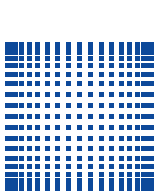


hochschule mannheim

Julius Bartel

Entwicklung robuster Übertragung sensibler Messwerte

Bachelorarbeit

Sommersemester 2021 Betreuer: Dr. Trebbels

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 Microcontroller (µC) zur Weiterleitung von sensiblen Messdaten und die anschließende Weiterleitung der Daten an ein Netzwerk. Die Entwicklung soll an Hand eines Praxisbeispiels demonstriert werden.

Grundlage der Arbeit ist eine 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 µC, 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 einerseits den Transport von Messwerten in die eine Richtung, als auch den Transport von Konfigurationsbefehlen in die andere Richtung ermöglicht.
Als Demonstrationsbeispiel wurde die kapazitive Messung von Feuchtigkeit im Erdreich gewählt. Die gemessenen Daten sollen verarbeitet werden und per Message Queuing Telemetry Transport (MQTT) an ein übergeordnetes Netzwerk weitergeleitet werden.

Abkuerzungsverzeichnis

HAL Hardware Abstraction Layer

IoT Internet of Things *en. Internet der Dinge*

µC Microcontroller

WLAN Wireless Local Area Network *en. Drahtlose Netzwerkverbindung*

RAM Random Access Memory

ADC Analog Digital Convert *en. Analog-Digital Wandler*

JTAG Joint Test Action Group

UART Universal Asynchronous Receiver Transmitter

USART Uinversal Synchronous Receiver Transmitter

USB Universal Serial Bus

MQTT Message Queuing Telemetry Transport

SPI Serial Periphal Interface

I2C Inter-Integrated Circuit

DMA Direct Memory Access

AHB-Bus Advanced High-Perfomance 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

Inhaltsverzeichnis

1 Einführung

1.1 Vorhergegangene Arbeit

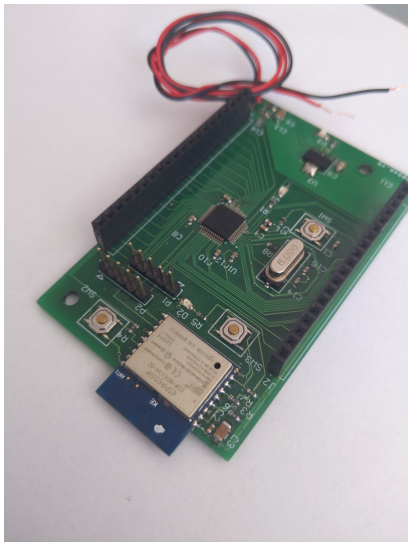


Abbildung 1: *IoT-Gateway [1]*

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

Die Platine verfügt über zwei µC. Einer der Prozessoren, genauer ein *STM32F103C8* ist für das auslesen von Messwerten und die verarbeitung dieser Zuständig, währen 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 µCs bereitzustellen. Desweiteren wurden die nötigen externen Beschaltungen für die beiden Prozessoren entwickelt. Beide Prozessoren verfügen über Status-LEDs. Die General Purpose Input Outputs (GPIOs) des STM32 werden über Buchsenleisten herausgeführt, um den einfachen Anschluss von Erweiterungsplatinen zu ermöglichen.

Die Programmierung des STM32 erfolgt mit Hilfe eines propietären 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
- Implementation der Funktionen in Software
- Test an Hand eines Praxisbeispiels
- Auswertung und Fazit sowie Aussicht

Im Beginn dieser Arbeit sollen zuerst die Grundlagen erklärt werden, auf denen diese Arbeit aufbaut. Diese spannen von den Eigenschaften der µC über die genutzten Protokolle zu diversen Datenstrukturen und Algorithmen.
Anschließend werden die zuvor besprochenen Grundlagen in Software implementiert um sie dann an Hand eines Praxisbeispiels zu testen.

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

2 Grundlagen

2.1 Microcontroller

2.1.1 ESP8266

Der ESP8266 ist ein 32-Bit µC mit einem Systemtakt von 80MHz - 160MHz. Er verfügt über 64kB Random Access Memory (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 Peripherien (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 [2].

Abbildung 2: *ESP8266 WROOM-02*

In diesem Anwendungsfall wird der ESP8266 als Modul eingesetzt (ESP8266-WROOM-02), da dieses bereits über die nötige externe Beschaltung (Oszillator, Flashspeicher, Antenne) verfügt [2].

Die Programmierung erfolgt über Universal Asynchronous Receiver Transmitter (UART) mittels einem USB-Adapter.

