Hochschule Augsburg University of Applied Sciences

Fakultät für Informatik

Bachelorarbeit

Studienrichtung
Technische Informatik

Entwicklung einer embedded Firmwarelösung zur Auswertung von Formgedächtnisdrähten auf Basis eines STM32-Mikrocontrollers

habemus! electronic + transfer GmbH

im Fachgebiet Technische Informatik

Prüfer: Prof. Dr. Hubert Högl

Verfasser:
Abdurrahman Celep
Wellenkampstr.29
80933 München
+49 176 80531782
Abdurrahman.Celep@HS-Augsburg.DE
Matrikelnr.: 2023650

Hochschule für angewandte Wissenschaften Augsburg
An der Hochschule 1
86161 Augsburg
Telefon: +49 (0)821-5586-0
Fax: +49 (0)821-5586-3222
info@hs-augsburg.de

Zusammenfassung

Die Bachelorarbeit beschäftigt sich mit der Aufgabe, eine Firmwarelösung zu entwickeln, die die ausgewerteten Ergebnisse der Formgedächtnisdrähte übernehmen und verarbeiten soll. Im Rahmen dieses Projektes werden die wesentlichen Grundinformationen der Formgedächtnislegierung und die verwendeten Bussysteme der Datenkommunikation sowie die benutzte Messmethode diskutiert. In der Arbeit wird außerdem auf den praxisbezogenen Versuchsaufbau und auf die wesentlichen Hardwarekomponenten eingegangen. Darüber hinaus bezieht sich die Thematik der Firmwarelösung auf den allgemeinen Aufbau und das Anwendungsprojekt. Abschließend können die Ergebnisse dieser Arbeit einen Beitrag zur Forschung auf diesem Gebiet leisten, indem sie einen Ansatz zur Auswertung von Formgedächtnislegierungen liefern.

Abstract

The purpose of this bachelor thesis is to develope a firmware solution which is to take over and process the evaluated results of the shape memory wires. Within the scope of the study the essential basic information of the shape memory alloy and the used bus systems of the data communication as well as the measurement method are discussed. The paper also describes the practical experimental setup and the main hardware components. Furthermore, the subject of the firmware solution refers to the general structure and the application project. To conclude the findings of this paper can contribute to research in this field through providing an approach for the evaluation of shape memory alloys.

Inhaltsverzeichnis

Abkürzungsverzeichnis

ADC	Analog Digital Converter
API	Application Programming Interface
AT	Attention
BSP	Board Support Package
CPHA	Clock Phase
CPOL	Clock Polarity
CS	Chip Select
DMA	Direct Memory Access
FG	Formgedächtnis
FGL	Formgedächtnislegierung
GND	Ground
HAL	Hardware Abstraction Layer
IO	Input Output
LL	Low Level
LSB	Least Significant Bit
MCU	Microcontroller Unit
MISO	Master Input Slave Output
MOSI	Master Output Slave Input
MSB	Most Significant Bit
RISC	Reduced Instruction Set Computer
RTOS	Real Time Operating System
SCLK	Signal Clock
SPI	Serial Peripheral Interface
SS	Slave Select
UART	Universal Asynchronous Receiver Transmitter

Abbildungsverzeichnis

Verzeichnis der Listings

1 Einleitung

1.1 Motivation

Heutzutage werden immer neue Möglichkeiten gesucht, um Aktoren oder auch Sensoren in einem komplexen System zu integrieren. Hierbei bieten sich Formgedächtnislegierung basierte Aktorsysteme bzw. Sensorsystem an, die vielfältig aufgebaut und eingesetzt werden können. Die FGL hat außerdem den Vorteil, dass es kostengünstig ist. Was eine wichtige Rolle spielt, ist die integrierte Sensorfunktion. In der Industrie und in der Forschung sowie in anderen Bereichen wächst die Nachfrage nach neuen Sensorsystemen. Deshalb werden FG-Sensoren in der Zukunft mehr Relevanz bekommen. FG-Sensoren können als Kraft- oder Wegsensoren eingesetzt werden. Andere Funktionsweisen wären auch die Detektion von Ermüdung und Verschleiß oder das Ermitteln von überhörten Lastzuständen. Auch in Bereichen, wo andere Geräte versagen oder keine Sensoren existieren, könnten die Formgedächtnissensoren als eine Lösung eingesetzt werden. Hierbei würden neue Aufgabenfelder entstehen.

1.2 Ziel der Arbeit

Das allgemeine Ziel des Projektes ist, ein Gerät zu entwickeln, das in der Lage ist, eine intelligente Auswertung der Widerstandssignale von FG-Sensordrähten zu ermitteln. Auch das Erkennen von Störgrößen sowie die Kompensation gehört zu den Aufgaben des Bauteils. Ein weiterer Bestandteil ist die Optimierung der FG-Sensordrähte und der elektrischen Kontaktierung hinsichtlich einer Minimierung von Störgrößen.

Die Aufgabe dieser Bachelorarbeit ist die Anfertigung einer Firmware, die die Baukomponente, die das Gerät beinhaltet, ansteuert, sowie eine harmonische Interaktion miteinander hervorhebt. Hierbei wird versucht, Programmkomponenten in einem STM32 Mikrocontroller zu implementieren, die zusammen das Programm abbilden. Das Endprodukt soll in der Lage sein, Werte der FGL einzulesen, umzuwandeln und

dem Benutzer lesbare Daten auszugeben. Gleichzeitig soll die Temperatur des Gerätes gemessen und den Benutzer übermittelt werden. Für die Umsetzung wird mit verschiedenen Bussystemen, die im Kapitel 2.1 erläutert werden, gearbeitet.

1.3 Aufbau der Arbeit

Die Thesis lässt sich in vier wesentliche Abschnitte unterteilen: Einleitung, Grundlagen, Versuchsaufbau und Fazit. In der Einleitung, werden die Motivation der Arbeit sowie das Ziel des Projektes behandelt. Im Grundlagenabschnitt wird auf die allgemeinen Informationen eingegangen. Der praktische Teil befindet sich im Kapitel Versuchsaufbau. In diesem Kapitel werden detailliert die Hardware und die Firmware untersucht. Im Kapitel Ausblick und Fazit werden auf die Vorgehensweise, Erkenntnisse der Arbeit und die Selbstreflexion eingegangen. Außerdem werden die zukünftigen Pläne des Projektes nach der Bachelorarbeit und Verwendungszwecke des Gerätes erwähnt.

2 Grundlagen

In diesem Kapitel werden die notwendigen Grundlagen beschrieben, die für das Verständnis der Arbeit sorgen soll. Dabei wird auf die Datenbussysteme, die Thematik der Formgedächtnislegierung, sowie über die Vierleitermessung eingegangen.

2.1 Bussysteme

Im Unterkapitel Bussysteme werden die Datenleitungen erklärt, die für die Kommunikation und Interaktion in dem Board verwendet werden.

2.1.1 Serial Peripheral Interface (SPI)

Das *Serial Peripheral Interface* ist ein Bussystem, welches 1987 von der Firma Motorola erfunden wurde. Hierbei handelt es sich um eine synchronen, seriellen Datenübertragung nach dem *Master-Slave*-Prinzip. (vgl. C u. a. (1989)) Es ist zu erwähnen, dass dem *Slave* nicht gestattet ist, eine Anfrage zu verschicken, dies ist nur dem *Master* vorbehalten. Während der Kommunikation dürfen die *Slaves* ohne eine Anfrage vom *Master* keine Daten verschicken. Das Taktsignal wird vom *Master* erzeugt, um den *Slave* Teilnehmer zu synchronisieren. Zwischen *Master* und *Slave* existieren maximal vier Leitungen, welche die beiden Komponenten verbinden. (vgl. MACHINE (2021)) Im Folgenden werden die vier Leitungen beschrieben, welches von der Quelle Association (2000) entnommen wurde.

- **SCLK (Signal Clock):** Der *Master* erzeugt einen Takt, um eine Synchronisation zu dem verbundenen Gerät herzustellen
- **MOSI (Master Out- Slave Input):** Der *Master* überträgt die Daten zum *Slave*
- **MISO (Master In- Slave Output):** Der *Slave* überträgt die Daten zum *Master*

- **CS (Chip Select):** Der *Master* stellt eine Kommunikation her, indem er das *CS* des bestimmten *Slaves* anspricht. Bei einem einzigen *Slave* ist der *Chip Select* optional, da für den *Master* keine Auswahlmöglichkeit besteht.

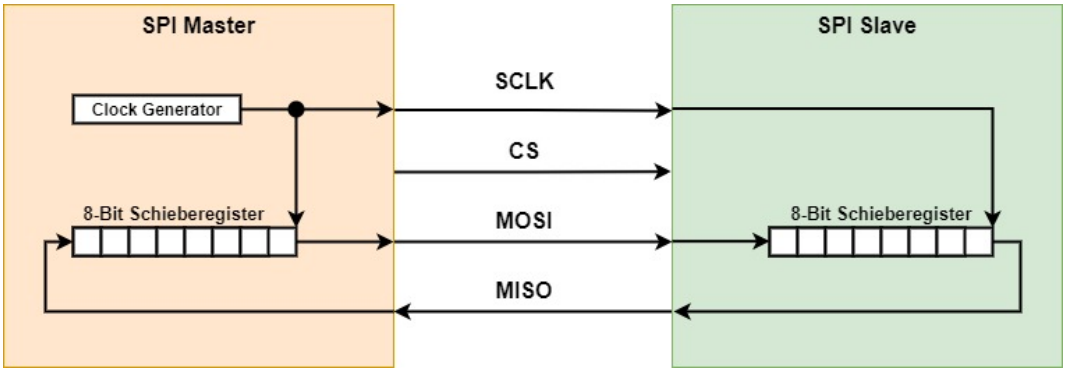


Abbildung 2.1: Datenübertragung zwischen *Master* und *Slave* über ein Schieberegister

Das Besondere an diesem Bus ist, dass die Daten in beide Richtungen gleichzeitig übertragen werden können. Dies wird auch als *full duplex* bezeichnet. Der *SPI* ist auch in der Lage, in *half duplex* zu arbeiten. Dabei werden die Daten abwechselnd und nicht gleichzeitig übertragen, wodurch der *MOSI* und *MISO* eine gemeinsame Leitung teilen können. Eine weitere Eigenschaft ist die Einstellmöglichkeit der Kommunikation. Hierbei sind folgende Einstellungen möglich:

- Auswahl mit welcher Taktflanke die Kommunikation stattfinden soll
- Anzahl von Bits, die übertragen werden sollen
- Die Reihenfolge der Bits

Die SPI-Datenübertragung:

Für eine Datenübertragung wählt der *Master* den *Slave* aus, mit dem er kommunizieren möchte. Dabei wird die spezifische *CS*-Leitung ausgewählt, über die der Pin auf *Masse (LOW)* gesetzt wird. Im nächsten Schritt werden während jedem Taktpuls Werte gleichzeitig über die *MOSI*- und *MISO*-Leitung empfangen und gesendet. Um eine Datenübertragung zu beenden, wird die *CS*-Leitung wieder auf *HIGH* gesetzt. (vgl. Dhaker u. Eckstein (2022))

Eine weitere Besonderheit des *SPI*-Busses ist die Betriebsart, die bei einer Kommunikation verwendet werden kann. Hierbei sind die Modi von den Parametern *Clock Polarity (CPOL)* und *Clock Phase (CPHA)* abhängig.

Tabelle 2.1: Verschiedenen *SPI*-Modi (vgl. MACHINE (2021))

Mode	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

In den meisten Fällen wird der Modus 0 benutzt. Doch für bestimmte Komponenten müssen *CPOL* und *CPHA* geändert werden. Dies kann der Benutzer mithilfe des Datenblattes der Komponente herausfinden.

Um zu verstehen, wie die Modi funktionieren, wird im Folgenden auf die Parameter *CPOL* und *CPHA* eingegangen. Der *CPOL* sorgt für die Polarität des Taktsignals. Hierbei liegt der Ruhestand der *SCLK* Leitung bei 0 auf *LOW* und bei 1 auf *HIGH*. Die *CPHA* sagt aus, in welcher Flanke des Taktsignals die Daten übertragen werden sollen. Der Wert null bedeutet, dass die Daten in der ersten Flanke übernommen werden, eins bedeutet die zweite Flanke. (vgl. MACHINE (2021)) In der unteren Abbildung wird die Betriebsart dargestellt.

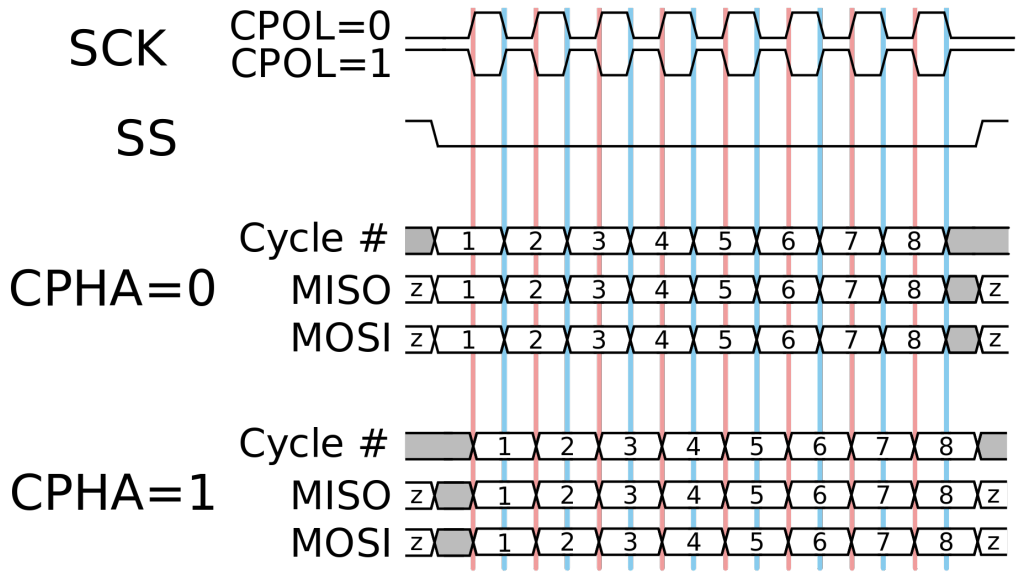


Abbildung 2.2: Überblick der verschiedenen Möglichkeiten für die *SPI*-Kommunikation (Colin M.L. Burnett (2010))

Im Abbildung 2.2 ist zu erwähnen, das der *CS* auch *Slave Select (SS)* genannt und der *SCLK* auch *SCK* beschrieben wird.

2.1.2 Universal Asynchronous Receiver Transmitter (UART)

Die serielle Schnittstelle dient der Kommunikation über eine Datenleitung. Allgemein bildet der *UART* eine Schnittstelle für Computer, Mikrocontroller sowie verschiedene Interfaces. (vgl. KUNBUS) Die Transaktion erfolgt durch zwei Leitungen, die Tx- und Rx-Leitung. Für das Senden der Daten wird die Tx (Transmit)-Leitung verwendet und für das Empfangen die Rx (Receive)-Leitung. Es ist zu erwähnen, dass bei der asynchronen Betriebsweise die Kommunikation kein Taktsignal benötigt. In diesem Fall synchronisiert sich der Empfänger mithilfe einer Rahmenlänge und einer eingestellten Bitrate. Die asynchrone Datenübertragung erfolgt durch ein Start-Stopp-Verfahren. (vgl. KUNBUS)

Die UART-Datenübertragung:

Die *UART*-Kommunikation fängt mit einem Startbit an, gefolgt von Daten-Bits. Im normalen Fall wird eine Größe von 8-Bits verwendet, doch in gewissen Anwendungen kann der Rahmen von 5- bis 9-Bits variiert werden. Als Nächstes kommt optional das Parität-Bit, das für die Erkennung von Übertragungsfehlern dient. Dieses Bit kann auch durch ein Stopbit ersetzt werden. Die Kommunikation endet mit 1 oder 1,5 oder 2 Stop-Bits. (vgl. KUNBUS)

Das *UART*-Protokoll wird in Abbildung 2.3 grafisch verdeutlicht.

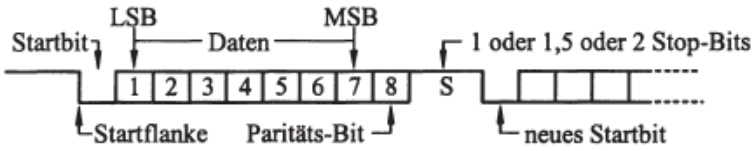


Abbildung 2.3: Standard *UART*-Protokoll mit dem Start-Stopp-Format (Wittgruber (2002), S.40)

Für die Kommunikation über *UART* spielt die Übertragungsrate eine wichtige Rolle. Die Datenübertragungsrate ist in diesem Fall die Bitrate, welche innerhalb einer bestimmten Zeit eine Anzahl von übertragenen Bits definiert. Dabei ist zu beachten, dass die beiden verbundenen Geräte die identische Bitrate enthalten müssen, um eine funktionierende Kommunikation miteinander zu gewährleisten. Die Bitrate wird Bit pro Sekunde (Bit/s) oder bps gemessen. Zu beachten ist, dass die Bitrate mit der Baudrate nicht zu verwechseln ist. Denn die Baudrate hingegen ist die Anzahl der übertragenen Signaleinheiten pro Zeit und hat die Einheit Baud. (vgl. Kompendium)

Die Baudrate bzw. Schrittgeschwindigkeit v_S wird mit folgender Formel berechnet.

$$v_S = \frac{1}{T_s} \quad [in\, Baud = 1/s] \tag{2.1}$$

In einem digitalen Signal wird die kleinstmögliche Zeitdifferenz zweier Zustandsänderungen als Schrittdauer T_S bezeichnet. Die Bitrate bzw. Übertragungsgeschwindigkeit v_D ist das Produkt der Übertragungskanäle m, Anzahl der Möglichkeiten n eines Signals sowie der Baudrate v_S. (vgl. Kompendium)

$$v_D = m \cdot v_S \cdot \log_2 n \quad [Bit/s] \tag{2.2}$$

Da in dem Fall der *UART* einen Übertragungskanal besitzt und die Anzahl der Kennzustände zwei beträgt, ist die Baudrate äquivalent der Bitrate ($v_D = v_S$).

Deswegen gilt,

$Schritt/s = Bit/s = bps(bitpersecond) = 1Baud$

In Tabelle 2.2 werden die normierten Bitraten sowie die Bitdauer tabellarisch dargestellt.

Tabelle 2.2: Auflistung der normierten Bitraten

Bitrate	**Bitdauer**
50 bit/s	20,0 ms
110 bit/s	9,09 ms
150 bit/s	6,67 ms
300 bit/s	3,33 ms
1.200 bit/s	833 μs
2.400 bit/s	417 μs
4.800 bit/s	208 μs
9.600 bit/s	104 μs
19.200 bit/s	52,1 μs
38.400 bit/s	26,0 μs
57.600 bit/s	17,4 μs
115.200 bit/s	8,68 μs
230.400 bit/s	4,34 μs
460.800 bit/s	2,17 μs
921.600 bit/s	1,08 μs
2.000.000 bit/s	500 ns
3.000.000 bit/s	333 ns

2.1.3 One-Wire

Der *One-Wire* oder auch Eindraht-Bus genannt ist eine serielle Schnittstelle, die von der Firma Maxim Integrated erfunden wurde. Mit dem Eindraht-Bus ist nur die Datenleitung *DQ* gemeint und kann für Verwirrung sorgen, da der *One-Wire* auch eine Masse Leitung *GND* besitzt. Bei den meisten *One-Wire* Geräten existiert keine Stromversorgungsleitung. Deshalb entnehmen sie den Strom durch die Datenleitung. Dies wird auch parasitäre Stromversorgung genannt. (vgl. maxim integrated (2008))

Eine Datenübertragung über einen *One-Wire* findet mit einer *half duplex* bidirektionalen Kommunikation statt. Durch die Trennung der Verbindung oder einen Kontaktverlust verfällt der *One-Wire* in einen definierten *Reset*-Zustand. Wenn die Verbindung wiederhergestellt ist, wacht der *Slave* auf und signalisiert dem *Master* seine Anwesenheit. Um Daten über das *DQ* zu verschicken, müssen dementsprechend eigene Protokolle, die aus Einsen und Nullen bestehen, erzeugt werden. Für den Protokollaufbau muss im Datenblatt des *One-Wire* Geräts nachgeschaut werden. (vgl. maxim integrated (2008))

Die One-Wire-Datenübertragung:

Reset Zustand:

Bei jeder Transaktion muss der *One-Wire* neu gestartet werden. Am Anfang wird ein *Reset* Puls erzeugt, indem der *Master* die Leitung mindestens für 480 μs auf *LOW* hält und wieder freigibt. Nach 15 bis 60 μs antwortet der *Slave*, indem die Leitung für 60 bis 240 μs auf *LOW* gesetzt wird, dies wird auch *Presence detect* Puls genannt. Nach dem Signal weiß der *Master*, dass der *Slave* anwesend ist. (vgl. max (2019), S.15)

