Fachhochschule
Dortmund
University of Applied Sciences and Arts

FB Informations- und Elektrotechnik

Bachelor Thesis
zur Erlangung des Akademischen Grades
Bachelor of Engineering

Programmierung eines STM32-Mikrocontrollers als steuerbares Spannungsmessgerät mit SCPI-Schnittstelle

Smaalia, Mohamed Mansour

Juni, 2023

Erstprüfer: **Prof. Dr.-Ing Michael Karagounis**
Zweitprüfer:**M.A. Alexander Walsemann**

Abstract

In dieser Bachelorarbeit wird ein steuerbares Spannungsmessgerät mit einem STM32 Mikrocontroller entwickelt. Für das Projekt wird der Analog-Digital-Wandler (ADC) auf einem STM32L476RG Nucleo Board zur präzisen Spannungsmessung verwendet.

Die Integration des SCPI-Protokolls (Standard Commands for Programmable Instruments) und des UART-Protokolls (Universal Asynchronous Receiver Transmitter) sind integraler Bestandteil des Systems und ermöglichen eine standardisierte Gerätesteuerung und zuverlässige Kommunikation.

Außerdem bietet das System eine Reihe von einstellbaren Messkonfigurationen, wie z.B. ADC-Auflösung, die Möglichkeit Oversampling zu aktivieren und bestimmte Kanäle auszuwählen.

Um mit dem Board zu kommunizieren und die Programmierung des Mikrocontrollers zu testen, wurde eine grafische Benutzeroberfläche (GUI) mit QT-Designer entwickelt. Die Schnittstelle ermöglicht die reibungslose Übertragung von Befehlen von der Qt-Anwendung zum Mikrocontroller, der effizient programmiert ist, um diese Befehle zu empfangen und auszuführen.

Die implementierte Lösung wird durch Software- und Hardwaretests getestet und die Ergebnisse werden vorgestellt. Die Arbeit schließt mit einer Zusammenfassung der erreichten Ziele und abschließenden Bemerkungen.

Inhaltsverzeichnis

Abbildungsverzichnis

Tabellenverzeichnis

1. Einleitung

In den letzten Jahren hat die Entwicklung von elektronischen Messgeräten und eingebetteten Systemen in Industrie und Forschung eine wichtige Rolle gespielt. Die kontinuierliche Verbesserung der Mikrocontrollertechnologie ermöglicht die Entwicklung komplexer und zuverlässiger Systeme, die eine Vielzahl von Aufgaben in unterschiedlichen Anwendungsbereichen erfüllen. Eine solche Anwendung ist die Entwicklung von Spannungsmessgeräten, die für die Überwachung und Steuerung elektrischer Schaltungen unerlässlich sind. Dabei ist eine einfache und effiziente Kommunikation zwischen diesen Geräten und einer Benutzerschnittstelle von großer Bedeutung, um eine benutzerfreundliche Bedienung und Analyse der Messergebnisse zu ermöglichen.

Die vorliegende Bachelorarbeit beschäftigt sich mit der Programmierung eines STM32-Mikrocontrollers als steuerbares Spannungsmessgerät mit SCPI-Schnittstelle. Dazu wird ein STM32L476RG Nucleo Board verwendet, dessen Analog-Digital-Wandler (ADC) zur Durchführung von Spannungsmessungen programmiert wird. Das entwickelte System wird über eine grafische Benutzeroberfläche (GUI) gesteuert und überwacht, die mit Qt Designer erstellt und in Python programmiert wurde. Die Kommunikation zwischen dem Mikrocontroller und der GUI-Anwendung erfolgt über UART- und SCPI-Befehle, die eine einfache und effiziente Steuerung des Messgeräts ermöglichen.

Die Arbeit beginnt mit einer Einführung in die verwendeten Technologien und Konzepte, gefolgt von einer detaillierten Beschreibung der Systemanforderungen und Spezifikationen. Anschließend wird die Programmierung des STM32-Mikrocontrollers und die Implementierung der SCPI-Befehle erläutert. Danach wird die Entwicklung der GUI-Anwendung mit Qt Designer und Python vorgestellt, einschließlich der Implementierung der verschiedenen Funktionen und der Kommunikation mit dem Mikrocontroller. Abschließend werden die Integration und der Test des entwickelten Systems beschrieben, bevor ein Fazit gezogen und ein Ausblick auf mögliche Erweiterungen und Verbesserungen gegeben wird.

2. Grundlagen

2.1 STM32-Mikrocontroller

STM32-Mikrocontroller sind eine Familie von 32-Bit-Mikrocontrollern, die auf der ARM Cortex-M-Serie von Prozessorkernen basieren. Sie werden von STMicroelectronics entwickelt und hergestellt und erfreuen sich aufgrund ihrer Leistungsfähigkeit, Energieeffizienz und Funktionsvielfalt großer Beliebtheit. STM32-Mikrocontroller sind in verschiedenen Serien erhältlich, die jeweils auf unterschiedliche Anwendungsanforderungen ausgerichtet sind und unterschiedliche Leistungs-, Speicher- und Peripheriemerkmale aufweisen.

