



Julius Bartel

# Entwicklung robuster Übertragung sensibler Messwerte

Bachelorarbeit

---

Sommersemester 2021 Betreuer: Prof. Dr.-Ing. Dennis Trebbels

---

## **Vorwort**

Diese Arbeit ist dem Studienrektor meines Gymnasiums gewidmet. Sie überreichten mir mein Abiturzeugnis mit dem Kommentar: "Herr Bartel, wir dachten nicht, dass Sie das überhaupt schaffen". Jetzt habe ich es sogar geschafft, meine Bachelorarbeit abzugeben - das hätten Sie wahrscheinlich auch nicht gedacht.

Des Weiteren widme ich diese Arbeit meinem damaligen Klassenlehrer - ohne Sie hätte ich das tatsächlich nicht gemeistert. Danke.

---

## **Selbstständigkeitserklärung**

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

---

Ort, Datum

---

Unterschrift

---

## Zusammenfassung

Thema dieser Arbeit ist die Entwicklung einer robusten Kommunikation zwischen zwei Microcontrollern ( $\mu$ Cs) zur Weiterleitung von sensiblen Messdaten und der anschließenden Weiterleitung der Daten an ein übergeordnetes Netzwerk. Die Entwicklung soll an Hand eines Praxisbeispiels demonstriert werden.

Grundlage der Arbeit ist meine vorhergegangene Studienarbeit [1], in welcher die Hardware, welche die Grundlage für dieses Projekt bildet, entwickelt, bestückt und in Betrieb genommen wurde.

Die entwickelte Platine verfügt über zwei  $\mu$ C, deren UART-Schnittstellen miteinander verbunden sind. Ein Prozessor ist hierbei für die Verarbeitung von Sensormesswerten zuständig, während der andere als WLAN-Relais an ein übergeordnetes Netzwerk fungiert.

Auf Basis der UART-Schnittstellen der beiden Prozessoren soll ein Protokoll implementiert werden, welches sowohl den Transport von Messwerten in die eine Richtung, als auch den Transport von Konfigurationsbefehlen in die andere Richtung ermöglicht.

Als Praxisbeispiel wurde die kapazitive Messung von Feuchtigkeit im Erdreich gewählt. Die gemessenen Daten sollen verarbeitet und per Message Queueing Telemetry Transport (MQTT) an ein Netzwerk weitergeleitet werden.

Der gesamte Code ist in einer GitHub-Repository zu finden [2].

---

## **Abkürzungsverzeichnis**

**HAL** Hardware Abstraction Layer

**IoT** Internet of Things

**µC** Microcontroller

**WLAN** Wireless Local Area Network

**RAM** Random Access Memory

**ADC** Analog Digital Converter

**UART** Universal Asynchronous Receiver Transmitter

**USART** Universal Synchronous Receiver Transmitter

**MQTT** Message Queuing Telemetry Transport

**SPI** Serial Peripheral Interface

**I2C** Inter-Integrated Circuit

**DMA** Direct Memory Access

**AHB-Bus** Advanced High-Performance Bus

**GPIO** General Purpose Input Output

**MSB** Most Significant Bit

**LSB** Least Significant Bit

**FIFO** First In First Out

**ASCII** American Standard Code for Information Interchange

**USB** Universal Serial Bus

**CAN** Controller Area Network

---

## Inhaltsverzeichnis

<b>Vorwort</b>	<b>3</b>
<b>Selbstständigkeitserklärung</b>	<b>4</b>
<b>Zusammenfassung</b>	<b>5</b>
<b>1 Einführung</b>	<b>9</b>
1.1 Vorhergegangene Arbeit . . . . .	9
1.2 Aufbau der Arbeit . . . . .	9
<b>2 Grundlagen</b>	<b>10</b>
2.1 Microcontroller . . . . .	10
2.1.1 ESP8266 . . . . .	10
2.1.2 STM32F103 . . . . .	10
2.2 Peripherie . . . . .	11
2.2.1 UART/USART . . . . .	11
2.2.2 DMA . . . . .	11
2.2.3 ADC . . . . .	12
2.2.4 GPIO . . . . .	13
2.2.5 Timer . . . . .	14
2.3 Protokolle . . . . .	15
2.3.1 UART . . . . .	15
2.3.2 MQTT . . . . .	17
2.4 Datenstrukturen und Algorithmen . . . . .	18
2.4.1 Ringbuffer . . . . .	18
2.4.2 Cyclic Redundancy Check . . . . .	18
2.5 STM32 Hardware Abstraction Layer und CubeMX . . . . .	18
<b>3 Umsetzung</b>	<b>20</b>
3.1 Protokoll zur Übertragung von Daten . . . . .	20
3.1.1 CRC16-CCITT . . . . .	20
3.1.2 Datenformat . . . . .	22
3.1.3 Nachrichtentypen . . . . .	22
3.2 Übertragung von Daten . . . . .	24
3.2.1 STM32: Strukturvariable DMA_STRUCT . . . . .	24
3.2.2 STM32: Konfiguration des UART . . . . .	24
3.2.3 STM32: Idle Line Detection . . . . .	25
3.2.4 STM32: Empfangsinterrupt / DMA Circular Buffer . . . . .	26
3.2.5 STM32: Überprüfen der Daten . . . . .	28
3.2.6 STM32: Verarbeitung der empfangenen Daten . . . . .	29
3.2.7 STM32: Erstellen von Telegrammen . . . . .	29
3.2.8 STM32: Versenden von Daten . . . . .	30

---

3.2.9	ESP8266: Empfang und Überprüfung der Daten . . . . .	31
3.2.10	ESP8266: Anmelden an WLAN und MQTT-Server . . . . .	34
3.2.11	ESP8266: Empfang und Versand per MQTT . . . . .	36
3.3	Erfassung und Verarbeitung von Messwerten . . . . .	39
3.3.1	Kapazitive Feuchtigkeitsmessung . . . . .	39
3.3.2	Analog-Digital Converter . . . . .	39
3.3.3	Strukturvariable SENS_STRUCT . . . . .	40
3.3.4	Kalibrierung des Sensors . . . . .	41
3.3.5	Berechnung des Messwertes . . . . .	44
3.3.6	Periodisches Versenden der Messwerte . . . . .	45
3.4	Inbetriebnahme MQTT-Server auf Raspberry Pi . . . . .	47
<b>4</b>	<b>Auswertung</b>	<b>49</b>
4.1	Fazit . . . . .	49
4.2	Fehler . . . . .	49
4.3	Ausblick . . . . .	49
<b>Abbildungsverzeichnis</b>		<b>50</b>
<b>Listings</b>		<b>51</b>
<b>Quellenverzeichnis</b>		<b>53</b>

---

# 1 Einführung

## 1.1 Vorhergegangene Arbeit

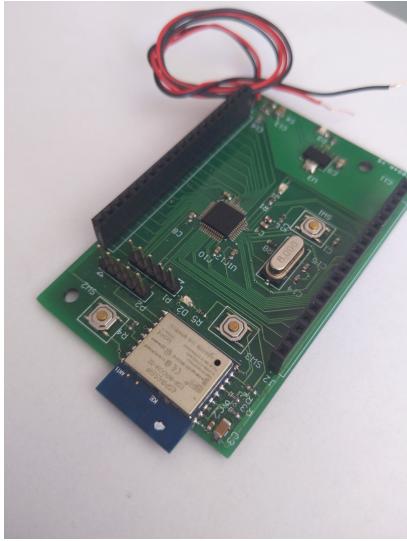


Abbildung 1: *IoT-Gateway [1]*

In der vorhergegangenen Arbeit (*Entwicklung eines IoT-Gateways [1]*) wurde eine Platine, zu sehen in Abb. 1, entwickelt, welche die Grundlage der vorliegenden Arbeit bildet.

Die Platine verfügt über zwei  $\mu$ C. Einer der Prozessoren, genauer ein *STM32F103C8*, ist für das Auslesen von Messwerten und die Verarbeitung dieser zuständig, während der andere Prozessor, ein *ESP8266*, die Netzwerkanbindung per WLAN ermöglicht.

Abgesehen von den beiden Prozessoren verfügt das Board über einen linearen Spannungsregler, um 3.3V für die  $\mu$ Cs bereitzustellen. Des Weiteren wurden die nötigen externen Schaltungen für die beiden Prozessoren entwickelt. Beide Prozessoren verfügen über Status-LEDs. Die GPIO des STM32 werden über Buchsenleisten herausgeführt, um den einfachen Anschluss von Erweiterungsplatinen zu ermöglichen.

Die Programmierung des STM32 erfolgt mit Hilfe eines proprietären Programmiergeräts (*ST-Link V2*), während der ESP8266 mit Hilfe eines UART-USB Interfaces programmiert wird.

## 1.2 Aufbau der Arbeit

Diese Arbeit ist in mehrere Abschnitte geteilt:

- Erklärung der Grundlagen (Kapitel 2)
- Implementation der Funktionen in Software (Kapitel 3)
- Auswertung und Fazit sowie Aussicht (Kapitel 4)

Zu Beginn sollen die Grundlagen erklärt werden, auf denen diese Arbeit aufbaut. Dies umfasst die Eigenschaften der  $\mu$ C, die genutzten Protokolle, Datenstrukturen und Algorithmen.

Anschließend werden die zuvor beschriebenen Grundlagen an Hand eines Praxisbeispiels (Feuchtigkeitsmessung von Erdreich) angewendet.

Schließlich wird die Arbeit ausgewertet, ein Fazit über die Entwicklungen gezogen und eine Aussicht präsentiert, wie das Projekt weiterentwickelt oder verbessert werden kann.

---

## 2 Grundlagen

### 2.1 Microcontroller

#### 2.1.1 ESP8266

Der ESP8266 ist ein 32-Bit µC mit einem Systemtakt von 80 - 160MHz. Er verfügt über 64kB RAM, welcher als Arbeitsspeicher genutzt wird, sowie über 96kB RAM, welcher als Datenspeicher genutzt wird. Während des Bootvorgangs wird die Firmware aus einem externen Flashspeicher geladen. Der µC verfügt über alle gängigen Peripherieeinheiten (ADC,UART,SPI,I2C) sowie über eine WLAN-Schnittstelle, welche mit dem Standard 802.11 b/g/n arbeitet und im 2.4-2,5GHz Band kommuniziert [3].

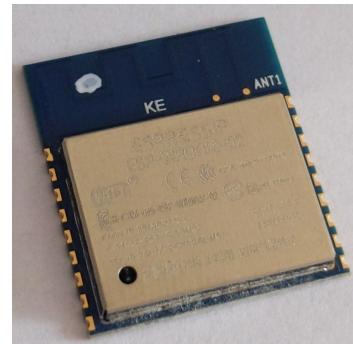


Abbildung 2: *ESP8266 WROOM-02*

In diesem Anwendungsfall wird der ESP8266 als Modul eingesetzt (ESP8266-WROOM-02), da diese über die nötige externe Beschaltung (Oszillator, Flashspeicher, Antenne) verfügt [3].

Die Programmierung erfolgt über UART mittels eines USB-Adapters.

#### 2.1.2 STM32F103

Der STM32F103 µC basiert auf der Cortex M3 Architektur von ARM und arbeitet mit einem Systemtakt von bis zu 72MHz. Die hier eingesetzte Version STM32F103C8 verfügt über 64kB Flashspeicher sowie 20kB RAM [4].

An den 37 Ein- und Ausgängen des µCs sind diverse Kommunikationsschnittstellen verfügbar (CAN,I2C,SPI,USART,USB). Des Weiteren verfügt der STM32F103 über einen 10-kanaligen 12-Bit ADC, diverse Timer mit verschiedenem Funktionsumfang sowie über einen DMA-Controller [4].

Für die Programmierung des µC ist ein ST-Link Programmiergerät notwendig, welches auch Debugging ermöglicht [4].

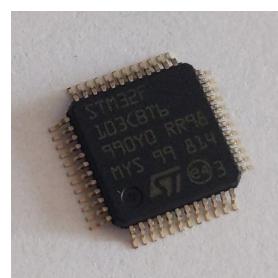


Abbildung 3: *STM32F103C8*

---

## 2.2 Peripherie

Im folgenden werden die verschiedenen benutzten Peripherien erläutert. Diese beziehen sich jedoch primär auf die Funktionen des STM32, da diese dort intensiver Nutzung unterliegen, während die einzige genutzten Funktionalitäten des ESP8266 seine WLAN- und UART-Schnittstelle sind, welche durch das Arduino-Framework verschleiert werden.

### 2.2.1 UART/USART

Der STM32F103 verfügt über drei Universal Synchronous Receiver Transmitter (USART)-Schnittstellen, welche jedoch auch als Universal Asynchronous Receiver Transmitter (UART) genutzt werden können [4]. Die dabei genutzten Spannungsspegel entsprechen hierbei die der TTL-Logik [4].

Die Einheiten verfügen unter anderem auch über Idle-Line Detection (*Erkennung von Kommunikationsstops*), Duplex und Hardware Flow Control [5].

### 2.2.2 DMA

Der Direct Memory Access Controller ist eine dedizierte Hardwareeinheit, welche das direkte Schreiben von Daten in den Speicher des µC erlaubt, ohne dass dafür Instruktionen durch den Prozessor ausgeführt werden müssen. Die unterstützten Peripherien sind Timer, ADC, SPI, I2C und UART [4].

Daten können von Speicherort zu Speicherort, von einer Peripherie zu einem Speicherort oder von einem Speicherort zu einer Peripherie transferiert werden. Der STM32F103 verfügt über sieben Direct Memory Access (DMA)-Kanäle.