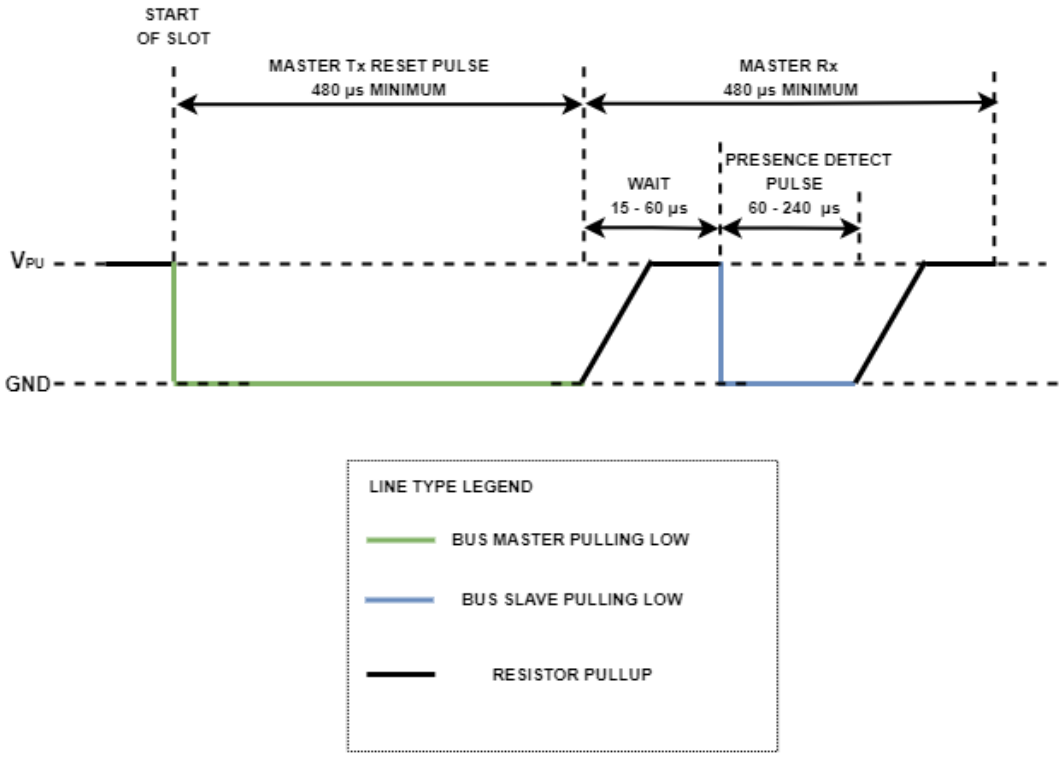


Abbildung 2.4: *One-Wire Reset* Protokoll nach dem Standard von Maxim Intergrated (vgl. max (2019), S.15)

Schreib Zustand:

Um über den *One-Wire* eine 1 zu verschicken, hält der *Master* die Leitung 1 bis 15 μs auf *LOW*. Anschließend wird für die restliche Zeit die Leitung auf *HIGH* gesetzt. Für die Übertragung einer 0 wird die Leitung vom *Master* für 60 bis 120 μs auf *LOW* gehalten und letztendlich losgelassen. Während dem Schreibvorgang tastet sich der *Slave* 30 μs ab. So erkennt der Baustein, ob eine 1 oder 0 gesendet wurde. (vgl. max (2019), S.15)

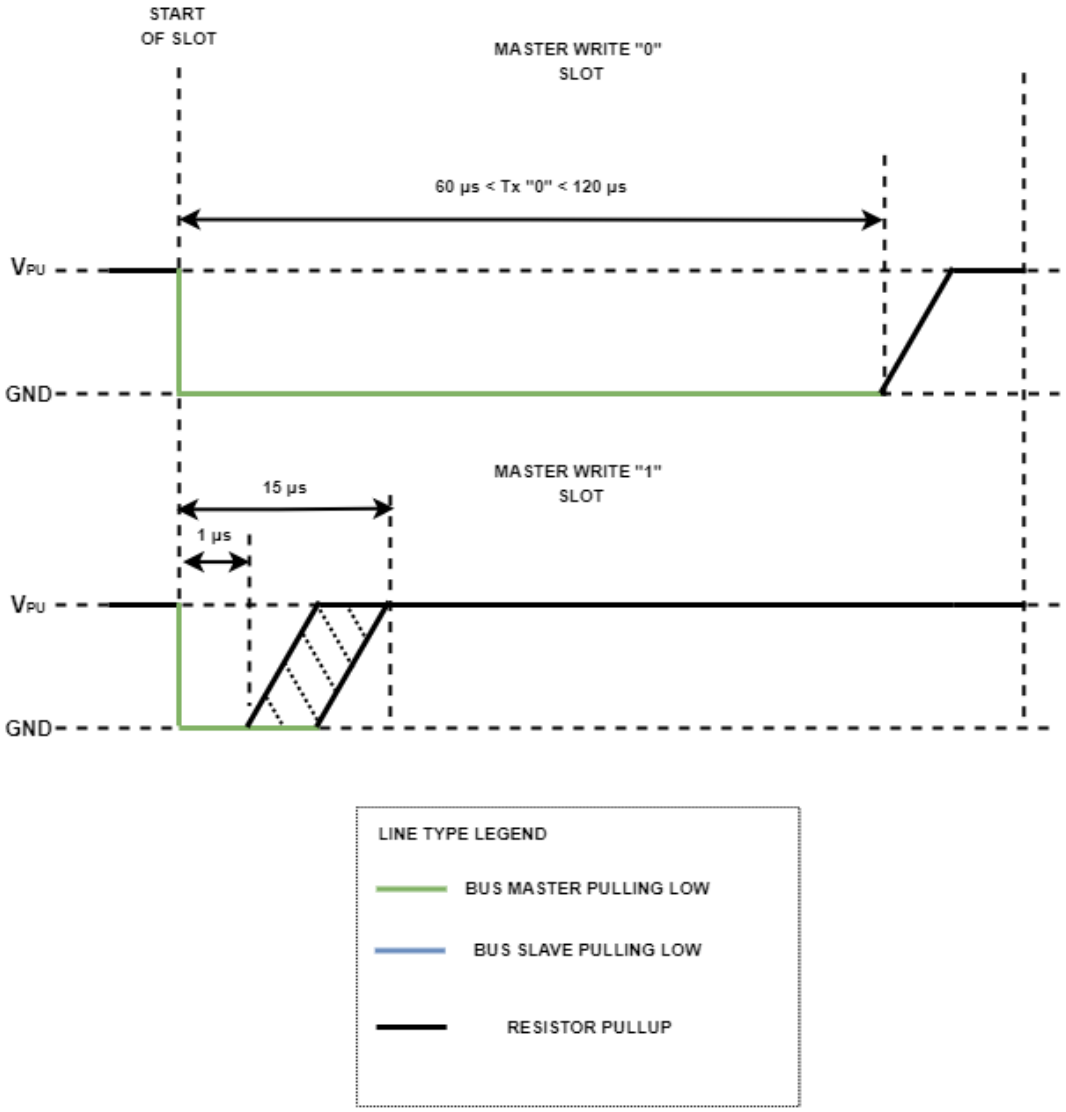


Abbildung 2.5: *One-Wire Write* Protokoll nach dem Standard von Maxim Intergrated (vgl. max (2019), S.16)

Lese Zustand:

Beim Lesen setzt der *Master* die Leitung für 1 μs auf *LOW*. Danach gibt der *Master* die Leitung frei und versucht die *DQ* zu lesen. Wenn der *Slave* die Leitung ein paar Mikrosekunden auf *HIGH* setzt, wird eine 1 gelesen. Doch wenn der *Slave* die Leitung auf *LOW* hält und nach 45 μs loslässt, so wird eine 0 erkannt. (vgl. max (2019), S.16)

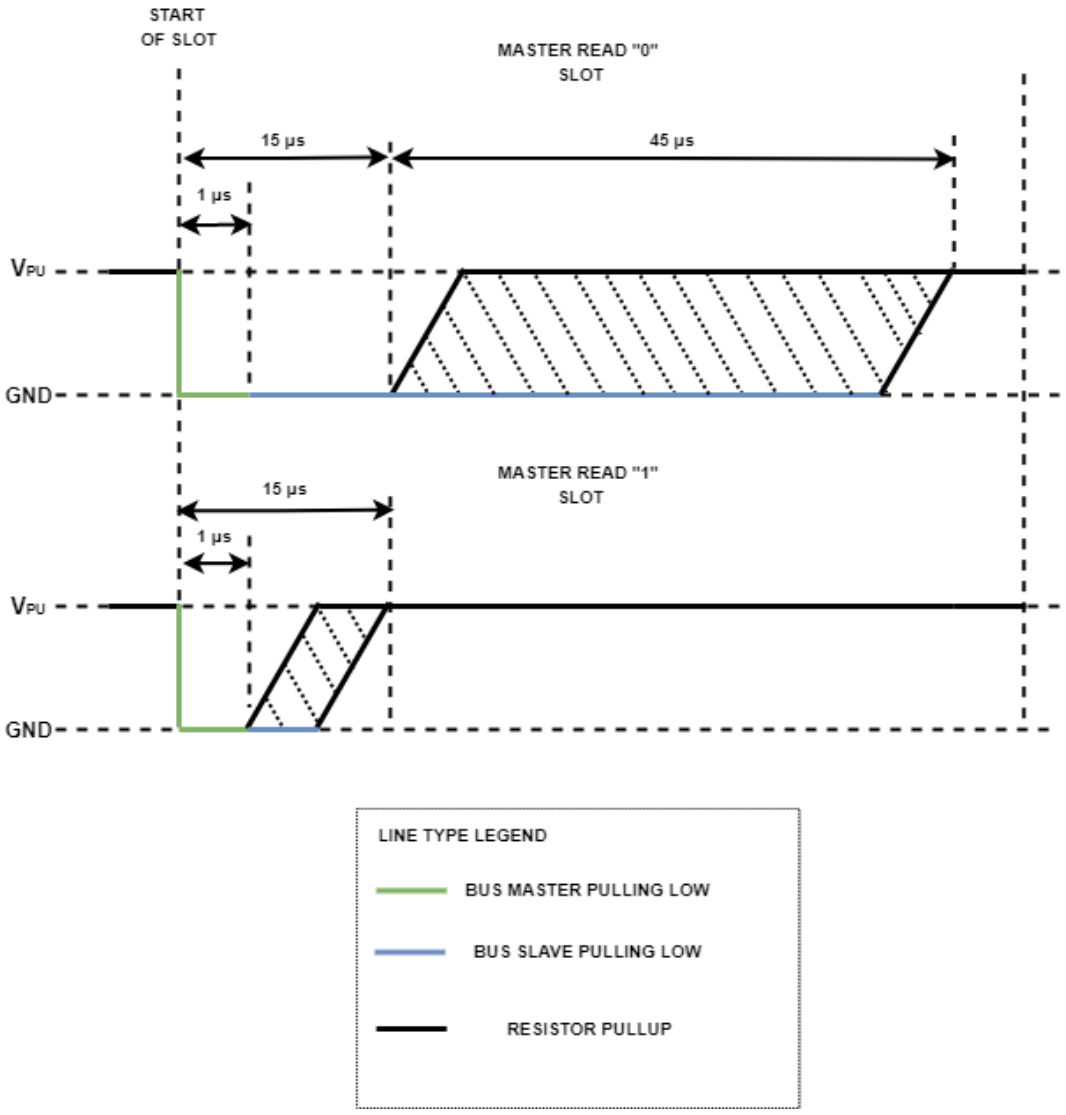


Abbildung 2.6: *One-Wire Read* Protokoll nach dem Standard von Maxim Intergrated (vgl. max (2019), S.16)

2.2 Direct Memory Acess (DMA)

Der DMA ist eine Zugriffsart, welche über ein Bussystem direkt auf den Speicher zugreift. Die Informationen wurden von der Quelle DeepBlue (2021) entnommen.

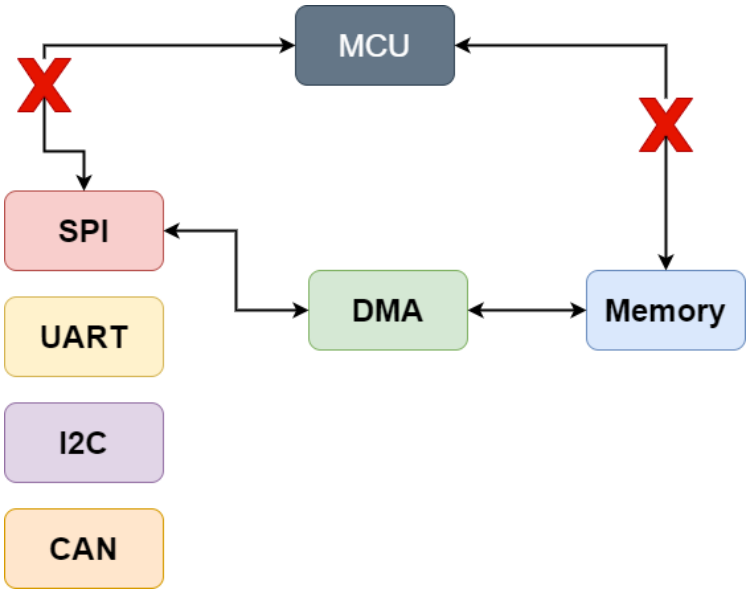


Abbildung 2.7: DMA-Kommunikation anhand des *SPI*-Busses

In der Abbildung 2.7 wird ein Beispiel der Datenkommunikation mithilfe der *DMA* dargestellt. Durch das Zugreifen der Daten, über den *SPI*-Bus, wird der Umweg über den Mikrocontroller vermieden und direkt in den Speicher geschrieben. Dadurch entsteht eine direkte Kommunikationsverbindung zwischen dem Speicher und den angeschlossenen Peripheriegeräten. Dies dient dazu, den Prozessor zu entlasten, sowie eine schnellere Datenübertragung herzustellen. Der *DMA* ist auch in der Lage eine Kommunikation zwischen zwei Speichern herzustellen.

Eine *Microcontroller Unit* kann mehrere *DMA*-Controller besitzen. Für eine Kommunikation gibt es die DMA-Channels. Hierbei wurde vom Hersteller definiert, für welche Peripherie, welcher Kanal verwendet wird. In der Tabelle 2.3 wird für jeden Kanal die Anforderungen der DMA1 dargestellt.

Tabelle 2.3: Zusammenfassung der DMA1 Anforderungen für jeden Kanal für den STM32F103rb (STM (2021), S.282)

Peripherals	**Channel 1**	**Channel 2**	**Channel 3**	**Channel 4**	**Channel 5**	**Channel 6**	**Channel 7**
ADC1	ADC1	-	-	-	-	-	-
SPI/I2C	-	SPI1 (RX)	SPI1 (TX)	SPI2/I2S2 (RX)	SPI2/I2S2 (TX)	-	-
USART	-	USART3 (TX)	USART3 (RX)	USART1 (TX)	USART1 (RX)	USART2 (RX)	USART2 (TX)
I2C	-	-	-	I2C2 (TX)	I2C2 (RX)	I2C1 (TX)	I2C1 (RX)
TIM1	-	TIM1 (CH1)	-	TIM1 (CH4) TIM1 (TRIG) TIM1 (COM)	TIM1 (UP)	TIM1 (CH3)	
TIM2	TIM2 (CH3)	TIM2 (UP)	-	-	TIM2 (CH1)	-	TIM2 (CH2) TIM2 (CH4)
TIM3	-	TIM3 (CH3)	TIM3 (CH4) TIM3 (UP)	-	-	TIM3 (CH1) TIM3 (TRIG)	-
TIM4	TIM4 (CH1)	-	-	TIM4 (CH2)	TIM4 (CH3)	-	TIM4 (UP)

Aus der Tabelle kann außerdem entnommen werden, dass für das Empfangen sowie das Senden der Daten jeweils unterschiedliche Kanäle benötigt werden.

2.3 Formgedächtnislegierung

Das Thema FGL ist der Hauptbestandteil dieser Arbeit. Um zu verstehen, was eine Formgedächtnislegierung ist, wird in den kommenden Unterthemen genauer auf die Thematik eingegangen.

2.3.1 Kristallstruktur

„Eine Kristallstruktur [wird] als eine Struktur [definiert], die durch periodische Wiederholung eines Bausteins, der sogenannten Einheitszelle [auch bezeichnet als Elementarzelle], aufgebaut werden kann.“ (Bäker (2014), S.32)

Was die Struktur eines Kristalls ist, wird mit Hilfe des Gitters sowie Basis/Motiv verständlicher. Ein Kristallgitter ist eine dreidimensionale Anordnung, die aus mathematischen Punkten im Raum besteht. Die Elementarzelle, aus der das Gitter aufgebaut ist, enthält alle relevanten Information des Kristalls. Die Basis oder auch Motiv genannt, besteht in diesem Fall aus Molekülen oder Atomen, welche sich in den Elementarzellen befinden. (vgl. Hoffmann (2016), S.28-30)

Formgedächtnislegierungen sind Metalle, die sich dadurch Auszeichnen, eine Kristallstruktur zu besitzen. (vgl. Bäker (2014), S.31-32) Hierbei handelt es sich um eine Fernordnung, da innerhalb eines kristallinischen Festkörpers eine periodische und regelmäßige Anordnung von Molekülen bzw. Atomen besteht. (vgl. Bäker (2014), S.2-3) Durch die Veränderung der Kristallstruktur entsteht der Formgedächtniseffekt. (vgl. Bäker (2014), S.32)

Insgesamt existieren 14 Arten von Kristalltypen, die auch als *Bravais-Gitter* bezeichnet werden. Also gibt es 14 verschiedene Möglichkeiten, die Atome in einem Kristallgitter anzuordnen. (vgl. Bäker (2014), S.32)

In der unteren Abbildung 2.8 wird die jeweiligen Typen dargestellt.

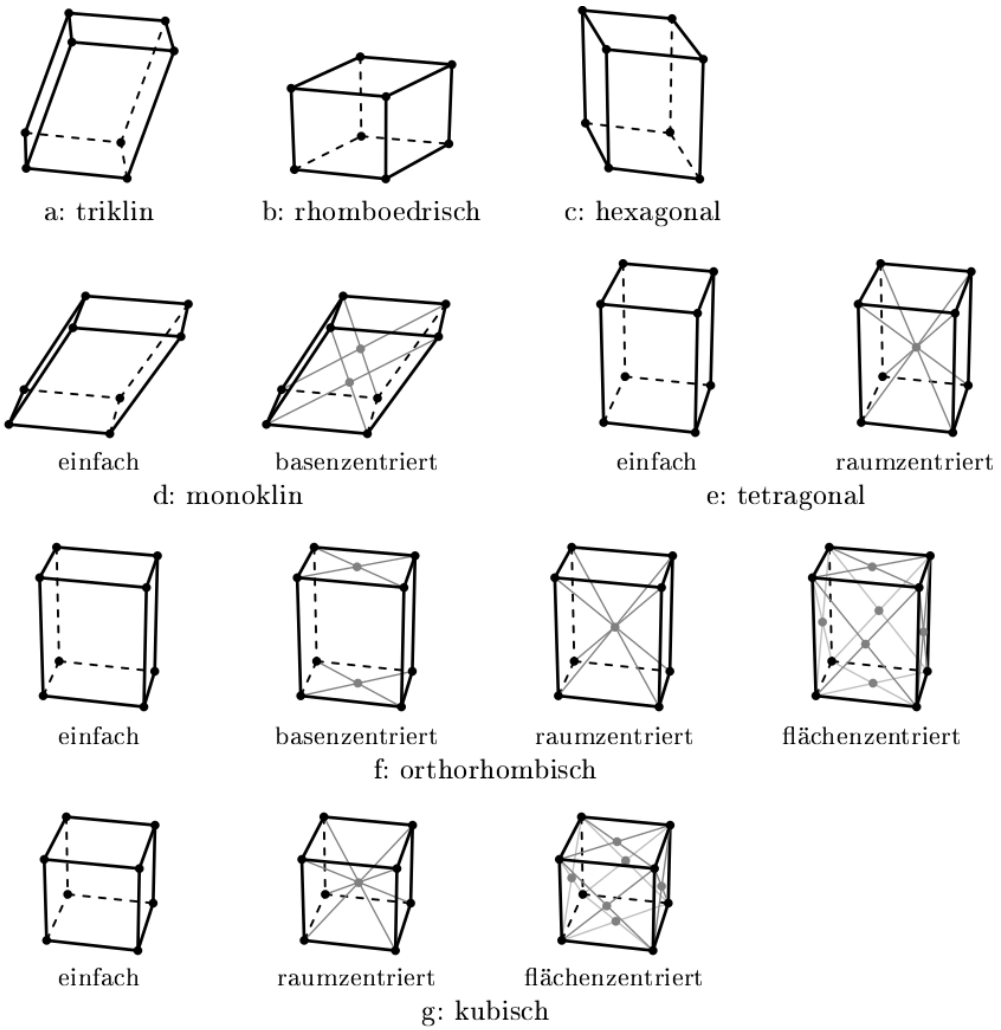


Abbildung 2.8: Einheitszellen der 14 möglichen Kristalltypen (Bäker (2014), S.32)

In der Formgedächtnislegierung sind zwei Kristallstrukturen wichtig: Das Austenit und Martensit.

Austenit:

Die Legierung geht bei hohen Temperaturen in die kubisch flächenzentrierte (fcc) Austenit-Phase über. (vgl. Kock (2010), S.1) Die Struktur besitzt eine hohe Verformbarkeit und kann eine maximale Bruchdehnung von 40 bis 50 % aufweisen. (vgl. BorTec) Austenitische Stähle werden für die Chemische Industrien sowie in der Meerestechnik eingesetzt. Auch in Apparate-, Behälterbau und Rohrleitungen sind sie vorhanden. (vgl. Gümpel (2000), S.27)

Martensit:

Beim Abkühlen entsteht die martensitische Kristallstruktur. Diese hat die Form eines flächenzentrierten Tetragonal (fct). (vgl. Kock (2010), S.1) Das Martensit kommt in

legierten und unlegierten Stählen sowie in gewissen Nichteisen-Metallen vor. (vgl. Hornbogen (1984), S.741-746) Eine wichtige Besonderheit liegt darin, dass Martensit eine sehr stabile Kristallstruktur aufweist. Anwendungsbereiche von martensitischen Stählen sind beispielsweise in der Erdöltechnik und Kältetechnik. Die Stähle werden auch für gewöhnliche Messer sowie Bremsscheiben und für die Konstruktionsteile in Wasserkraftwerden verwendet. (vgl. Gümpel (2000), S.22)

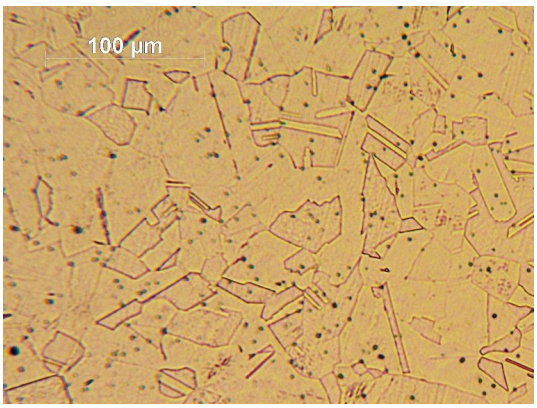


Abbildung 2.9: Austenitische Struktur von SAE 304 (Cr Ni 18 10) Stahl (itwiki (2014))

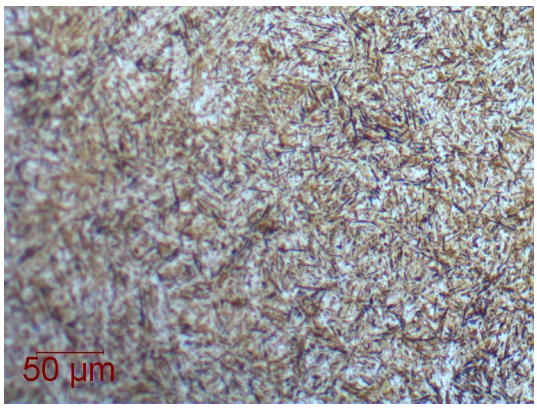


Abbildung 2.10: Martensit Struktur von SAE 1045 Stahl (koganam (2007))

Um das Verhalten der beiden Strukturen zu verstehen, wird im Thema 2.3.2 darauf eingegangen.

