schüler ist der App Inventor aufgrund seiner grafischen Programmiersprache nach wie vor bestens geeignet.
Daher war es längst überfällig, den ArduSmartPilot entsprechend zu erweitern.

Systemdesign

Auf dem ESP wird ein „WiFi-Server“ programmiert. Dies ist eigentlich ein rudimentärer Webserver.

Jedoch sendet er an seinen Client (das Androidgerät) keinerlei Daten zurück wie es normaler Webserver im Internet machen. Bei dem hier beschriebenen WiFi-Server werden praktisch keine HTML-Inhalte übertragen, weshalb die Kommunikation ähnlich schnell ist wie mit der WLAN-Kommunikation mit der Processing-App.

Man kann auf dem ESP auch einen „normalen Webserver“ programmieren, wie im Projekt „Legoino“ der Hochschule Reutlingen dokumentiert ist. Hierbei ist aber die Latenzzeit zu groß, um ein Flugzeug zu steuern, da das komplette Layout der App über WLAN übertragen wird.

Hardwareaufbau

In Tabelle 1 und 2 ist die Anschlussbelegung des ESP-201 für die Programmierung bzw. für den Betrieb angegeben. Die darin genannten 10 kΩ Pull Up und Pull Down Widerstände sind nicht zwingend erforderlich, werden aber laut Datenblatt des ESP8266 empfohlen, um undefinierte Zustände an den Eingängen und eine noch höhere Stromaufnahme zu vermeiden.

Den Hardwareaufbau baut man am besten zuerst auf einem Steckbrett auf (siehe Abb. Fehler: Referenz nicht gefunden). Dabei ist zu beachten, dass die Steckbrett-Spannungsversorgung einen genügend hohen Strom liefern kann: Der ESP8266 kann Ströme bis ca. 200 mA ziehen. Der Betrieb des ESP-201 mit dem Foca- oder FTDI-Adapter mit aktiviertem WLAN ist aus diesem Grund instabil. Zum Programmieren liefern diese beiden Adapter jedoch ausreichend Strom.

Die PWM-Ausgänge werden am besten einzeln mit einem Servo getestet, ebenfalls damit die Strombelastung nicht zu hoch wird. Meistens ist aber gar nicht die Steckbrettspannungsversorgung für einen Spannungseinbruch verantwortlich: Manche USB Anschlusskabel haben derart kleine Adernquerschnitte, so dass der Spannungsabfall nicht vom Spannungsregler auf dem Adapter sondern vom USB-Kabel verursacht wird!

Prof. Dr. rer. nat. Stefan Mack, Fakultät Technik / Mechatronik, Hochschule Reutlingen, Alteburgstraße 150, 72762 Reutlingen,
www.reutlingen-university.de, T. +49 (0)7121 271-7070, F. +49 (0)7121 271-7004, stefan.mack@reutlingen-university.de

Auf dem ESP-201 ist eine Platinenantenne vorhanden. Über den U.FL Stecker kann mit einer externen Antenne die Reichweite etwas vergrößert werden, nachdem die Platinenantenne über das Durchtrennen der entsprechenden Leiterbahn „abgeklemmt“ wurde.

Pin am ESP-201...	**… wird laut Datenblatt zum Programmieren belegt mit:**
IO0	Direkt auf GND
IO2	Über 10 kΩ Pull Up Widerstand auf 3,3 V [1] [2]
IO15	Über 10 kΩ Pull Down Widerstand auf GND [1]
CHIP_EN bzw. CHIP_PD	Direkt auf 3,3 V
RST (Reset)	Über 10 kΩ Pull Up Widerstand auf 3,3 V, bei Reset kurz direkt auf GND oder floatend (=unbeschaltet) Dann Reset beim Einschalten.
3,3 V	Direkt auf 3,3 V (Stromversorgung)
RX	TX (Foca/FTDI auf 3,3 V einstellen!)
TX	RX (Foca/FTDI auf 3,3 V einstellen!)
GND	Direkt auf GND (Stromversorgung)

Tabelle 1: Belegung der Anschlusspins des ESP-201 für die Programmierung.

Pin am ESP-201...	**… wird laut Datenblatt im Betrieb belegt mit:**
IO0	Direkt auf 3,3 V [2]
IO2	Über 10 kΩ Pull Up Widerstand auf 3,3 V [1] [2]
IO15	Über 10 kΩ Pull Down Widerstand auf GND [1]
CHIP_EN bzw. CHIP_PD	Direkt auf 3,3 V
RST (Reset)	Floatend (=unbeschaltet). Dann automatisch Reset beim Einschalten.
3,3 V	Direkt auf 3,3 V (Stromversorgung)
GND	Direkt auf GND (Stromversorgung)
IO13	Servo Höhenruder (PWM)
IO12	Servo Seitenruder (PWM)
IO14	Eingang ESC Motorregler (PWM)

Tabelle 2: Belegung der Anschlusspins des ESP-201 für den Betrieb.