Durch die Nutzung von DMA wird der Prozessor entlastet, da er dann nicht mit der Übertragung von Daten blockiert wird. Die Daten werden über eine interne Busmatrix direkt übertragen [6].

Abbildung 4 zeigt den Aufbau eines einzelnen Controllers. Der Nutzer kann den verschiedenen Kanälen, die jeweils mit einer Peripherieeinheit verknüpft werden können, Prioritäten zuweisen. Die Priorität der DMA-Controller werden durch den Arbiter verwaltet. Der DMA ist über den Advanced High-Performance Bus (AHB-Bus), mit den Peripherieeinheiten und dem Speicher des µC verknüpft [6].

Der DMA-Controller verfügt über einen Slave-Port. Mittels dieses Anschlusses lässt sich der DMA-Controller konfigurieren [6].

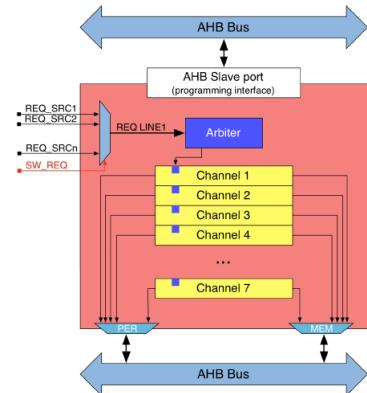


Abbildung 4: DMA [6]

### 2.2.3 ADC

Ein ADC ermöglicht die Konversion von analogen Signalen in digitale Werte, um diese anschließend durch den µC weiterzuverarbeiten. Der Analog Digital Converter (ADC) des STM32F103 hat eine Auflösung von 12-Bit bei bis zu 16 Kanälen und arbeitet nach dem Prinzip der sukzessiven Approximation [4].

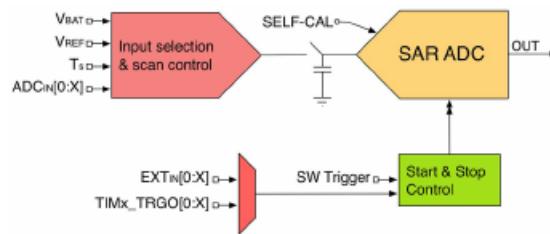


Abbildung 5: Aufbau [6]

Es ist möglich, den ADC automatisch zu kalibrieren und ihn entweder per Software-Trigger, Timer oder externen Interrupt zu starten. In Abbildung 5 ist der vereinfachte Aufbau des ADC abgebildet.

Der ADC verfügt über verschiedene Betriebsmodi [5]:

- Single Mode
- Continuous Mode
- Discontinuous Mode

Im Single Mode wird eine Konversion durchgeführt, während im Continuous Mode ständig weitere Konversionen durchgeführt werden. Im Discontinuous Mode wird die nächste Konversion durchgeführt, sobald ein benutzerdefinierter Trigger ausgelöst wird. Es ist im (Dis-)Continuous Mode möglich, bei jeder Konversion einen anderen Kanal des ADC anzusprechen.

## 2.2.4 GPIO

Der STM32F103 besitzt diverse GPIO. Mit Hilfe dieser Ein- und Ausgänge können Signale ein- oder ausgegeben werden. Es ist möglich, den Anschlüssen interne Pull-Up oder Pull-Down Widerstände zuzuweisen [4]. Ausgänge können entweder als Push-Pull oder Open-Drain konfiguriert werden [5].

Die Eingänge können je nach anliegendem Signal, Interrupts auslösen [5]:

- Steigende Flanke
- Fallende Flanke
- Steigende oder Fallende Flanke

Die interne Beschaltung der General Purpose Input Output (GPIO) ist Abb. 6 zu entnehmen.

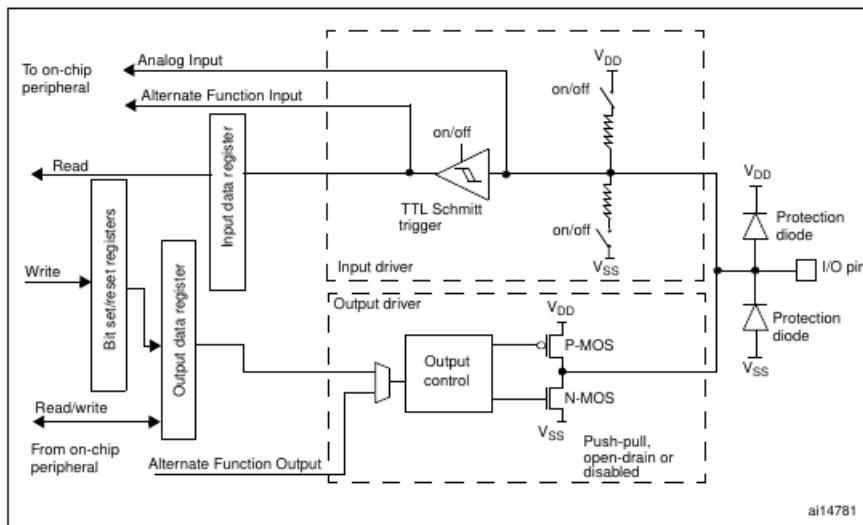


Abbildung 6: *Beschaltung /5/*

---

## 2.2.5 Timer

Die Timer des STM32F103 teilen sich, wie in Abb. 7 zu sehen, in zwei Gruppen auf:

Timer	Counter resolution	Counter type	Prescaler factor	DMA request generation	Capture/compare channels	Complementary outputs
TIM1	16-bit	Up, down, up/down	Any integer between 1 and 65536	Yes	4	Yes
TIM2, TIM3, TIM4	16-bit	Up, down, up/down	Any integer between 1 and 65536	Yes	4	No

Abbildung 7: *Timer [4]*

TIM1 ist ein Advanced-Control Timer, während TIM2, TIM3 und TIM4 General-Purpose Timer sind.

Advanced-Control Timer implementieren erweiterte Funktionen, wie z.B. dreiphasige Pulsweitenmodulation oder programmierbare Totzeiten[4].

Abgesehen von den bereits vorgestellten Timern stehen zwei Watchdog-Timer sowie ein SysTick-Timer zur Verfügung. Der SysTick-Timer wird einerseits genutzt, um ein Real-Time Operating System auf dem STM32F103 zu realisieren und andererseits, um dem Hardware Abstraction Layer des STM32F103 eine Zeitkonstante zu geben. Er kann auch als simpler Timer benutzt werden, da er jede ms aktualisiert wird.

Watchdog-Timer werden genutzt, um abnormale Systemzustände zu erkennen. Ein Beispiel hierfür ist das Festhängen in spezifischen Codeabschnitten.

---

## 2.3 Protokolle

### 2.3.1 UART

Eine wichtige Grundlage ist die Parität. Das Paritätsbit dient als Ergänzung einer Folge von Bits. Durch das Ergänzen und das entsprechende Setzen des Bits, wird sichergestellt, dass die Anzahl der Bits gerade ist.

Die Nutzung eines Paritätsbits ist die einfachste aller Möglichkeiten, Übertragungsfehler zu erkennen. Kippt während der Übertragung eines der zu übertragenen Bits, ist die Zahl der Bits nicht mehr gerade - ein Fehler liegt vor.

Es ist allerdings nicht möglich, festzustellen, wo genau der Fehler aufgetreten ist [8].

Abb. 8 zeigt die Applikation eines Paritätsbits bei der binären Repräsentation der Dezimalzahlen eins bis acht. Sobald die Anzahl der gesetzten Bits ungerade ist, wird ein Paritätsbit (E) hinzugefügt.

Eine weitere Grundlage stellt die Baudrate dar. Die Baudrate, auch Symbolrate genannt, beschreibt die Geschwindigkeit, mit der Zeichen übertragen werden.

Ein Baud entspricht hierbei ein Zeichen pro Sekunde [8].

Beispiele für gängige, standarisierte Baudraten für die Übertragung per UART sind:

- 4800 Baud
- 9600 Baud
- 115200 Baud

Diese Baudraten werden von den meisten Computer- oder Prozessorsystemen unterstützt [8].

Wertigkeit	8	4	2	1	E
Dezimalziffern	0				
1					
2					
3					
4					
5					
6					
7					
8					
9					

Abbildung 8: Parität [7]

---

Auf Basis dieser Grundlagen bildet der Universal Asynchronous Receiver Transmitter eine elektronische Schaltung, oder im Falle eines µC, eine Peripherieeinheit, welche die Datenübertragung mit einem anderen System ermöglicht. Die Übertragung läuft hierbei asynchron, d.h. ohne ein Taktsignal, welches ebenfalls übertragen wird, ab [8]. Die Übertragungsgeschwindigkeit wird in Baud angegeben.

Jede Seite der Übertragung verfügt über zwei Anschlüsse, **RX** und **TX**. **RX** steht hier für Receiver (*deutsch Empfänger*), während **TX** für Transmitter (*deutsch Sender*) steht. Für eine korrekte Funktion müssen die beiden Anschlüsse über Kreuz, wie in Abb. 9 dargestellt, verbunden werden [10].

Es existieren verschiedene Implementationen von UART, welche sich in der Nachrichtenlänge und der Art der Parität unterscheiden (gerade oder ungerade Parität). Ein oft genutztes Format ist das **8N1**-Format.

Die Abkürzung steht hierbei für 8 Bits Nachrichtenlänge und *keine* Parität und ein Stop-Bit. Es wird jedoch neben dem *Stop*-Bit immer noch ein weiteres Bit übertragen, das *Start*-Bit. Diese zwei Bits repräsentieren das Framing (*deutsch Einrahmen*, siehe Abb. 10) und signalisieren den Beginn und das Ende der Nachricht. Der Beginn einer Nachricht wird mit einem Wechsel von '1' auf '0' signalisiert, während das Ende einer Nachricht mit einem Wechsel von '0' auf '1' signalisiert wird [10].

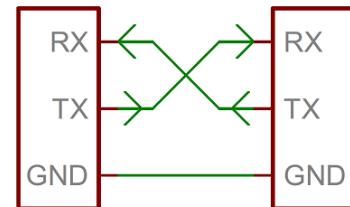


Abbildung 9: *Verbindung* [9]



Abbildung 10: *Framing* [9]

Sollen nun z.B. die ASCII-Zeichen 'O' und 'K' im 8N1-Format übertragen werden, sähe die Bitreihenfolge wie in Abb. 11 dargestellt aus. Es ist dabei zu beachten, dass das LSB zuerst übertragen wird.



Abbildung 11: *Nachricht 'OK'* [9]

Die Bitfolge '01001111' entspricht dem Buchstaben 'O', die Bitfolge '01001011' die dem Buchstaben 'K'.

---

### 2.3.2 MQTT

MQTT ist ein simples Client-Server-Protokoll zur Nachrichtenübertragung, häufig genutzt in Internet of Things (IoT)-Anwendungen oder im industriellen Umfeld. Durch die geringe Belegung von Bandbreite eignet es sich in Netzwerken mit hoher Latenz, bei schlechten Verbindungen und zur Nutzung auf Geräten mit vergleichsweise geringer Rechenleistung [11].

Geräte, welche Daten empfangen oder versenden (s.g. Clients), sind mit einem zentralen Server (s.g. Broker) verbunden. Nach erfolgreicher Verbindung der Clients mit dem Broker versenden diese ihre Nachrichten mit einem Topic, welches die versendete Nachricht hierarchisch einstuft. Ein Beispiel wäre zum Beispiel `Hochschule/B/Temp`. Sendet nun ein Client eine Nachricht mit diesem Topic, wird diese Nachricht an alle Clients weitergeleitet, welche dieses Topic abonniert haben. Diese Art der Kommunikation wird Publish/Subscribe-Modell genannt [12] und in Abb. 12 dargestellt.

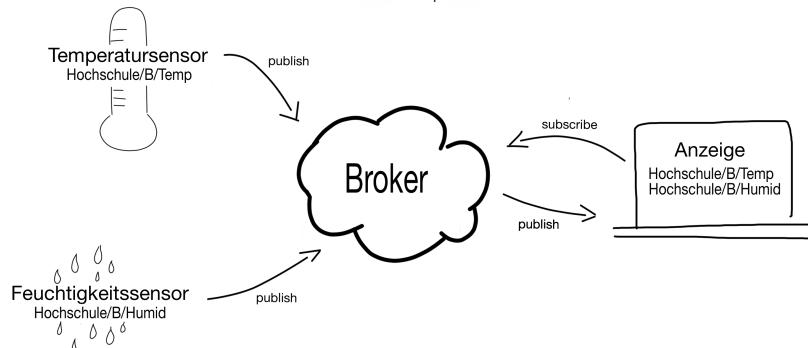


Abbildung 12: *Publish/Subscribe-Modell*

MQTT besitzt diverse Features, welche, abgesehen von der Simplizität, attraktiv für den Einsatz im industriellen Umfeld sind [12]:

- Durch den **Quality of Service Level** kann sichergestellt werden, dass eine Nachricht beim Empfänger ankommt, auch wenn es zu Verbindungsabbrüchen kommt.
- Die letzte gesendete Nachricht eines Topics wird im Broker hinterlegt. Verbindet sich ein neuer Client mit diesem Topic, erhält er automatisch die gespeicherte Nachricht (**Retained Messages**).
- Sobald der Broker feststellt, dass ein Client die Verbindung ohne eine Abmeldung verloren hat, wird eine Nachricht an alle Clients, welche das selbe Topic abonniert haben, versendet (**Last Will and Testament**).
- Falls mit häufigen Verbindungsabbrüchen zu rechnen ist, kann der Broker alle durch einen Verbindungsabbruch verpasste Nachrichten automatisch erneut an den Client versenden (**Persistent Sessions**).