Die in STM32-Mikrocontrollern verwendeten Prozessorkerne der ARM Cortex-M-Serie bieten eine skalierbare und energieeffiziente Lösung für eingebettete Anwendungen. Sie verfügen über eine RISC-Architektur (Reduced Instruction Set Computing), die eine schnellere Befehlsausführung und einen geringeren Stromverbrauch ermöglicht. Cortex-M-Cores gibt es in verschiedenen Versionen wie Cortex-M0, Cortex-M3, Cortex-M4 und Cortex-M7, die unterschiedliche Leistungsstufen und Zusatzfunktionen wie digitale Signalverarbeitung (DSP) oder Fließkommaeinheiten (FPU) bieten. [1]

STM32-Mikrocontroller bieten eine breite Palette an Peripheriegeräten und Kommunikationsschnittstellen und eignen sich daher für eine Vielzahl von Anwendungen.

Zu den allgemein verfügbaren Peripheriegeräten gehören GPIO-Pins (General Purpose Input/Output), Timer, Analog-Digital-Wandler (ADCs), Digital-Analog-Wandler (DACs), serielle Kommunikationsschnittstellen wie UART, SPI und I2C sowie erweiterte Kommunikationsprotokolle wie USB, Ethernet und CAN. Darüber hinaus bieten STM32-Mikrocontroller verschiedene Stromsparmodi, mit denen Entwickler den Stromverbrauch ihrer Anwendungen entsprechend den jeweiligen Anforderungen optimieren können.

Um die Entwicklung mit STM32-Mikrocontrollern zu erleichtern, bietet STMicroelectronics ein umfangreiches Set an Software-Tools und -Bibliotheken an. Das Softwarepaket STM32Cube enthält beispielsweise HAL-Bibliotheken (Hardware Abstraction Layer), Middleware-Komponenten und Beispielprojekte, die dem Entwickler einen

schnellen Einstieg ermöglichen. Mit dem STM32CubeMX-Tool steht ein grafisches Konfigurationswerkzeug zur Verfügung, das die Einrichtung der Pinbelegung des Mikrocontrollers, die Taktkonfiguration und die Initialisierung der Peripherie vereinfacht. Darüber hinaus gibt es verschiedene integrierte Entwicklungsumgebungen (IDEs) und Toolchains für die STM32-Entwicklung.

2.2 STM32L476RG Nucleo-Board

Das STM32L476RG Nucleo Board ist ein Entwicklungsboard von STMicroelectronics für die STM32L4-Serie, die für ihre Mikrocontroller mit extrem niedrigem Stromverbrauch bekannt ist. Es ist mit einem STM32L476RGT6 Mikrocontroller ausgestattet, der auf einem ARM Cortex-M4 Kern basiert und Frequenzen bis zu 80 MHz verarbeiten kann. Dieser Mikrocontroller bietet eine ausgewogene Kombination aus Leistung, Energieeffizienz und einer großen Auswahl an Peripheriegeräten, wodurch er für eine Vielzahl von Anwendungen geeignet ist, einschließlich der in diesem Artikel beschriebenen Spannungserfassungssysteme.

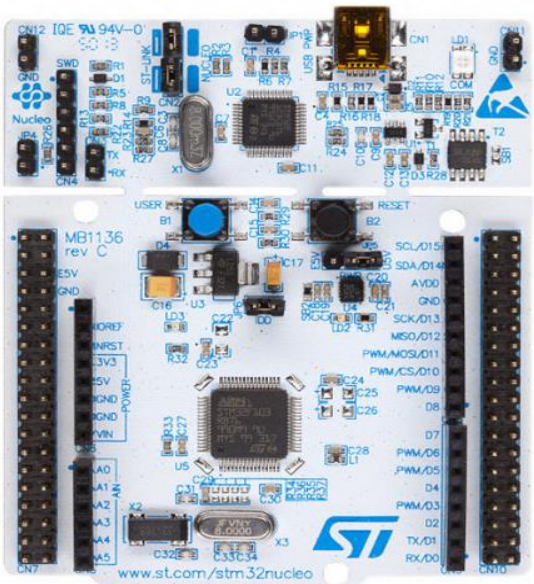

Abbildung 1: Stm32-L476rg Nucleo Board [2]

Zu den Hauptmerkmalen des STM32L476RG Nucleo-Boards gehören Speicher, Stromversorgung, Erweiterungsports, integrierter Debugger/Programmierer, Kommunikationsschnittstellen usw. Der Mikrocontroller STM32L476RGT6 verfügt über 1 MB

Flash-Speicher und 128 KB SRAM und bietet ausreichend Platz für Firmware und Laufzeitdaten. Hinsichtlich der Stromversorgung kann das Nucleo Development Board über einen USB-Port oder ein externes Netzteil mit Strom versorgt werden und unterstützt mehrere Stromsparmodi, um den Stromverbrauch je nach Anwendung zu optimieren. Das Board ist außerdem mit einem Arduino Uno-kompatiblen Erweiterungsanschluss ausgestattet, der die einfache Integration verschiedener Erweiterungskarten und Zusatzplatinen ermöglicht. Darüber hinaus verfügt es über einen Morpho-Erweiterungsport, der Zugriff auf alle I/Os des Mikrocontrollers bietet und eine Erweiterung mit benutzerdefinierter Hardware ermöglicht. Das Nucleo-Board verfügt über einen integrierten ST-LINK/V2-1-Debugger und Programmer für einfache Firmware-Entwicklung und Debugging. [3]

Weiterhin bietet der Mikrocontroller eine breite Palette an Kommunikationsschnittstellen wie USART, SPI, I2C, USB und CAN für den einfachen Anschluss externer Geräte und Komponenten. Das STM32L476RG Nucleo Board bietet somit eine flexible und leistungsfähige Plattform für die Entwicklung des Spannungsmesssystems. Seine umfangreiche Funktionalität ermöglicht in Kombination mit den zahlreichen Entwicklungstools und Softwarebibliotheken von STMicroelectronics ein schnelles Prototyping von Hardware- und Softwarekomponenten.