2.3.2 Phasenübergänge

„Für die meisten Phasenübergänge ist es notwendig, dass Atome durch das Kristallgitter diffundieren, also Strecken von mehr als einer Gitterkonstante Länge zurücklegen.“ (Bäker (2014), S.38)

Um genau zu verstehen, wie genau ein Phasenübergang funktioniert, wird Beispielweise ein Übergang von flüssigen zum festen Zustand betrachtet. Dabei wird kurz in die Thematik des Boltzmann-Gesetztes sowie die Entropie eingegangen.

Boltzmann-Gesetz:

Das Gesetzt von Boltzmann beschreibt, die Wahrscheinlichkeit p_i, eines gegebenen Systems, in einem bestimmten Zustand Z_i anzutreffen, welches Beispielsweise durch den Kontakt mit einem Wärmebad entsteht. Dadurch findet ein Wärmeaustausch statt, der dazu führt, dass das System die gleiche Temperatur wie das Wärmebad besitzt. Dies wird als thermisches Gleichgewicht bezeichnet. (vgl. Bäker (2014), S.33)

Die dazugehörige Formel lautet:

$$p_i = \frac{1}{Z} e^{-\frac{E}{k_B T}} \tag{2.3}$$

Laut des Boltzmann-Gesetzes hängt die Wahrscheinlichkeit p_i des Endzustands eines Systems von der Energie E und der Temperatur T ab. (vgl. Bäker (2014), S.34) „Dabei ist Z ein Normierungsfaktor, der dafür sorgt, dass die Gesamtwahrscheinlichkeit, das System in irgendeinem Zustand anzutreffen, gleich Eins ist.“ (Bäker (2014), S.33) Die Boltzmann-Konstante k_B beträgt $1,3807 * 10^{-}23$ J/K. (vgl. Bäker (2014), S.33)

$$Z = \sum_{Zustände} e^{-\frac{E_{Zustände}}{k_B T}} \tag{2.4}$$

Durch die obere Formel wird erkannt, dass bei hoher Energie die Wahrscheinlichkeit der Zustände sinkt. Mit steigender Temperatur steigt die Wahrscheinlichkeit der Systemzustände. Bei einer Temperatur von null nimmt das System den energetisch günstigen Zustand an. (vgl. Bäker (2014), S.34)

Entropie:

„Die Entropie stellt eine Verbindung zwischen der mikroskopischen Anordnung der Atome eines Systems und den makroskopisch beobachteten Größen her.[...] Die Entropie eines Systems ist ein Maß dafür, wie viele Möglichkeiten es gibt, einen bestimmten makroskopischen Zustand (oder kurz Makrozustand) durch Anordnung der einzelnen Atome (also durch unterschiedliche Mikrozustände) zu erreichen.“ (Bäker (2014), S.35)

Je mehr ein Zustand an Mikrozuständen M gewinnt, desto mehr steigt die Entropie S des Systems.

Die dazugehörige Formel lautet

$$S = k_B \ln M \tag{2.5}$$

Ein weiterer wichtiger Punkt ist die freie Energie F. Dabei wir die innere Energie U und die Entropie S eines Systems betrachtet.

$$F = U - TS \tag{2.6}$$

Durch die Temperatur T und die Entropie S wird die freie Energie beeinflusst. Dies wird durch den Formelanteils ***-TS*** abgebildet. Die Entropie hat einen geringeren Einfluss, wenn die Temperatur niedrig ist. Doch erst wenn das System erwärmt wird, minimiert sich die freie Energie, da das Produkt der Entropie und Temperatur dominiert. (vgl. Bäker (2014), S.37)

Zusammenfassung:

Laut dem Boltzmann-Gesetzes nimmt die Wahrscheinlichkeit der Zustände eines Systems bei steigender Temperatur zu. Dabei wird die freie Energie kleiner, da der Einfluss der Entropie dominiert. In diesem Fall stellt sich ein Zustand ein, wodurch die Moleküle beweglicher werden. Es wird davon ausgegangen, dass bei einer hohen Temperatur die Legierung sich in einer flüssigen Phase befindet. Bei sinkender Temperatur bilden sich Flüssigkristalle (Eine Substanz, die einen flüssig und festen Eigenschaft aufweist). Dies sorgen dafür, dass zunächst die freie Energie minimal gehalten wird. Beim Abkühlen sinkt die Entropie, da die Orientierung der Moleküle kleiner werden. Doch durch die Beweglichkeit der Moleküle in der flüssigkristallinen Phase ist die Entropie größer als im festen Zustand. Dies führt dazu, dass die freie Energie den kleinstmöglichen Wert im festen Zustand erhält, was man in der unteren Abbildung entnehmen kann. (vgl. Bäker (2014), S.37)

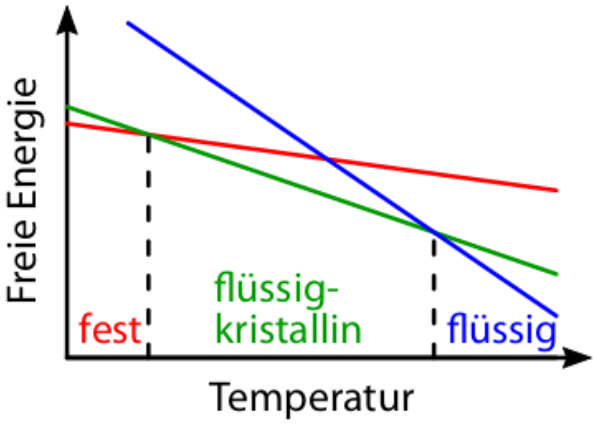


Abbildung 2.11: Verhalten der freien Energie während der Phasenübergang (Bäker (2014), S.38)

In Abbildung 2.11 wird erkannt, das bei einer niedrigen Temperatur die freie Energie im festen Zustand den kleinsten Wert besitzt. Dies liegt darin, da die innere Energie in der festen Phase kleiner ist als in der flüssigen Phase. Bei einer hohen Temperatur ist die freie Energie in der flüssigen Phase kleiner als in der festen Phase, da die Entropie dominiert. (vgl. Bäker (2014), S. 38)

Ein Phasenübergang in der Formgedächtnislegierung wird im festen Zustand durchgeführt. (vgl. Bäker (2014), S.38) Wie im Kapitel 2.3.1 beschrieben geht die Formgedächtnislegierung bei niedriger Temperatur in die Martensit-Phase und beim Erhitzen in die Austenit-Phase über.

In der Abbildung 2.12 wird das Verhältnis zwischen der freien Energie und der Temperatur sichtbar.

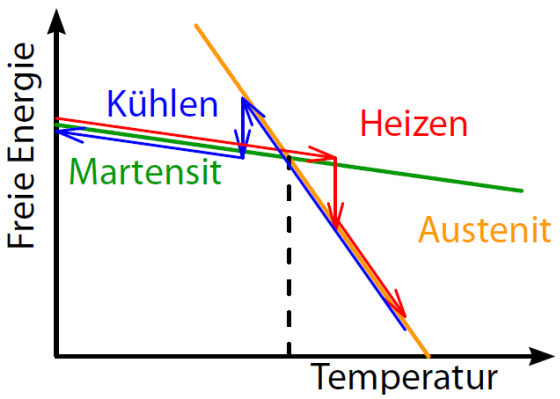


Abbildung 2.12: Freie Energie einer Formgedächtnislegierung in Abhängigkeit von der Temperatur (Bäker (2014), S.44)

Die freie Energie sinkt in der martensitischen Phase langsam und erst in der austenitischen Phase rapide nach unten. Beim Abkühlen der Legierung steigt die freie Energie schnell und nach der Martensit-Phase langsam nach oben. Was im Abbildung 2.11 auch zu erkennen ist, dass bei dem Übergang vom Martensit zu Austenit und umgekehrt, eine verzögerte Umwandlung stattfindet. Dies wird durch die Hysterese in Abbildung 2.13 deutlicher. (vgl. Bäker (2014), S.44)

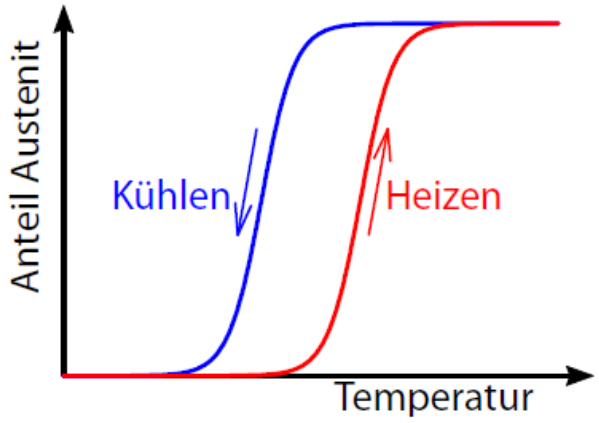


Abbildung 2.13: Hysterese der Austenit Anteil in Abhängigkeit der Temperatur (Bäker (2014), S.44)

Die Entstehung dieser Hysterese liegt daran, dass die Entropie sowie die innere Energie nicht von der Temperatur abhängig ist. Ein Beispiel wäre das langsame Runterkühlen vom reinen Wasser. Das Wasser kann sich auch unterhalb von 0 °C in einem flüssigen Zustand befinden, da die Bildung von Eiskristallen nur an Kristallisationskeimen erfolgt. (vgl. Bäker (2014), S.44)

2.3.3 Formgedächtnis-Effekt

Der Formgedächtnis-Effekt wird in drei Arten unterteilt.

- Ein-Weg-Effekt
- Zwei-Weg-Effekt
- Superelastizität

Dabei enthält jede Art ein Spannungs-Dehnungs-Diagramm in Abhängigkeit der Temperatur T. Die Fließspannung ist in diesem Fall σ und die Dehnung ε.

2.3.3.1 Ein-Weg-Effekt

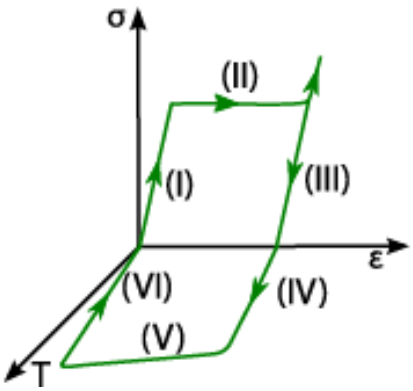
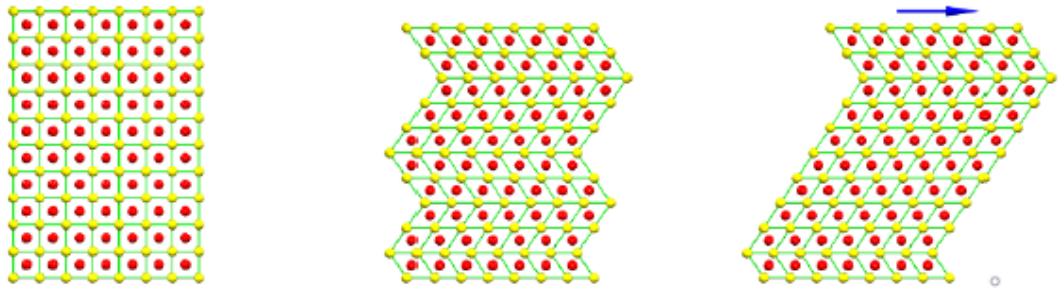


Abbildung 2.14: Spannungs-Dehnungs-Diagramm vom Ein-Weg-Effekt (Bäker (2014), S.30)

Wie in Abbildung 2.14 dargestellt, wird im Bereich I der FGL in einem kalten Zustand elastisch gedehnt. Hierbei wird erkannt, dass die Spannung und Dehnung ein lineares Verhältnis mit einer großen Steigung besitzt. Nach einer bestimmten Spannung, nimmt die Steigung der Dehnung langsam zu. Im Bereich II, wird eine plastische Verformung durchlaufen. Dies wird nach dem Knick der Kurve deutlich. Nach der Entlastung, die im Bereich III zu erkennen ist, geht die Formgedächtnislegierung nicht in ihre Ursprungsform zurück, sondern bleibt in einer neuen Form bestehen. Wenn anschließend eine Energie zugeführt wird, wie im Bereich IV zu sehen ist, entsteht zuerst eine thermische Ausdehnung des Materials. Anschließend geht die Legierung rapide in ihre Ursprungsform zurück (Bereich V). Erst wenn eine Abkühlung stattfindet, verschwindet die thermische Dehnung des FGLs (Bereich VI). (vgl. Bäker (2014), S.30)

Im Bereich IV bis VI findet der Formgedächtnis-Effekt statt. „Überschreitet die Temperatur einen bestimmten Wert, dann »erinnert« sich das Material an seine ursprüngliche Form und verformt sich ohne weitere Krafteinwirkung wieder in die Ausgangsform zurück [...].“ (Bäker (2014). S.30)

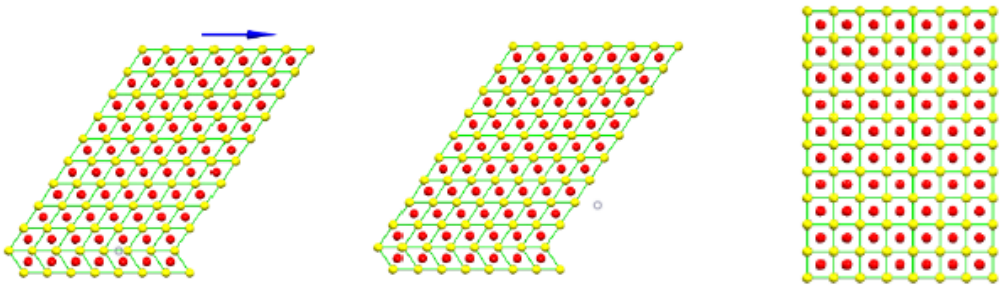
Um genauer zu verstehen, wie das funktioniert, spielt die Martensit-Phase und die Austenit-Phase eine wichtige Rolle. Beim Erhitzen geht die Legierung in die Austenit-Phase über. Nach einer Abkühlung verformt sich das Gitter in Martensit. Wenn das Material belastet wird, klappt sich das Gitter um. Nach der Entlastung bleibt die Endstruktur erhalten. Es wird wieder Austenit gebildet, wenn anschließend das FGL erhitzt wird. (vgl. Bäker (2014), S.41)



(a) Austenit-Phase (hohe Temp.)

(b) Martensit-Phase (niedrige Temp.)

(c) Belastung: Umklappen der Martensit-Phase



(d) Belastung: Umklappen der Martensit-Phase

(e) Entlastung: Struktur bleibt erhalten

(f) Aufheizen: Austenit-Phase bildet sich

Abbildung 2.15: Veränderung der Gitterstruktur im Ein-Weg-Effekt (Bäker (2014), S.41)

„[...] Bei Belastung der Martensitphase klappen einzelne Bereiche des Gitters um. Beim Aufheizen bildet sich wieder Austenit. [...]“ (Bäker (2014), S.41)

2.3.3.2 Zwei-Weg-Effekt

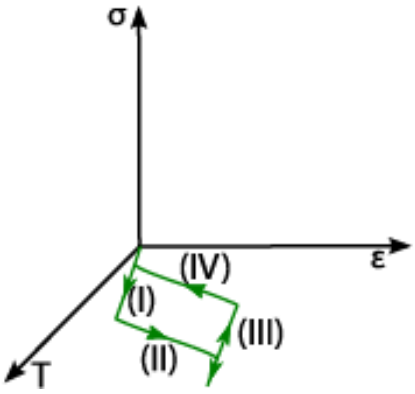


Abbildung 2.16: Spannungs-Dehnungs-Diagramm vom Zwei-Weg-Effekt (Bäker (2014), S.30)

Was in Abbildung 2.16 entnommen werden kann ist, dass bei der Formgedächtnislegierung nur eine thermische Veränderung vorgenommen wird und nicht gedehnt wird. Hierbei wird im Zwei-Weg-Effekt die mechanische Spannung nicht angelegt sondern nur die Legierung erhitzt.

Am Anfang dehnt sich das Material nach einer steigenden Temperatur, thermisch aus (Bereich I). Nach einer bestimmten Erhitzung wird die Legierung stark verformt bis zu einer gewissen Dehnung (Bereich II). Bei Abkühlung nimmt die thermische Dehnung ab, wie in Bereich III dargestellt. Wenn anschließend bei der Abkühlen eine gewisse Temperatur erreicht wurde, geht das Material in die ursprüngliche Form zurück (Bereich IV). (vgl. Bäker (2014), S.31)

In diesem Effekt ist die Besonderheit, dass das Material bei einer hohen Temperatur eine andere Form besitzt als in abgekühltem Zustand. Die Formgedächtnislegierung wird durch einen thermomechanischen Behandlungszyklus »trainiert«, um bei einer Abkühlung wieder in die definierte Form zurückzukehren. (vgl. Bäker (2014), S.43) Dies kann sich in bestimmten Bereichen als nützlich erweisen, was im Kapitel 2.3.5 beschrieben wird.

2.3.3.3 Superelastizität

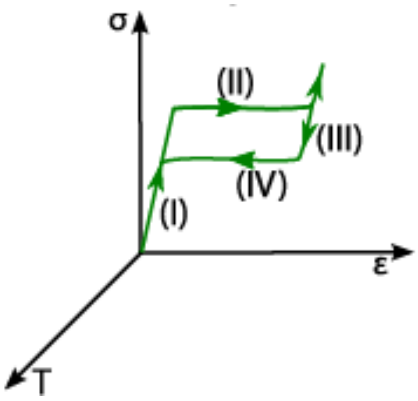
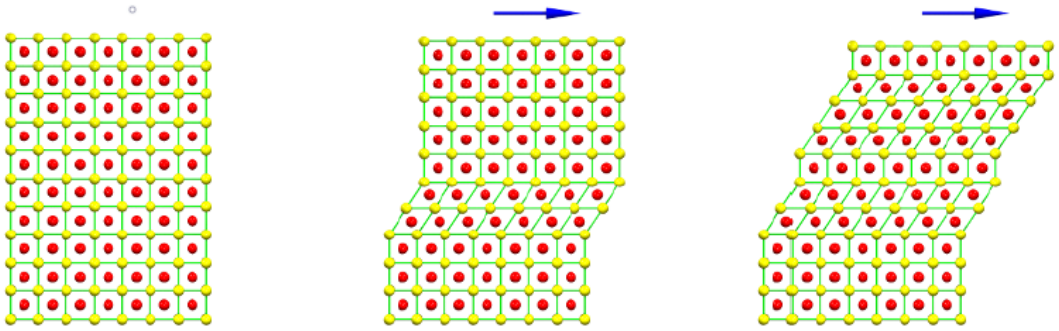


Abbildung 2.17: Spannungs-Dehnungs-Diagramm von Superelastizität (Bäker (2014), S.30)

Bei der Superelastizität wird die Temperatur, im Gegensatz zum Ein-Weg-Effekt, nicht verändert, sondern konstant gehalten. Dabei wird das Material belastet (Abbildung 2.17).

In Bereich I und Bereich II verhält sich die Legierung identisch zum Ein-Weg-Effekt. Nach der Entlastung dehnt sich das Material erst leicht (Bereich III) und anschließend stark (Bereich IV) in die Ausgangsform zurück. (vgl. Bäker (2014), S.31)

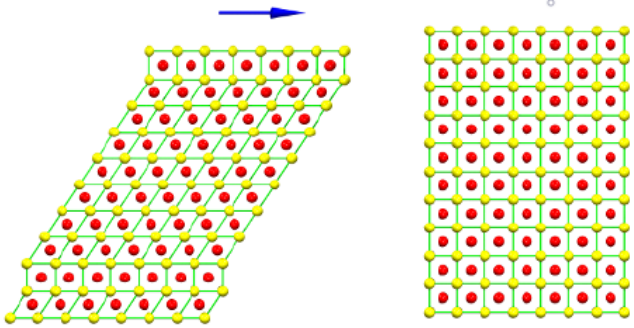
In diesem Fall wirkt auf die FGL eine andauernde konstante Temperatur. Mithilfe eines Stromflusses, welches durch die Legierung fließt, ist es möglich, diese konstante Temperatur zu erzeugen. (vgl. Czechowicz u. Langbein (2000), S.5) Wie in Abbildung 2.18 geht nach jeder Belastung, die Formgedächtnislegierung in die Ursprungsposition zurück. Dies entsteht dadurch, das sich die Legierung im Ursprung in einer Austenit-Phase befindet und während der Belastung in einer Martensit-Phase übergeht. Dabei klappt sich, bei einer wachsenden Spannung, das Gitter weiter um. Beim Entlasten kehrt das Material, wegen der konstanten Temperatur in die Austenit-Phase zurück. (vgl. Bäker (2014), S.42)



(a) Beginn: Austenit-Phase

(b) Belastung:Umklappen in Martensit-Phase

(c) Belastung: Umklappen der Martensit-Phase



(d) Belastung: Umklappen der Martensit-Phase

(e) Entlastung: Umklappen in Austenit-Phase

Abbildung 2.18: Veränderung der Gitterstruktur im Superelastizität (Bäker (2014), S.42)

2.3.4 Elektrischer Widerstand der FGL

Die Information über den elektrischen Widerstand des FGLs wurde von der Quelle Czechowicz u. Langbein (2000) entnommen.