---

## 2.4 Datenstrukturen und Algorithmen

### 2.4.1 Ringbuffer

Der Ringbuffer ist eine Datenstruktur, die nach dem First In First Out (FIFO)-Prinzip arbeitet. Dies bedeutet, dass die Daten, welche zuerst in den Buffer geschrieben wurden, auch zuerst wieder ausgelesen werden.

Die zu schreibenden Daten werden in ein Array

von bestimmter Länge N geschrieben. Läuft das Array voll, werden die neuen Zeichen wieder an den Anfang geschrieben. Zur Orientierung werden zwei Zähler eingeführt, der Lese- und Schreibindex. Der Leseindex zeigt die aktuelle Position im Array an. Der Schreibindex zeigt, bis an welche Stelle neue Daten geschrieben wurden.

Haben Lese- und Schreibindex den selben Wert, gilt der Buffer als leer.

Wird ein Zeichen in den Ringbuffer geschrieben, wird der Schreibindex inkrementiert. Sobald ein Zeichen gelesen wird, wird der Leseindex inkrementiert. Erreichen die beiden Indizes das Ende des Arrays, werden sie auf 0 zurückgesetzt.

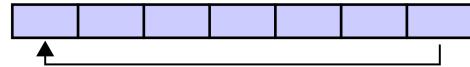


Abbildung 13: *Ringbuffer* [13]

### 2.4.2 Cyclic Redundancy Check

Cyclic Redundancy Check (*deutsch Zyklische Redundanzprüfung*), kurz **CRC**, ist eine Methode zur Erkennung von Fehlern bei der Übertragung. Es ist nur möglich, zufällige Fehler zu erkennen, wie sie z.B. durch Übertragungsfehler oder Rauschen auf der Leitung entstehen [14].

Den zu übertragenden Daten wird ein zuvor berechneter Wert angehängt. Mittels dieses Wertes kann die Empfängerseite feststellen, ob die Daten korrekt übertragen wurden oder ob ein Fehler vorliegt.

CRC nutzt zur Überprüfung der Daten die Polynomdivision. Die zu übertragenden Daten werden als Polynom dargestellt.

Die Bitfolge 10101010 entspricht dem Polynom  $1 * x^7 + 0 * x^6 + 1 * x^5 + 0 * x^4 + 1 * x^3 + 0 * x^2 + 1 * x + 0$ . Das Polynom der Bitfolge wird durch ein zuvor festgelegtes CRC-Polynom geteilt. Der Rest dieser mathematischen Operation repräsentiert den CRC-Wert. Dieser Wert wird anschließend bei der Datenübertragung an die zu übertragende Nachricht angehängt.

Empfängerseitig wird nun abermals eine Polynomdivision durchgeführt. Ist das Ergebnis der Polynomdivision empfangene Nachricht inkl. CRC-Wert dividiert durch das CRC-Polynom gleich 0, wurde die Nachricht korrekt übertragen [14].

## 2.5 STM32 Hardware Abstraction Layer und CubeMX

Der Hardware Abstraction Layer (HAL) ist eine Schnittstelle zwischen der untersten Firmwareschicht des STM32 und der Software, welche der Nutzer schreibt. Durch die

Benutzung des HALs müssen keine einzelnen Register mittels Assembler gesetzt werden. Die Konfiguration wird dadurch stark vereinfacht und die Fehleranfälligkeit verringert. Die dazu notwendigen Libraries werden vom Hersteller (ST) bereitgestellt und dokumentiert [15].

Funktionen, welche durch den HAL bereitgestellt werden, sind zu erkennen an einem vorgestellten *HAL*. Folgende Funktion schaltet z.B. einen GPIO um:

Listing 1: *Umschalten von GPIO*

```
void HAL_GPIO_TogglePin(GPIOx, GPIO_Pin);
```

Neben dem HAL stellt der Hersteller ein weiteres Tool zur Verfügung (Cube MX [16]), welches die schnelle Erstellung des Initialisierungscodes für die Peripherie und sonstige Funktionen ermöglicht. Dies verringert ebenfalls die Fehleranfälligkeit und vereinfacht die Programmierung.

Mittels einer graphischen Oberfläche können den verschiedenen Anschlüssen des µC Funktionen zugewiesen (z.B. I2C) und Taktfrequenzen konfiguriert werden.

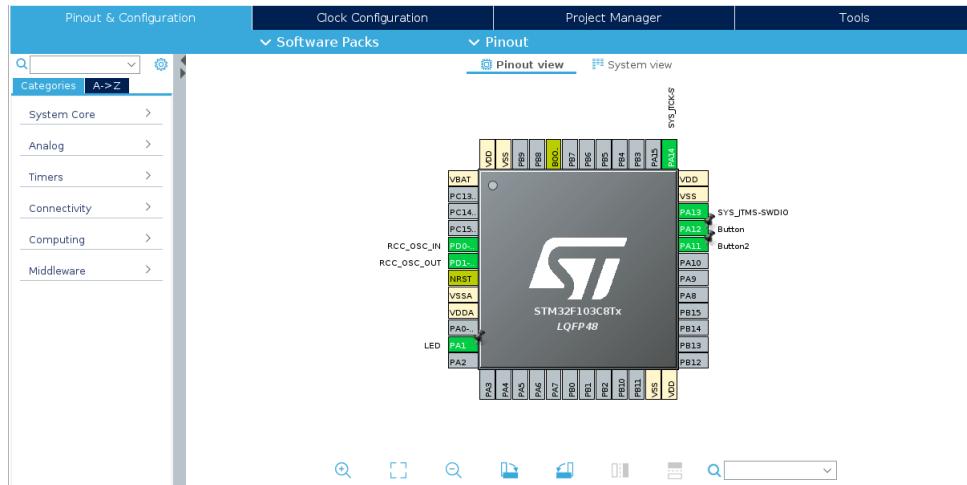


Abbildung 14: *CubeMX*

---

## 3 Umsetzung

### 3.1 Protokoll zur Übertragung von Daten

Im Folgenden soll die Entwicklung des Protokolls zur Übertragung von Daten erklärt werden. Zuerst wird auf den Algorithmus zur zyklischen Redundanzprüfung eingegangen, anschließend auf den Aufbau der zu übertragenden Telegramme.

#### 3.1.1 CRC16-CCITT

Als Implementation der zyklischen Redundanzprüfung wurde die Version *CRC16-CCITT* gewählt [17].

Das CRC-Polynom lautet  $x^{16} + x^{15} + x^2 + 1$ . Die hexadezimale Repräsentation ergibt sich deshalb zu  $0x1021$ .

Als Startwert für die Berechnung wird  $0x0$  gewählt - fehlerhafte Implementierungen beginnen oftmal mit  $0xFFFF$ .

a	b	y
0	0	0
0	1	1
1	0	1
1	1	0

Tabelle 1: XOR

Es handelt sich bei der Berechnung des CRC-Wertes um eine Polynomdivision. Diese wird mit Hilfe eines Exklusiv-Oder Gatters (*XOR*) und Schieberegistern realisiert. Den zu prüfenden Daten werden abhängig von der Länge des CRC-Polynoms Nullen angehängt. Bei CRC16 entspricht dies 16 Nullen.

Um die Daten zu prüfen, wird das CRC-Schieberegister mit '0' initialisiert. Anschließend wird der zu prüfende Wert, mit angehängten Nullen, von rechts in das Schieberegister geschoben, bis das Most Significant Bit (MSB) gleich '1' ist. Anschließend wird das Schieberegister um eine Einheit weitergeschoben, sodass das MSB herausfällt. Dann wird das Schieberegister mit Hilfe des XOR-Vergleiches mit dem CRC-Polynom verglichen. Der so entstandene Wert wird wieder in das Schieberegister übernommen [14].

Nun werden immer wieder Daten von rechts in das Register hineingeschoben. Immer wenn das MSB einer '1' entspricht und im nächsten Schritt aus dem Register geschoben wird, wird ein XOR-Vergleich mit dem CRC-Polynom durchgeführt. Dies wird so lange wiederholt, bis keine neuen Daten mehr in das Register geschoben werden können. Der Wert, welcher nun im Schieberegister verbleibt, entspricht der Prüfsumme [14].

Die Implementation in C bedient sich zweier Kniffe, um die zuvor erklärte Berechnung zu beschleunigen. Auf ein Initialisieren mit  $0x0000$  kann verzichtet werden, da keine XOR-Vergleiche mit Nullen durchgeführt werden. Der Startwert kann also direkt mit den Eingangsdaten initialisiert werden. Ein Anhängen von Nullen ist ebenfalls nicht erforderlich, da der «-Operator (*left shift*) automatisch am Least Significant Bit (LSB) Nullen anhängt.

---

Umgesetzt in C Code entsteht folgende Funktion:

Listing 2: Berechnung CRC16

```
uint16_t CRC16_buf(const uint8_t * pBuf, uint16_t len)
{
    const uint16_t poly = 0x1021;
    uint16_t crc = 0;

    for (uint8_t i = 0; i < len; i++)
    {
        crc ^= pBuf[i] << 8;

        for(uint8_t j = 0; j < 8; j++)
        {
            if((crc & 0x8000) != 0)
            {
                crc = (crc<<1) ^ poly;
            }
            else
                crc <<= 1;
        }
    }
    return crc;
}
```

Der Funktion wird ein Zeiger auf ein Array übergeben, sowie die Länge des Arrays. Die Variable `poly` repräsentiert das CRC-Polynom, während die Variable `crc` für das Schieberegister steht.

Die Berechnung des CRC-Wertes wird für jedes Byte des Arrays durchgeführt, der Startwert für jedes Byte nach dem ersten Byte ist die CRC-Prüfsumme des letzten Durchlaufes. Für jedes Byte wiederum müssen, auf Grund der Länge eines Bytes, acht mal die Shift- und XOR-Operationen durchgeführt werden.

Mit dem Wert `0x8000` wird geprüft, ob das MSB gesetzt ist. Der Rückgabewert entspricht der CRC-Prüfsumme.

---

### 3.1.2 Datenformat

Bei den übertragenen Daten handelt es sich um Zeichen im ASCII-Format. Dies vereinfacht die Auswertung und hat den Vorteil, dass die Kommunikation zu Testzwecken leicht mitgelesen werden kann. Im folgenden wird unter *Telegramm* die Gesamtheit der übertragenen Daten verstanden, während der Ausdruck *Nachricht* den eigentlichen Informationsinhalt beschreibt.

Ein Telegramm besteht aus: Startzeichen, Länge, Nachricht, CRC-Prüfsumme.

Die einzelnen Teile bestehen aus einer verschiedenen Anzahl an Bytes. Während Start- und Endzeichen nur ein Byte benötigen, werden für die Länge der Nachricht und die CRC-Prüfsumme zwei Bytes benötigt.

Da die Länge der Nachricht aus zwei Bytes besteht, ergibt sich eine maximale Nachrichtenlänge von 99 Bytes. Auf ein Endzeichen wird verzichtet - das Ende des Telegramms berechnet sich aus der Länge der Nachricht. Das Startzeichen entspricht dem ASCII-Zeichen '<'.

1	2	3	4	n+4	n+5	n+6
Start	Länge	Nachricht	CRC16			

Tabelle 2: Telegramm

### 3.1.3 Nachrichtentypen

Basierend auf dem in 3.1.2 beschriebenen Format werden nun verschiedene Nachrichtentypen implementiert, welche die Steuerung der Platine ermöglichen oder der Information dienen. Dabei ist zu unterscheiden, ob es sich um Nachrichten handelt, welche an das Board geschickt werden, oder um Nachrichten, welche vom Board versendet werden. Die Nachrichten werden dabei durch den STM32 verarbeitet, der ESP8266 leitet sie weiter.

Um die Nachrichten zu differenzieren, wird bei Nachrichten, welche an das Board gesendet werden, die Nachricht durch eine ASCII-Zahl kodiert, bei Nachrichten, welche das Board sendet, durch einen ASCII-Buchstaben.

Nachrichten in Senderichtung:

- STATUS (1) - Abfrage der Versionsnummer, genutzter Sensor
- CALIBRATE (2) - Kalibrieren des Sensors
- SENDVAL (3) - Start der automatischen Versendung von Messwerten

Der Nachricht SENDVAL wird eine '1' oder eine '0' angehängt, um die Funktion ein- oder auszuschalten.

---

Soll z.B. der Status des Boards abgefragt werden, wird die Nachricht STATUS versendet. Diese Nachricht ist genau 1 Byte lang, deshalb werden als Längeninformationen die American Standard Code for Information Interchange (ASCII)-Zahlen '0' und '1' übertragen. Danach folgt die eigentliche Nachricht, welche ebenfalls durch eine '1' repräsentiert wird. Abgeschlossen wird das Telegramm durch die CRC-Prüfsumme. Das Telegramm ergibt sich somit wie folgt:

Zeichen	1	2	3	4	5	6
ASCII	<	0	1	1	CRC16	
Hex	0x3C	0x30	0x31	0x31	0xB6	0xA8

Tabelle 3: SENDVAL

Nachrichten in Empfangsrichtung:

- BOARD (B) - Antwort auf STATUS, enthält zusätzlich Versionsnummer und Sensorsortyp
- ANVALUE (A) - Zeigt einen Messwert an
- ERROR (E) - Zeigt einen Fehler an, gefolgt vom Errortyp
- ACK (A) - Acknowledge, automatische Antwort auf empfangene Nachrichten

Die Nachricht ACK gibt den empfangenen Nachrichtentyp zurück, um sicherzugehen, dass die gesendete Nachricht korrekt empfangen wurde. Dem Nachrichtentyp ERROR folgt immer der Typ des aufgetretenen Fehlers, repräsentiert durch eine Zahl:

- ER\_UNSPEC (1) - unspezifizierter Fehler
- ER\_CAL (2) - Fehler bei der Kalibrierung des Sensors
- ER\_NOT\_CAL (3) - Sensor nicht kalibriert
- ER\_UN\_MSG (4) - Die empfangene Nachricht kann nicht interpretiert werden

---

## 3.2 Übertragung von Daten

Die Daten müssen auf beiden µC ordnungsgemäß empfangen werden. Die Implementation unterscheidet sich hierbei stark. Der STM32 ermöglicht die Nutzung seiner DMA-Funktionalität, während auf dem ESP8266 eine interruptbasierte Abfrage implementiert wird.