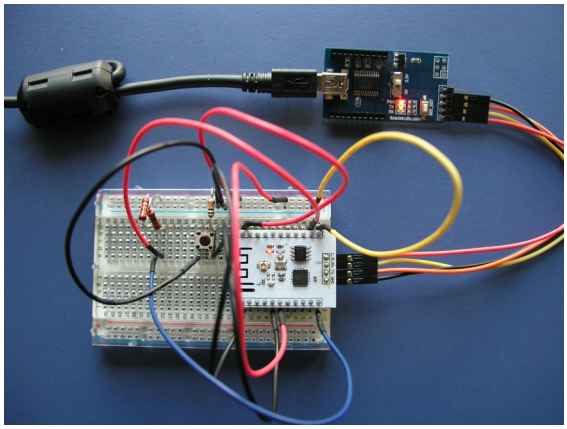


Abbildung 1: Aufbau zum Programmieren ("Flashen") des ESP-201 mit einem Foca-Adapter.

Programmieren („Flashen“)

Bei der Arduino IDE (Version 1.6.5 oder neuer) muss zuerst der ESP8266 als neue Platine hinzugefügt werden: Dazu sind folgende Schritte notwendig:

1) In der Arduino IDE unter „*Datei -> Voreinstellungen -> Additional Boards Manager URLs:*“ den Link „*http://arduino.esp8266.com/staging/package_esp8266com_index.json*“ eingeben.

2) In der Arduino IDE wählen: „*Werkzeuge -> Platine -> Boards Manager -> esp8266 installieren*“

1 Die Pull Up bzw. Down Widerstände sind nicht zwingend nötig, siehe Abschnitt Hardwareaufbau.

2 Kann auch floaten (= unbeschaltet). Jedoch dann störanfälliger z.B. bei Berührung der Platine oder elektromagnetischer Einstrahlung.

Für das Programmieren des ESP-201 sind die in Abb 2 dargestellten Einstellungen bei der Arduino IDE vorzunehmen.

Für das Übertragen des Programms ist ein USB-Seriell-Wandler wie z.B. der Foca- oder der FTDI-Adapter nötig. Der Aufbau (siehe Abb. 1) hierfür erfolgt auch am besten auf einem Steckbrett, wobei bei einem guten USB-Kabel die 3,3 V Spannungsversorgung durch den Adapter ausreicht. Anders als beim Arduino Pro Mini wird der ESP beim Herunterladen des Programms nicht automatisch in den Programmiermodus versetzt und erhält auch keinen Resetbefehl.

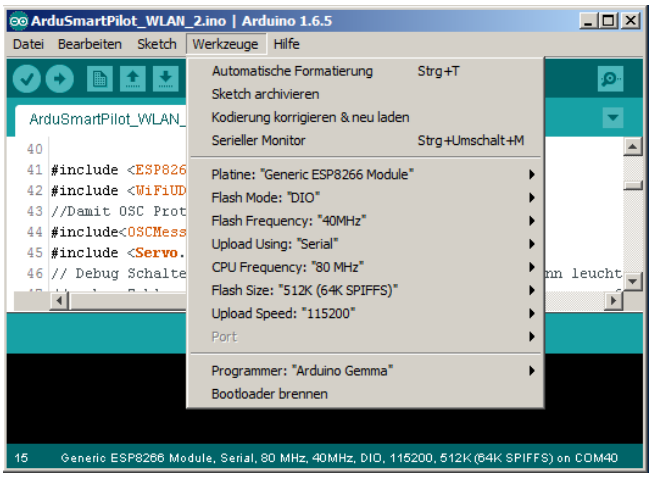


Abbildung 2: Einstellungen der Arduino IDE zur Programmierung des ESP8266.

Ist der RST Pin nicht angeschlossen (floatet), so führt der ESP8266 nach dem Einschalten der Spannungsversorgung automatisch einen Reset aus, was durch ein kurzes Aufblinken der blauen LED angezeigt wird. Ansonsten mus der RST-Pin kurz auf GND gelegt werden.

Die Kommunikation mittels WLAN verwendet das HTTP Protokoll.
Der Quellcode für den ESP ist dem Buch „Das ESP8266-Praxisbuch“ von E. Bartmann entnommen [1].
In diesem Codebeispiel bucht sich der ESP wie auch das Androidgerät in ein vorhandenes WLAN ein.