In der Thematik des elektrischen Widerstandes gehen wir davon aus, dass durch die Formgedächtnislegierung ein Strom fließt. Hierbei herrscht, wie in Kapitel 2.3.3 erwähnt, bei einer mechanischen Belastung, der Superelastizitätseffekt. Bei einem Formgedächtniseffekt findet eine Veränderung der Kristallgitter statt. Dies wirkt hervor, dass der elektrische Widerstand der Formgedächtnislegierung sich verändert. In der unteren Abbildung 2.19 wird das Verhältnis der Dehnung und des elektrischen Widerstandes, des FGLs dargestellt.

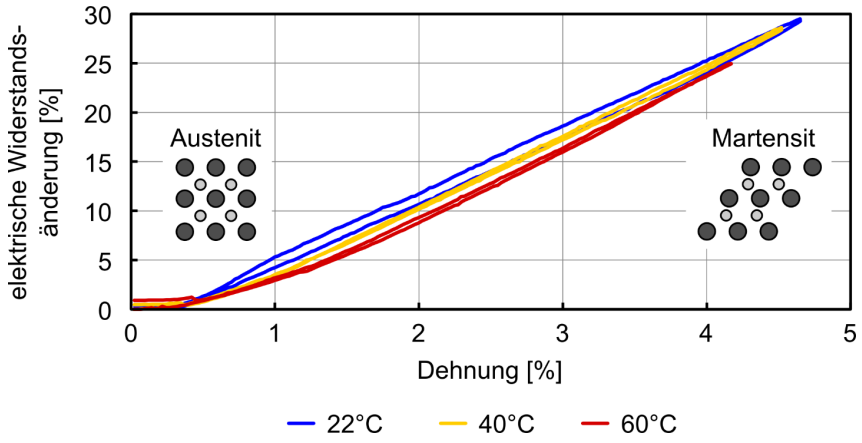


Abbildung 2.19: Verhältnis der Dehnung und des elektrischen Widerstandes bei konstanten Temperaturen von 22 °C, 40 °C und 60 °C (ADAPTRONIK (2016))

Die Formgedächtnislegierung wird bei einer konstanten Temperatur mechanisch ausgedehnt. Durch das Dehnen entsteht ein Phasenübergang von Austenit zum Martensit. Bei einer weiteren Belastung in der Martensit-Phase wird der Kristallgitter weiter umgeklappt. Hierbei steigt die elektrische Widerstandsänderung. Wenn anschließend das FGL entlastet wird, geht es in die Austenit-Phase über und die Größe des Widerstandes sinkt.

Die Messwertgröße des FGL-Widerstandes hängt von der Legierungsart sowie die Querschnittsfläche ab.

2.3.5 Anwendungsmöglichkeit

Die Formgedächtnislegierung kann für verschiedene Anwendungsmöglichkeiten verwendet werden, wie zum Beispiel in der Industrie, Wissenschaft sowie der Medizintechnik.

Heutzutage wird die Legierung vermehrt in der Industrie und neuer Technologien verwendet. Die Anwendungsmöglichkeiten der thermischen Effekte (Ein-Weg-Effekt und Zwei-Weg-Effekt) und die Superelastizität sind in der Tabelle 2.4 dargestellt.

Tabelle 2.4: Anwendungen im Bereich thermische Effekte und Superelastizität (vgl. Gümpel u. a. (2018), S.94)

Technologie	**FG-Effekt**	
	Thermischer Effekt	**Superelastizität**
Verbindung, Befestigung	Rohrverbindungen, Schrumpfringe zum Dichten, Befestigen und Schließen	Brillenrahmen, Kopfhörer, Sicherungsmuttern
Regelung, Steuerung	Schalter, Temperaturregelung, Dampfdruckregulierer, Ventilsteuerung	
Kraftfahrzeug, Antrieb	Dichtungselement bei Dieseleinspritzung, Schließsysteme und Lampenabdeckungen	Dämpfungs-elemente
Daten-verarbeitung	Steckverbindung für Schaltkreise Magnetische Datenspeicher	
Energie, Motor	Schaufelspaltregulierung bei Gasturbinen, Wärmekraftmaschinen	
Automation, Aktorik, Robotik	Greifer, Roboterglieder	
Medizin	Implantate und Klammern zu Osteosynthese, Prostatastent, Handprothese	Zahnspange und orthodontische Drähte, Medizinische Instrumente und Stens
Kleidung	Versteifung von Gewebe	Gummi-Ersatz als Tragekompfort
Luft- und Raumfahrt	Auslöse-und Öffnungsmechanismen Rohrverbindungsmuffen	Sicherungsmutter

2.4 Vierleitermessung

Die Vierleitermessung ist eine Messmethode, die dafür eingesetzt wird, um einen elektrischen Widerstand, der einen kleinen Wert besitzt, zu messen. Diese Messtechnik wird dafür verwendet, um eine Verfälschung, die durch Leitungs- und Anschlusswiderständen entstehen könnte, zu vermeiden. In der Quelle ITWissen (2011) wurden die Informationen aufgegriffen.

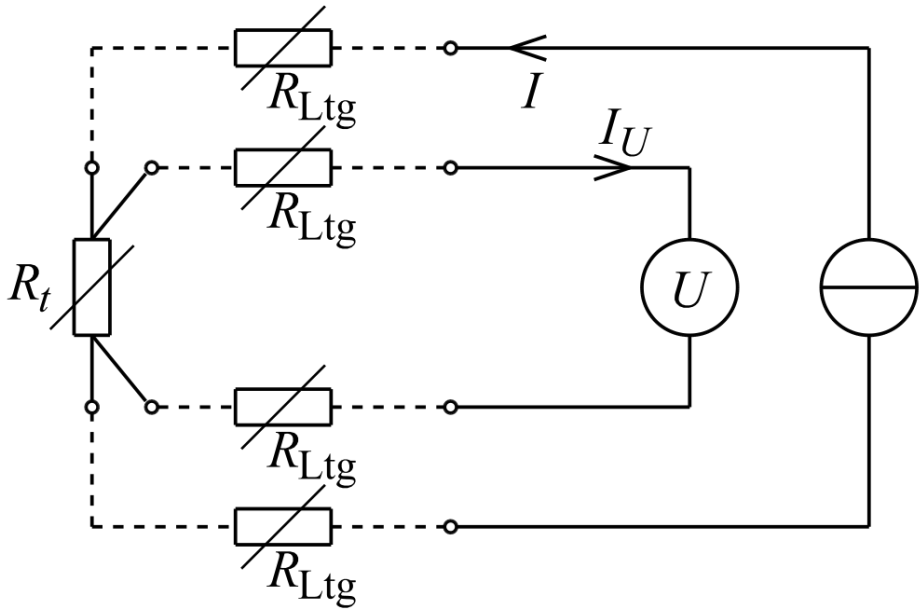


Abbildung 2.20: Anwendung der Vierleitermessung an R_t (Saure (2011))

Wie in der Abbildung 2.20 dargestellt, existieren vier Leitungen, die jeweils einen eigenen Widerstand R_{Ltg} besitzen. Dies führt dazu, dass bei der Messung des Widerstands R_t die Werte nicht richtig ermittelt werden.

Der Strom I fließt durch die beiden äußeren Leitungen und ist für die Stromversorgung zuständig. Bei den beiden inneren Leitungen, durch die der Strom I_U fließt, wird eine Spannungsmessung durchgeführt. Hier wird der Widerstand R_t gemessen, ohne die Beeinflussung der Leitungswiderstände R_{Ltg}.

Die Messung ist nur möglich, wenn der Strom des Spannungsmessers vernachlässigbar klein ist. Die Voraussetzung wird durch den hohen Innenwiderstand des Spannungsmessers erfüllt.

$$I_U << I \tag{2.7}$$

Da der Strom I_U eine vernachlässigbare kleinen Wert besitzt, entsteht nahezu kein Spannungsabfall an den Leitungswiderständen.

$$2I_U R_{Ltg} << IR_t \tag{2.8}$$

Letztendlich fallen die Werte des Leitungswiderstandswertes weg und es ergibt sich der Wert für R_t.

$$R_t = U/I \tag{2.9}$$

3 Versuchsaufbau

In diesem Kapitel werden die wesentlichen Komponenten des Praxisprojekts beschrieben. Hierbei wird unter anderem auf die Aufgabenstellung der Bachelorarbeit sowie die verwendete Hardware und die dazu entwickelte Firmware eingegangen.

3.1 Allgemein über das Projekt

Das Ziel dieses Projekts teilt sich in zwei Richtungen auf, die allgemeine Absicht und das Ziel der Bachelorarbeit. Dies wird im Kapitel 1.2 genauer erläutert. In dieser Arbeit wird nicht auf die Projektplanung und die Hardwareentwicklung eingegangen, sondern nur auf die Firmwareentwicklung, die im Laufe des Projekts entwickelt wurde.

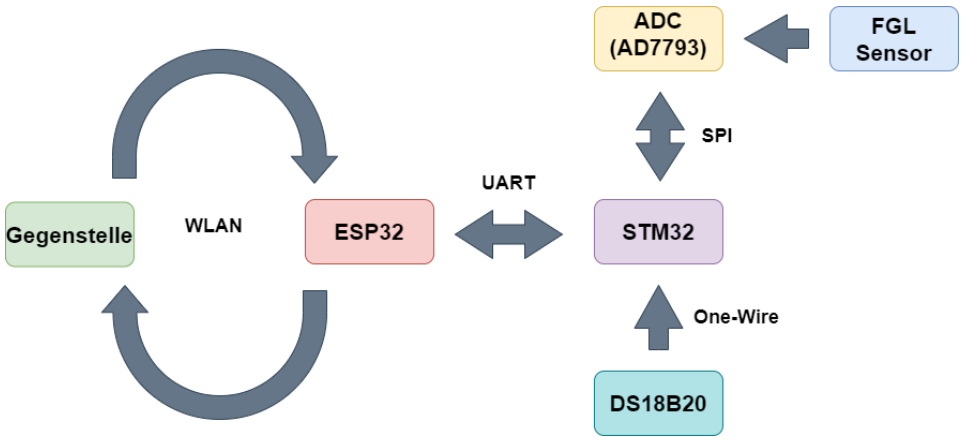


Abbildung 3.1: Allgemeine Interaktion innerhalb des Boards

In Abbildung 3.1 wird die gesamte Interaktion des Boards dargestellt. Der im Board integrierte *STM32*-Mikrocontroller übernimmt die Hauptaufgabe, da er die laufende Firmware enthält. Die Kommunikation zwischen den Bauteilen, läuft über die Bussysteme *UART*, *SPI* und *One-Wire*. Im Kapitel 2.1 wird die Thematik Bussysteme genauer erläutert.

Der *STM32* soll die Daten vom Analog-Digital-Wandler (*AD7793*) sowie Temperatursensor (*DS18B20*) kontinuierlich auslesen und dies dem *ESP32*-Mikrocontroller

übergeben. Mithilfe des *ESP32* werden über WLAN die Daten auf eine Gegenstelle übertragen. Dies geschieht mit sogenannten AT (Attention)-Kommandos, welche vom *STM32* über den *UART* empfangen bzw. versendet werden können. Die Gegenstelle soll in der Lage sein, Konfigurationsänderungen am *ADC* auszuführen. Hierbei werden über WLAN die Befehle an den *ESP32* versendet. Nach dem Empfangen wird die Konfigurationsinformation an den *STM32* weitergeleitet und der *AD7793* wird konfiguriert.

Auf den *ESP32*-Mikrocontroller, sowie die WLAN-Kommunikation und die Gegenstelle wird in der Arbeit nicht eingegangen. Hierbei wird die *UART*-Schnittstelle nicht über dem *ESP32* geführt, sondern über einen Rechner. Die AT-Befehle werden mithilfe eines Terminals simuliert und ausgelesen.

Projektaufbau

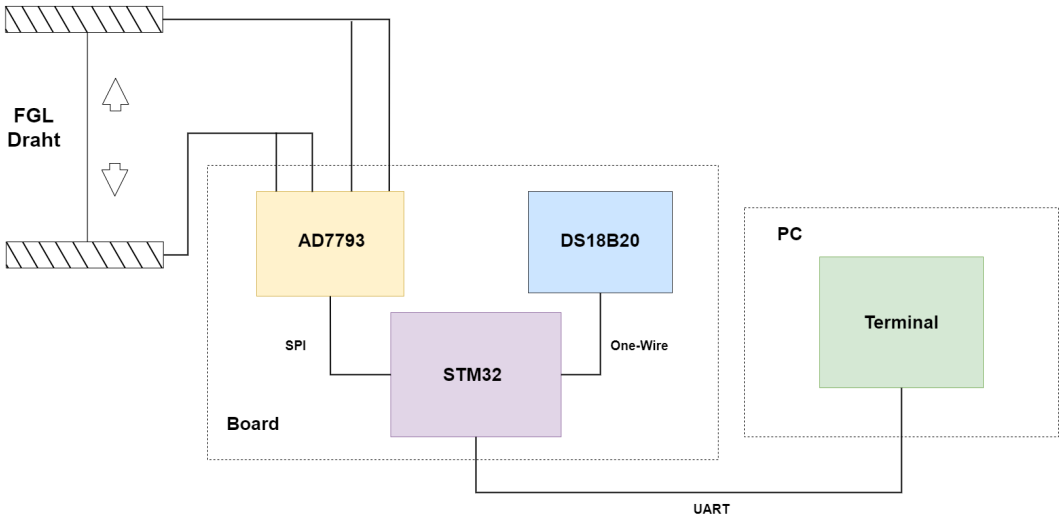


Abbildung 3.2: Praktische Projektaufbau für die Arbeit

Der Projektaufbau wird grafisch in Abbildung 3.2 dargestellt. Der FGL-Draht, wird mit einer Vierleitermessung gemessen, um den Widerstand der Messleitung auszuschließen. Dabei sind die vier Leitungen an steckbare Schraubklemmen am Board angeschlossen, was direkt zum *AD7793*-Bauteil führt. Der Analog-Digital-Wandler (*AD7793*) sowie der Temperatursensor (*DS18B20*) sind direkt mit dem *STM32*-Mikrocontroller verbunden. Anschließen wird mit einem *UART*-Kabel das Board mit dem Rechner Verbunden.

3.2 Hardware

Um genauer zu verstehen, wie die gesamte Firmware aufgebaut ist und funktioniert, wird in diesem Kapitel auf die Hardware eingegangen, die dafür entwickelt wurde.

3.2.1 Die gesamte Hardware

Für die Planung sowie die Hardwareentwicklung war die Firma *habemus* zuständig. Hierbei soll das Board nur als experimentelle Zwecke dienen.

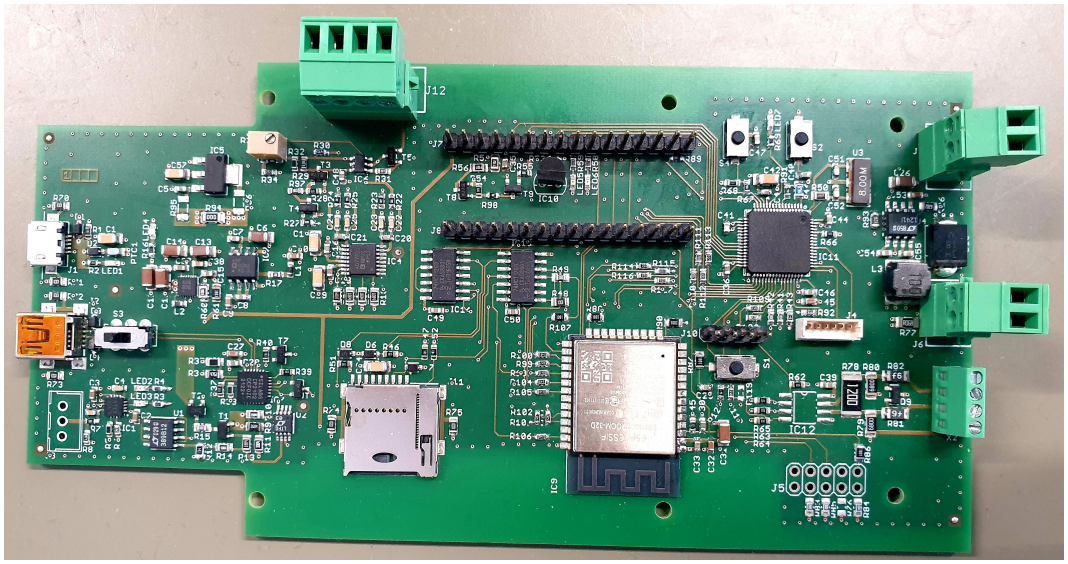


Abbildung 3.3: Das Board, welches für die Firmwareentwicklung benutzt wurde

In Abbildung 3.3, wird der Board dargestellt, auf dem die Firmware laufen soll.

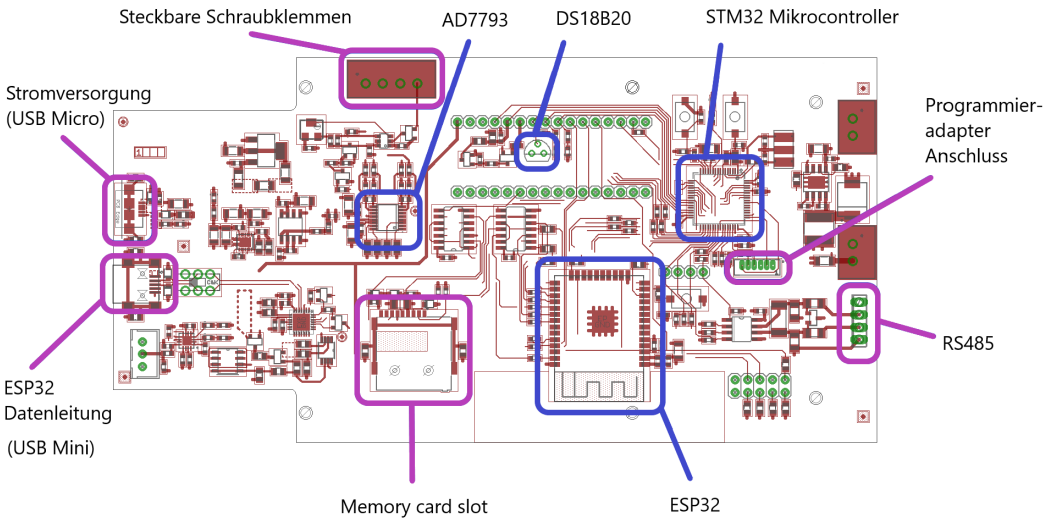


Abbildung 3.4: Gesamte Schaltung des Boards

Die Darstellung 3.4 zeigt das Layout des gesamten Boards an, welches die internen Verbindungen der Komponenten deutlich repräsentiert. Die Hauptkomponenten, werden blau und die Anschlüsse für die Peripherie, sind lila markiert.

Die lila markierten Peripherien des Boards werden tabellarisch in 3.1 aufgelistet.

Tabelle 3.1: Alle Möglichen Board Peripherien

Peripherieanschlüsse	Beschreibung
Spannungsversorgung (USB Micro)	Dieser Anschluss versorgt das Board mit einer Spannung von 5 V.
ESP32 Datenleitung (USB Mini)	Mithilfe der Leitung wird der *ESP32* Mikrocontroller programmiert.
RS485	Eine serielle Busleitung, welches als Kommunikationsmittel mit anderen Systemen dienen kann. Hierbei ist es möglich, das Board in einem automatischen System mit zu integrieren und es als eine erweiterte Komponente zu benutzen.
Memory card slot	In diesem Slot werden die gemessenen Daten und deren jeweilige Zeitstempel gespeichert.
Steckbare Schraubklemmen	In den Schraubklemmen werden die Daten des FGL ausgelesen. Die Daten des FG-Drahtes werden mithilfe der 4-Leitungsmessmethode ermittelt.
Programmieradapter Anschluss	Der *STM32*-Debugger wird mit diesem Anschluss verbunden.

Für die Entwicklung der Firmware sind unter anderem die Stromversorgung sowie der Programmieradapter notwendig. Wie im Kapitel 3.1 erläutert werden die vier Leitungen, für die Messung in den steckbaren Schraubklemmen angesteckt. Um eine Kommunikation über den *UART* herzustellen, wird mit einem TTL-232R *UART* Kabel, mit dem Board verbunden. Der STLINK-V3 von *STMicroelectronics*, dient als Programmieradapter für den *STM32* Mikrocontroller. Hierbei wird mit Hilfe des STLINKs die entwickelte Firmware in die *MCU* geladen.

In Abbildung 3.5 werden die verwendeten Peripherien verdeutlicht.