### 3.2.1 STM32: Strukturvariable DMA\_STRUCT

Aus Gründen der Übersichtlichkeit wurde für die Datenverarbeitung des STM32 eine Strukturvariable definiert, welche hiermit eingeführt wird. Die Strukturvariable DMA\_STRUCT enthält mehrere weitere Variablen für Flags und Zähler.

Listing 3: DMA Strukturvariable

```
typedef struct
{
    volatile uint8_t t_flag;
    uint8_t tx_flag;
    volatile uint8_t av_flag;
    uint16_t timer;
    uint16_t prevCOUNT;
    uint8_t data[DMA_BUF_SIZE];
} DMA_STRUCT;
```

Die Flags `t_flag` und `tx_flag` dienen der Identifikation von Interrupts. Die Interrupts werden im Falle von `t_flag` durch ein Timeout ausgelöst und im Falle von `tx_flag` durch das erfolgreiche Senden von Daten. Die Variable `timer` legt die Zeitkonstante für den Timeout fest, während `prevCOUNT` einen Zähler speichert, der in 3.2.4 genauer erklärt wird. Das Flag `av_flag` zeigt an, wenn neue Daten zur Verarbeitung verfügbar sind. Im Array `data[]` werden die neuen Daten gespeichert.

### 3.2.2 STM32: Konfiguration des UART

Wichtig bei der Konfiguration des UART, ist das Format und die Baudrate. Wie in Kapitel 2 erläutert, wird der UART im 8N1-Modus konfiguriert. Dies entspricht einer Nachrichtenlänge von acht Bit und keiner Parität. Die Geschwindigkeit wird auf 115200 Baud festgelegt (siehe Abb. 15a).

Um die Nutzung des UART in Kombination mit DMA zu ermöglichen, muss der globale Interrupt aktiviert werden. Der DMA wird so konfiguriert, dass empfangsseitig die Daten direkt zum Speicher übertragen werden. Senderseitig werden die Daten direkt vom Speicher an den UART weitergeleitet. Der Empfang von Daten per DMA wird durch einen Ringbuffer umgesetzt (siehe Abb. 15b).

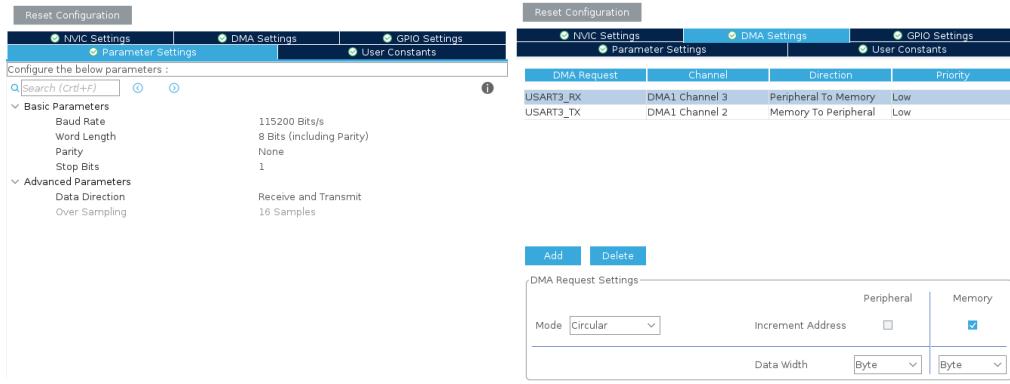


Abbildung 15: Konfiguration des UARts

### 3.2.3 STM32: Idle Line Detection

Eine Idle Line Detection, welche erkennt, wenn keine Daten mehr empfangen werden, wird implementiert. Dies vermeidet das Schreiben von sinnlosen Daten in den Buffer, wenn keine Daten mehr empfangen werden.

Der STM32 bietet dafür einen Interrupt, welcher manuell aktiviert werden muss [5]:

Listing 4: Aktivierung Idle Line Interrupt

```
--HAL_UART_ENABLE_IT(&huart3, UART_IT_IDLE);
```

Zudem muss in der Datei `stm32f1xx_it.c` die Funktion `void USARTX_IRQHandler(void)` erweitert werden:

Listing 5: Idle Line Interrupt Clear

```
if (_HAL_UART_GET_FLAG(&huartx, UART_FLAG_IDLE))
{
    _HAL_UART_CLEAR_IDLEFLAG(&huartx);
    dma_info.timer = DMA_TIMEOUT_MS;
}
```

Immer, wenn ein Interrupt in Zusammenhang mit dem UART ausgelöst wird, wird die Routine `void USARTX_IRQHandler(void)` aufgerufen und geprüft, ob es sich um ein Idle Line Interrupt handelt [5].

Es ist allerdings nicht ausreichend, nur zu prüfen, ob der Interrupt aufgetreten ist, es kann durchaus vorkommen, dass es sich nur um eine kurze Unterbrechung in der Kommunikation handelt. Deshalb wird die Funktionalität um einen Timeout erweitert. Wenn der Interrupt auftritt, wird gleichzeitig die Strukturvariable `dma_info.timer` mit dem zuvor definierten Zeitwert `DMA_TIMEOUT_MS` geladen.

Um für zukünftige Erweiterungen keinen Timer zu blockieren, wird für den Timeout der Systick-Timer (2.2.5) genutzt. Dieser implementiert eine Routine, welche im 10ms-Takt

---

aufgerufen wird. Für diese Funktion wird die Routine in der Datei `stm32fxx_it.c` um folgenden Code erweitert:

Listing 6: *Systick Timer*

```
if(dma_info.timer == 1)
{
    dma_info.t_flag = 1;
    HAL_UART_RxCpltCallback(&huartx);
}
if(dma_info.timer)
{
    --dma_info.timer;
}
```

Mittels dieser Erweiterung wird der Zähler des Timeouts dekrementiert, bis er 1 erreicht. Wenn dies geschieht, wird ein Flag gesetzt und die Interruptroutine `HAL_UART_RxCpltCallback (&huartx)` aufgerufen, in welcher anschließend der aufgetauchte Interrupt mittels des gesetzten Flags identifiziert und verarbeitet wird.

### 3.2.4 STM32: Empfangsinterrupt / DMA Circular Buffer

Während der Kommunikation mit UART werden verschiedene Interrupts ausgelöst werden. Bei der Nutzung von DMA werden Interrupts ausgelöst, wenn der Buffer halb voll oder ganz voll ist [5]. Des Weiteren wurde der µC so konfiguriert, dass auch bei einem Idle Line Interrupt die entsprechende Interruptroutine aufgerufen wird 3.2.3.

Die Interruptroutinen sind nach der Generation von Code mittels CubeMX in der Datei `stm32f1xx_hal.c` als `__weak` definiert, werden also neu gesetzt sobald sie ohne das Keyword `__weak` definiert werden [15].

In `main.c` wird die Interruptroutine für den Empfang von Daten per UART mit dem Namen

Listing 7: *Empfangsinterrupt*

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
```

initialisiert.

Die Variablen `pos`, `start` und `length` werden für den Ringbuffer benötigt. Die Variable `currCount` speichert die aktuelle Position des Ringbuffers und wird über den Befehl

Listing 8: *Position Ringbuffer*

```
__HAL_DMA_GET_COUNTER(huart->hdmarx)
```

beschrieben.

---

Die Variable `start` enthält die Startposition, ab welcher neue Daten im Ringbuffer enthalten sind. In der Variable `length` ist die Länge der Daten gespeichert. Tritt ein Interrupt auf, weil der Empfangsbuffer voll ist, berechnet sich die Länge der empfangenen Daten simpel durch folgenden Befehl:

Listing 9: Berechnung Länge

```
length = DMA_BUF_SIZE - start;
```

Ein Timeout-Interrupt kann ausgelöst werden, nach dem der Buffer voll ist. Um die falsche Verarbeitung von Daten zu verhindern, wird die aktuelle Position des Buffers auf die Größe des Buffers gesetzt.

Listing 10: Setzen des Zählers bei Timeout

```
dma_info.prevCOUNT = DMA_BUF_SIZE;
```

Wird ein Timeout-Interrupt ausgelöst, wird die Routine durch folgenden Code frühzeitig abgebrochen und das Flag zurückgesetzt:

Listing 11: Abbruch Timeoutinterrupt

```
if(dma_info.t_flag && currCOUNT == DMA_BUF_SIZE)
{
    dma_info.t_flag = 0;
    return;
}
```

Tritt ein Interrupt auf, weil ein Timeout stattgefunden hat, muss unterschieden werden ob im Buffer bereits alte Daten liegen, welche ignoriert werden müssen oder ob der Buffer mit zu verarbeitenden Daten gefüllt ist. Dies wird durch folgenden Code implementiert:

Listing 12: Längenberechnung Timeout

```
if(dma_info.t_flag)
{
    if(dma_info.prevCOUNT < DMA_BUF_SIZE)
    {
        length = dma_info.prevCOUNT - currCOUNT;
    }
    else
    {
        length = DMA_BUF_SIZE - currCOUNT;
    }

    dma_info.prevCOUNT = currCOUNT;
    dma_info.t_flag = 0;
}
```

Tritt ein Timeout-Flag auf, wird geprüft, ob der alte Zähler des Ringbuffers kleiner als die Buffergröße ist. Ist dies der Fall, berechnet sich die Länge der Daten durch die Differenz zwischen dem alten und dem neuen Zählerwert, da alte Daten im Buffer gespeichert sind.

---

Wenn keine alten Daten im Buffer gespeichert sind, berechnet sich die Länge durch die Differenz zwischen der Buffergröße und der aktuellen Zählerposition.

Im Anschluss wird der Zählerwert des DMA übergeben und das Timeout-Flag auf 0 zurückgesetzt.

Die Startposition der neuen Daten berechnet sich ähnlich. Ist der alte Zähler des Ringbuffers kleiner als die Buffergröße, ist die Startposition die Differenz zwischen der Buffergröße und des alten Zählerwerts. Wenn nicht, ist die Startposition gleich 0.

Listing 13: Berechnung Startposition

```
if(dma_info.prevCOUNT<DMA_BUF_SIZE)
{
    start = (DMA_BUF_SIZE-dma_info.prevCOUNT);
}
else
{
    start = 0;
}
```

Mit Hilfe der berechneten Werte für die Startposition und der Länge der empfangenen Daten werden diese Daten in ein dediziertes Array zur Weiterverarbeitung kopiert. Außerdem wird ein Flag gesetzt, welches anzeigt, dass neue Daten vorhanden sind:

Listing 14: Kopieren neuer Daten

```
for(uint16_t i=0, pos=start; i<length; ++i, ++pos)
{
    dma_info.data[i] = dma_rx_buf[pos];
}
dma_info.av_flag = 1;
```

Die Telegramme lassen sich mit den Informationen über das Protokoll aus 3.1.2, aus dem Array in dem die angekommenen Daten gespeichert wurden, extrahieren.

### 3.2.5 STM32: Überprüfen der Daten

Die empfangenen Daten müssen auf ihre Integrität getestet werden. Dazu wurde in 2.4.2 das Konzept der zyklischen Redundanzprüfung eingeführt.

Um die empfangenen Daten zu prüfen, wird das Array, welches die Daten enthält, der Funktion `uint8_t data_check(uint8_t *dat)` übergeben. Diese Funktion extrahiert die CRC-Prüfsumme sowie die Länge der Nachricht und validiert diese. Wenn die zyklische Redundanzprüfung erfolgreich ist, wird die Länge der Nachricht übergeben. Wenn sie fehlschlägt, wird eine 0 zurückgegeben.

---

### 3.2.6 STM32: Verarbeitung der empfangenen Daten

Wird durch das Flag `av_flag` der Strukturvariable `DMA_STRUCT` (siehe 3.2.1) angezeigt, dass neue Daten verfügbar sind, werden diese in der Main-Schleife verarbeitet. Mit Hilfe der Funktion `data_check()` aus 3.2.5 wird das empfangene Telegramm geprüft und die Länge der Nachricht in der globalen Variable `uint8_t data_len` gesichert sowie das Flag auf 0 zurückgesetzt.

Listing 15: *Datenverarbeitung*

```
if (dma_info.av_flag == 1)
{
    dma_info.av_flag = 0;
    data_len = data_check(dma_info.data);
```

Ist die Länge ungleich 0, war die Prüfung erfolgreich. Mittels eines Switch-Cases wird anschließend geprüft, um welche Art Nachricht es sich handelt (Nachrichtentypen siehe 3.1.3). Es folgt ein Beispiel für die Nachricht STATUS.

Listing 16: *Switch Case*

```
if (data_len != 0) {
    switch (dma_info.data[3]) {

        case STATUS:
            TxLen = ack_send(STATUS, TxBuffer);
            HAL_UART_Transmit_DMA(&huart3, TxBuffer, TxLen);
            tx_wait(&dma_info);