2.3 Analog-Digital-Wandler (ADC)

Analog-Digital-Wandler (ADCs) sind ein integraler Bestandteil vieler eingebetteter Systeme. Sie spielen eine wichtige Rolle bei der Umwandlung kontinuierlicher analoger Signale in diskrete digitale Werte, die zur Verarbeitung, Analyse oder Speicherung durch einen Mikrocontroller oder andere digitale Systeme verwendet werden können. ADCs werden in einer Vielzahl von Anwendungen wie Datenerfassung, Sensorschnittstellen und Signalverarbeitung eingesetzt.

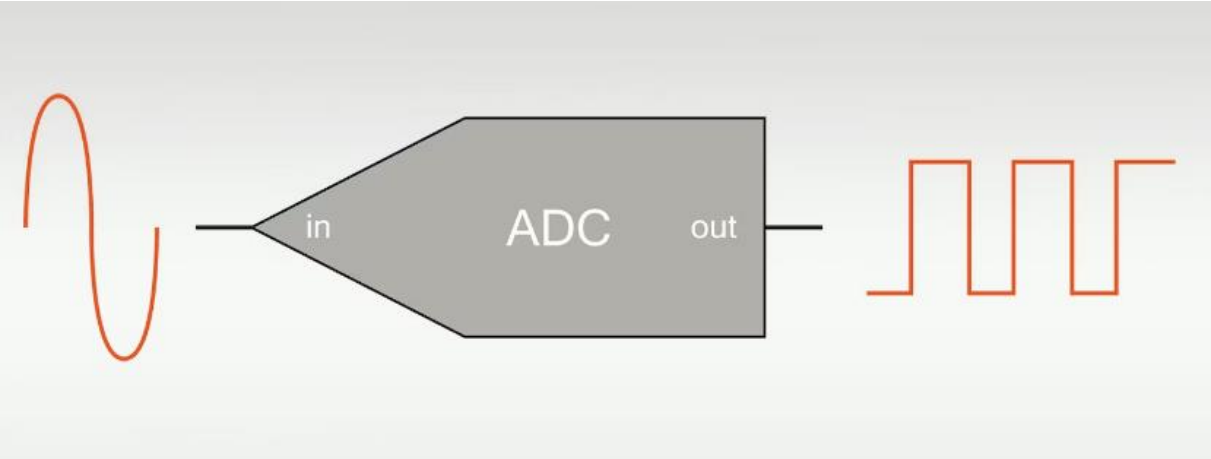

Abbildung 2: ADC-Signalverarbeitung: Von analog zu digital [4]

Das STM32L476RG Nucleo Board, welches in diesem Projekt verwendet wird, verfügt über eine 12-bit SAR ADC ausgestattet, der bis zu 16 Kanäle unterstützt. Dieser ADC ist eine geeignete Lösung für Spannungsmesssysteme, da er ein ausgewogenes Verhältnis zwischen Wandlungsgeschwindigkeit, Genauigkeit und Stromverbrauch bietet. Er kann in verschiedenen Modi wie single-ended oder differential betrieben werden und unterstützt einstellbare Abtastraten und Auflösungen. Da mehrere Kanäle verwendet werden können, ist das System in der Lage, Spannungen an mehreren Eingängen gleichzeitig zu messen, was die Gesamtfunktionalität und Flexibilität des Systems deutlich erhöht.

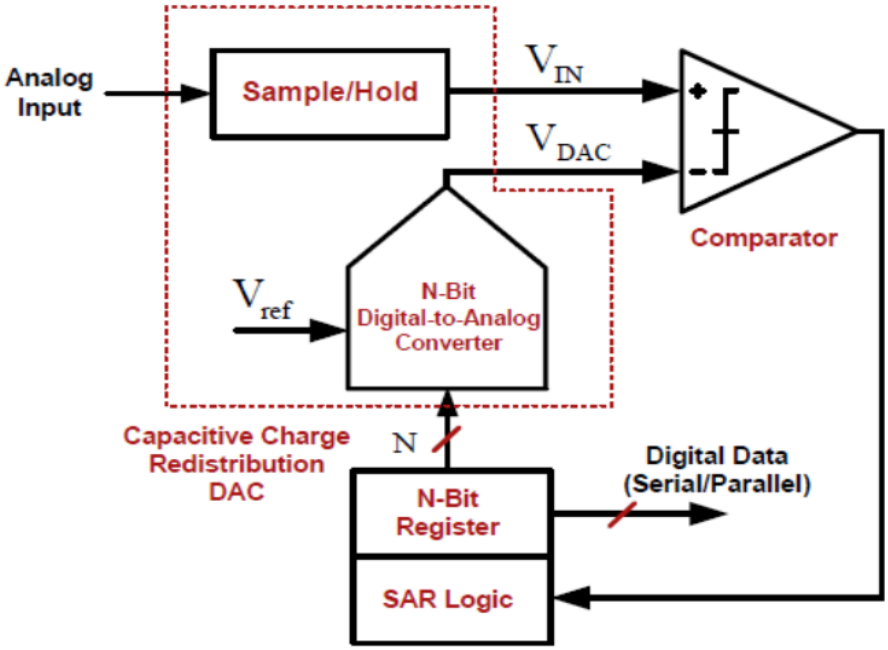

Abbildung 3: Blockschaltbild eines SAR-ADC-Wandlers [5]