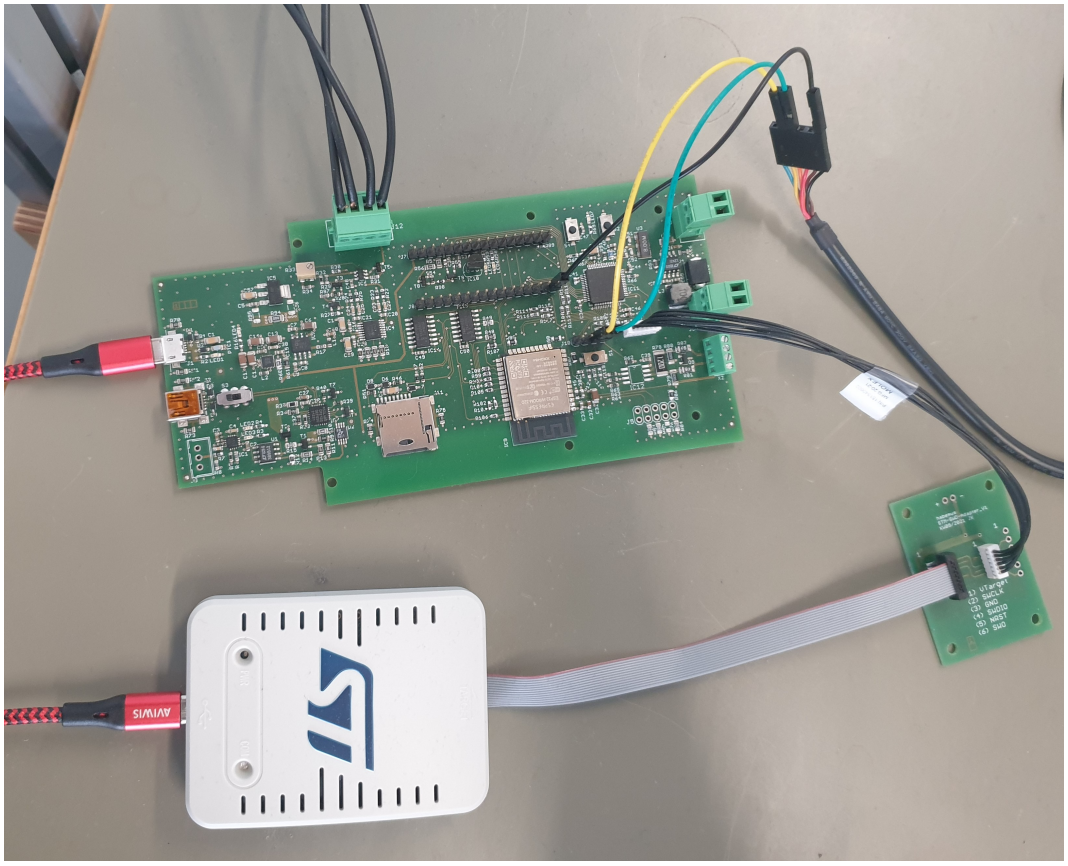


Abbildung 3.5: relevant Peripherien für die Firmwareentwicklung

3.2.2 Wichtige Baukomponenten

In diesem Unterkapitel wird auf die in Abbildung 3.3 blau markierten Hardwarekomponenten eingegangen.

3.2.2.1 STM32 Mikrocontroller

Für das Projekt wird ein *STM32F103RB*-Mikrocontroller verwendet, der einen leistungsstarken *ARM®Cortex®-M3* 32-Bit *RISC*-Kern besitzt und mit einer Frequenz von 72 MHz arbeitet. Außerdem enthält er eingebettete Hochgeschwindigkeitsspeicher: einen 128 KByte Flashspeicher und einem 20 KByte SRAM.

Der Mikrocontroller wurde dafür ausgewählt, da es die gewünschten Schnittstellen für das Projekt aufweist und leistungsfähig ist.

3.2.2.2 ESP32

Der *ESP32* gehört zu der 32-Bit-Mikrocontroller Familie und besteht, abhängig vom Typ, aus einem oder zwei Prozessorkernen. Der *ESP32* arbeitet mit einem Systemtakt von 160 bis 240 MHz und enthält einen *RAM* Speicher, welcher 160 KiB statisch (*SRAM*) und 160 KiB dynamisch (*DRAM*) zugewiesen werden. Der *ESP32* enthält wie sein Vorgänger keinen internen Flashspeicher. Außerdem besitzt der Mikrocontroller den Vorteil eines geringen Leistungsbedarfs sowie kostengünstig zu sein. Eine weitere Besonderheit liegt darin, das der *ESP32* je nach Prozessortyp ein Bluetooth und WLAN Antenne besitzt.

Im Allgemeinen Projekt wird der Mikrocontroller nur für die WLAN-Verbindung benutzt, was mit Hilfe von AT-Kommandos möglich ist. Diese Komponente wir im Projekt nicht berücksichtigt, da die Arbeit sich in die Länge überziehen würde.

3.2.2.3 AD7793

Der *AD7793* ist ein stromsparender, rauscharmer und hoch präziser Analog-Digitalwandler, der einen 24-Bit Ausgang besitzt. Dies wird für das Übersetzen der Analogdaten des FGLs verwendet. Die Besonderheit dieses *ADC* ist, dass die Abtastrate und der Eingangsstrom konfigurierbar sind.

AD7793 Werteausgabe

Um den Widerstandswert des FGL auszugeben, muss verstanden werden, was der *AD7793* ausgibt und was bei einer Dehnung des Drahtes mit dem Wert passiert. Die Baukomponente kann entweder im bipolaren oder unipolaren Betrieb arbeiten. Der Wert *Code* im bipolaren Betrieb kann negativ sowie positiv dargestellt werden.

Laut dem Datenblatt des *ADC* ist die dazugehörige Formel des bipolaren Betriebs

$$Code = 2^{N-1} * [(AIN * GAIN/V_{Ref}) + 1] \tag{3.1}$$

Im unipolaren Modus wird der Ausgangswert nur positiv dargestellt. Dabei kann im Gegensatz zu bipolaren Betrieb der unipolare Modus größere positive Zahlen darstellen, da die negativen Werte wegfallen.

$$Code = (2^N * AIN * GAIN)/V_{Ref} \tag{3.2}$$

Der Wert N ist die Größe des Ausgang-Bits. Dies würde beim *AD7793* 24-Bit betragen. Der $GAIN$ repräsentiert den einstellbaren Eingangsstrom. Die analoge Eingangsspannung AIN ist die Spannung, welche vom FG-Draht abfällt. Hierbei kann sich die Eingangsspannung durch eine Belastung des Drahtes verändern.

Durch den FG-Draht fließt ein konstanter Strom, wodurch ein Spannungsabfall U_{Meas} entsteht. Wenn anschließend der Draht mit einer mechanischen Kraft ausgedehnt wird, wächst U_{Meas}, da der Widerstand des FGLs größer wird. Im Kapitel 2.3.4 wurde auf die Thematik des elektrischen Widerstandes eingegangen.

Dabei wurde eine Schaltung kreiert, die das Verhältnis der Messspannung U_{Meas} und der angelegten Referenzspannung U_{Ref} messen soll. In Abbildung 3.6 wird die Schaltung dargestellt.

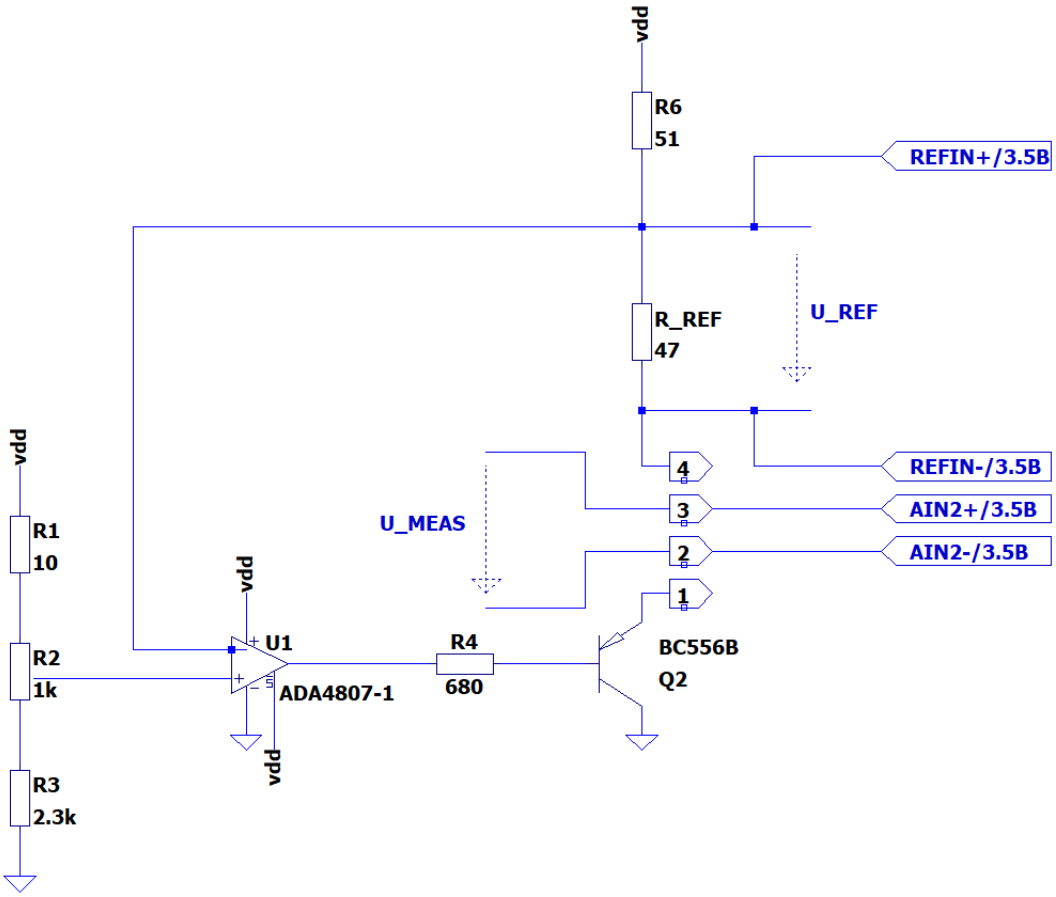


Abbildung 3.6: Beschaltung des *AD7793*

Die Spannungsverhältnisse sind äquivalent zum Verhältnis der ermittelten ADC-Wert ADC_{MEAS} und dessen Maximalwert ADC_{Max}.

$$\frac{U_{Meas}}{U_{Ref}} = \frac{ADC_{Meas}}{ADC_{Max}} \tag{3.3}$$

U_{Meas} und U_{Ref} können durch das Produkt des Stromes und den Widerstand ersetzt werden. Hierbei kann der Strom weggelassen werden, da durch den Messwiderstand sowie der Referenzwiderstand derselbe Strom fließt. Durch die untere Formel wird erkannt, dass die Verhältnisse des Referenzwiderstands und des Messwiderstands identisch der ADC Verhältnisse sind. Wenn anschließend die R_{Ref} mit dem Verhältnis ADC_{Meas}/ADC_{Max} multipliziert wird, kann der Widerstand des gemessenen Objektes ermittelt werden. Dies ist der veränderbare Widerstand des Formgedächtnisdrahtes.

$$R_{Meas} = \frac{ADC_{Meas}}{ADC_{Max}} R_{Ref} \tag{3.4}$$

Durch diese Methode ist eine Strommessung nicht erforderlich und die Messergebnisse unabhängig des Stromwerts. Hierbei ist zu beachten, dass der maximale Messwiderstand kleiner sein muss als der Referenzwiderstand der Schaltung, welches in diesem Versuch 45Ω beträgt.

Wen anschließend die Formel für den bipolaren (3.1) sowie unipolaren (3.2) Betrieb umgeformt und mit der Gleichung der Schaltung gleichgesetzt wird, kann dadurch die gesuchte Größe ermittelt werden.

Bipolar:

$$\frac{R_{Meas}}{R_{Ref}} = \frac{AIN}{U_{Ref}} = \frac{\frac{Code}{2^{N-1}} - 1}{GAIN} \tag{3.5}$$

Unipolar:

$$\frac{R_{Ref}}{R_{Meas}} = \frac{U_{Ref}}{AIN} = \frac{2^N * GAIN}{Code} \tag{3.6}$$

Letztendlich kann mithilfe der Schaltung der Widerstand des Formgedächtnisdrahtes im bipolaren sowie unipolaren Betrieb ermittelt werden.

AD7793 Hardwarehintergrund

Um genauer zu verstehen, wie der *AD7793* mit der Firmware interagiert, wird speziell auf das Bauteil eingegangen. Der *AD7793* wird mithilfe des *SPI*-Busses angesprochen. Hierbei ist zu erwähnen, dass der Analog-Digital-Wandler insgesamt 16 Pins besitzt. Wie in Abbildung 3.7 dargestellt.

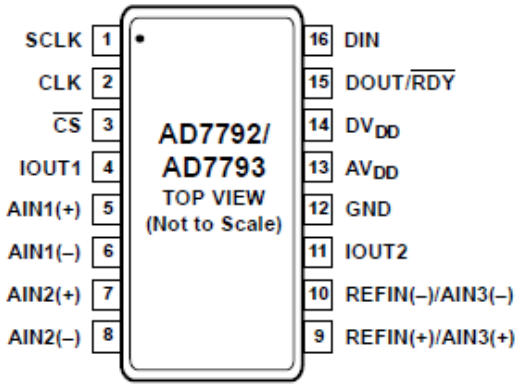


Abbildung 3.7: *AD7793* in Frontsicht (Ana (2004), S.9)

In der Tabelle 3.2 werden die Pins, die im Projekt verwendet werden, aufgelistet und erklärt.

Tabelle 3.2: *AD7793* Pins (vgl. Ana (2004). S.9-10)

Relevante ADC Pins

Pin Nummer	Name	Beschreibung
7	AIN2(+)	Der positive Anschluss für den analogen Eingang.
8	AIN2(-)	Der negative Anschluss für den analogen Eingang.
9	REFIN(+)/AIN3(+)	Der positive Referenz-/Analogeingang. Der Pin kann für externe positive Referenzspannungen dienen.
10	REFIN(-)/AIN3/(-)	Der negative Referenz-/Analogeingang. Der Pin kann für externe negative Referenzspannungen dienen.
13	AV_{DD}	Die Versorgungsspannung von 2,7 V bis 5,25 V.
14	DV_{DD}	Die Versorgungsspannung der digitalen Schnittstelle, welche eine Spannung von 2,7 V bis 5,25 V beträgt. AV_{DD} und DV_{DD} sind unabhängig von einander und können deshalb unterschiedliche Spannungen betragen.

***ADC*-Pin für den *SPI*-Bus**

Pin Nummer	Name	Beschreibung
1	SCLK	Der *Serial Clock Input* empfängt das Taktsignal vom *Master*.
3	$\overline{CS}$	Der *Chip Select* dient für das Auswählen des *ADC*, wenn es sich in einem System befindet, wo mehrere Bauteile dieselbe *SPI*-Leitung teilen.
15	DOUT/$\overline{RDY}$	Dies ist die *MISO*-Leitung, welche die Daten an *Master* versendet. Der Pin dient auch als *RDY*-Pin. Dabei geht der Pin auf *LOW*, wenn der *ADC* bereit für eine Kommunikation ist.
16	DIN	Dies ist die *MOSI*-Leitung, welche die Daten vom *Master* empfängt.

Die Daten wurden vom Datenblatt (Ana (2004)) entnommen. Der *ADC* besitzt neun Register, die entweder gelesen oder beschrieben werden können. Hierbei werden nur auf die Register eingegangen, die für die Arbeit relevant sind.

Kommunikationsregister (8-Bit)

Das Register besteht aus 8-Bit und ist für die allgemeine Kommunikation der jeweiligen Register des *ADC* zuständig. Anfangs der *SPI*-Kommunikation wird auf das Kommunikationsregister zugegriffen, um festzulegen, in welches Register die nächste Operation stattfinden und ob Daten gelesen oder beschrieben werden sollen.

Statusregister (8-Bit)

Der Statusregister beinhaltet die Information, ob der *ADC* momentan in der Lage ist zu lesen bzw. beschrieben werden darf. Auch ob die vorgesehene Operation ein Erfolg war, wird im Statusregister angezeigt. Das Register zeigt außerdem an, welcher Kanal durch den *ADC* umgewandelt wird.

Moderegister (16-Bit)

Im Moderegister kann ausgewählt werden, mit welcher Abtastrate sowie Taktsignalquelle der *ADC* arbeiten soll. Auch in welcher Operationsmodus der *AD7793* arbeitet, ist in diesem Register möglich. Der Moderegister kann beschrieben sowie gelesen werden.

Konfigurationsregister (16-Bit)

Für die Konfiguration des *ADC* ist dieses Register zuständig. Der *ADC* kann in einem unipolaren oder bipolaren Modus konfiguriert werden. Eine Besonderheit des Konfigurationsregisters ist der *burnout current*, welches die Aufgabe hat, den Benutzer zu informieren, ob der *ADC* durch den Strom durchgebrannt ist oder nicht. Im Konfigurationsregister wird ausgewählt, mit welchen analog Kanal und mit wie viel Eingangsstrom gearbeitet werden soll. Außerdem kann der Puffer des *ADC*s in diesem Register aktiviert werden.

Datenregister (8-Bit)

Das Datenregister kann nur gelesen werden und beinhaltet den Digitalwert bzw. den umgewandelten Analogwert.

IO Register (8-Bit)

In diesem Register wird der Erregerstrom ausgewählt und freigegeben.

Continuous Read

Eine weitere Besonderheit des Analog-Digital-Wandlers, was erwähnt werden muss, ist das kontinuierliche Lesemodus. Hierbei wird in das Kommunikationsregister der Wert 0x5C beschrieben, um dies zu aktivieren.

Im *Continuous Read* wird das $\overline{CS}$ auf LOW gesetzt, um das ADC kontinuierlich zu besetzen. Wie in Abbildung 3.8 dargestellt, wird in das Datenregister zugegriffen. Nach dem Zugreifen werden, ohne in das Kommunikationsregister zurückzukommen, die Daten ausgelesen. Dabei zieht der ADC die Leitung DOUT/$\overline{RDY}$ auf Masse, wenn das Bauteil für die nächste Leseoperation bereit ist.

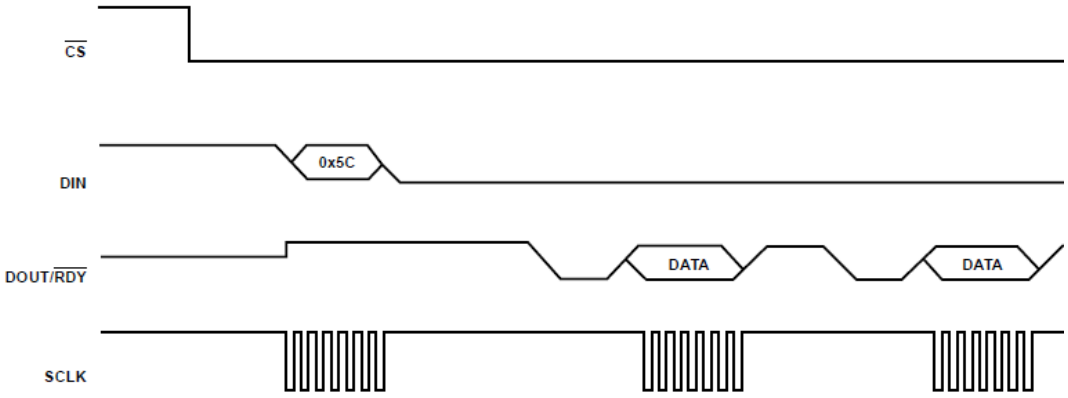


Abbildung 3.8: Kontinuierliche lesen des Datenregisters (Ana (2004), S.23)

Um das Kontinuierliche lesen zu deaktivieren, wird in das Kommunikationsregister der Wert 0x58 geschrieben.

3.2.2.4 DS18B20

Die Informationen wurden vom Datenblatt des *DS18B20* entnommen. (max (2019))

Der *DS18B20* ist ein Temperatursensor, welcher mithilfe eines *One-Wire*-Bus mit dem *Master* kommunizieren kann. Hierbei bietet der Sensor eine Messung von 9 bis 12-Bit-Celsius an. Die Versorgungsspannung kann der Temperatursensor entweder durch die Spannungsversorgungsleitung oder durch die Datenleitung (parasitär) aufnehmen. Um eine parasitäre Spannungsaufnahme zu bekommen, muss der *parasite power mode* des *DS18B20* aktiviert und die Spannungsleitung mit *GROUND* verbunden werden.

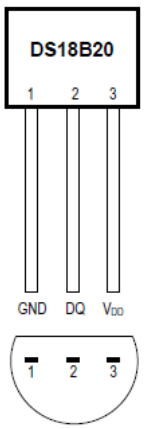


Abbildung 3.9: *DS18B20* im Front- sowie Draufsicht (max (2019), S.1)

In Abbildung 3.9 werden drei Pin Ausgänge dargestellt. Wie im Kapitel 2.1.3 erwähnt, kann die Datenübertragungsleitung *DQ* gleichzeitig als Stromversorgung für den *DS18B20* Bus dienen.

Um zu erklären, wie die Kommunikation des *DS18B20* funktioniert, wird auf das Blockschaltbild in Abbildung 3.10 eingegangen.

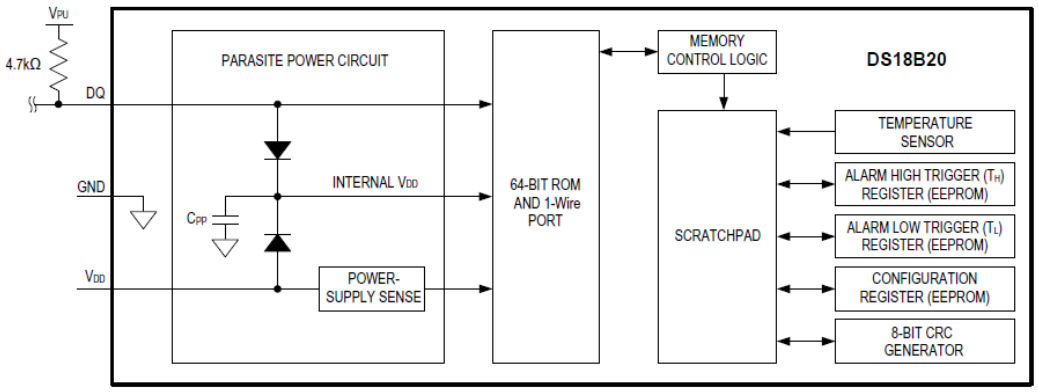


Abbildung 3.10: Gesamte Blockschaltbild von *DS18B20* (max (2019), S.5)

Im Blockschaltbild ist nur die 64-Bit *ROM*, *One-Wire*-PORT und die Kommunikation über den *SCRATCHPAD* relevant. Mit Hilfe des 64-Bit-*ROM*-Code und der *ROM*-Funktionssteuerlogik ist der *DS18B20* in der Lage, das *One-Wire*-Protokoll zu verwenden. Der *SCRATCHPAD* ist ein 8-Byte großes SRAM und beinhaltet alle Register des *DS18B20*.