In einer zukünftigen Version wird der ESP wie in vorhergehenden Processing-Version als Access Point (AP) sein eigenes WLAN aufspannen. Erst dann kann der ArduSmartPilot ohne zusätzlichen WLAN-AP (am Boden) fliegen.

```
Serial port COM23 opened
Try to connect to SSID: Wehlan
Try to connect to SSID: Wehlan
SSID: Wehlan
IP Adresse: 192.168.178.49
```

Abbildung 3: Ausgabe der seriellen Schnittstelle auf einem Terminalprogramm: Hier ist zu sehen, dass der ESP sich erfolgreich in das WLAN "Wehlan" eingebucht hat, und dort die IP-Adresse 192.168.178.49 erhalten hat.

Technisch und organisatorisch ist es jedoch kein großes Problem, auf dem Flugfeld für diesen WLAN-AP zu sorgen: Man kann dafür z.B. einen TP-Link TL-WR802N verwenden (ca. 20 €), den man mit einer USB-Powerbank mit Strom versorgt.

Über die serielle Schnittstelle des ESPs (Baudrate 9600) wird kontrolliert, ob das Einbuchen in das WLAN funktioniert hat, und welche IP-Adresse der ESP vom Router erhalten hat (Siehe Abb. 3).

Quellcode des ESP:

Der Quellcode stammt aus dem ESP8266-Praxisbuch von E. Bartmann [1] und ist dort ab Seite 270 beschrieben. Er wurde für den ArduSmartPilot nur geringfügig modifiziert. Hier muss noch die SSID und das Passwort des WLANs entsprechend geändert werden.

```
//Quellcodebeispiel entnommen aus E. Bartmann: Das ESP 8266-Praxisbuch.
#include <ESP8266WiFi.h>

char ssid[] = "SSID des Wlan APs"; // SSID
char pass[] = "Das Passwort"; // Netzwerk-Passwort

int status = WL_IDLE_STATUS; // Speichert Verbindungsstatus
String cmd; // Speichert Client-Zeilen
WiFiServer server(80); // WiFi-Server erstellen

void setup() {
  Serial.begin(9600);
  while (status != WL_CONNECTED) {
    Serial.print("Try to connect to SSID: ");
    Serial.println(ssid);
```

```
    status = WiFi.begin(ssid, pass);
    delay(2000); // Kurze Pause
  }
  server.begin(); // Server starten
  printWifiStatus(); // WiFi-Status anzeigen
}

void loop() {
  WiFiClient client = server.available();
  // Kommt eine neue Client—Anfrage?
  if (client) {
    Serial.println("New client...");
    boolean newLine = true;
    while (client.connected())
    {
      if (client.available())
      {
        char c = client.read();
        if (c == '\n' && newLine) {
          // Standard—HTTP—Response Header senden
          client.println("HTTP/1.1 200 OK");
          client.println("Content—Type: text/html");
          client.println("Connection: close");
          client.println();
          client.println("<!DOCTYPE HTML>");
          client.println("<html>");
          client.println("</html>");
          break;
        }
        if (c == '\n') {
          //Efine neue Zeile beginnt
          newLine = true;
          //Serial.println("Empfangene Zeile: ");
          //Serial.println(cmd);
          //Zum Debuggen, um zu sehen, welche Stings via HTTP beim ESP angkommen
          parseStream(cmd);
          cmd = ""; // Zeileninformation loeschen
        }
        else if (c != '\r') {
          // Zeichen einer Zeile werden empfangen
          newLine = false;
          cmd += c; // Zeichen der Zeile hinzufuegen
        }
      }
    }
    delay(5); // Dem Web—Browser Zeit geben
    client.stop(); // Verbfindung schliessen
    Serial.println("Client disconnected");
  }
}


void parseStream(String c) {
  String all, servo;
  int value;

  if (c.indexOf("GET") != -1) { // Nach GET parsen
    all = c.substring(5); // Abschneiden von GET und des ersten Schraegstrichs
    servo = all.substring(0, all.indexOf(":")); //Bis zum Doppelpunkt Text ausschneiden
    value = all.substring(all.indexOf(":") + 1, all.indexOf(" ")).toInt(); //Nach dem
Doppelpunkt bis zum ersten Leerzeichen Text ausschneiden
    Serial.println(servo);
    Serial.println(value);
  }
}

void printWifiStatus() {
  // Anzeigen des WiFi-Status
  Serial.print("SSID: ");
  Serial.println(WiFi.SSID());
```

```
  // Anzeigen der WiFi-IP-Adresse
  IPAddress ip = WiFi.localIP();
  Serial.print("IP Adresse: ");
  Serial.println(ip);
}
```

Android App

Das Erstellen der App ist ebenfalls in [1] ab Seite 302 beschrieben. Wichtig ist hier, dass die richtige IP-Adresse (sicher eine andere als in Bild 5 zu sehen!) des ESPs angegeben wird.