Der SAR-ADC arbeitet nach dem Prinzip der sukzessiven Approximation. Dieser spezielle ADC-Typ verwendet einen binären Suchalgorithmus, um die Eingangsspannung zu approximieren. Er führt eine Reihe von Vergleichen durch und nähert sich sukzessive dem digitalen Wert an, der dem Eingangssignal entspricht. Das grundlegende Funktionsprinzip eines ADC besteht darin, das analoge Eingangssignal in festen Zeitabständen abzutasten. Dieser Abtastwert wird dann quantisiert, d.h. in eine diskrete digitale Darstellung umgewandelt. Die so gewonnenen Zahlenwerte werden dann in ein spezielles Format wie Binär- oder Zweierkomplementformat kodiert.

2.4 SCPI-Protokoll

Das SCPI-Protokoll (Standard Commands for Programmable Instruments) ist ein weit verbreitetes Kommunikationsprotokoll zur Steuerung und Kommunikation mit elektronischen Prüf- und Messgeräten. SCPI ist eine ASCII-basierte, menschenlesbare Sprache, die einen standardisierten Satz von Befehlen und Antworten zur Programmierung und Abfrage von Geräteeinstellungen, Messungen und Statusinformationen definiert.

SCPI wurde Anfang der 1990er Jahre in Zusammenarbeit mit mehreren großen Messgeräteherstellern entwickelt. Das Hauptziel bestand darin, eine einheitliche Befehlsstruktur zu schaffen, um die Geräteprogrammierung zu vereinfachen und die Interoperabilität zwischen Geräten verschiedener Hersteller zu verbessern. Infolgedessen hat sich SCPI zum De-facto-Standard für die Gerätesteuerung entwickelt und seine Einführung hat zu einer höheren Produktivität und kürzeren Entwicklungszeiten für Prüfsysteme geführt. [6]

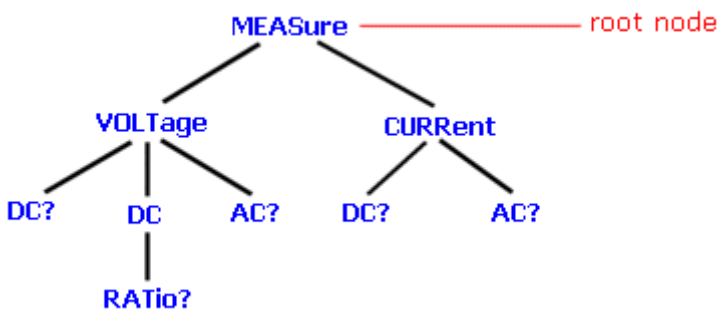

Abbildung 4: Beispiel für die SCPI-Befehlsstruktur [7]

Zu den Eigenschaften des SCPI-Protokolls gehört eine hierarchische Befehlsstruktur, bei der die SCPI-Befehle hierarchisch organisiert sind, mit Befehlen auf höherer Ebene und spezifischeren Unterbefehlen. Diese Struktur ermöglicht eine klare Organisation und vereinfacht das Hinzufügen oder Ändern von Befehlen. Darüber hinaus verwenden SCPI-Befehle mnemonische Abkürzungen, die leicht verständlich und selbsterklärend sind. Dieses für den Menschen lesbare Format erleichtert das Verständnis und die Anwendung des Protokolls und verkürzt die Lernkurve für die Geräteprogrammierung.

Befehlsparameter sind ein weiteres Merkmal des SCPI-Protokolls. Sie ermöglichen es Befehlen, bestimmte Einstellungen oder Aktionen zu definieren, und bieten dem Anwender Flexibilität bei der Steuerung verschiedener Aspekte des Gerätebetriebs. Darüber hinaus ermöglicht SCPI die Abfrage des Gerätestatus oder der Messergebnisse mit Hilfe von Abfragebefehlen, die in der Regel die gleiche Mnemonik wie der entsprechende Befehl jedoch mit einem Fragezeichen am Ende verwenden.

Schließlich enthält SCPI sowohl Standardbefehle, die für die meisten Geräte gelten, als auch gerätespezifische Befehle, die nur für bestimmte Geräte oder Geräteklassen relevant sind. Diese Kombination aus Standard- und gerätespezifischen Befehlen gewährleistet die Kompatibilität zwischen verschiedenen Geräten und ermöglicht gleichzeitig die Anpassung an kundenspezifische Anforderungen und Sonderfunktionen.

Im Rahmen dieser Arbeit wird das SCPI-Protokoll verwendet, um die Kommunikation zwischen dem STM32-Mikrocontroller-basierten Spannungsmesssystem und der grafischen Benutzeroberfläche (GUI) zu ermöglichen. Durch die Implementierung des SCPI-Protokolls kann das System über einen einheitlichen und standardisierten Befehlssatz gesteuert und abgefragt werden, was den Entwicklungsprozess vereinfacht und die Benutzerfreundlichkeit erhöht.