            TxLen = stat_send(TxBuffer);
            HAL_UART_Transmit_DMA(&huart3, TxBuffer, TxLen);
            tx_wait(&dma_info);
            break;
    }
}
```

Wenn die Länge ungleich 0 ist, wird die vierte Stelle des Arrays, welches das Telegramm enthält, geprüft, um den Nachrichtentyp zu bestimmen. Anschließend wird ein Acknowledge-Telegramm versendet, um den Empfang zu bestätigen. Dies geschieht mittels der Funktion `ack_send()`. Der Aufbau dieser Funktion ist ähnlich der in 3.2.7 erklärten Funktion. Auf den Versandprozess wird in 3.2.8 genauer eingegangen.

Nachdem das Acknowledge-Telegramm verschickt wurde, wird das eigentliche Antworttelegramm versendet (Funktion `stat_send()`) und der Switch-Case beendet.

Ist der Nachrichtentyp keiner bekannten Nachricht zuzuordnen, wird mittels des Default-Cases eine Errornachricht versandt.

### 3.2.7 STM32: Erstellen von Telegrammen

Es werden verschiedene Telegramme, abhängig von der Nachricht, versendet. Die Telegramme enthalten neben der eigentlichen Nachricht auch immer eine CRC-Prüfsumme,

---

Längeninformationen und ein Startzeichen (siehe 3.1.2). Um dies zu ermöglichen, wurden diverse Funktionen zum Erstellen von Telegrammen entwickelt. Dieser Sachverhalt wird an Hand einer Funktion, welche die Antwort auf den Befehl STATUS erstellt, dargestellt. Andere Funktionen, welche z.B. die Messwerte des Sensors oder eine Empfangsbestätigung versenden, sind ähnlich aufgebaut und unterscheiden sich nur durch den eigentlichen Nachrichteninhalt.

Allen Funktionen wird ein Zeiger auf ein Array übergeben, in welches die Daten geschrieben werden. Zudem geben die Funktionen die Länge des Telegramms zurück.

Die Funktion `uint8_t stat_send(uint8_t * txbuf)` schreibt Informationen über den Softwarestand, den genutzten Sensor und die Boardversion in das Array. Diese Daten sind durch die Defines `BOARD`, `VERSION` und `SENSOR` festgelegt.

Listing 17: *Status-Telegramm*

```
uint8_t stat_send(uint8_t * txbuf)
{
    uint16_t crc_value;

    txbuf[0] = START;
    txbuf[1] = 0 + OFF_ASCII;
    txbuf[2] = 3 + OFF_ASCII;
    txbuf[3] = BOARD;
    txbuf[4] = VERSION;
    txbuf[5] = SENSOR;
    crc_value = CRC16_buf(txbuf, 6);
    txbuf[6] = (crc_value >> 8) & 0xFF;
    txbuf[7] = (crc_value >> 0) & 0xFF;

    return 8;
}
```

Die Längeninformation wird als ASCII-Zahl dargestellt, sie muss deshalb um den entsprechenden Offset (48) verschoben werden. Da die CRC16-Prüfsumme 16 Bit lang ist, muss diese in zwei jeweils 8 Bit lange Teile aufgeteilt werden. Dies geschieht durch entsprechendes Verschieben nach rechts und der logischen UND-Operation mit der Zahl 0xFF.

Da das gesamte Telegramm acht Zeichen lang ist, wird eine 8 zurückgegeben.

### 3.2.8 STM32: Versenden von Daten

Das Versenden von Daten geschieht mittels folgender Funktion [15]:

Listing 18: *Sendefunktion*

```
HAL_StatusTypeDef HAL_UART_Transmit_DMA(UART_HandleTypeDef *huart,
                                         uint8_t *pData, uint16_t Size)
```

---

Der Handler, welcher dem genutzten UART zugeordnet ist, der zu übertragende Buffer und die Länge des Buffers werden übergeben.

Wichtig bei der Nutzung der Funktion ist das Abwarten, ob die Daten versendet wurden. Wird die Funktion zu früh ein zweites Mal aufgerufen, werden die Daten, die sich gerade im Sendebuffer befinden, überschrieben [15]. Eine Prüfung, ob die Funktion `HAL_OK` zurückgibt, ist nicht ausreichend, da dies bereits geschieht, bevor alle Daten übertragen wurden.

Ähnlich wie bei dem Empfang von Daten gibt es auch beim Versenden von Daten einen Interrupt, sobald die Daten erfolgreich versendet wurden [15]. Mittels des Flags `uart_info.tx_flag`, welches innerhalb der zugehörigen Interruptroutine gesetzt wird, wird die Vollendung der Übertragung kontrolliert.

Dieser Zusammenhang wird in folgender Funktion umgesetzt:

Listing 19: *Warten auf Versand*

```
void tx_wait(DMA_STRUCT * dma)
{
    while (!dma->tx_flag);
    dma->tx_flag=0;
}
```

Es wird die DMA-Strukturvariable übergeben und gewartet, bis das Flag in der Interruptroutine gesetzt wurde. Sobald dies geschehen ist, wird es zurückgesetzt; neue Daten können versendet werden.

### 3.2.9 ESP8266: Empfang und Überprüfung der Daten

Die Implementation für den Empfang von Daten ist auf Grund des Arduino-Frameworks [18] anders aufgebaut. Die Funktion `void receive()` implementiert die Funktionalität des Datenempfangs. Ähnlich wie bei der Implementation des STM32 wird hier ein Timeout genutzt, jedoch auf DMA verzichtet.

Folgende Variablen werden innerhalb der Funktion genutzt:

Listing 20: *Variablen receive()*

```
static uint16_t count = 0;
static bool InProg = false;
static uint8_t len= 0;
uint16_t crc_calc;
static uint32_t tim;
```

Die Variable `count` zählt die Anzahl der empfangenen Bytes, während `InProg` anzeigt, dass zurzeit Daten empfangen werden. Zudem speichert `len` die Länge der empfangenen Nachricht, während `tim` die Zählervariable für den Timeout implementiert. Um das Ergebnis der zyklischen Redundanzprüfung zu speichern, wird `crc_calc` genutzt.

---

Zudem werden die globalen Variablen `receivedChars[BUF_SIZE]` und `newData` benötigt. Das Array speichert die empfangenen Daten, während durch das Flag `newData` die Verfügbarkeit von neuen Daten angezeigt wird.

Die Implementation bedient sich diverser Befehle aus dem Arduino-Framework [18]:

- `Serial.available()` - Prüfen ob neue Daten vorhanden sind
- `Serial.read()` - Einlesen eines Bytes
- `millis()` - Auslesen der verstrichenen Zeit in ms

Um Funktionen, die im Zusammenhang mit dem UART stehen, nutzen zu können, muss dieser während des Starts des µC mit der Baudrate initialisiert werden [18]:

Listing 21: Konstruktor *UART*

```
Serial.begin(BAUD);
```

Die Funktion `void receive()` wird kontinuierlich in der Main-Loop aufgerufen.

Mittels einer While-Schleife wird geprüft, ob Daten verfügbar sind und ob Daten, welche zuvor empfangen wurden, verarbeitet werden müssen. Wenn diese Bedingung erfüllt ist, wird der Zähler des Timers auf den aktuellen Wert aktualisiert und ein Byte eingelesen.

Listing 22: Prüfen auf neue Daten

```
while (Serial.available() > 0 && newData == false)
{
    tim = millis();
    receivedChars[count] = Serial.read();
```

Nach jedem Durchlauf der kompletten While-Schleife wird der Zähler `count` inkrementiert, um das nächste Zeichen einzulesen.

Wird ein Start-Zeichen im empfangenen Telegramm (siehe 3.1.2) erkannt, wird die Variable `InProg` gesetzt, um die weitere Verarbeitung zu erlauben:

Listing 23: Erkennung Start-Marker

```
if (receivedChars[count] == START_MARKER)
{
    InProg = true;
}
```

Auf Basis der aktuellen Anzahl an eingelesenen Zeichen und dem gesetzten Flag wird nun das Einlesen der Längeninformation ermöglicht:

Listing 24: Einlesen Längeninformation

```
if (InProg == true && count == 2)
{
    len = (receivedChars[count]-OFF_ASCII)
        +((receivedChars[count-1]-OFF_ASCII)*10);
}
```

---

Die Länge muss dafür zu einer Integervariablen konvertiert werden. Dies geschieht durch die Subtraktion des Offsets (48) und das Zusammenführen der verschiedenen Stellen.

Basierend auf der eingelesenen Länge der Nachricht kann festgestellt werden, wann die CRC-Prüfsumme eingelesen wurde (siehe 3.1.2). Mittels der Prüfsumme wird dann die zyklische Redundanzprüfung durchgeführt. Bei Erfolg werden die Variablen der Funktion zurückgesetzt sowie das Flag zum Anzeigen von neuen Daten gesetzt. War die Prüfung nicht erfolgreich, wird das Flag nicht gesetzt.

Listing 25: *Zyklische Redundanzprüfung*

```
if(InProg == true && count == len+4){  
    crc_calc = CRC16_buf(receivedChars ,count+1);  
    if(crc_calc == 0)  
    {  
        length_pub = len;  
        InProg = false;  
        newData = true;  
        len = 0;  
        count = 0;  
        return;  
    }  
    else{  
        InProg = false;  
        newData = false;  
        len = 0;  
        count = 0;  
        return;  
    }  
}
```

Da der Zähler `count` am Ende der While-Schleife inkrementiert wird, muss der Funktion der zyklischen Redundanzprüfung `count+1` übergeben werden.

Bevor jedoch die soeben erklärte While-Schleife aufgerufen wird, muss geprüft werden, ob ein Timeout eingetreten ist. Diese Überprüfung bedient sich der gespeicherten Zeit, der aktuellen Zeit sowie dem Flag `InProg`, welches anzeigt, ob eine Transaktion läuft. Ist die Differenz zwischen aktueller Zeit und der gespeicherten Zeit größer als das definierte Timeout und das Flag gesetzt, werden die Variablen `count` und `InProg` zurückgesetzt und das Einlesen abgebrochen:

Listing 26: *Abbruch durch Timeout*

```
if (InProg == true && (millis() - tim > TIMEOUT))  
{  
    count = 0;  
    InProg = false;  
}
```

---

### 3.2.10 ESP8266: Anmelden an WLAN und MQTT-Server

Um die Datenübertragung per MQTT an ein übergeordnetes Netzwerk zu ermöglichen, wird eine drahtlose Netzwerkverbindung per WLAN eingesetzt. Hierfür ist es notwendig, den ESP8266 beim Netzwerk anzumelden. Dies geschieht mittels der ESP8266-WiFi Library [19]. Sobald die Verbindung erfolgreich hergestellt ist, meldet sich der ESP8266 beim MQTT-Server an, welcher sich im selben Netzwerk befindet.

Das Anmelden am WLAN erfolgt über die Funktion `void setup_wifi()`, diese enthält die Logik zum Aufbau der Verbindung. Es muss jedoch während der Variablen Deklaration der Konstruktor aufgerufen werden, um die entsprechende Klasse zu initialisieren:

Listing 27: Konstruktor WLAN-Library

```
WiFiClient espClient;
```

Um den Verbindungsauflauf zu signalisieren, leuchtet, während die Verbindung aufgebaut wird, die auf der Platine angebrachte LED auf. Die globalen Variablen `ssid` und `password` speichern den Namen und das Kennwort des drahtlosen Netzwerkes, mit welchem die Verbindung hergestellt wird.

Listing 28: Anmelden an WLAN

```
void setup_wifi() {
    bool led_flag = 0;
    delay(10);
    WiFi.begin(ssid, password);
    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        if (led_flag==0)
        {
            digitalWrite(ledPin,HIGH);
            led_flag = 1;
        }
        else
        {
            digitalWrite(ledPin,LOW);
            led_flag = 0;
        }
    }
    digitalWrite(ledPin,LOW);
}
```

Die Funktion `WiFi.begin` initiiert die Verbindung mit den Anmelddaten des Netzwerkes. Solange die Verbindung nicht aufgebaut wurde, signalisiert durch `WL_CONNECTED`, blinkt die LED alle 0.5s auf. Sobald die Verbindung erfolgreich hergestellt wurde, wird die LED abgeschaltet.

---

Die Verbindung zum MQTT-Server wird hergestellt, wenn die Verbindung zum Wireless Local Area Network (WLAN) erfolgreich hergestellt wurde. Der Server muss sich im selben Netzwerk befinden. Der Server wird mittels eines RaspberryPis realisiert (siehe 3.4).

Um die Kommunikation per MQTT zu vereinfachen, wird auf die Library PubSubClient [20] zurückgegriffen. Die IP des MQTT-Servers wird in der globalen Variable `mqtt_server` gespeichert. Die Funktion `void reconnect()` übernimmt den Verbindungsauftbau. Um Daten per MQTT zu empfangen, wird ein Callback implementiert, eine Funktion, welche bei Empfang von Daten aufgerufen wird:

*Listing 29: MQTT Callback*

```
void callback(String topic, byte* message, unsigned int length)
```

Die Variablen der Funktion enthalten Informationen über die Länge der Nachricht, das Topic, unter welchem die Nachricht empfangen wurde, sowie den eigentlichen Nachrichteninhalt.

Während der Variablen Deklaration wird der Konstruktor der Bibliothek aufgerufen, welcher die Informationen über die WLAN-Verbindung übernimmt:

*Listing 30: MQTT Konstruktor*