Während einer Kommunikation befindet sich der Benutzer zuerst im *ROM*-PORT. Um die Daten sowie Konfigurationen der Temperatur zu lesen bzw. zu schreiben, muss der *ROM* übersprungen werden. Dabei muss als Erstes der Wert 0xCC, was für *Skip Rom* steht, in das *DS18B20* beschrieben werden. Anschließend wird mit nachfolgenden Befehl in dem *SCRATCHPAD* auf das gewünschte Register zugegriffen.

In diesem Projekt wird nur auf die Daten des Temperatursensors zugegriffen und keine weiteren Einstellungen vorgenommen.

f

3.3 Firmware

In diesem Kapitel wird auf die Firmware eingegangen. Dabei wird als Erstes die Firmwarekomponente erklärt und die Interaktion miteinander zusammengefasst.

3.3.1 Allgemeine Firmwareaufbau

Die allgemeine *STM32*-Firmwareaufbau wird in Abbildung 3.11 dargestellt. Sie wird in verschiedenen Komponenten getrennt, um eine bessere Ordnung und Struktur zu erhalten. Außerdem dient der Aufbau als Wiederverwendung für weitere Projekte. Von der Quelle STMicroelectronics (2022) wurden die Informationen entnommen.

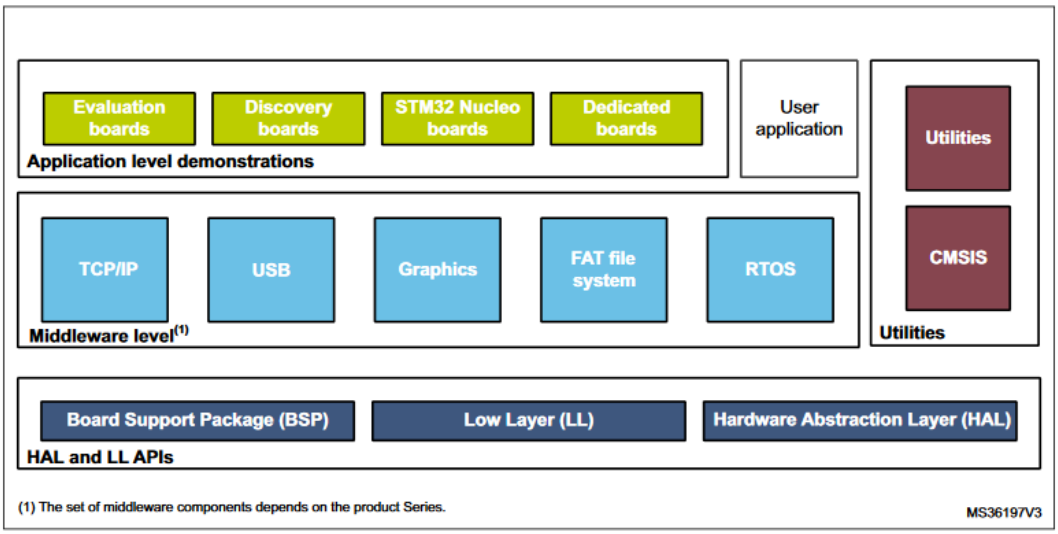


Abbildung 3.11: Der Aufbau der Firmware des *STM32Cube* wird in verschiedenen Ebenen separiert (STM (2017))

Application level demonstrations:

In dieser Ebene befinden sich alle Anwendungen und Beispiele. Außerdem enthält sie für jedes unterstützte Board alle *STM32CubeIDE*-Projekte sowie die Quelldaten des *Templates*. Es werden Projekte erstellt, die für Beispiele, Anwendungen, Demonstrationen und für Vorlagen dienen können.

Es gibt vier verschiedene Varianten von Ausführbare Projekten, welche die Verwendung der Software *Application Programming Interface* (*API*) darstellten.

Beispielprojekte: In einem Beispielprojekt werden die *Hardware Abstraction Layer*-APIs und *Low level* Treibern benutzt, mit einer leichten Verwendung der *Board support package* Schicht.

Applikationsprojekte: In einem Applikationsprojekt werden die *Middleware*-Komponente benutzt und dies mit der Hardware und die *HAL/LL*-APIs Ebene mitintegriert.

Demonstrationsprojekte: In einer Demonstration werden so viel wie möglich an Peripheriegeräte und *Middleware Stacks* verwenden, um die Leistung des Produkts zu demonstrieren.

Vorlagenprojekte: In dieser Ebene befindet sich alle Anwendungen sowie Beispiele. Außerdem enthält sie für jedes unterstützte Board alle *STM32CubeIDE*-Projekte, sowie die Quelldaten des Templates.

Middlewarelevel:

Die *Middleware*-Ebene enthält eine Reihe von Bibliotheken, welche eine große Anzahl von Diensten anbieten. Die beinhalteten Komponenten hängen von Produkt Serie ab. Jede *Middleware*-Komponente beinhaltet einen Bibliothekskern, welcher die Hauptbibliotheks-Zustandsmaschine und den Datenfluss zwischen den verschiedenen Modulen verwaltet. Außerdem besteht jede Komponente aus einer Schnittstellenschicht, was für die Verbindung des Komponentenkerns mit den unteren Schichten, wie dem *HAL*- und den *BSP*-Treibern verwendet wird.

HAL- und LL-APIs:

Diese Ebene enthält drei Softwarekomponenten.

1. Hardware Abstraction Layer (HAL):

Im *HAL*-Treiber wird die Interaktion zwischen den *Low Level*-Treiber sowie die Hardware-Schnittstellenmethoden mit dem oberen Schichten (Anwendung, Bibliotheken und Stacks) zur Verfügung gestellt. Sie enthält für die Kommunikationsperipheriegeräte (*SPI*, *UART*, etc.) die APIs, welche die Implementierung der Peripheriegeräten, durch die Bereitstellung von gebrauchsfertigen Prozessen vereinfacht. Dies wären unter anderem die Initialisierung, sowie Konfiguration von Datenübertragungen auf der Grundlage von *Polling*, *Interrupt*s und *DMA*-Prozessen. Hierbei hat es den Vorteil, die Peripheriegeräten besser zu verwalten und während der Kommunikation Fehler zu vermeiden.

2. Low-Level (LL):

Diese Treiber sind im *STM32Cube*-Firmware *HAL* mit inbegriffen. Hierbei stellt der *Low Level* optimierte sowie einmalige Dienste zur Verfügung. Der Vorteil ist, dass der

LL-Treiber für gewisse Diensten bereitgestellt werden kann, wofür der *HAL*-Treiber nicht in der Lage ist.

3. Board-Support-Package (BSP)):

Der *BSP* abstrahiert die Peripherie für eine einfachere Portierung der darauf aufbauenden Module auf andere Hardwareinterfaces.

Utilities:

Die Ebene besteht hauptsächlich aus zwei Komponenten, die *Common Microcontroller Software Interface Standard* (*CMSIS*) und die *Utilities*. Außerdem werden allgemeine Hilfsmittel sowie Dienste zur Verfügung gestellt.

In Abbildung 3.12 wird der Aufbau der Firmware für das Projekt dargestellt.

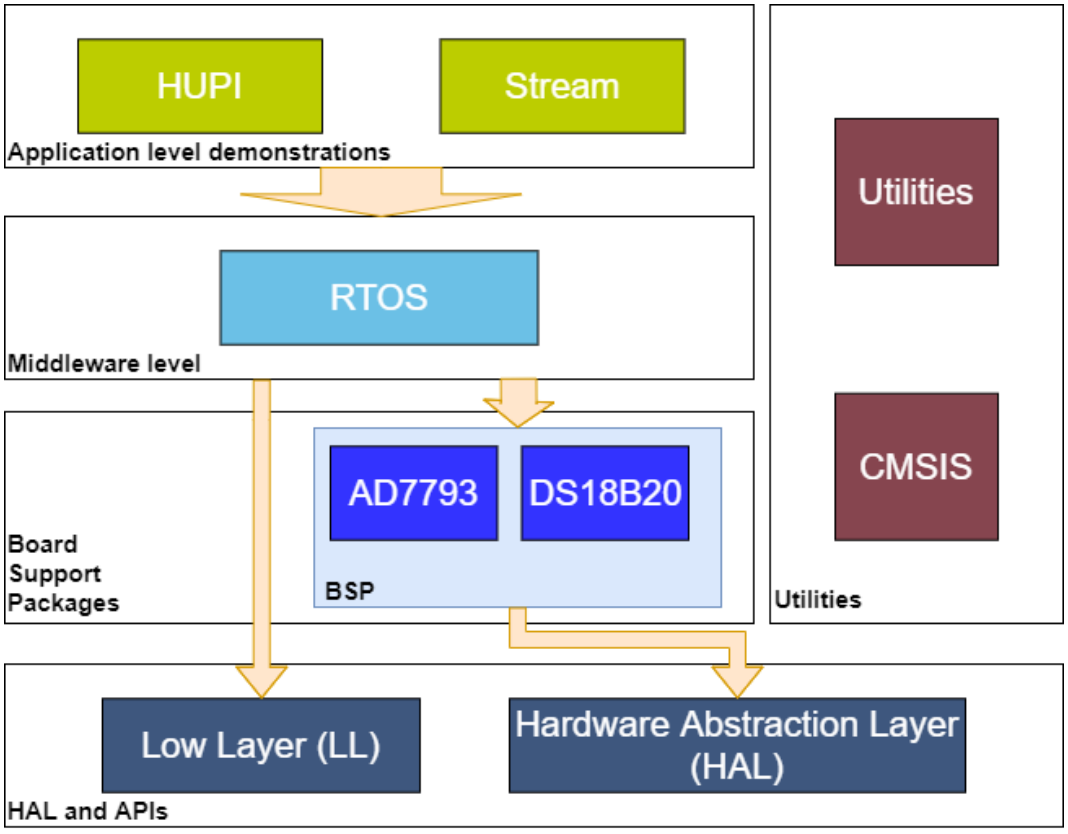


Abbildung 3.12: Aufbau der Firmware für die Bachelorarbeit

Für die Arbeit wurde ein Applikationsprojekt entwickelt. Es wurde für den Kunden ein Board angefertigt, auf dem die Firmware laufen soll.

Wie in Abbildung 3.12 dargestellt, befinden sich die Komponenten *habemus Universal Production Interface* (HUPI) und *Stream* in der höchsten Ebene. Im Kapitel 3.3.2 wird genauer auf die Komponenten eingegangen.

Im *Middleware-Level* wurde die Komponente *Real Time Operating System* (RTOS) bzw. *Free Real Time Operating System* (FreeRTOS) benutzt. Das *RTOS* ist ein Betriebssystem, welches präemptive Echtzeitleistung mit optimierten Kontextwechsel- und *Interrupt*-Zeiten die schnelle und gut vorhersehbare Reaktionszeiten ermöglichen, anbietet.

Es wurde für die *Board Support Package*s der Treiber für den *AD7793* sowie des *DS18B20* entwickelt. Für die Peripherien, welche für die *BSP*-Komponente verwendet wird, wurde auf die *HAL*-Bibliothek zugegriffen. Die *UART*-Verbindung, welche für den *HUPI* sowie *Stream* benötigt wird, wurde mit dem *Low Level* entwickelt, da die Dienste der *HAL*-Bibliothek für die benötigte Anwendung nicht geeignet ist.

Im folgenden Kapitel wird genauer auf die Entwicklung des Applikationsprojekts eingegangen.

3.3.2 Entwicklung der Firmware

In diesem Kapitel wird auf das Applikationsprojekt eingegangen. Dabei wird zunächst genauer die einzelnen Komponenten und die Interaktion miteinander beschrieben. Als Nächstes wird detailliert die Firmware dargestellt, welches die Initialisierung und die Arbeitsweise beinhaltet. Die jeweiligen Programme, die in den Listings dargestellt werden, sind wegen Verständigungszwecken leicht verändert und abgebildet.

3.3.2.1 Applikations- und BSP-Komponenten

In diesem Unterkapitel wird genauer auf die beiden entwickelten Treiber *AD7793* und *DS18B20* sowie die Applikationskomponenten *HUPI* und *Stream* eingegangen.

A. AD7793-Treiber

Folgende Voreinstellungen der *SPI* wurden für den *AD7793* vorgenommen, um mit dem *ADC* ordnungsgemäß zu kommunizieren. Laut dem Datenblatt muss die Taktfrequenz des *SPI* maximal 5 MHz betragen. Hierbei wurde der *SCK* des *SPI* auf 4 MHz eingestellt, um Probleme, die bei niedriger und hoher Frequenz entstehen könnten, auszuschließen. Die Betriebsart läuft unter Modus 3, siehe Tabelle 2.1 und die Länge der übertragenen Daten beträgt 8-Bit.

Die *SPI*-Kommunikation findet unter *Direct Memory Acess* satt. Durch eine *SPI*-Datenübertragung gehen die Daten durch das *MCU* direkt in den Speicher. Dabei wird die Performance der *MCU* beeinträchtigt. Um dies zu vermeiden, wird der *DMA* verwendet, um den Mikrocontroller zu entlasten und mehr Zeit einzusparen. Um die Art von Kommunikation zu nutzen, müssen die *DMA*-Channels für die *Rx* (*MISO*) sowie die *Tx* (*MOSI*) Leitung konfiguriert werden.

Listing 3.1: Initialisierung der *Direct Memory Acess*

```
454     static void DMA_Init(void)
455     {
456       /*main.c*/
457       /*Aktiviert den DMA Kontroller Taktfrequenz*/
458       RCC_DMA_CLK_ENABLE();
459 
460       /*Die Initialisierung  der DMA Interrupt*/
461       /*Die Konfiguration des DMA_Channel4_IRQn Interrupts*/
462       NVIC_SetPriority(DMA_Channel4_IRQn, 5, 0);
463       NVIC_EnableIRQ(DMA_Channel4_IRQn);
464       /*Die Konfiguration des DMA_Channel5_IRQn Interrupts*/
465       NVIC_SetPriority(DMA_Channel5_IRQn, 5, 0);
466       NVIC_EnableIRQ(DMA_Channel5_IRQn);
467     }
```

Wie im Listing 3.1 dargestellt wird als Erstes für die Initialisierung, die Taktfrequenz für den *DMA* aktiviert. Im nächsten Schritt werden die *Interrupt*s der *DMA*-Channel initialisiert, konfiguriert und anschließend aktiviert. Die Zuordnung der Kanäle kann von der Tabelle 2.3 entnommen werden.

Der Zustandsautomat

Der Treiber für den *AD7793* wurde nach einer Zustandsmaschinenschematik programmiert. Der Grund liegt darin, dass mithilfe dieser Methode Zustandsabläufe besser dargestellt werden können. Ein weiterer Vorteil liegt darin, dass der Automat mit wenig Aufwand beliebig verändert werden kann. Dabei können die Zustände asynchron abgefragt werden, ohne andere Systeme zu beeinflussen.

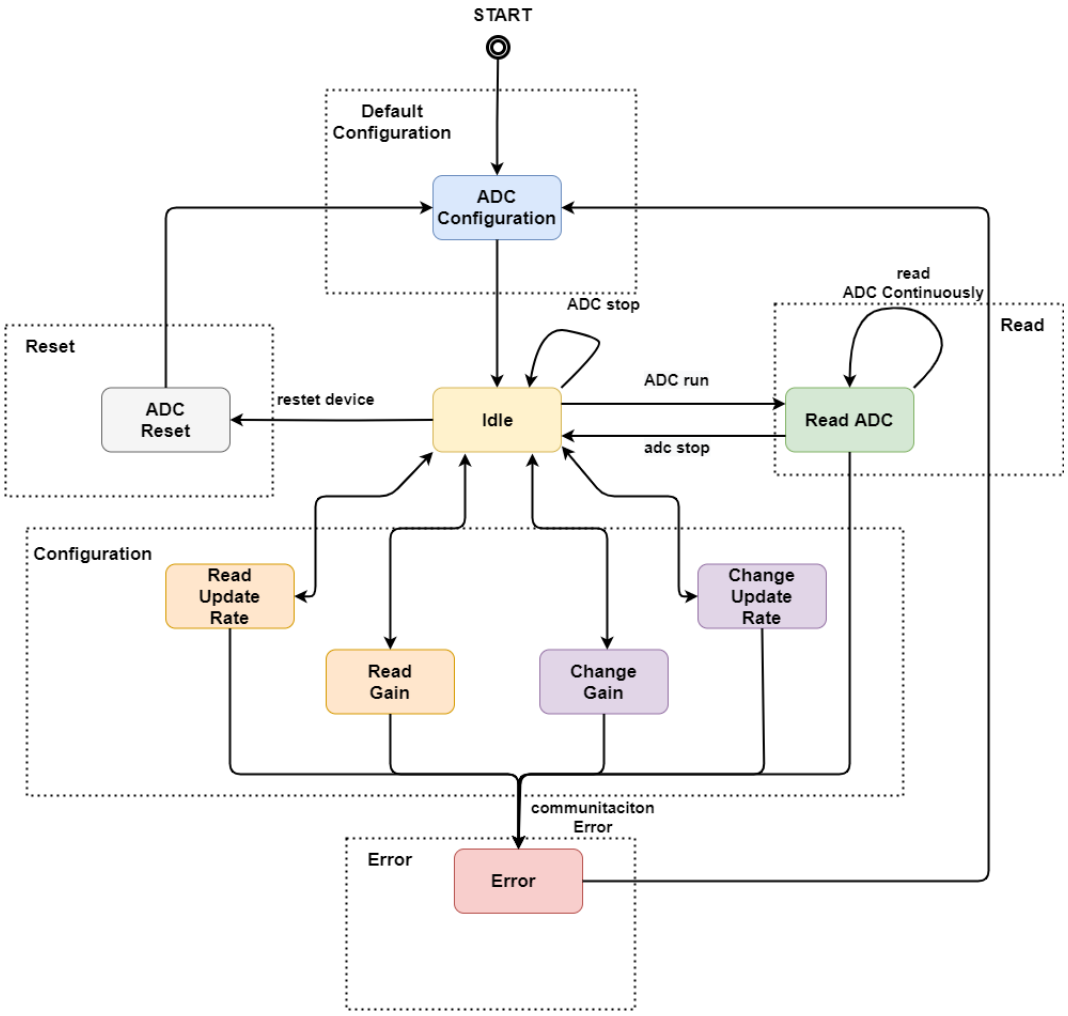


Abbildung 3.13: AD7793 Zustandsautomat

In Abbildung 3.13 wird die gesamte Zustandsmaschine der Firmware dargestellt. Um zu verstehen wie der Treiber funktioniert wird die Zustandsmaschine Abstrakt unterteilt.

Die Initialisierung

Es ist sicherzustellen, dass die Initialisierung des Treibers vor der Anwendung erfolgen muss. Dabei werden die Standardwerte, der Eingangstrom und die Abtastrate definiert. Während der Initialisierung wird bestimmt, ob der *AD7793* die Werte unipolar oder bipolar ausgeben soll.

Eine weitere Besonderheit der Initialisierung ist die Übergabe einer Formel. Hierbei kann die Werteausgabe vom *ADC*-Treiber, welcher im Kapitel 3.2.2.3 beschrieben wird, bestimmt werden. Im Codebeispiel 3.3, wird die Funktion für die Formel *RMeas* dargestellt. Die Formel ergibt sich durch die Verschaltung des *ADC*s.

Für die Widerstandswerte, die zum Schluss ausgerechnet werden, wird nur der bipolare Betrieb benötigt, da keine negativen Widerstände existieren. Um den Widerstand zu erhalten, wird in die übergebende Formel das Produkt des Referenzwiderstands und des Ausgabewerts des *AD7793* bzw. die Spannungsverhältnisse definiert. Dabei ist zu beachten, dass bei der Initialisierung alle Standardwerte in einem Puffer geschrieben werden und nicht in die *AD7793* konfiguriert wird.

Im Codebeispiel Listing 3.2, wird die Initialisierung des *AD7793*-Treibers dargestellt. Als Erstes wird ein *struct* erstellt, welches die Standardkonfigurationselemente, wie der Eingangstrom, die Abtastrate und die Ausgabeart für den Treiber beinhaltet.

Listing 3.2: Standardwerte des *AD7793*

```
318  /*AD7793.h*/
319  typedef struct configurationValues_s{
320    uint8_t gain;
321    float updateRateHz;
322    bool unipolar;
323  }configurationValues_t;
```

```
84   /*main.c*/
85   configurationValues_t configuration;
```

Im nächsten Schritt wird die Formel anhand einer Funktion erstellt, die das Endergebnis zurückgibt (Siehe Listing 3.3). In diesem Fall wurde der Rückgabewert für den Widerstand *float* hergenommen. Die beiden Formeln können den Kapiteln 3.5 bzw. 3.6 entnommen werden.

Listing 3.3: Funktion für die umgerechnete Formel des *RMeas*

```
101  /*main.c*/
102  float formulaForRMeas(float voltageRelationshipValue){
103    if(configuration.unipolar){
104      /*unipolar*/
105      return(R_Ref/voltageRelationshipValue);
106    }
107    else{
108      /*bipolar*/
109      return(voltageRelationshipValue*R_Ref);
110    }
111  }
```

Der Wert *voltageRelationshipValue* entsteht durch die Formel des bipolaren bzw. des unipolaren Betriebs (Siehe Listing 3.4).