2.5 UART (Universal Asynchron Receiver Transmitter)

2.5.1 Grundlagen der UART-Kommunikation

Der Universal Asynchronous Receiver Transmitter (UART) ist eine serielle Kommunikationsschnittstelle, die häufig in Mikrocontrollern und anderen integrierten Schaltungen (ICs) verwendet wird. UART ermöglicht die asynchrone Datenübertragung zwischen zwei Geräten über zwei Signalleitungen: eine zum Senden (TX) und eine zum Empfangen (RX) von Daten. Asynchrone Kommunikation bedeutet, dass die Datenübertragung nicht auf einen gemeinsamen Takt abgestimmt ist. Stattdessen werden die Daten in Paketen mit Start- und Stoppbits übertragen, um den Beginn und das Ende jedes Datenpakets zu kennzeichnen.

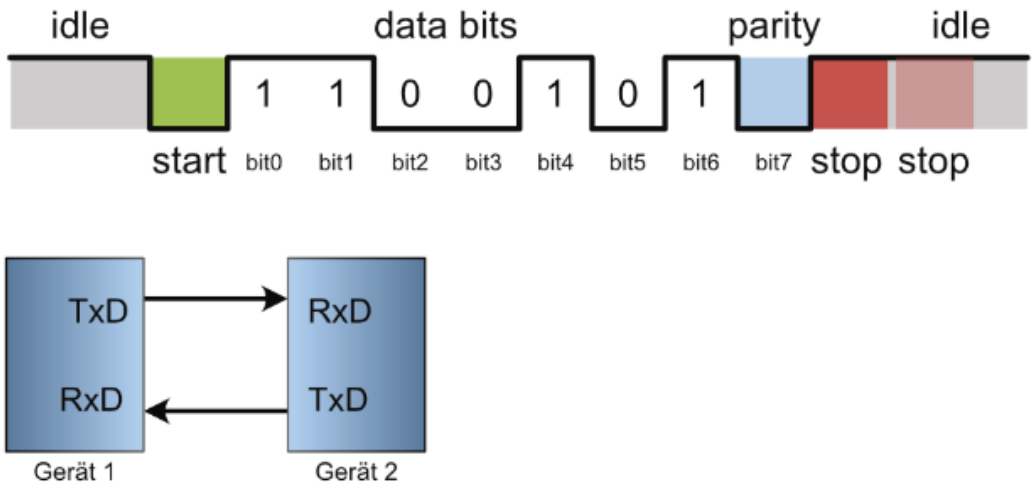

Abbildung 5: UART Datenübertragung [8]

2.5.2 Baudrate und Datenformat

Die Baudrate ist die Geschwindigkeit, mit der Daten über die UART-Schnittstelle übertragen werden und wird in Bit pro Sekunde (bps) angegeben. Beide Kommunikationsgeräte müssen mit der gleichen Baudrate konfiguriert werden, um eine erfolgreiche Datenübertragung zu gewährleisten. Häufig verwendete Baudraten sind 9600, 19200, 38400, 57600 und 115200 bps.

Das Datenformat in einer UART-Übertragung umfasst die Anzahl der Datenbits, die Parität und die Anzahl der Stoppbits. Die Anzahl der Datenbits kann zwischen 5 und 9 liegen, wobei 8 Datenbits am häufigsten verwendet werden. Die Parität kann gerade (even), ungerade (odd) oder keine (none) sein und dient der Fehlererkennung. Schließlich werden ein oder zwei Stoppbits verwendet, um das Ende des Datenpakets anzuzeigen. Beide Geräte müssen das gleiche Datenformat verwenden, um eine korrekte Kommunikation zu gewährleisten.

2.5.3 Einsatz von UART in der STM32 Mikrocontroller Kommunikation

In dieser Bachelorarbeit wird UART verwendet, um die Kommunikation zwischen dem STM32-Mikrocontroller und der GUI-Anwendung zu ermöglichen. Die SCPI-Befehle werden als Textzeichen über die UART-Schnittstelle gesendet und empfangen, um verschiedene Funktionen des steuerbaren Spannungsmessgerätes auszuführen.

Um die UART-Kommunikation auf dem STM32L476RG Nucleo Board einzurichten, muss die entsprechende Peripherie im STM32CubeIDE konfiguriert werden. Dies beinhaltet die Aktivierung des UART-Moduls, die Festlegung der Baudrate und des Datenformats sowie die Auswahl der verwendeten TX- und RX-Pins. Nach der Konfiguration kann die UART-Schnittstelle in der Firmware des Mikrocontrollers programmiert werden, um SCPI-Befehle zu empfangen, auszuwerten und entsprechende Aktionen auszuführen.

2.6 Qt Designer und Python für GUI-Entwicklung

2.6.1 Einführung in den Qt Designer

Qt Designer ist ein leistungsstarkes und benutzerfreundliches Werkzeug zum Entwerfen und Erstellen von grafischen Benutzeroberflächen (GUIs) für Desktop-Anwendungen. Es ermöglicht Benutzern, GUIs durch einfaches Ziehen und Ablegen verschiedener Widgets wie Schaltflächen, Beschriftungen und Schieberegler auf ein Hauptfenster oder einen Dialog zu entwickeln. Dadurch entfällt die Notwendigkeit, manuell Code zu schreiben, um die Schnittstelle zu erstellen, was den Designprozess effizienter und zugänglicher macht. [14]