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.

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

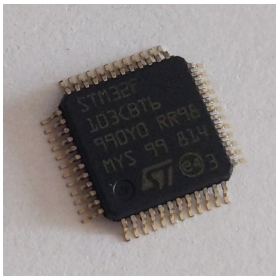


Abbildung 3: *STM32F103C8*

Für die Programmierung des µC ist ein s.g. ST-Link Programmiergerät notwendig, welches auch Debugging ermöglicht.

2.2 Peripherie

Dieser Abschnitt erklärt die verschiedenen genutzten Peripherien. Diese beziehen sich jedoch primär auf die Funktionen des STM32, da diese dort intensiver Nutzung unterliegen, während die einzigen 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 USART-Schnittstellen, welche jedoch auch als UART genutzt werden können [3]. Die dabei genutzten Spannungspegel entsprechen hierbei die der TTL-Logik [3].

Die Einheiten verfügen unter anderem auch über Idle-Line Detection (*Erkennung von Kommunikationsstops*), Duplex und Hardware Flow Control [4].

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 unterstützen Peripherien sind Timer, ADC, Serial Periphal Interface (SPI), I2C und UART [3].

Es ist möglich, Daten von Speicherort zu Speicherort, von einer Peripherie zu einem Speicherort oder von einem Speicherort zu einer Peripherie zu transferieren. Der STM32F103 verfügt über sieben Direct Memory Access (DMA)-Kanäle.

Durch die Nutzung von DMA wird der Prozessor entlastet, da dieser somit nicht mit der Übertragung von Daten blockiert wird. Die Daten werden über eine interne Busmatrix direkt übertragen [5].

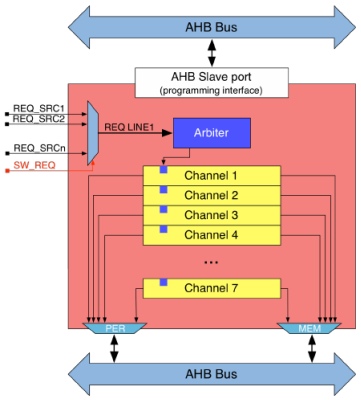


Abbildung 4: *DMA [5]*

Abbildung 4 zeigt den Aufbau eines einzelnen Controllers. Jeder DMA-Controller verfügt über bis zu sieben Kanäle, deren Priorität von s.g. Arbiter verwaltet werden. Der Nutzer kann den verschiedenen Kanälen, die jeweils mit einer Peripherieeinheit verknüpft werden können, Prioritäten zuweisen. Der DMA ist über den s.g. AHB-Bus mit den Peripherieeinheiten und dem Speicher des µC verknüpft [5].

Des weiteren verfügt der DMA-Controller über einen Slave-Port. Mittels dieses Anschlusses lässt sich der DMA-Controller konfigurieren [5].

2.2.3 ADC

Ein Analog Digital Convert *en. Analog-Digital Wandler* ermöglicht die Konversion von analogen Signalen in digitale Werte, um diese dann mittels des µC weiterzuverarbeiten. Der Analog Digital Convert *en. Analog-Digital Wandler* (ADC) des STM32F103 hat eine Auflösung von 12-Bit bei bis zu 16 Kanälen und arbeitet nach dem Prinzip der sukzessiven Approximation [3].

Abbildung 5: *Aufbau [5]*

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.

Des weiteren verfügt der ADC über verschiedene Betriebsmodi [4]:

- 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 [3]. Ausgänge können entweder als Push-Pull oder Open-Drain konfiguriert werden [4].

Die Eingänge können,je nach anliegendem Signal, Interrupts auslösen [4]:

- Steigende Flanke
- Fallende Flanke
- Steigende oder Fallende Flanke

Die interne Beschaltung der GPIO ist Abb. 6 zu entnehmen.