```
PubSubClient client(espClient);
```

Während der Initialisierung, nachdem die Verbindung zum WLAN-Netzwerk hergestellt wurde, werden die Verbindungsinformationen des MQTT-Servers gespeichert und der Callback gesetzt:

*Listing 31: MQTT Initialisierung*

```
setup_wifi();
client.setServer(mqtt_server, 1883);
client.setCallback(callback);
```

Innerhalb der Main-Schleife wird fortlaufend geprüft, ob die Verbindung besteht. Wenn dies nicht der Fall ist, wird diese mittels der Funktion `reconnect()` wiederhergestellt. Mit der Funktion `client.loop()` wird geprüft, ob neue Daten per MQTT empfangen wurden und gegebenenfalls der Callback aufgerufen.

*Listing 32: MQTT Verbindungsprüfung, Datenempfang*

```
if (!client.connected()) {
    reconnect();
}
client.loop();
```

Die Funktion `void reconnect()` implementiert den Verbindungsauftbau zum MQTT-Server. Dem Client (in diesem Falle der ESP8266) muss ein einzigartiger Name zur Identifikation vergeben werden. Wenn ein anderes Gerät denselben Namen besitzt, kommt es zu Fehlern. Dieser Name wird in der Variable `clientId` gespeichert.

---

Während die Verbindung aufgebaut wird, leuchtet die LED auf der Platine, um den Prozess anzuzeigen. Wurde die Verbindung erfolgreich aufgebaut, wird eine Nachricht („Sensor connected!“) geteilt und das Topic, auf welchem in Zukunft Befehle empfangen werden sollen, abonniert. Anschließend wird die LED abgeschaltet.

Listing 33: *Funktion reconnect()*

```
void reconnect()
{
    while (!client.connected())
    {
        if (client.connect(clientId.c_str()))
        {
            client.publish("Humid", "SensorConnected!");
            client.subscribe("SensorSetup");
            digitalWrite(ledPin, LOW);
        } else
        {
            digitalWrite(ledPin, HIGH);
            delay(5000);
        }
    }
}
```

### 3.2.11 ESP8266: Empfang und Versand per MQTT

Der Versand von Nachrichten geschieht durch die Funktion `client.publish()`. Ihr werden das Topic, die Daten, welche übertragen werden sollen (in Form eines Arrays) sowie die Länge des Arrays übergeben. Die Funktion wird innerhalb der Main-Schleife immer dann aufgerufen, wenn durch das Flag `newData` angezeigt wird, dass neue Daten verfügbar sind (siehe 3.2.9).

Es ist dabei zu beachten, dass die Startzeichen sowie die Längenangabe nicht übertragen werden sollen, sondern nur der Nachrichteninhalt. Dies erklärt den Offset +3 des Arrays. Das Flag `newData` muss zurückgesetzt werden, um ein erneutes Versenden zu ermöglichen.

Listing 34: *Versand per MQTT*

```
if (newData == true)
{
    client.publish("Humid", receivedChars+3, length_pub);
    newData = false;
}
```

Der Empfang von Daten geschieht mittels des in 3.2.10 erwähnten Callbacks. Dieser wird aufgerufen, sobald neue Daten verfügbar sind. In dieser Routine werden die Daten direkt per UART an den STM32 weitergereicht. Dem Callback werden Variablen übergeben,

---

die das Topic und die Länge der Nachricht enthalten, sowie ein Zeiger auf das Array mit der eigentlichen Nachricht.

Die Variable `uint16_t crc16` dient der Speicherung der berechneten CRC-Prüfsumme. Das Array `char len_buf[2]` speichert die Längenangabe der empfangenen Nachricht. Dem Callback wird die Längenangabe als Integervariable übergeben, darum ist eine Konversion zu einem Char-Array notwendig, da die Kommunikation unter Nutzung des ASCII-Standards geschieht. Das globale Array `uint8_t mqtt_in_data` speichert die Daten, welche empfangen wurden, inkl. der hinzugefügten Startmarker, Längeninformationen etc.

Listing 35: *Längenkonversion*

```
void callback(String topic, byte* message, unsigned int length) {
    char len_buf[2];
    uint16_t crc16;

    sprintf(len_buf, "%d", length);

    mqtt_in_data[0] = START_MARKER;

    if (length < 10){
        mqtt_in_data[1] = '0';
        mqtt_in_data[2] = len_buf[0];
    }
    else{
        mqtt_in_data[1] = len_buf[0];
        mqtt_in_data[2] = len_buf[1];
    }
}
```

Die Funktion `sprintf()` ermöglicht die Konversion einer Integervariablen zu einem Char-Array unter Angabe der Länge `length`. Es muss allerdings darauf geachtet werden, dass unter Nutzung des in 3.1.2 erklärten Formates bei einer Länge `<10` das erste Byte der Längeninformation 0 sein muss und in diesem Falle die Länge in `len_buf[0]` gespeichert wird.

Ist die Länge `>10`, wird in `len_buf[0]` die Einerstelle und in `len_buf[1]` die Zehnerstelle gespeichert. Des Weiteren wird, dem Protokoll entsprechend, der erste Wert des Arrays mit dem Startmarker beschrieben.

Die eigentliche Nachricht wird anschließend dem Ausgabearray übergeben und die CRC-Prüfsumme berechnet. Der Länge `length` muss dabei die Anzahl der hinzugefügten Bytes addiert werden (+3). Anschließend wird die 16-Bit breite Prüfsumme geteilt und angehängt.

Um das Telegramm im Anschluss an den STM32 auszugeben wird das Array `mqtt_in_data[]` mit Hilfe der Arduino-Funktion `Serial.print()` per UART versendet.

---

Listing 36: Ausgabe MQTT-Nachricht

```
for (uint8_t i = 0; i < length; i++)
{
    mqtt_in_data[i+3] = message[i];
}

crc16 = CRC16_buf(mqtt_in_data,length+3); //calc the crc16

mqtt_in_data[length+3] = (crc16 >> 8) & 0xFF;
mqtt_in_data[length+4] = (crc16 >> 0) & 0xFF;

for (uint8_t c = 0; c <= (length+4); c++)
{
    Serial.print((char)mqtt_in_data[c]);
}
```

---

### 3.3 Erfassung und Verarbeitung von Messwerten

Im Praxisbeispiel sollen die erklärten Technologien Anwendung finden. Es wird ein kapazitiver Sensor verwendet, um die Feuchtigkeit des Erdreiches zu messen. Die ermittelten Messwerte sollen per UART vom STM32 an den ESP8266 übertragen werden. Anschließend werden die Messwerte dem übergeordneten Netzwerk per MQTT zur Verfügung gestellt.

Um präzise Messungen vorzunehmen, muss der Sensor kalibriert werden. Die Messwerte, die durch den ADC gemessen werden, müssen zudem in Prozent umgewandelt werden, da reine Spannungsspeicher wenig aussagekräftig sind.

Der Sensor muss während des Betriebs in das Erdreich gesteckt werden, zu sehen in Abb.16.

#### 3.3.1 Kapazitive Feuchtigkeitsmessung

Der genutzte Sensor [21], gibt die gemessene Feuchtigkeit als eine Spannung zwischen 0V und 3V aus und wird mit einer Spannung von 3.3V betrieben. Die Spannung entspricht demselben Spannungsspeicher, mit welchem auch die beiden µC betrieben werden. Dies vereinfacht die Verwendung.

Verbunden wird der Sensor mit dem Anschluss PA1 des STM32, welcher wiederum prozessorseitig mit dem ADC verbunden ist [4]. Auf der Platine führt dieser Anschluss zu Pin 15 [1]. Des Weiteren sind Verbindungen zur Versorgungsspannung und dem Massepotential erforderlich [21].

#### 3.3.2 Analog-Digital Converter

Es wird ein einzelner Kanal des ADC1 des STM32s verwendet. Dieser wird so konfiguriert, dass eine einzelne Konversion durch Software ausgeführt wird und bei Abschluss dieser ein Interrupt ausgelöst wird [5]. Die Konfiguration geschieht mittels CubeMX [16].

Der ADC verfügt über eine Funktion zur Selbstkalibration [5]. Diese wird während der Initialisierung des µC durch folgende Funktion ausgeführt [15]:

Listing 37: *ADC Kalibrierung*

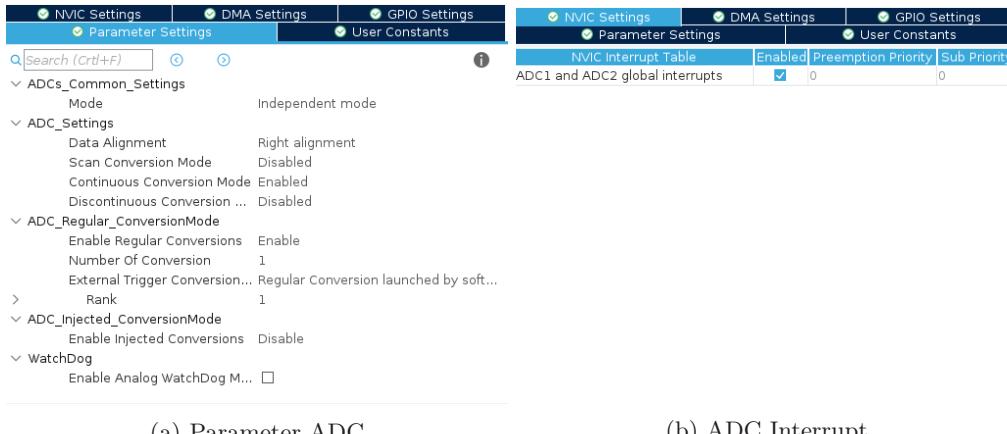
```
HAL_ADCEx_Calibration_Start(&hadcx);
```



Abbildung 16: *Pflanze mit Sensor*



Abbildung 17: *Kapazitiver Sensor*



(a) Parameter ADC (b) ADC Interrupt

Abbildung 18: *Konfiguration des ADCs*

Die Verarbeitung des Interrupts des ADC wird ähnlich realisiert, wie in 3.2.4 für den UART beschrieben. Nach dem Auftreten eines Interrupts wird dieser in der Routine

Listing 38: *ADC Callback*

```
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef *hadc)
```

verarbeitet.

Die Konversion wird mittels der Funktion `HAL_ADC_Start_IT(&hadcx)` gestartet und kann durch `HAL_ADC_Stop_IT(&hadcx)` gestoppt werden.

### 3.3.3 Strukturvariable SENS\_STRUCT

Aus Gründen der Übersichtlichkeit wird eine Strukturvariable mit diversen Flags und Variablen eingeführt.

Listing 39: *Sensor Strukturvariable*

```
typedef struct
{
    uint8_t cal_flag;
    uint16_t dry_value;
    uint16_t wet_value;
    volatile uint8_t timer_flag;
    uint16_t an_value;
    uint8_t percentage;
    volatile uint8_t adc_flag;
    uint8_t send_flag;
} SENS_STRUCT;
```

Das Flag `cal_flag` zeigt an, dass der Sensor kalibriert wurde. Das Flag `adc_flag` wird genutzt, um eine vollendete Konversion des ADC anzuzeigen. Um die kontinuierliche

---

Übertragung von Messwerten freizuschalten, wird `send_flag` genutzt. `timer_flag` speichert, ob ein Interrupt durch einen Timer ausgelöst wurde.

Die Variablen `dry_value`, `wet_value` und `an_value` speichern Werte zur Kalibrierung sowie den aktuellen Messwert. Dessen Repräsentation in Prozent wird mittels der Variablen `percentage` gespeichert.

### 3.3.4 Kalibrierung des Sensors

Der Sensor muss vor Benutzung kalibriert werden, damit eine zuverlässige Messung möglich ist. Dazu muss ein Messwert erfasst werden, wenn der Sensor komplett trocken ist und ein Messwert, wenn der Sensor in Wasser getränkt wird (siehe Abb. 19)[21].

Die Kalibrierung wird durch den Befehl `CALIBRATE`, beschrieben in 3.1.3, ausgelöst. Da der Sensor für die Messung der Werte in Wasser getaucht werden soll, muss der Nutzer durch den Prozess begleitet werden. Dies geschieht durch die Benutzung der auf der Platine aufgelöten LED [1], welche abhängig vom aktuellen Schritt aufleuchtet. Folgende Funktion der Datei `sensor.c` implementiert die Kalibrierung:

Listing 40: Funktion Kalibrierung

```
uint8_t cal_sens(ADC_HandleTypeDef hadc, SENS_STRUCT * sens)
```

Die Funktion gibt eine '1' bei Erfolg, oder eine '0' bei Misserfolg zurück.

Der Ablauf lautet wie folgt:

- Starten der Kalibrierung durch `CALIBRATE`
- LED an, Sensor muss trocken sein
- LED aus, Wert wird gemessen
- LED an, Sensor muss nass sein
- LED aus, Wert wird gemessen

Um zuverlässige Messwerte zu erhalten, werden immer 10 Messungen durchgeführt und die gemessenen Werte gemittelt. Die Messung wird exemplarisch für den Messwert des trockenen Sensors erklärt. Der Zähler `cal_counter` speichert die Anzahl der Messungen. Die Variable `dry` speichert den gemittelten Messwert.



Abbildung 19: Kalibrierung des Sensors

---

Listing 41: *Kalibrationsroutine*