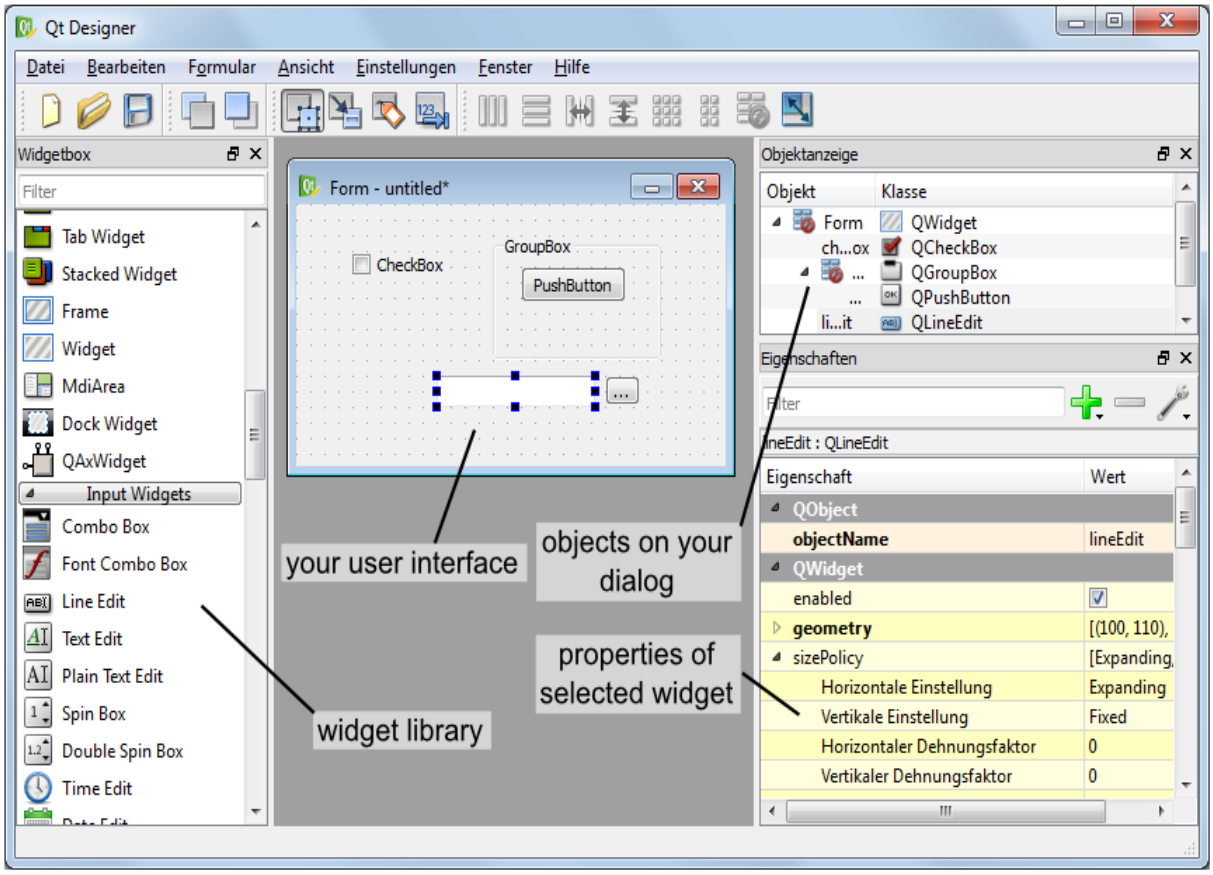

Abbildung 6: QT Designer Oberfläche [9]

2.6.2 Integration mit Visual Studio

Qt Designer lässt sich leicht in gängige integrierte Entwicklungsumgebungen (IDEs) wie Visual Studio integrieren, was eine nahtlose Entwicklungserfahrung ermöglicht. Die Integration ermöglicht es Entwicklern, sowohl am GUI-Design als auch am

zugrunde liegenden Python-Code in derselben Umgebung zu arbeiten. Visual Studio bietet erweiterte Debugging- und Codekomplettierungsfunktionen, welche die Entwicklung und Wartung des Anwendungscodes erleichtern.

```
25      def __init__(self):
26          super(ADC,self).__init__()
27          loadUi("ADC.ui" , self)
```

In diesem Code wird das GUI, das in einer .ui-Datei mit dem Qt Designer erstellt wurde, in eine Python-Anwendung mit Visual Studio geladen. Die loadUi-Methode des UIC-Moduls(User Interface Compiler) wird verwendet, um das GUI-Layout aus der Datei 'ADC.ui' zu laden und auf das aktuelle Projekt anzuwenden.

2.6.3 Funktionsweise der GUI

Die entwickelte grafische Benutzeroberfläche (GUI) bietet mehrere Funktionen, die eine intuitive Interaktion mit dem STM32-Mikrocontroller zur Steuerung und Überwachung des Spannungsmesssystems ermöglichen. Es werden Befehle an das Board gesendet, um den aktuellen Kanal auszuwählen oder auszugeben. Außerdem kann die GUI Befehle senden, um die ADC-Auflösung zu ändern oder abzufragen. Eine weitere Funktion ist die Möglichkeit, Befehle zur Änderung oder zum Auslesen der Blende, auch Oversampling-Zeit genannt, zu senden. Schließlich bietet die GUI die Möglichkeit, einzelne ADC-Konvertierungen zu starten und die Ergebnisse direkt in der Benutzeroberfläche anzuzeigen.