Listing 3.4: Formel für die unipolare und bipolare Ausgabe

```
36   /*AD7793.c*/
37   /*Die Makros repreasentieren das 2^n sowie 2^(n-1)*/
38   /*In die variable n muss mit 1 subtrahiert werden, da die*/
39   /*groesse bzw. Index bei null anfeangt*/
40   #define POWER_OF_TWO_BIPOLAR(n) (2 << ((n - 1) - 1))
41   #define POWER_OF_TWO_UNIPOLAR(n) (2 << (n  - 1))
```

```
192  /*AD7793.c*/
193  float FormelBipolar(uint8_t *outputCode, int maxBit, int gain){
194    uint32_t code = 0;
195
196    /*Die Arraywerte werden in einem Dezimal Wert dargestellt*/
197    for(uint8_t dataElement = 0; dataElement < (maxBit/ONE_BYTE);
           dataElement++){
198      code += (outputCode[dataElement] << (ONE_BYTE * (2 -
           dataElement)));
199    }
200
201    /*Die Umgeformte Formel fuer den Bipolaren betrieb(Formel 3.5)*/
202    return ((((float)code)/(POWER_OF_TWO_BIPOLAR(maxBit))-1)/gain);
203  }
```

```
203  float FormelUnipolar(uint8_t *outputCode,int maxBit, int gain){
204    uint32_t code = 0;
205
206    /*Die Arraywerte werden in einem Dezimal Wert dargestellt*/
207    for(uint8_t dataElement = 0; dataElement < (maxBit/ONE_BYTE);
           dataElement++){
208      code += (outputCode[dataElement] << (ONE_BYTE * (2 - dataElement
             )));
209    }
210
211    /*Die Umgeformte Formel fuer den Unipolaren betrieb(Formel 3.6)*/
212    return ((POWER_OF_TWO_UNIPOLAR(maxBit) * gain)/((float)code));
213  }
```

Für das Umrechnen werden hierfür der Ausgangswert des *AD7793*, die maximale Anzahl der Bits sowie der Eingangstromwert benötigt. Der Ausgangswert des *AD7793* liegt in einem *Array* vor, welche drei Elemente mit 8-Bit besitzt. Dies wir in einem gesamten Wert umgerechnet und mit der dementsprechenden Formel berechnet und ausgegeben.

Nach den Vordefinitionen werden die Standardvariablen für den *AD7793* befüllt. Anschließend wird in die Initialisierungsfunktion die Formel sowie die *struct*, welche die Konfiguration enthält, übergeben (Siehe Listing 3.5).

Listing 3.5: Initialisierung des *AD7793*-Treibers

```
153      ...
154      /*main.c*/
155      configuration.gain = 1;        // 2.5V
156      configuration.updateRateHz = 123;    // 123 Hz
157      configuration.unipolar = true;
158      ...
```

```
574      ...
575      AD7793_Init(formulaForRMeas,&configuration);
576      ...
```

ADC Standard Konfiguration

Nach der Initialisierung wird der *ADC*-Treiber in den *default configuration state* geführt. Bei diesem Zustand werden die Standardeinstellungen des *ADC* vorgenommen. Hier müssen davor gewisse Register beschrieben werden, um den *AD7793* einsatztüchtig zu machen. Wie aus der Schaltung 3.6 zu entnehmen ist, werden die analogen Daten im *AIN2*-Kanal eingelesen. Hierbei muss im *IO*-Register des *AD7793* dementsprechend beschrieben werden. Was wiederum auch eine Relevanz besitzt,

sind die vordefinierten Standardwerte wie die Abtastrate und der Eingangsstrom, welche in den *AD7793* eingestellt werden. Die Einstellung für die Ausgabe in unipolaren und bipolaren wird wiederum in diesem Zustand vorgenommen. Wenn der Treiber in dem Fall neu initialisiert wird oder ein Neustart des *ADC*s durchgeführt wird, werden die existierenden Konfigurationen in die vordefinierte Standardeinstellung zurückgesetzt. Nach der Konfiguration geht der Treiber in den *Idle*-Zustand über.

ADC Lesen

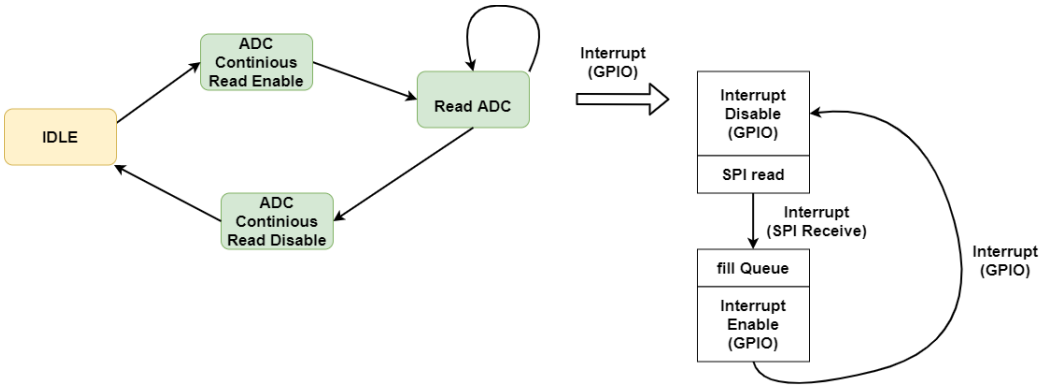


Abbildung 3.14: Genaue Darstellung des Lesezustandes sowie die *Interrupt*s

Nach der Konfiguration, gefolgt vom *Idle*-Zustand wird der *ADC* automatisch in den kontinuierlichen Lesemodus versetzt, wie in Abbildung 3.14 dargestellt. Dabei ist das Ziel, durch einen *Interrupt* die Werte des *ADC*s schnell und oft wie möglich auszulesen ohne die Verzögerungszeit der Zustandsmaschine mitzuberücksichtigen.

Der *ADC* versetzt periodisch den DOUT/$\overline{RDY}$-Ausgang bzw. den *MISO*-Ausgang auf *LOW*, welches im Kapitel 3.2.2.3 genauer erläutert wurde. Darauffolgend wird im *Read-ADC*-Zustand ein *GPIO Interrupt* aktiviert, um eine fallende Taktflanke am *MISO*-Ausgang zu detektieren. Wenn anschließend eine fallende Taktflanke erfolgt, wird der *EXTI Interrupt* aufgerufen und der Wert des *ADC* wird über den *SPI*-Bus gelesen. Nach einem erfolgreichem lesen wird der *Interrupt* des *SPI*s aufgerufen und füllt den Wert in einer Warteliste. Anschließend wird der *Interrupt* des *GPIO* wieder aktiviert und der Zyklus beginnt von vorne los. Das gegenseitige Aufrufen der *Interrupt*s hat den Vorteil, dass der *MCU* weniger belastet wird und so kein blockierendes System herrscht. Parallel wird im Read-Zustand überprüft, ob sich Werte in der Queue befinden. Wen dies der Fall ist, wird die Liste ausgelesen.

Das Kontinuierliche lesen kann beliebig oft gestoppt und gestartet werden. Es ist zu beachten, dass bei einer Konfiguration und einem Reset der Lesemodus angehalten

werden muss. Wenn das Lesen gestoppt wird, gelangt der Treiber wieder in den *Idle State* und verweilt so lange im Zustand, bis der Lesemodus wieder gestartet wird. Der *Interrupt*-Zyklus wird durch das Anhalten des Lesemodus unterbrochen.

Um genauer zu verstehen, wie die Prozedur abläuft, wird auf die entsprechenden Codebeispiele eingegangen. Es ist anzumerken, dass die Beispiele für das Verständnis gekürzt sowie leicht verändert wurden.

Nachdem der *GPIO EXTI* ausgelöst wurde, wird dementsprechende *Callbackfunktion* aufgerufen (Siehe Listing 3.6). Im *Callback*, wird der *EXTI Interrupt* deaktiviert, um während der *SPI*-Kommunikation sie nicht auszulösen. Hierbei wird mit einer Abfrage sichergestellt, ob der Lesemodus aktiviert ist. Wenn dies der Fall ist, wird die *SPI*-Kommunikation über den *DMA* durchgeführt. Für den Fall, dass der Lesemodus deaktiviert ist, wird ein *stopADC*-Flag gesetzt, um sicher den *Interrupt*-Zyklus zu stoppen.

Listing 3.6: *EXTI Callback*

```
943   /*AD7793.c*/
944   void GPIO_EXTI_Callback(uint16_t GPIO_Pin){
945     NVIC_DisableIRQ(EXTI_IRQn);
946
947     if(stopRead){
948       stopADC = true;
949       ...
950     }
951     else{
952       ...
953       SPI_TransmitReceive_DMA(spi2, dataRegisterBuffer, 3);
954     }
955   }
```

Wenn die *SPI*-Kommunikation erfolgreich war, wird der *SPI Interrupt* ausgelöst und der *Callback* aufgerufen (Siehe Listing 3.7). Im *Callback* wird abgefragt, ob der *stopADC* gesetzt wurde, um keine weiteren Werte in die Warteliste reinzuschreiben. Doch wen der Lesemodus aktiv ist, werden die 24-Bit Werte, die sich im *dataRegisterBuffer* befinden in den *adcReadDataBuffer* geschrieben. Anschließend, werden die Daten in die *Queue* geschrieben und der *EXTI Interrupt* wird wieder aktiviert.

Listing 3.7: *SPI* communikation complete *Callback*

```
968   /*AD7793.c*/
969   void SPI_CpltCallback(SPI_HandleTypeDef *spi){
970     if(stopADC){
971       ...
972     }
973     else{
974       ...
975       memcpy(adcReadDataBuffer,dataRegisterBuffer,3);
976       writeToQueue(xQueue,qMessage);
977       NVIC_EnableIRQ(EXTI_IRQn);
978       ...
979     }
980   }
```

Im Lesezustand werden die Werte in der *Queue* gelesen (Siehe Listing 3.8). Hierbei wird überprüft, ob sich in der Warteliste Werte befinden.

Listing 3.8: Auslesen der Daten in der Warteliste

```
285   /*AD7793.c*/
286   void readQueue(void){
287     if(xQueue != NULL){
288       readFromQueue(xQueue, rxMessage);
289     }
290     ...
291   }
```

3.9

Der Funktionsaufruf für das Lesen des *ADC*s wird im Listing 3.9 dargestellt. Hierbei muss ein *Callback* definiert werden, welches, dass Wert des FG-Widerstands beinhaltet. Der *Callback* wird dann aufgerufen, sobald ein Element aus der *Queue* gelesen wurde.

Listing 3.9: Funktion für das Lesen des Widerstandes und die dazugehörige *Callback*

```
28    /*STREAM.c*/
29    void readValueCallback(resistanceValue){
30    /*Im resistanceValue befindet sich der Widerstandswert des FG-
          Drahtes.Der Entwickler kann entscheiden, was er mit diesem
          Werten machen moechte.*/
31    }
32    ...
```

```
54    /*STREAM.c*/
55    ...
56    AD7793_readValue(readValueCallback);
57    ...
```

Wie viel Werte in einer Sekunde ausgelesen werden, ist von der Abtastrate des *ADC* abhängig. Dies kann maximal 4,17 bis 470 Hz betragen.

ADC Konfiguration

Während der *ADC*-Treiber sich im *Idle*-Zustand befindet, ist der Treiber in der Lage, in den Konfigurationszustand zu gelangen. In diesem Zustand ist der Treiber fähig, die Abtastrate sowie den Eingangstrom zu verändern oder sie zu lesen, siehe Abbildung 3.13. Nach der Veränderung wird der *ADC*-Treiber wieder in den *Idle*-Zustand zurückgeführt.

Die jeweiligen Konfigurationsfunktionen werden im Listing 3.10 dargestellt. Dabei muss sichergestellt werden, dass der *ADC* beim Konfigurieren und beim Lesen der Konfiguration nicht im kontinuierlichen Lesezustand befindet. Durch diesen Grund wird der Lesezustand angehalten und nach dem Konfigurieren wieder gestartet.

Listing 3.10: Funktion für das Konfigurieren des *ADC*s

```
121     /*HUPI_Commands.c*/
122     ...
123     AD7793_stopRead();
124     ...
125     /*Die Abtastrate betreagt 4.17 bis 470 Hz*/
126     AD7793_changeUpdateRate(updateRateValue);
127     ...
128     AD7793_runRead();
129     ...
```

```
171     /*HUPI_Commands.c*/
172     ...
173     AD7793_stopRead();
174     ...
175     /*Der Gain kann einen Wert von 1, 2 und 4 betragen. Bei hoeren
            Werten wuerden falsche Ausgabewerte entstehen.*/
176     AD7793_changeGain(gainValue);
177     ...
178     AD7793_runRead();
179     ...
```

Um die aktuellen Konfigurationen auszulesen, muss nur die gewisse Lese Funktion, welches im Listing 3.11 dargestellt wird, aufgerufen werden. Das Leseprinzip ähnelt wie im Listing 3.9.

Listing 3.11: Funktion für das Lesen der *ADC*-Konfiguration

```
240     /*HUPI_Commands.c*/
241     ...
242     AD7793_stopRead();
243     ...
244     /*Die Abtastrate wird im Callback dargestellt.*/
245     AD7793_readUpdateRate(readUpdateRateCallback);
246     ...
247     AD7793_runRead();
248     ...
```

```
269     /*HUPI_Commands.c*/
270     ...
271     AD7793_stopRead();
272     ...
273     /*Der Gain Wert wird im Callback dargestellt.*/
274     AD7793_readGain(readGainCallback);
275     ...
276     AD7793_runRead();
277     ...
```

ADC Reset

Der *ADC* kann nur resettet werden, wenn der Treiber sich im *Idle*-Zustand befindet. Bei einem Reset wird der *ADC* komplett von neu gestartet und die Konfigurationswerte werden in die Standardeinstellungen zurückversetzt. Dabei durchläuft es den Standardkonfigurationszustand und landet im *Idle*-Zustand. Der Funktionsaufruf wird im Listing 3.12 dargestellt.

Listing 3.12: Funktion für das neu starten des *ADC*s

```
318     /*HUPI_Commands.c*/
319   ...
320   AD7793_stopRead();
321   ...
322   AD7793_resetDevice();
323   ...
324   AD7793_runRead();
325   ...
```

ADC Error

Der ADC Treiber landet im *Error*-Zustand, wenn die *SPI*-Transaktion fehlgeschlagen wurde. Hierbei bleibt es so lange im selben Zustand und versendet über den

SPI Daten, bis eine stabile *SPI*-Kommunikation möglich ist. Nachdem das Problem gelöst wurde, wird ein Neustart durchgeführt.

B. DS18B20-Treiber

Im Treiber des Temperatursensors wurden nur die Temperaturwerte des Sensors ausgelesen und keine weiteren Konfigurationen durchgeführt. Hierbei wird kontinuierlich auf das Register, welches die Temperaturwerte beinhalten, zugegriffen. Im Kapitel 3.2.2.4 wird die Architektur des *DS18B20* sowie der Zugriff auf das Temperaturregister genau erläutert.

Anhand eines Codebeispiels wird die Verwendung des Treibers dargestellt (Siehe Listing 3.13).

Listing 3.13: Auslesen der Temperaturdaten

```
18  /*DS18B20.h*/
19  #define SKIP_ROM 0xCC
20  ...
21  #define CONVERT_TEMPERATUR 0x44
```

```
21  /*DS18B20.c*/
22  #define TEMPERATUR(msb,lsb) (float)(((msb<<8)|(lsb))/16.0)
23  ...
```

```
37  float getTemperature(void){
38    OneWire_Reset(oneWire);
39    ...
40    OneWire_WriteByte(oneWire, SKIP_ROM);
41    OneWire_WriteByte(oneWire, CONVERT_TEMPERATUR);
42    ...
43    OneWire_Reset(oneWire);
44    ...
45    OneWire_WriteByte(oneWire, SKIP_ROM);
46    OneWire_WriteByte(oneWire, FUNCTION_COMMANDS_READ_SCRATCHPAD);
47
48    temperaturByte[0] = OneWire_ReadByte(oneWire);
49    temperaturByte[1] = OneWire_ReadByte(oneWire);
50
51    return TEMPERATUR(temperaturByte[1], temperaturByte[0]);
52  }
```

Es wurde ein Makro definiert, welcher in der Lage ist, einen 16-Bit AD-Wert in einen Temperaturwert umzurechnen. Anfangs einer *One-Wire*-Übertragung wird der *DS18B20* neu gestartet, um diesen in einem definierten Zustand zu versetzten. Da keine weiteren Einstellungen vorgenommen werden, wird außerdem in jeder Kommunikation der *ROM* mit dem Wert 0xCC, übersprungen, um in den *SCRATCHPAD* zu gelangen. Um die Temperaturwerte auszulesen wird als Erstes der Wert 0x44 gesendet, um den AD-Wert in Grad Celsius umzurechnen. Als Nächstes wird ein Lesebefehl gesendet, um das Temperaturregister, welches 2-Byte groß ist, auszulesen. Anschließend wird in einem *Array* die Temperaturdaten geschrieben. Zum Schluss werden die Daten in dem definierten Makro übergeben und als *float*-Wert zurückgegeben.

C. Habemus Universal Production Interface (HUPI)

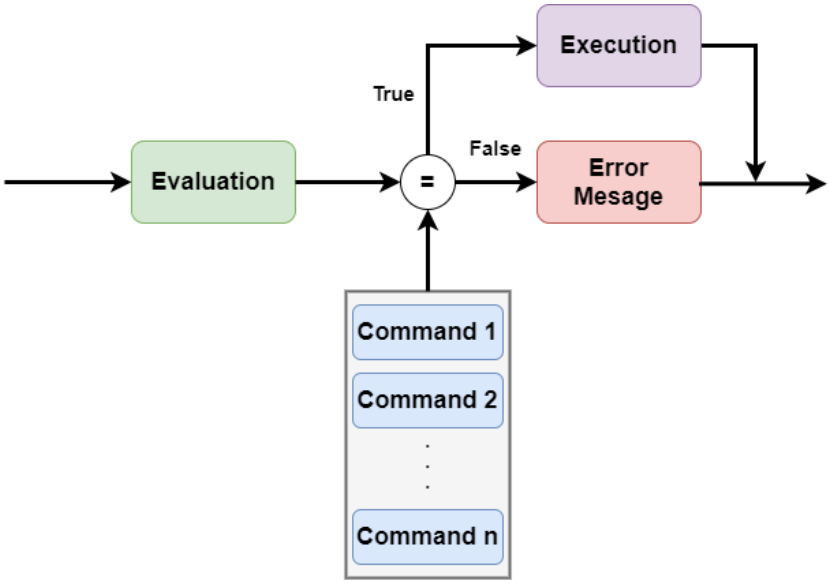


Abbildung 3.15: Arbeitsweise des *HUPI*

Der *HUPI* ist ein Modul, was von der Firma habemus entwickelt wurde und ist ein *Command Parser* und *Handler*, welches für verschiedene Interfaces benutzt werden kann. In diesem Projekt dient es dazu, selbstdefinierte Kommandos über UART zu erkennen und auszuführen. Die Gegenstelle für die Kommunikation ist ein Terminal, wodurch die Befehle eingegeben werden. Im Anschluss werden die Befehle ausgewertet und mit den definierten Kommandos verglichen. Wenn der Befehl im *HUPI* definiert ist, wird dementsprechend das Kommando ausgeführt, andernfalls wird eine Fehlermeldung im Terminal ausgegeben.

D. Stream

Der *HUPI* wurde nur für Befehle definiert, die synchron ausgeführt werde. Hierbei ist es alleine mit dem *HUPI* nicht möglich, kontinuierliche Werte des *AD7793*, sowie *DS18B20* auszulesen.

Das *Stream*-Modul dient als Ergänzung und kann nur über den *HUPI*-Modul aktiviert werden. Wenn letztendlich der *Stream*-Modul aktiviert wurde, werden die Werte des *ADC*, sowie des Temperatursensors wiederholt ausgelesen. Die Einstellung des ADCs sind während des Streamings nicht möglich. Mit einem Stopp-Befehl wird der *Stream* angehalten und der ADC kann wieder Eingestellt werden. Da der *UART full Duplex* fähig ist, können während des *Stream*s die Daten empfangenen und währenddessen ein Stopp-Befehl versendet werden.

Die Interaktion aller Komponente miteinander wird im kommenden Kapitel genauer erläutert.

Die gesamte Applikation

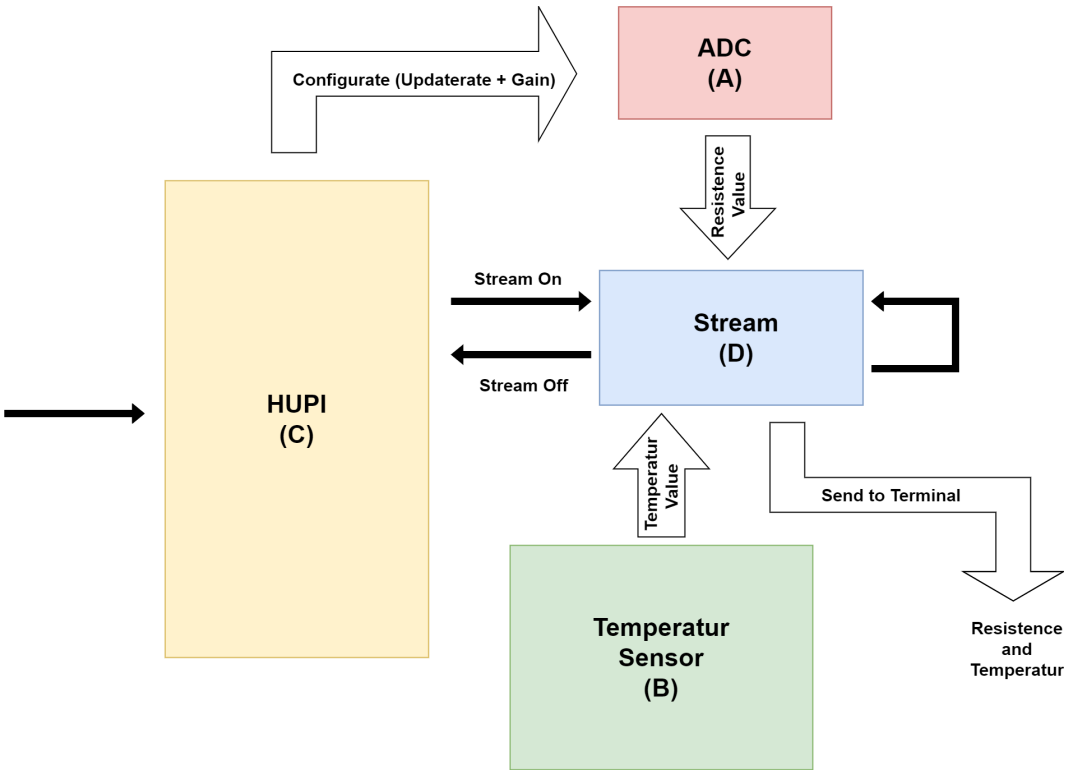


Abbildung 3.16: Aufbau aller Firmware-Module

Wie in Abbildung 3.16 entnommen werden kann, wird die gesamte Firmware in vier Module unterteilt, welches im vorherigen Kapiteln genauer erläutert wurden.