```
HAL_ADC_Start_IT(&hadc);
while(cal_counter<10)
{
    if(sens->adc_flag == 1)
    {
        cal_counter++;
        sens->an_value = hadc.Instance->DR;
        dry += sens->an_value;
        sens->adc_flag=0;
        HAL_ADC_Start_IT(&hadc);
    }
}
cal_counter = 0;
dry = dry / 10;
sens->dry_value = dry;
```

Die Messung wird durch `HAL_ADC_Start_IT(&hadc)` gestartet. Wenn die Konversion abgeschlossen ist, wird in der entsprechenden Interruptroutine das Flag `adc_flag` gesetzt. Anschließend wird der Zähler, welcher die While-Schleife der Messung steuert, inkrementiert und der aktuelle Messwert aus dem Register des ADC ausgelesen. Dieser Messwert und seine folgenden werden in der Variablen `dry` summiert.

Nach dem Zurücksetzen des Flags wird eine erneute Konversion gestartet, bis der Zähler seinen Endwert erreicht hat. Anschließend wird der Zähler zurückgesetzt, der Messwert gemittelt und an die Strukturvariable übergeben.

Die LED für die Benutzerinformation wird durch einen Timer (siehe 2.2.5), genauer TIM2, mit einer Periodendauer von 0.5s gesteuert. Der Timer wird mit der Funktion `HAL_TIM_Base_Start_IT(&htimx)` gestartet [15]. In seiner Interruptroutine wird das Flag `timer_flag` gesetzt.

Listing 42: *LED Timer*

```
HAL_TIM_Base_Start_IT(&htim2);
while(sens->timer_flag)
{
    HAL_GPIO_WritePin(LED_GPIO_Port, LED_Pin, GPIO_PIN_SET);
}
sens->timer_flag = 0;
HAL_GPIO_WritePin(LED_GPIO_Port, LED_Pin, GPIO_PIN_RESET);
```

Während das Abbruchkriterium der While-Schleife nicht erfüllt ist, wird gewartet und die LED angeschaltet. Wird der entsprechende Interrupt ausgelöst und das Flag gesetzt, führt dies zum Verlassen der Schleife. Sobald die Schleife verlassen wurde, wird das Flag rückgesetzt und die LED wieder abgeschaltet.

Innerhalb der Interruptroutine wird der Timer wieder gestoppt, um ein ständiges Auslösen des Interrupts zu vermeiden.

---

Mit den gemessenen Werten für den feuchten und den trockenen Sensor kann nun geprüft werden, ob die Ergebnisse realistisch sind. Es ist davon auszugehen, dass die Kalibration erfolgreich war, wenn der Messwert des trockenen Sensors höher ist, als der des feuchten Sensors. Dies wird am Ende der Funktion geprüft und abhängig davon der Rückgabewert ausgegeben.

Listing 43: *Prüfung Ergebnisse*

```
if((dry-wet)>0){  
    return 1;  
}  
else{  
    return 0;  
}
```

Der Rückgabewert wird in der Variablen `cal_flag` des Structs `SENS_STRUCT` gespeichert.

---

### 3.3.5 Berechnung des Messwertes

Mittels der durch die Kalibrierung ermittelten Werte lässt sich nun der eigentliche Messwert auf einer Skala von 0% bis 100% darstellen. Die Variable  $x$  steht für den aktuellen Messwert, während  $x_{wet}$  und  $x_{dry}$  für die jeweiligen Kalibrationswerte stehen.

Die Formel, welche den Zusammenhang darstellt, lautet:

$$f(x) = 100 - \frac{(x-x_{wet})*100}{x_{wet}-x_{dry}}$$

Bei der Implementation dieser Funktion in C ist bei der Nutzung von Integervariablen ohne Vorzeichen darauf zu achten, dass der Nenner nicht negativ oder 0 wird, da dies zu unerwarteten Ergebnissen führen kann.

Die Funktion `uint8_t val_sens(ADC_HandleTypeDef hadc, SENS_STRUCT * sens)` implementiert die Berechnung und ist in Datei `sensor.c` zu finden. Um unerwarteten Ergebnisse zu vermeiden, wird geprüft, ob die Differenz zwischen den Kalibrierungswerten nicht negativ ist. Als Rückgabewert wird die gemessene Feuchtigkeit in Prozent ausgegeben.

Listing 44: Berechnung des Messwerts

```
uint8_t val_sens(ADC_HandleTypeDef hadc, SENS_STRUCT * sens)
{
    uint8_t percentage;
    sens->an_value = HAL_ADC_GetValue(&hadc);
    if(sens->an_value-sens->wet_value > 0)
    {
        percentage = 100-((sens->wet_value-sens->an_value)*100)
                    /(sens->wet_value-sens->dry_value);
    }
    else
    {
        percentage = 0;
    }
    return percentage;
}
```

---

### 3.3.6 Periodisches Versenden der Messwerte

Wurde der Sensor erfolgreich kalibriert, kann mit dem Befehl SEND\_VAL der periodische Versand der Messwerte beginnen. Wenn der Sensor nicht kalibriert ist, wird eine Fehlermeldung versendet.

Listing 45: *Periodisches Versenden von Messwerten*

```
if(sensor.cal_flag == 1){  
    HAL_TIM_Base_Start_IT(&htim1);  
}  
else{  
    TxLen = er_send(ER_NOT_CAL, TxBuffer);  
    HAL_UART_Transmit_DMA(&huart3, TxBuffer, TxLen);  
    tx_wait(&dma_info);  
}
```

Ist der Sensor kalibriert, wird der Timer 1 (siehe 2.2.5) gestartet, welcher anschließend mit einer Periodendauer von 0.5s den Timer-Interrupt auslöst. Innerhalb der Interruptroutine wird die Konversion des ADC gestartet.

Listing 46: *ADC Start*

```
if(htim->Instance == TIM1){  
    HAL_ADC_Start_IT(&hadc1);  
}
```

Wenn dieser mit der Konversion des analogen Messwertes fertig ist, wird das entsprechende Flag `adc_flag` der Strukturvariablen `SENS_STRUCT` gesetzt (siehe 3.3.3).

Innerhalb der Main-Schleife wird ständig geprüft, ob dieses Flag gesetzt wurde und gegebenenfalls der Messwert in einen Prozentwert (siehe 3.3.5) umgerechnet, um anschließend versendet zu werden. Auch hier ist darauf zu achten, das Flag des ADC `adc_flag` wieder zurückzusetzen. Damit kein fehlerhafter Wert versendet werden kann, wird auch geprüft, ob der Sensor kalibriert ist. Ist dies nicht der Fall, wird eine Fehlermeldung versendet (siehe 47).

---

Listing 47: Versenden Messwert

```
if(sensor.adc_flag == 1)
{
    if (sensor.cal_flag == 1)
    {
        sensor.percentage = val_sens(hadc1, &sensor);
        sensor.adc_flag = 0;
        TxLen = an_send(sensor.percentage, TxBuffer);
        HAL_UART_Transmit_DMA(&huart3, TxBuffer, TxLen);
        tx_wait(&dma_info);
    }
    else
    {
        TxLen = er_send(ER_CAL,TxBuffer);
        HAL_UART_Transmit_DMA(&huart3, TxBuffer, TxLen);
        tx_wait(&dma_info);
    }
}
```

---

### 3.4 Inbetriebnahme MQTT-Server auf Raspberry Pi

Als Server für den MQTT-Broker (siehe 2.3.2) fungiert ein Raspberry Pi 3B+. Als Serverseitiges Programm wird Mosquitto genutzt, welches von der Eclipse Foundation zur Verfügung gestellt wird [22].

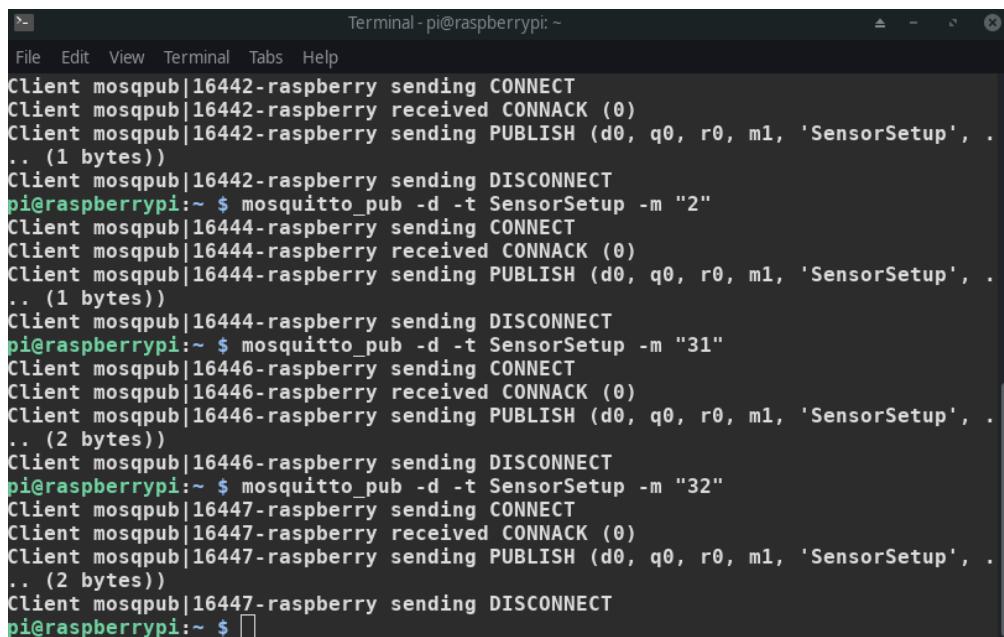
Wenn der Paketmanager APT genutzt wird, reicht der Befehl `sudo apt install mosquitto` aus, wobei ein vorheriges Update des Systems und der Datenbanken des Paketmanagers empfehlenswert ist. Dies geschieht durch die Befehle `sudo apt-get update` sowie `sudo apt-get upgrade`.

Um selbst Nachrichten per MQTT empfangen und versenden zu können, wird die Applikation Mosquitto-Clients benötigt, welche durch den Befehl `sudo apt install mosquitto-clients` installiert wird.

Da es die Nutzung vereinfacht, wenn der Broker automatisch nach dem Bootvorgang des Servers startet, wird dieser zusätzlich als Systemd-Service registriert. Dies geschieht mittels des Kommandos `sudo systemctl enable mosquitto`.

Um Topics zu abonnieren wird der Befehl `mosquitto-sub -t /topic/beispiel` genutzt. Um eine Nachricht an ein bestimmtes Topic zu versenden, steht der Befehl `mosquitto-pub -t /topic/beispiel -m "HelloWorld"` zur Verfügung [22].

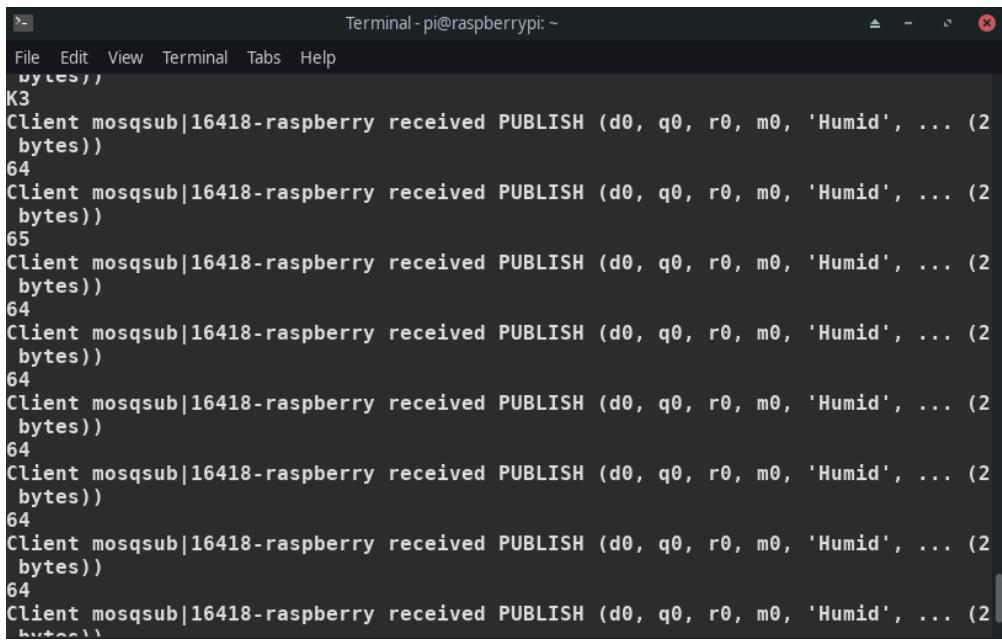
Folgende Abbildungen zeigen einerseits die von der Platine versendeten Nachrichten (Abb. 20) und die an die Platine gesendeten Nachrichten (Abb. 21). In diesem Beispiel wurde der Status des Boards abgefragt, die Kalibrierung gestartet und anschließend die Übertragung von Messwerten gestartet und wieder gestoppt.