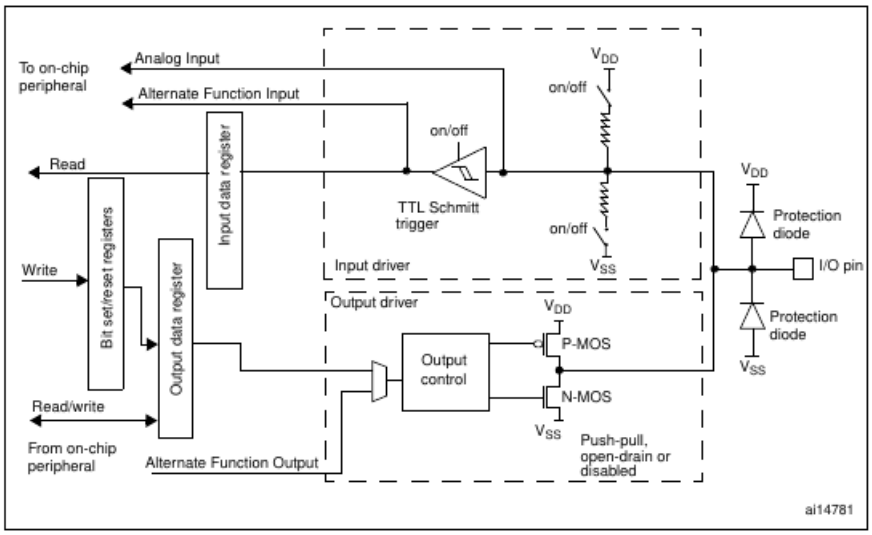


Abbildung 6: *Beschaltung [4]*

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 [3]*

TIM1 ist ein s.g. Advanced-Control Timer, während TIM2,TIM3 und TIM4 s.g. General-Purpose Timer sind.
Advanced-Control Timer implementieren erweiterteFunktionen, wie z.B. dreiphasige PWM oder programmierbare Totzeiten[3].

Abgesehen von den bereits vorgestellten Timern stehen zwei Watchdog-Timer sowie ein s.g. 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 wird, 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

Grundlagen

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 [7].

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.

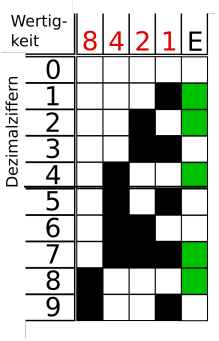


Abbildung 8: *Parität [6]*

Baudrate Die Baudrate, auch Symbolrate genannt, beschreibt die Geschwindigkeit mit der Zeichen übertragen werden.
Ein Baud entspricht hierbei ein Zeichen pro Sekunde [7].
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 [7].

Erklärung Der Universal Asynchronous Receiver Transmitter ist 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 [7]. Die Übertragungsgeschwindigkeit wird in Baud 2.3.1 angegeben.

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

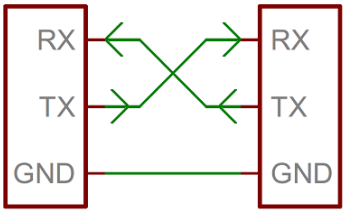


Abbildung 9: *Verbindung [8]*

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 s.g. **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 s.g. Framing *engl.: 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 [9].

Abbildung 10: *Framing [8]*

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' [8]*

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

2.3.2 MQTT

2.4 Datenstrukturen und Algorithmen

2.4.1 Ringbuffer

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

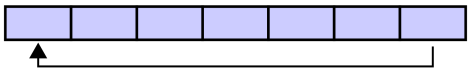


Abbildung 12: *Ringbuffer [10]*

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 s.g. Lese- und Schreibindex. Der Leseindex zeigt die aktuelle Position im Array an, an der gelesen wird, während der Schreibindex anzeigt, bis an welche Stelle neue Daten geschrieben wurden.
Haben Lese- und Schreibindex den selben Wert, wird der Buffer als leer angesehen.

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

2.4.2 Zyklische Redundanzprüfung

Die Zyklische Redundanzprüfung, im englischen *cyclic redundancy check* genannt, 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 [11].

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

CRC nutzt zur Überprüfung der Daten die Polynomdivsion. Die Daten, welche übertragen werden sollen, 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 null, wurde die Nachricht korrekt übertragen [11].

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 [12].

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

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

Neben dem HAL stellt der Hersteller ein weiteres Tool zur Verfügung (Cube MX [13]), welche die schnelle Erstellung des Initialiserungscodes für die Peripherie und sonstige Funktionen ermöglicht. Dies verringert ebenfalls die Fehleranfälligkeit und vereinfacht die Programmierung.
Mittels einer graphischen Oberfläche können den verschieden Anschlüssen des µC Funktionen zugewiesen (z.B. Inter-Integrated Circuit (I2C)) werden und die Timings und Taktfrequenzen konfiguriert werden.