Der grafische Quellcode ist in Abb. 4 und Abb. 5 zu sehen.

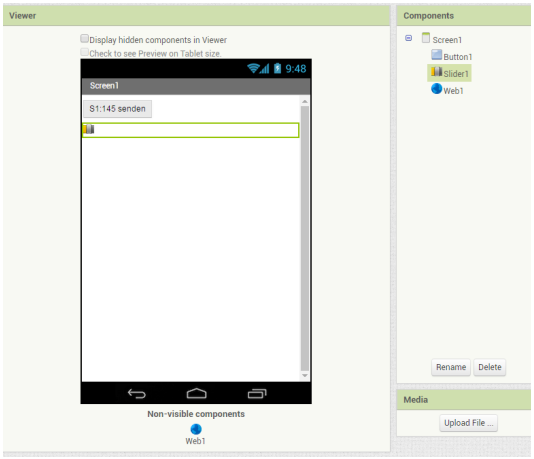


Abbildung 4: Benutzeroberfläche der Android APP.

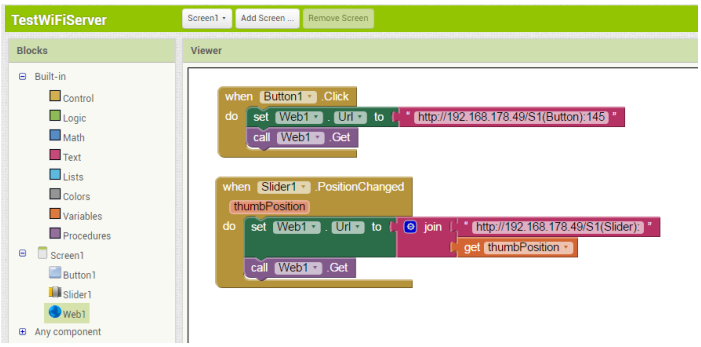


Abbildung 5: Grafischer Quellcode der Android-App.

```
New client...
S1(Button)
145
Client disconnected
New client...
S1(Button)
145
Client disconnected
New client...
S1(Slider)
31
Client disconnected
New client...
S1(Slider)
32
```

Abbildung 6: Textausgabe über die serielle Schnittstelle beim Drücken des Buttons bzw. bewegen des Sliders auf dem Androidgerät.

Die so erzeugte APP kann man entweder als APK-Datei herunterladen und manuell auf dem Androidgerät oder mit Hilfe der APP „MIT AI2 Companion“ (aus dem Google Store) automatisch installieren.

Das Androidgerät muss sich zuerst in das selbe WLAN wie der ESP einbuchen.

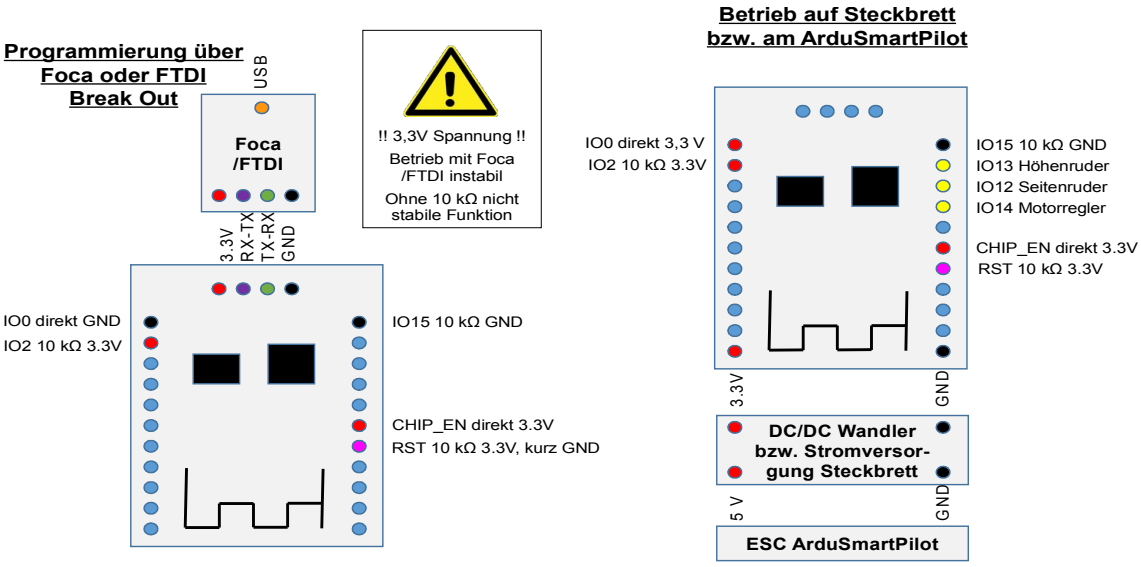
Drückt man bei der Android-App nun den Button oder bewegt den Slider, dann wird zur Kontrolle auf der seriellen Schnittstelle des ESP der in Abb. dargestellte Text ausgegeben.