```
Terminal - pi@raspberrypi: ~
File Edit View Terminal Tabs Help
Client mosqpub|16442-raspberry sending CONNECT
Client mosqpub|16442-raspberry received CONNACK (0)
Client mosqpub|16442-raspberry sending PUBLISH (d0, q0, r0, m1, 'SensorSetup', . .
.. (1 bytes))
Client mosqpub|16442-raspberry sending DISCONNECT
pi@raspberrypi:~ $ mosquitto_pub -d -t SensorSetup -m "2"
Client mosqpub|16444-raspberry sending CONNECT
Client mosqpub|16444-raspberry received CONNACK (0)
Client mosqpub|16444-raspberry sending PUBLISH (d0, q0, r0, m1, 'SensorSetup', . .
.. (1 bytes))
Client mosqpub|16444-raspberry sending DISCONNECT
pi@raspberrypi:~ $ mosquitto_pub -d -t SensorSetup -m "31"
Client mosqpub|16446-raspberry sending CONNECT
Client mosqpub|16446-raspberry received CONNACK (0)
Client mosqpub|16446-raspberry sending PUBLISH (d0, q0, r0, m1, 'SensorSetup', . .
.. (2 bytes))
Client mosqpub|16446-raspberry sending DISCONNECT
pi@raspberrypi:~ $ mosquitto_pub -d -t SensorSetup -m "32"
Client mosqpub|16447-raspberry sending CONNECT
Client mosqpub|16447-raspberry received CONNACK (0)
Client mosqpub|16447-raspberry sending PUBLISH (d0, q0, r0, m1, 'SensorSetup', . .
.. (2 bytes))
Client mosqpub|16447-raspberry sending DISCONNECT
pi@raspberrypi:~ $
```

Abbildung 20: Senden von Nachrichten

---



A screenshot of a terminal window titled "Terminal - pi@raspberrypi: ~". The window shows a series of log messages from a Mosquitto client. The messages are identical, indicating that the client has received a PUBLISH message from a broker. Each message includes a timestamp (e.g., K3, 64, 65, 64, 64, 64, 64), the client ID ("mosqsub|16418-raspberry"), the topic ("Humid"), and the payload length (2 bytes). The payload content is not explicitly shown.

```
File Edit View Terminal Tabs Help  
uy lcs,,  
K3  
Client mosqsub|16418-raspberry received PUBLISH (d0, q0, r0, m0, 'Humid', ... (2 bytes))  
64  
Client mosqsub|16418-raspberry received PUBLISH (d0, q0, r0, m0, 'Humid', ... (2 bytes))  
65  
Client mosqsub|16418-raspberry received PUBLISH (d0, q0, r0, m0, 'Humid', ... (2 bytes))  
64  
Client mosqsub|16418-raspberry received PUBLISH (d0, q0, r0, m0, 'Humid', ... (2 bytes))  
64  
Client mosqsub|16418-raspberry received PUBLISH (d0, q0, r0, m0, 'Humid', ... (2 bytes))  
64  
Client mosqsub|16418-raspberry received PUBLISH (d0, q0, r0, m0, 'Humid', ... (2 bytes))  
64  
Client mosqsub|16418-raspberry received PUBLISH (d0, q0, r0, m0, 'Humid', ... (2 bytes))  
64  
Client mosqsub|16418-raspberry received PUBLISH (d0, q0, r0, m0, 'Humid', ... (2 bytes))  
64  
Client mosqsub|16418-raspberry received PUBLISH (d0, q0, r0, m0, 'Humid', ... (2 bytes))
```

Abbildung 21: *Empfang von Nachrichten*

---

## 4 Auswertung

### 4.1 Fazit

Es wurde eine robuste Übertragung sensibler Messwerte erfolgreich implementiert. Mittels der genutzten Technologien wie zyklischer Redundanzprüfung, UART und MQTT werden die Messwerte zuverlässig übertragen und auf Fehler in der Übertragung überprüft. Durch die Nutzung von MQTT ist die Anbindung an ein übergeordnetes Netzwerk möglich.

Durch die Implementation eines kapazitiven Feuchtigkeitssensors wurde die Funktionalität der Übertragung erfolgreich demonstriert. Durch diverse Befehle kann der Sensor kalibriert und das Versenden von Messwerten freigeschaltet werden.

### 4.2 Fehler

Die Implementation einer robusten Kommunikation ist komplex. Die ursprüngliche Idee, das Telegrammende ebenfalls durch ein ASCII-Zeichen anzuzeigen, erwies sich als unzuverlässig. Es kam immer wieder zu falsch erkannten Telegrammenden. Ursache dafür war die Nutzung der CRC-Prüfsumme, da diese durchaus dasselbe Zeichen ergeben kann, welches auch für das Telegrammende definiert war.

Lösung des Problems war die Übertragung von Längeninformationen und der Verzicht auf ein Ende-Zeichen.

Der STM32 besitzt eine Peripherieeinheit zur Berechnung von CRC32-Prüfsummen. Da der genutzte Algorithmus keinem etablierten Standard folgt, wurde diese Peripherieeinheit nicht genutzt. Die Implementation der zyklischen Redundanzprüfung mittels CRC16-CCITT erwies sich jedoch als zuverlässig.

### 4.3 Ausblick

Diese Arbeit bildet eine Grundlage für zahlreiche mögliche weitere Entwicklungen und Verbesserungen. Es ist durchaus interessant, andere Sensortypen zu implementieren und eine einfache Konfiguration dieser zu ermöglichen. Die Verschlüsselung der Kommunikation zwischen den Prozessoren ist eine weitere Möglichkeit, die Datenübertragung weiter zu optimieren. Schlanke Algorithmen wie XTEA bieten sich hierfür an, da die Berechnung dieser vergleichsweise wenig Rechenleistung benötigt.

Um den Code zu optimieren, wäre die Umstellung auf die Programmiersprache C++ und die damit einhergehende Objektorientierung zukunftsorientiert. Dies würde in Hinblick auf die Implementation mehrerer Sensoren (auch Sensoren gleichen Typs) den Programmieraufwand verringern und die Anbindung der Sensoren vereinfachen.

Auch der Aspekt der serverseitigen Datenverarbeitung der gesammelten Daten ist interessant. Mittels Projekten wie Node-Red oder anderen Automatisierungsplattformen lassen sich MQTT-Nachrichten leicht verarbeiten und versenden. Gekoppelt mit einer graphischen Repräsentation der Messwerte ist dies ein logischer nächster Schritt in der Fortführung des Projekts.

---

## Abbildungsverzeichnis

1	<i>IoT-Gateway [1]</i>	9
2	<i>ESP8266 WROOM-02</i>	10
3	<i>STM32F103C8</i>	10
4	<i>DMA [6]</i>	11
5	<i>Aufbau [6]</i>	12
6	<i>Beschaltung [5]</i>	13
7	<i>Timer [4]</i>	14
8	<i>Parität [7]</i>	15
9	<i>Verbindung [9]</i>	16
10	<i>Framing [9]</i>	16
11	<i>Nachricht 'OK' [9]</i>	16
12	<i>Publish/Subscribe-Modell</i>	17
13	<i>Ringbuffer [13]</i>	18
14	<i>CubeMX</i>	19
15	<i>Konfiguration des UARTs</i>	25
16	<i>Pflanze mit Sensor</i>	39
17	<i>Kapazitiver Sensor</i>	39
18	<i>Konfiguration des ADCs</i>	40
19	<i>Kalibrierung des Sensors</i>	41
20	<i>Senden von Nachrichten</i>	47
21	<i>Empfang von Nachrichten</i>	48

---

## Listings

1	<i>Umschalten von GPIO</i>	19
2	<i>Berechnung CRC16</i>	21
3	<i>DMA Strukturvariable</i>	24
4	<i>Aktivierung Idle Line Interrupt</i>	25
5	<i>Idle Line Interrupt Clear</i>	25
6	<i>Systick Timer</i>	26
7	<i>Empfangsinterrupt</i>	26
8	<i>Position Ringbuffer</i>	26
9	<i>Berechnung Länge</i>	27
10	<i>Setzen des Zählers bei Timeout</i>	27
11	<i>Abbruch Timeoutinterrupt</i>	27
12	<i>Längenberechnung Timeout</i>	27
13	<i>Berechnung Startposition</i>	28
14	<i>Kopieren neuer Daten</i>	28
15	<i>Datenverarbeitung</i>	29
16	<i>Switch Case</i>	29
17	<i>Status-Telegramm</i>	30
18	<i>Sendefunktion</i>	30
19	<i>Warten auf Versand</i>	31
20	<i>Variablen receive()</i>	31
21	<i>Konstruktor UART</i>	32
22	<i>Prüfen auf neue Daten</i>	32
23	<i>Erkennung Start-Marker</i>	32
24	<i>Einlesen Längeninformation</i>	32
25	<i>Zyklische Redundanzprüfung</i>	33
26	<i>Abbruch durch Timeout</i>	33
27	<i>Konstruktor WLAN-Library</i>	34
28	<i>Anmelden an WLAN</i>	34
29	<i>MQTT Callback</i>	35
30	<i>MQTT Konstruktor</i>	35
31	<i>MQTT Initialisierung</i>	35
32	<i>MQTT Verbindungsprüfung, Datenempfang</i>	35
33	<i>Funktion reconnect()</i>	36
34	<i>Versand per MQTT</i>	36
35	<i>Längenkonversion</i>	37
36	<i>Ausgabe MQTT-Nachricht</i>	38
37	<i>ADC Kalibrierung</i>	39
38	<i>ADC Callback</i>	40
39	<i>Sensor Strukturvariable</i>	40
40	<i>Funktion Kalibrierung</i>	41
41	<i>Kalibrationsroutine</i>	42

---

42	<i>LED Timer</i>	42
43	<i>Prüfung Ergebnisse</i>	43
44	<i>Berechnung des Messwerts</i>	44
45	<i>Periodisches Versenden von Messwerten</i>	45
46	<i>ADC Start</i>	45
47	<i>Versenden Messwert</i>	46

---

## Quellenverzeichnis

- [1] Julius Bartel. Entwicklung eines IoT-Gateways, Wintersemester 2020.
- [2] Julius Bartel. Listings der Bachelorarbeit. URL <https://github.com/pitchbent/stm32-esp8266com>. Zuletzt aufgerufen am: 6.6.2021.
- [3] Espressif Systems. Esp-wroom-02 datasheet. URL [https://www.espressif.com/sites/default/files/documentation/0c-esp-wroom-02\\_datasheet\\_en.pdf](https://www.espressif.com/sites/default/files/documentation/0c-esp-wroom-02_datasheet_en.pdf). Zuletzt aufgerufen am: 30.4.2021.
- [4] ST. Stm32f103x8 datasheet, . URL <https://www.st.com/resource/en/datasheet/stm32f103v8.pdf>. Zuletzt aufgerufen am: 14.10.2020.
- [5] ST. Rm0008 reference manual, . URL <https://www.st.com/en/microcontrollers-microprocessors/stm32f103.html#overviewt>. Zuletzt aufgerufen am: 30.4.2021.
- [6] Carmine Noviello. *Mastering STM32*. Learnpub, 2018.
- [7] RosarioVanTulpe. Codetafel – Dualergänztes gerades Paritätsbit (E = even = gerade). URL [https://de.wikipedia.org/wiki/Parit%C3%A4tsbit#/media/Datei:Code\\_Even\\_dualergaenzt.svg](https://de.wikipedia.org/wiki/Parit%C3%A4tsbit#/media/Datei:Code_Even_dualergaenzt.svg). Zuletzt aufgerufen am: 4.5.2021.
- [8] Klaus Dembowski. *Computerschnittstellen und Bussysteme : für PC, Tablets, Smartphones und Embedded-Systeme*. VDE Verlag, 2016.
- [9] Sparkfun. Wiring and Hardware. URL <https://cdn.sparkfun.com/assets/2/5/c/4/5/50e1ce8bce395fb62b000000.png>. Zuletzt aufgerufen am: 6.5.2021.
- [10] Helmut Müller. *Mikroprozessortechnik*. Vogel Buchverlag, 2012.
- [11] MQTT.org. Mqtt FAQ. URL <https://mqtt.org/faq/>. Zuletzt aufgerufen am: 29.5.2021.
- [12] Dominik Obermaier und Sebastian Gerstl. Was ist MQTT? *embedded-software-engineering*, 2018.
- [13] Cburnett. An empty 7-element circular buffer. URL [https://en.wikipedia.org/wiki/Circular\\_buffer#/media/File:Circular\\_buffer--empty.svg](https://en.wikipedia.org/wiki/Circular_buffer#/media/File:Circular_buffer--empty.svg). Zuletzt aufgerufen am: 9.5.2021.
- [14] Dr. Christof Hübner. Industrielle Kommunikationstechnik, Sommersemester 2020.
- [15] ST. Description of stm32f1 hal and low-layer drivers, . URL <https://www.st.com/en/embedded-software/stm32cube-mcu-mpu-packages.html?querycriteria=productId=LN1897#overview>. Zuletzt aufgerufen am: 30.4.2021.
- [16] ST. Cubemx, . URL <https://www.st.com/en/development-tools/stm32cubemx.html>. Zuletzt aufgerufen am: 11.5.2021.

- 
- [17] Bastian Molkenthin. Understanding CRC. URL [http://www.sunshine2k.de/articles/coding/crc/understanding\\_crc.html](http://www.sunshine2k.de/articles/coding/crc/understanding_crc.html). Zuletzt aufgerufen am: 9.5.2021.
  - [18] Arduino. Arduino Language Reference. URL <https://www.arduino.cc/reference/en/>. Zuletzt aufgerufen am: 21.5.2021.
  - [19] Ivan Grokhotkov. Esp8266wifi library. URL <https://arduino-esp8266.readthedocs.io/en/latest/esp8266wifi/readme.html>. Zuletzt aufgerufen am: 6.6.2021.
  - [20] Nick O'Leary. Arduino client for MQTT. URL <https://github.com/knolleary/pubsubclient>. Zuletzt aufgerufen am: 6.6.2021.
  - [21] dfrobot. Capacitive soil moisture sensor. URL [https://wiki.dfrobot.com/Capacitive\\_Soil\\_Moisture\\_Sensor\\_SKU\\_SEN0193](https://wiki.dfrobot.com/Capacitive_Soil_Moisture_Sensor_SKU_SEN0193). Zuletzt aufgerufen am: 28.5.2021.
  - [22] Eclipse Foundation. Eclipse Mosquitto. URL <https://mosquitto.org/>. Zuletzt aufgerufen am: 6.6.2021.