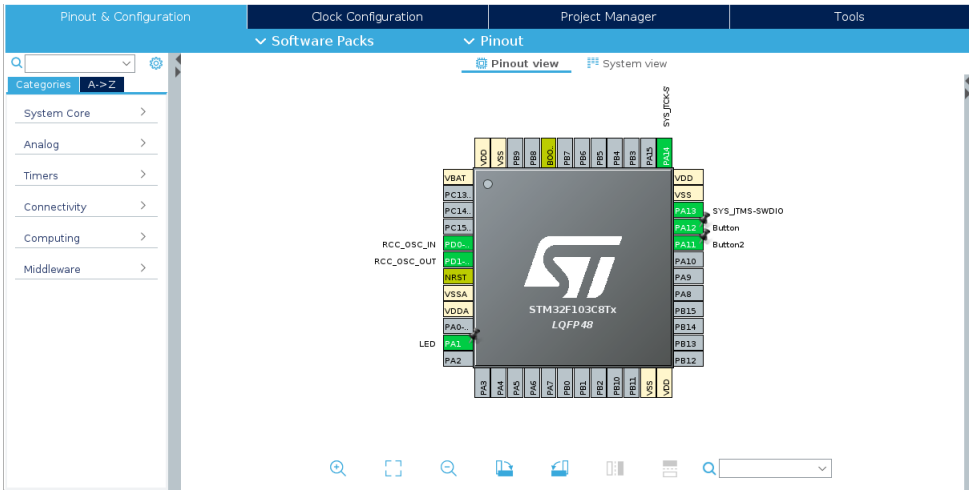


Abbildung 13: *CubeMX*

3 Umsetzung

3.1 Protokoll

Im folgenden soll die Entwicklung des Protokolls erklärt werden. Zuerst wird auf den Algorithmus zur zyklischen Redundanzprüfung eingegangen, anschließend auf den Aufbau der übertragenen Nachrichten.

3.1.1 CRC16-CCITT

Als Implementation der zyklische Redundanzprüfung wurde die Version *CRC16-CCITT* gewählt, da diese bewährt und gut dokumentiert ist. Leider kursieren viele fehlerhafte Implementationen dieses Algorithmus - es wurde jedoch wert darauf gelegt, die Richtige Version zu implementieren.

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 Implementationen beginnen oftmal mit $0xFFFF$.

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

Tabelle 1: XOR

Zwar handelt es sich bei der Berechnung des CRC-Wertes um eine Division, allerdings um eine *Polynom*divsion - 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, in diesem Falle also 16 Stück.

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 [11].

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 [11].

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 wird also direkt mit den Eingangsdaten initialisiert werden. Auch auf ein anhängen der Nullen kann verzichtet werden, da der *«-Operator (left shift)* automatisch am Least Significant Bit (LSB) Nullen anhängt.

Umgesetzt in C Code entsteht folgende Funktion:

Listing 1: *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; //Move Byte into 16Bit CRC Register

    for(uint8_t j = 0; j < 8; j++)
    {
        if((crc & 0x8000) != 0) //Test for MSB
        {
            crc = (crc<<1) ^ poly; //If MSB = 1 shift & XOR
        }
        else
            crc <<= 1;  //If not just shift
    }
}
return crc;
}
```

Der Funktion wird ein Zeiger zu einem Array übergeben, sowie die Länge dieses 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, wobei der Startwert für jedes Byte nach dem ersten die CRC-Prüfsumme des letzten Durchlaufes ist. Für jedes Byte wiederrum müssen, auf Grund der länge eines Bytes, acht mal die Shift- und XOR-Operationen durchgeführt werden.

Mittels des Wertes `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 Informationsgehalt beschreibt.

Ein Telegramm teilt sich in mehrere Teile auf:

- Startzeichen
- Länge
- Nachricht
- CRC-Prüfsumme

Die verschiedenen 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 American Standard Code for Information Interchange (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 desweiteren noch eine ’1’ oder eine ’0’ angehängt, um die Funktion ein- oder auszuschalten.

Soll nun z.B. der Status des Boards abgefragt werden, wird die Nachricht STATUS versendet. Diese Nachricht ist genau 1 Byte lang, deshalb wird als Längeninformation die 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 zu:

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 Sensortyp
- 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 Error
- 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.

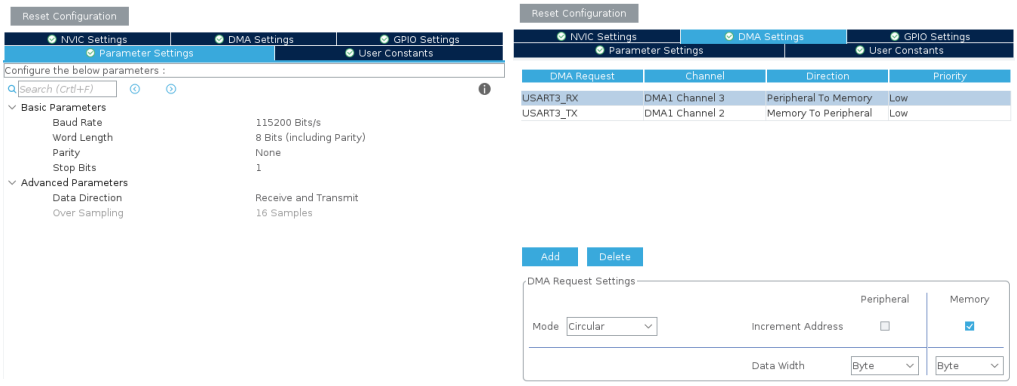
Listing 2: *DMA Strukturvariable*