Die hier vorgestellte App wie auch der Quellcode für den ESP stellen nur Prinzipbeispiele dar. Für den ArduSmartPilot müssen sie noch entsprechend erweitert werden.

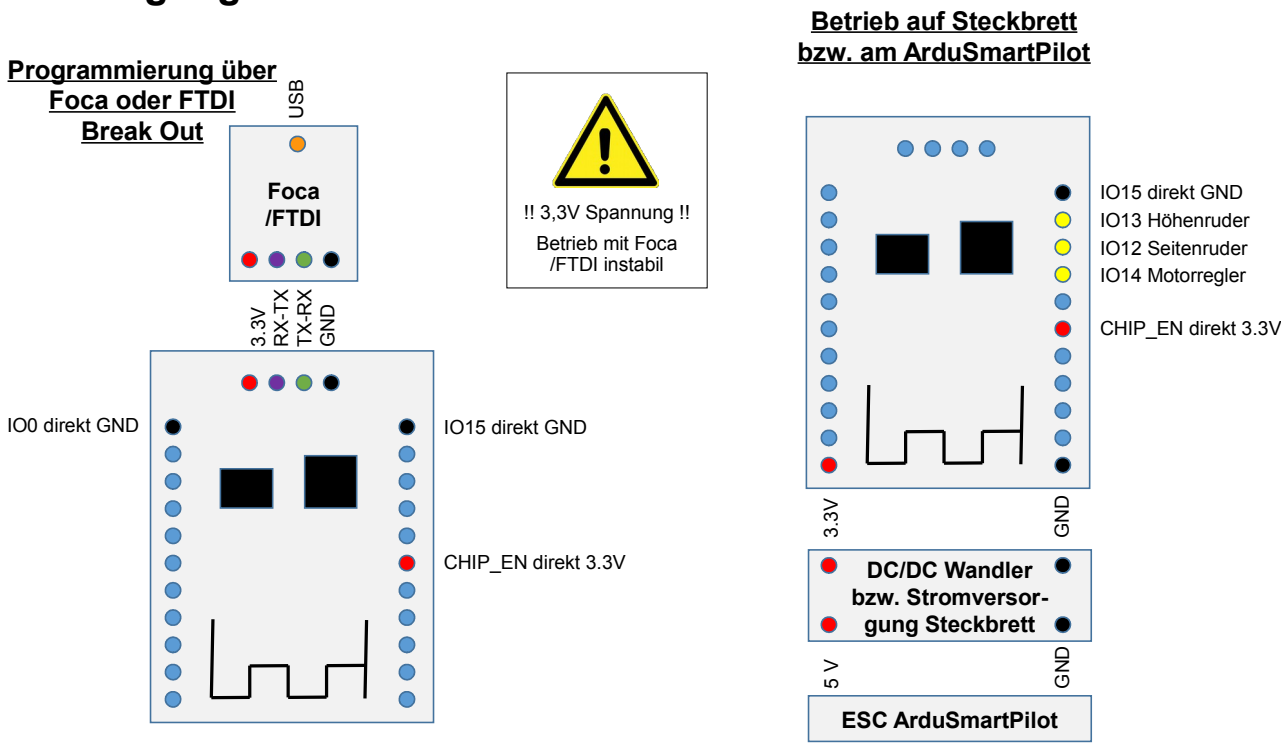
Praktische Erfahrungen

Mangels Zeit und Sommerwetter wurde diese Version des ArduSmartPilot noch nicht komplett an einem Flugzeug realisiert und getestet.
Die hier vorliegende Anleitung befasst sich nur prinzipiell mit der Software auf dem ESP bzw. auf dem Androidgerät.

Pinbelegung nach Datenblatt ESP8266



Pinbelegung: Minimalversion



Literaturverzeichnis

1: Bartmann, E.: Das ESP-8266 Praxisbuch. Elektor-Verlag, Aachen, 2016.

Prof. Dr. rer. nat Stefan Mack Januar 2017