2.6.4 GUI-Implementierung mit Python

Nachdem das GUI-Layout mit Qt Designer entworfen wurde, wird die Funktionalität der Anwendung mit Python implementiert. Um die Integration der entworfenen Oberfläche in den Python-Code zu erleichtern, wird die Bibliothek PyQt verwendet. PyQt bietet eine Reihe von Modulen, die es Entwicklern ermöglichen, auf die Funktionalität von Qt zuzugreifen und voll funktionsfähige Anwendungen zu erstellen.

Python wurde aufgrund seiner Einfachheit, Vielseitigkeit und der großen Anzahl verfügbarer Bibliotheken ausgewählt und ist somit die ideale Wahl für die Entwicklung der GUI-Anwendung. Der Python-Code ist für die Verarbeitung der Benutzereingaben, die Verarbeitung der SCPI-Befehle und die Kommunikation mit dem STM32-Mikrocontroller über die UART-Schnittstelle verantwortlich.

Somit bietet die Kombination von Qt Designer und Python eine leistungsfähige und effiziente Lösung für die Erstellung benutzerfreundlicher GUIs, die es dem Benutzer ermöglichen, auf STM32-Mikrocontrollern implementierte Strommesssysteme zu steuern und zu überwachen. Die resultierende Anwendung ermöglicht eine effiziente Kommunikation zwischen dem Benutzer und dem Mikrocontroller und bietet eine reibungslose und intuitive Benutzererfahrung.

3. Systemanforderungen und Spezifikationen

3.1 Kanalauswahl

Das auf dem STM32-Mikrocontroller implementierte Spannungsmesssystem verfügt über insgesamt 9 wählbare Kanäle zur Spannungsmessung. Diese Kanäle werden über ADC 1 auf dem STM32L476RG Nucleo Board definiert. Zur Auswahl stehen die Kanäle Kanal 1, Kanal 2, Kanal 3, Kanal 4, Kanal 5, Kanal 11, Kanal 12, Kanal 13 und Vrefint (Interne Spannungsreferenz).Jeder Kanal entspricht einem bestimmten Eingangspin auf dem STM32L476RG Nucleo Board. Die den verfügbaren Kanälen zugeordneten Pins sind :

- Kanal 1: PC0 (Pin C0)
- Kanal 2: PC1 (Pin C1)
- Kanal 3: PC2 (Pin C2)
- Kanal 4: PC3 (Pin C3)
- Kanal 5: PA0 (Pin A0)
- Kanal 11: PA6 (Pin A6)
- Kanal 12: PA7 (Pin A7)
- Kanal 13: PC4 (Pin C4)

Zur Übersicht ist die untenstehende Pinbelegung sehr praktisch.

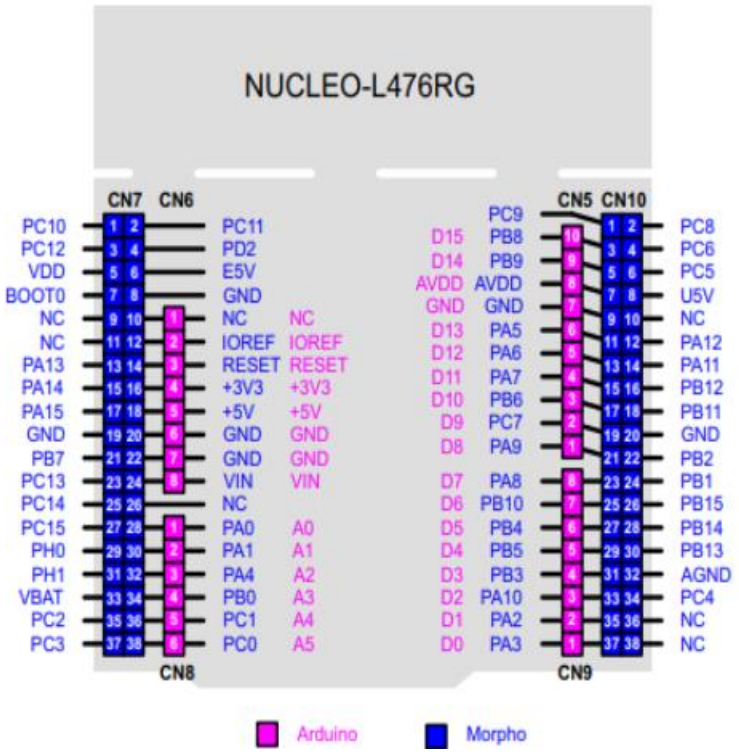

Abbildung 7: Pins-Out des Nucelo Stm32L476rg Boards [10]

Die GUI-Anwendung ermöglicht es dem Benutzer, einen der verfügbaren Kanäle für die Spannungsmessung auszuwählen. Der ausgewählte Kanal wird als Eingangsquelle für den ADC verwendet und der entsprechende Pin wird als Analogeingang konfiguriert. Dadurch wird sichergestellt, dass die ADC-Wandlung die am ausgewählten Kanal anliegende Spannung genau wiedergibt.

3.2 ADC-Auflösung

Die ADC-Auflösung bezieht sich auf die Anzahl der diskreten digitalen Werte, die der ADC bei der Umwandlung eines analogen Eingangssignals in einen digitalen Wert darstellen kann. Eine höhere Auflösung bedeutet eine höhere Genauigkeit des Wandlungsprozesses, was eine genauere Messung der Eingangsspannung ermöglicht. Das STM32L476RG Nucleo Board ist mit einem 12-Bit ADC ausgestattet, was bedeutet, dass es bis zu 2^12 (4.096) diskrete Digitalwerte darstellen kann.