```
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, welche durch im Falle von `t_flag` durch ein Timeout ausgelöst wurden oder 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 UARTs ist das Format und die Baudrate. Wie in 2 erklärt, 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. 14a).
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, während senderseitig die Daten direkt vom Speicher zum UART weitergeleitet werden. Zudem wird der Empfang von Daten per DMA als Ringbuffer umgesetzt (siehe Abb. 14b).



(a) Baudrate,Format

(b) DMA

Abbildung 14: Konfiguration des UARTs

3.2.3 STM32: Idle Line Detection

Es ist von großem Vorteil, zu erkennen, wenn keine Daten mehr Empfangen werden, um das schreiben von sinnlosen Daten in den Buffer zu vermeiden. Deshalb wird eine s.g. *Idle Line Detection* implementiert, welche erkennt, sobald keine Daten mehr empfangen werden.
Der STM32 bietet dafür einen Interrupt, welcher manuell aktiviert werden muss [4]. Um den Interrupt zu aktivieren, muss während der Initialisierung der Interrupt aktiviert werden:

```
__HAL_UART_ENABLE_IT(&huart3,UART_IT_IDLE);
```

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

Listing 3: *Idle Line Interrupt*

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

Jedes mal, 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 [4].

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 einer 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. Diese Routine ist ebenfalls in der Datei `stm32fxx_it.c` zu finden und wird um folgenden Code erweitert:

Listing 4: *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 nun immer der Zähler des Timeouts dekrementiert, bis er eins 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. Bei der Nutzung von DMA werden Interrupts ausgelöst, wenn der Buffer halb oder ganz voll ist [4]. Desweiteren 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 [12].

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

```
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

```
__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:

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

Es kann nun jedoch dazu kommen, dass nachdem der Buffer voll ist, ein Timeout-Interrupt ausgelöst wird. Um nun falsche Verarbeitung von Daten zu verhindern, wird die aktuelle Position des Buffers auf die Größe des Buffers gesetzt.

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

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

Listing 5: *Abbruch Timeoutinterrupt*

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

Tritt ein Interrupt auf, weil ein Timeout aufgetreten ist, muss unterschieden werden ob im Buffer bereits alte Daten liegen, welche ignoriert werden müssen oder ob der Buffer mit zu verarbeitenden Daten gefüllt ist. Folgender Code implementiert dies:

Listing 6: *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;
}
```

Wenn ein Timeout-Flag aufgetreten ist, 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 noch 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 rückgesetzt.

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 null.

Listing 7: *Berechnung Startposition*

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

Mit Hilfe der berechneten Werten für die Startposition und die Länge der empfangenen Daten werden diese Daten zu guter Letzt in ein dediziertes Array zur Weiterverarbeitung kopiert und ein Flag gesetzt, welches anzeigt, dass neue Daten vorhanden sind:

Listing 8: *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;
```

Mit den Informationen über das Protokoll aus 3.1.2 lassen sich nun aus dem Array in dem die angekommenen Daten gespeichert wurden, die Telegramme 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 null zurückgegeben.

3.2.6 STM32: Versenden von Daten

Das Versenden von Daten geschieht mittels der Funktion [12]

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

Es werden der Handler, welcher dem genutzten UART zugeordnet ist, übergeben, sowie den zu übertragenden Buffer und die Länge des Buffers.

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 welche sich gerade im Sendebuffer befinden überschrieben [12]. 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 bei der Versendung von Daten einen Interrupt, sobald die Daten erfolgreich versendet wurden [12]. 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 9: *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 rückgesetzt - neue Daten können versendet werden.