(A) AD7793

(B) DS18B20

(C) HUPI

(D) Stream

Als Erstes werden die Treiber, die für das Messen der Temperatur sowie des FGL-Widerstandes benötigt werden, initialisiert. Im *HUPI*-Modul werden die Befehle, die im Terminal versendet wurden, empfangen und ausgewertet. Hierbei kann beliebig die Konfiguration des *ADC* vorgenommen werden. Wenn anschließend vom Terminal der Befehl für das Streamen empfangen wird, befindet sich die Firmware im *Stream*-Modul. Im *Stream* werden kontinuierlich die Daten des *AD7793*, sowie *DS18B20* ausgelesen und die Werte werden im Terminal dargestellt. Während des Streamens können keine Konfigurationen vorgenommen werden. Der *Stream* kann nur verlassen werden, wenn ein Stopp-Befehl vom Terminal empfangen wurde.

Im folgenden Kapitel wird genauer auf das Programm der Firmware eingegangen.

3.3.2.2 Die Firmwareapplikation

In diesem Kapitel wird genauer über die entwickelte Firmware eingegangen.

Die allgemeine Arbeitsweise der Firmware

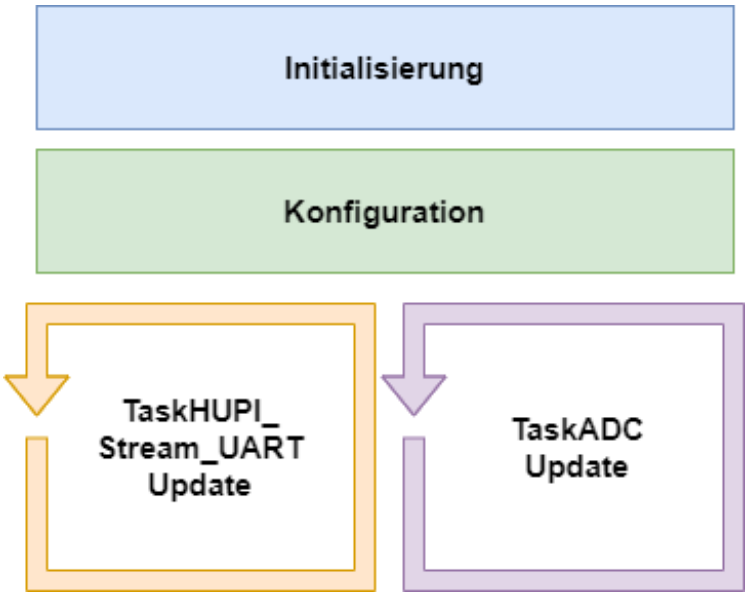


Abbildung 3.17: Gesamte Programm Ablauf abstrakt erklärt

Wie im 3.17 dargestellt wird im Vorfeld die benötigten Komponenten initialisiert und konfiguriert. Anschließen arbeitet die Firmware mit zwei FreeRTOS Tasks, welche der Aktualisierung des Zustandsautomaten des *ADC* und die *UART*-Kommunikationserhaltung des *HUPI*s und *Stream*s dient. Das Lesen der Temperaturwerte erfolgt im *HUPI_Stream_UART Update* Task, da die Auswertung der Messwerte, nicht so oft erfolgen muss, wie vom *AD7793*. Im Folgenden wird genauer darauf eingegangen, wie genau dies erfolgt.

Die Initialisierung der einzelnen Komponenten

Als ersten Schritt werden die wesentlichen Headerdateien benötigt (siehe 3.14).

Listing 3.14: Initialisierung der benötigten Bibliotheken

```
22    /*main.c*/
23    #include "main.h"
24    #include "cmsis_os.h" //Wird fuer das FreeRTOS benoetigt
25
26    ...
27
28    /*BSP*/
29    #include "AD7793.h"  //Treiber fuer den ADC
30    #include "DS18B20.h" //Treiber fuer den Temperatursensor
31
32    /*UART-Kommunikation mit dem Rechner*/
33    #include "HUPI.h"
34    #include "STREAM.h"
35    ...
```

Dabei wird die Bibliothek für den *FreeRTOS*, der *BSP* für den *ADC* sowie des Temperatursensors und die Bibliotheken für die Applikationskomponenten eingefügt. Im nächsten Schritt werden die relevanten Initialisierungen und Konfiguration durchgeführt, um das Programm auszuführen.

Listing 3.15: Initalisierung und Konfiguration der Firmware

```
134   /*main.c*/
135   ...
136   /*Initialisierung der HAL Bibliothek*/
137   HAL_Init();
138   ...
139   /*Die Konfiguration der Taktfrequenz*/
140   SystemClock_Config(); //
141   ...
142   /*Die Initialisierung der Peripherien*/
143   MX_GPIO_Init();
144   MX_DMA_Init();
145   MX_SPI_Init();
146   MX_USART_UART_Init();
147   ...
```

Um die *HAL*- und die *LL*-Bibliothek zu benutzen, müssen diese zuerst initialisiert werden. (siehe 3.15, Zeile 137) Im nächsten Schritt wird die Taktfrequenz konfiguriert und anschließend die benötigte Peripherie initialisiert.

Listing 3.16: Das Kreieren der Tasks

```
163     /*main.c*/
164     /*Die Initialisierung des Scheduler*/
165     osKernelInitialize();
166     ...
167     /*Die Erstellung der Task*/
168     /*Task1*/
169     TaskUART_UpdateHandle = osThreadNew(StartTaskUART_Update, NULL, &
            TaskUART_Update_attributes);
170
171     /*Task2 */
172     TaskADC_UpdateHandle = osThreadNew(StartTaskADC_Update, NULL, &
            TaskADC_Update_attributes);
173     ...
174     /*Starte Scheduler*/
175     osKernelStart();
```

Im Listing 3.16 werden die benötigten Tasks erstellt. Dabei wird als Erstes der Scheduler initialisiert, um die Prioritäten zwischen den Task zu definieren. Im nächsten Schritt werden die jeweiligen Tasks instanziiert. Dabei werden die Prioritäten, Größen, sowie der Name des Tasks bestimmt. Anschließend wird das System gestartet.

Die gesamte Arbeitsweise der Firmwareapplikation

Listing 3.17: Task für den HUPI_Stream_UART-Kommunikation

```
540     /*main.c*/
541     void TaskHUPI_Stream_UART_Update(void *argument)
542     {
543       HUPI_configurateCommands();
544       /*Die Initialisierung des Temperatursensors*/
545       DS18B20_Init();
546
547       /*Unendlichkeitsschleife*/
548       for(;;)
549       {
550         /*Das An- und Ausschalten der Streams*/
551         if(streamOn){
552           STREAM_Update();
553         }
554         else{
555           HUPI_ReceiveAndUpdate();
556         }
557       }
558     }
```

Der erste Task, welches im Listing 3.17 darstellt wird, hat die Aufgabe den *UART*-Kommunikation zu erhalten sowie die Werte des *DS18B20* auszulesen. Die Priorität wurde im Gegensatz zum zweiten Task höher gestellt, um während des Programmablaufs, die vom Terminal empfangene Befehle auszulesen und dementsprechend zu reagieren. Als Erstes wird *HUPI* konfiguriert sowie der Treiber des *DS18B20* initialisiert. Durch die Variable *streamOn* wird entschieden, ob die Firmware sich im *Stream*- oder *HUPI*-Modus befindet. Das Auslesen der Temperaturwerte erfolgt im *Stream*, durch einen einfachen Funktionsaufruf.

Listing 3.18: Task für die Aktualisierung der ADC-Zustandsmaschine

```
573     /*main.c*/
574     void TaskADC_Update(void *argument)
575     {
576       /*Die Initialisierung des ADC-Treibers*/
577       AD7793_Init(formulaForRMeas,&config,initSuccess);
578 
579       /*Unendlichkeitsschleife*/
580       for(;;)
581       {
582         AD7793_Update();
583       }
584 
585     }
```

Im zweiten Task geht es nur um den *AD7793*-Treiber (siehe 3.18). Es wird im ersten Schritt die Treiber initialisiert. Dabei wird der Zustandsautomat automatisch in den Lesemodus versetzt. Anschließend wird in einer Schleife der Zustandsautomat aktualisiert. Dies hat den Vorteil, dass die Daten der FGL kontinuierlich ohne Unterbrechung ausgelesen werden und dabei parallel andere Prozesse bearbeiten können.

4 Fazit und Ausblick

4.1 Fazit

Die Aufgabenstellung "Entwicklung einer embedded Firmwarelösung zur Auswertung von Formgedächtnisdrähten auf Basis eines *STM32*-Mikrocontrollers" konnte erfolgreich umgesetzt werden. Dabei wurde über die Thematik der physikalischen Eigenschaften der Formgedächtnislegierung sowie die Anwendungsmöglichkeiten in der Praxis recherchiert. Außerdem wurden über die Hardwarekomponenten und die Bussysteme, welche für die Kommunikation der Hardwareteile zuständig sind, auseinander gesetzt. Nach der Anforderungsanalyse wurden die Treiber *AD7793* und *DS18B20* für die Firmware entwickelt.

Der Widerstand der Formgedächtnislegierung wurde mithilfe der Vierleitermessung gemessen, um keine Verfälschung der Messdaten zu erhalten. Für die Entwicklung der *AD7793*-Treiber wurde die Architektur in Form eines Zustandsautomaten geplant. Dabei wurde überlegt, wie der Mikrocontroller vom Analog-Digital-Wandler die Widerstandsmessdaten des FGLs am effizientesten lesen kann. Es wurde auf ein nicht blockierendes System zugegriffen, welches mithilfe des *DMA* sowie des *SPI Interrupt* realisiert werden konnte. Während der Entwicklung wurde der Treiber zunächst auf einem Evaluationsboard ausgetestet. Im nächsten Schritt wurde der Treiber für den *DS18B20* entwickelt. Hierbei wurde ein einfaches Zugreifen mithilfe des *One-Wire*-Busses in das Temperaturregister vorgenommen.

Nach den Treibern wurde *HUPI* in die Firmware mit integriert, welches dafür sorgt, mithilfe eines Terminal die Daten des *AD7793* sowie *DS18B20* über den *UART* auf dem Bildschirm darzustellen und auszulesen. Hierbei dient Terminal als Gegenstelle und Simulation für die Eingabe der AT-Kommandos. Für das kontinuierliche Auslesen der Daten wurde der *Stream*-Modul entwickelt.

Die finale Firmware wurde auf den geplanten Board geladen sowie ausgetestet. Das Board wurde außerdem mit einem TTL-232R *UART*-Kabel mit dem Rechner verbunden, um über den Terminal die Werte auszulesen sowie Befehle zu verschicken. Auf dem Terminal wird der Widerstand des FGLs sowie der Temperaturwert des

Systems dargestellt. Über den Terminal ist der Anwender in der Lage, die Abtastrate sowie den Eingangstrom für die Auswertung des FGL-Widerstandswertes zu konfigurieren.

Die Firmware kann für andere Anwendungen erweiter sowie verändert werden. Außerdem ist es möglich, die Software für verschiedene elektrische Schaltungen zu verwenden. Hierbei muss die Firmware dementsprechend angepasst werden.

Durch Vorausgegangene Messungen wurden für die FG-Drähte gewisse Erkenntnisse gewonnen. Dabei wurden mit Formgedächtnisdrähten gearbeitet, die einen Querschnittsfläche von 0,15 und 0,3 mm besitzen. Der Veränderbare Widerstand weist eine Spannweite von 3 bis 11Ω auf. Bei höheren Werten besteht das Risiko, dass die Drähte reißen könnten. Durch das Einstellen des Eingangsstroms kann die Messungenauigkeit verringert werden, indem man den Gain erhöht. Bei einer Update Rate von 50 Hz und mit einer Gain von 2 erhält man eine Abweichung von ca. ±500mΩ. Wen die Verstärkung auf maximal 4 eingestellt wird, sinkt die Abweichung auf ca. ±200$\mu\Omega$. Die Messgenauigkeit des Temperatursensors beträgt ca. ±0,5 °C

4.2 Ausblick

Für eine weitere Herangehensweise nach dem Projekt wäre die Kommunikation des *ESP32* mit einem Empfängermodul geplant. Hierbei muss zunächst die *UART*-Kommunikation des *STM32*, sowie des *ESP32* hergestellt werden, welches die AT-Kommandos übergibt. Zu guter Letzt müssen die Daten vom *ESP32* zu Empfängermodul über WLAN verschickt werden. Hier würde sich eine *MQTT*-Verbindung als Möglichkeit anbieten.

Das Gerät kann für kommerzielle Zwecke verwendet werden, wie zum Beispiel für die Überwachung von jeglichen maschinellen Modulen und Bauteile innerhalb einer Produktionslinie. Auch im Bereich der Messtechnik könnte das Produkt Fuß fassen. Hierbei müsste es für die jeweilige Nutzung angepasst, erweitert oder verbessert werden. Außerdem könnte es als Ergänzungskomponente eines gesamten automatischen Systems dienen.

Literaturverzeichnis

[**ADAPTRONIK 2016**]
ADAPTRONIK, Frauenhofer: *HOCHELASTISCHE FORM-GEDÄCHTNIS-SENSORIK ZUR DEHNUNGSMESSUNG*. https://www.adaptronik.fraunhofer.de/content/dam/adaptronik/documents/Handzettel2016. – Eingesehen am 04.11.2021 2.19

[**Ana 2004**]
Analog Devices (Hrsg.): *AD7792/AD7793*. Analog Devices, 2004–2007. – Rev. B 3.7, 3.2, 3.2.2.3, 3.8

[**Association 2000**]
Association, Open Source H.: *A Resolution to Redefine SPI Signal Names*. https://www.oshwa.org/a-resolution-to-redefine-spi-signal-names/, 2000. – Eingesehen am 11.02.2022 2.1.1

[**Bäker 2014**]
Bäker, Martin: *Funktionswerkstoffe*. Springer Vieweg, Wiesbaden, 2014. – 1–299 S. http://dx.doi.org/10.1007/978-3-658-02970-8. http://dx.doi.org/10.1007/978-3-658-02970-8. – ISBN 978–3–658–02969–2 2.3.1, 2.8, 2.3.2, 2.3.2, 2.3.2, 2.3.2, 2.11, 2.3.2, 2.12, 2.3.2, 2.13, 2.3.2, 2.14, 2.3.3.1, 2.15, 2.3.3.1, 2.16, 2.3.3.2, 2.17, 2.3.3.3, 2.18

[**BorTec**]
BorTec: *AUSTENIT*. https://www.edelstahl-haerten.de/glossar/austenit/, . – Eingesehen am 26.01.2022 2.3.1

[**C u. a. 1989**]
C, Hill S. ; Joseph, Jelemensky ; R, Heene M.: *Queued serial peripheral interface for use in a data processing system*. https://worldwide.espacenet.com/patent/search/family/022138902/publication/US4816996A?q=pn198 – Eingesehen am 11.02.2022 2.1.1

[**Colin M.L. Burnett 2010**]
Colin M.L. Burnett: *SPI timing diagram2*. https://commons.wikimedia.org/wiki/File:SPI_timing_diagram2.svg, 2010. – Eingesehen am 22.10.2021 2.2

[**Czechowicz u. Langbein 2000**]
Czechowicz, Alexander ; Langbein, Sven: *Formgedächtnislegierungen: Charakteristika und Potentiale in optischen und feinwerktechnischen Anwendungen.* Optik-Verlag, 2000. – Eingesehen am 12.02.2022 2.3.3.3, 2.3.4

[**DeepBlue 2021**]
DeepBlue: *STM32 DMA Tutorial – Using Direct Memory Access (DMA) In STM32.* https://deepbluembedded.com/stm32-dma-tutorial-using-direct-memory-access-dma-in-stm32/, 2021. – Eingesehen am 07.02.2021 2.2

[**Dhaker u. Eckstein 2022**]
Dhaker, Piyu ; Eckstein, Michael: *So funktioniert das Serial Peripheral Interface.* https://www.elektronikpraxis.vogel.de/so-funktioniert-das-serial-peripheral-interface-a-1006985/, 2022. – Eingesehen am 20.10.2022 2.1.1

[**Gümpel 2000**]
Gümpel, Paul: *Rostfreie Stähle: Grundwissen, Konstruktions- und Verarbeitungshinweise ; mit 28 Tabellen.* Expert-Verlag, 2000. – 1–264 S. – ISBN 978–3–8169–1735–9 2.3.1

[**Gümpel u. a. 2018**]
Gümpel, Paul ; Gläser, Stefan ; Jost, Norbert ; Mertmann, Matthias ; Seitz, Norman ; Strittmatter, Joachim: *Formgedächtnislegierung.* experten verlag, 2018. – 1–147 S. – ISBN 978–3–8169–2727–3 2.4

[**Hoffmann 2016**]
Hoffmann, Frank: *Faszination Kristalle und Symmetrie.* Springer Spektrum, Wiesbaden, 2016. – 1–321 S. http://dx.doi.org/10.1007/978-3-658-09581-9. http://dx.doi.org/10.1007/978-3-658-09581-9. – ISBN 978–3–658–09580–2 2.3.1

[**Hornbogen 1984**]
Hornbogen, Erhard: On the Martensite Start Temperature MS. In: *Zeitschrift für Metallkunde* (1984) 2.3.1

[**maxim integrated 2008**]
integrated maxim: *GUIDE TO 1-WIRE COMMUNICATION.* https://www.maximintegrated.com/en/design/technical-documents/tutorials/1/1796.html, Juni 2008. – Eingesehen am 18.12.2021 2.1.3

[**itwiki 2014**]
itwiki, Melancholia: *Austenit Struktur.* htt-

ps://commons.wikimedia.org/wiki/File:AISI_304_-_austenitic_structure.jpg, April 2014. – Eingesehen am 09.10.2021 2.9

[**ITWissen 2011**]
ITWissen: *Vierleitermessung*. https://www.itwissen.info/Vierleitermessung-four-wire-measuring.html, 2011. – Eingesehen am 09.02.2021 2.4

[**Kock 2010**]
Kock, Iris: *Phasenbildung, Phasenübergang und mechanische Eigenschaften des Funktionsmaterials Eisen-Palladium*. 2010 2.3.1

[**koganam 2007**]
koganam, Israel: *Martensite, Braude College Karmiel Israel, 2007 Kogan*. https://commons.wikimedia.org/wiki/File:Martensite,_Braude_College_Karmiel_Israel,_2007_Kogan.jpg?uselang=de, 2007. – Eingesehen am 09.10.2021 2.10

[**Kompendium**]
Kompendium, Elektronik: *Übertragungsgeschwindigkeit und Datenrate*. https://www.elektronik-kompendium.de/sites/kom/0212095.htm, . – Eingesehen am 09.02.2022 2.1.2, 2.1.2

[**KUNBUS**]
KUNBUS: *UART - Universal Asynchronous Receiver Transmitter*. https://www.kunbus.de/uart.html, . – Eingesehen am 11.02.2022 2.1.2, 2.1.2

[**MACHINE 2021**]
MACHINE, WAYBACK: *SPI - Serial Peripheral Interface*. https://web.archive.org/web/20190116220910/http://www.mct.de/faq/spi.html, 2021. – Eingesehen am 11.02.2022 2.1.1, 2.1, 2.1.1

[**max 2019**]
maximi intergrated (Hrsg.): *DS18B20*. maximi intergrated, 2019. – Rev. 6 2.1.3, 2.4, 2.1.3, 2.5, 2.1.3, 2.6, 3.2.2.4, 3.9, 3.10

[**Saure 2011**]
Saure: *TempMess 4 Leit*. https://commons.wikimedia.org/wiki/File:TempMess_-4_Leit.svg, September 2011. – Eingesehen am 14.12.2021 2.20

[**STM 2017**]
STMicroelectronics (Hrsg.): *STM32Cube firmware examples for STM32F1 Series*. STMicroelectronics, 2017. – Rev. 2 3.11

[**STM 2021**]
STMicroelectronics (Hrsg.): *RM0008*. STMicroelectronics, 2021. – Rev. 21 2.3

[**STMicroelectronics 2022**]
STMicroelectronics: *STM32CubeMP1 architecture*. https://wiki.st.com/stm32mpu/wiki/STM32CubeMP1_architecture, 2022. – Eingesehen am 09.11.2021 3.3.1

[**Wittgruber 2002**]
Wittgruber, Friedrich: *Digitale Schnittstellen und Bussysteme*. Vieweg+Teubner Verlag, Wiesbaden, 2002. – 1–181 S. http://dx.doi.org/10.1007/978-3-663-01615-1_7. http://dx.doi.org/10.1007/978-3-663-01615-1_7. – ISBN 978–3–528–17436–1 2.3

Ich, Abdurrahman Celep, Matrikel-Nr. 2023650, versichere hiermit, dass ich die vorliegende Arbeit mit dem Thema

> *Entwicklung einer embedded Firmwarelösung zur Auswertung von Formgedächtnisdrähten auf Basis eines STM32-Mikrocontrollers - habemus! electronic + transfer GmbH*

selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, wobei ich alle wörtlichen und sinngemäßen Zitate als solche gekennzeichnet habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

München, den 21. Februar 2022

Abdurrahman Celep