3.2.7 ESP8266: Empfang und Überprüfung der Daten

Die Implementation für den Empfang von Daten ist simpler gehalten, primär auf Grund des vereinfachenden Arduino-Frameworks. Die Funktion `void receive()` implementiert die Funktionalität des Datenempfangs. Ähnlich wie bei der Implementation des STM32, wird hier ein Timeout genutzt, jedoch auf fortgeschrittene Technologien wie DMA verzichtet.

Folgende Variablen werden innerhalb der Funktion genutzt:

Listing 10: *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.

Desweiteren werden weitere globale Variablen benötigt:

Listing 11: *Globale Variablen receive()*

```
uint8_t receivedChars[BUF_SIZE];
bool newData = false
```

Das Array `receivedChars[]` speichert die empfangenen Daten. Durch die Variable `newData` wird die Verfügbarkeit von neuen Daten angezeigt.

Die Implementation bedient sich diversen Befehlen aus dem Arduino-Framework [14]:

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

Um Funktionen welche im Zusammenhang mit dem UART stehen, nutzen zu können, muss dieser während des Starts des µC mit der Baudrate initialisiert werden [14]:

Listing 12: *Initialiserung 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 13: *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 14: *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:

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

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

Basierend auf der eingelesenen Länge der Nachricht, kann festgestellt werden, wann die CRC-Prüfsumme eingelesen wurde 3.1.2. Mittels der Prüfsumme wird dann die zyklische Redundanzprüfung durchgeführt und bei Erfolg die Variablen der Funktion rückgesetzt und das Flag zum anzeigen von neuen Daten gesetzt. War die Prüfung nicht erfolgreich, wird das Flag nicht gesetzt.

Listing 15: *Zyklsiche 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, wird geprüft ob ein Timeout eingetreten ist. Diese Überprüfung bedient sich der gespeicherten Zeit und der aktuellen Zeit sowie dem Flag `InProg`, welches anzeigt, ob eine Transaktion im Gange ist. Ist die Differenz zwischen aktueller Zeit und der gespeicherten Zeit zu groß und das Flag gesetzt, werden die Variablen `count` und `InProg` rückgesetzt und das Einlesen abgebrochen:

Listing 16: *Abbruch durch Timeout*

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

Abbildungsverzeichnis

Listings

Quellenverzeichnis

[1] Julius Bartel. Entwicklung eines iot-gateways, Wintersemester 2020.

[2] Espressif Systems. Esp-wroom-02 datasheet. URL https://www.espressif.com/sites/default/files/documentation/0c-esp-wroom-02_datasheet_en.pdf. Zuletzt aufgerufen am: 30.4.2021.

[3] ST. Stm32f103x8 datasheet, . URL https://www.st.com/resource/en/datasheet/stm32f103v8.pdf. Zuletzt aufgerufen am: 14.10.2020.

[4] ST. Rm0008 reference manual, . URL https://www.st.com/en/microcontrollers-microprocessors/stm32f103.html#overviewt. Zuletzt aufgerufen am: 30.4.2021.

[5] Carmine Noviello. *Mastering STM32*. Learnpub, 2018.

[6] RosarioVanTulpe. Codetafel – dualergänztes gerades paritätsbit (e = even = gerade). URL https://de.wikipedia.org/wiki/Parit%C3%A4tsbit#/media/Datei:Code_Even_dualergaenzt.svg. Zuletzt aufgerufen am: 4.5.2021.

[7] Klaus Dembowski. *Computerschnittstellen und Bussysteme : für PC, Tablets, Smartphones und Embedded-Systeme*. VDE Verlag, 2016.

[8] Sparkfun. Wiring and hardware. URL https://cdn.sparkfun.com/assets/2/5/c/4/5/50e1ce8bce395fb62b000000.png. Zuletzt aufgerufen am: 6.5.2021.

[9] Helmut Müller. *Mikroprozessortechnik*. Vogel Buchverlag, 2012.

[10] Cburnett. An empty 7-element circular buffer. URL https://en.wikipedia.org/wiki/Circular_buffer#/media/File:Circular_buffer_-_empty.svg. Zuletzt aufgerufen am: 9.5.2021.

[11] Dr. Christof Hübner. Industrielle kommunikationstechnik, Sommersemester 2020.

[12] 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.

[13] ST. Cubemx, . URL https://www.st.com/en/development-tools/stm32cubemx.html. Zuletzt aufgerufen am: 11.5.2021.

[14] Arduino. Arduino language reference. URL https://www.arduino.cc/reference/en/. Zuletzt aufgerufen am: 21.5.2021.