Bei diesem Spannungsmesssystem kann der Benutzer zwischen mehreren Auflösungsoptionen wählen, um ein Gleichgewicht zwischen Messgenauigkeit und Wandlungsgeschwindigkeit zu erreichen. Niedrigere Auflösungen führen in der Regel zu schnelleren Wandlungszeiten, allerdings auf Kosten einer geringeren Genauigkeit. Für den ADC sind folgende Auflösungen verfügbar:

- 6-Bit Auflösung
- 8-Bit Auflösung
- 10-Bit Auflösung
- 12-Bit Auflösung

Die GUI-Anwendung ermöglicht es dem Benutzer, die gewünschte Auflösung für den ADC-Wandlungsprozess auszuwählen. Die gewählte Auflösung wirkt sich direkt auf die Genauigkeit und Geschwindigkeit der Spannungsmessung aus und ermöglicht es dem Benutzer, die Leistung des Systems an seine spezifischen Anforderungen anzupassen. Es ist wichtig zu beachten, dass die Wahl einer höheren Auflösung zu einer genaueren Messung führt, aber zusätzliche Zeit für den Wandlungsprozess erfordern kann.

3.3 Oversampling und Apertur

3.3.1 Oversampling

Oversampling ist eine Technik zur Verbesserung der Auflösung und Genauigkeit von ADC-Messungen, indem mehrere Abtastwerte genommen und gemittelt werden. Auf diese Weise kann die effektive Anzahl der Bits bei der ADC-Wandlung erhöht und das Rauschen und der Quantisierungsfehler verringert werden. Diese Technik ist besonders nützlich bei der Messung kleiner Spannungsdifferenzen oder bei der Verarbeitung verrauschter Signale.

Im Spannungsmesssystem kann der Benutzer die Oversampling-Funktion je nach Bedarf aktivieren oder deaktivieren. Wenn sie aktiviert ist, führt das System mehrere ADC-Wandlungen durch und berechnet den Mittelwert, bevor das endgültige Messergebnis zurückgegeben wird. Dies kann zu präziseren und genaueren Spannungsmessungen beitragen.

3.3.2 Apertur

Im Zusammenhang mit ADC-Messungen bezieht sich Apertur auf die Dauer des Abtastvorgangs oder die Zeit, in der der ADC eine einzelne Wandlung durchführt. Eine längere Aperturzeit kann zu einer genaueren Messung führen, da der ADC das Eingangssignal besser erfassen kann. Allerdings erhöht eine längere Aperturzeit auch die Gesamtwandlungszeit, was bei zeitkritischen Anwendungen ein kritischer Faktor sein kann.

Die GUI-Anwendung ermöglicht dem Benutzer die Auswahl der Aperturzeit für den ADC-Wandlungsprozess. Durch die Einstellung der Aperturzeit kann der Benutzer ein optimales Gleichgewicht zwischen Messgenauigkeit und Wandlungsgeschwindigkeit finden. Dabei ist zu berücksichtigen, dass längere Aperturzeiten zu genaueren Messungen führen können, aber zusätzliche Zeit für den Wandlungsprozess erfordern.

Allgemein bieten die Aperturfunktionen des Spannungsmesssystems dem Benutzer die Flexibilität, die Genauigkeit und Geschwindigkeit von ADC-Messungen entsprechend seinen spezifischen Anforderungen zu optimieren. Durch die Anpassung dieser Parameter kann der Benutzer das für seine Anwendung erforderliche Genauigkeits- und Leistungsniveau erreichen.

3.4 Definierte SCPI Befehle

Die Kommunikation zwischen der GUI-Anwendung und dem STM32-Mikrocontroller erfolgt über die UART-Schnittstelle. Um eine effiziente und standardisierte Kommunikation zwischen den beiden Komponenten zu gewährleisten, werden SCPI-Befehle verwendet.

SCPI-Befehle sind hierarchisch aufgebaut und gruppiert. Sie folgen einer bestimmten Syntax. Dabei wird zwischen Groß- und Kleinschreibung unterschieden. In den Befehlen stellen die Großbuchstaben die obligatorischen Zeichen dar, während die Klein-

buchstaben die optionalen Zeichen darstellen. Das bedeutet, dass Großbuchstaben für die Identifizierung des SCPI-Befehls ausreichen, während Kleinbuchstaben zur zusätzlichen Klarheit beitragen können, aber für die erfolgreiche Ausführung nicht unbedingt erforderlich sind.

Für diese Applikation sind folgende SCPI-Befehle implementiert:

Kanalsteuerung :

- SENSe:CHANnel? - Liefert die Nummer des aktuellen Kanals
- SENSe:CHANnel <Channel No> - Wählt den Kanal mit der angegebenen Nummer als aktuellen Kanal aus

Auflösung Steuerung :

- SENSe:RESOlution? - Abfrage der Auflösung des aktuellen Kanals
- SENSe:RESOlution? <MIN|MAX> - Abfrage der Auflösungsgrenzen des aktuellen Kanals
- SENSe:RESOlution <Wert|MIN|MAX> - Setzt die Auflösung für den aktuellen Kanal auf einen benutzerdefinierten Wert oder auf einen der Grenzwerte

Apertur Steuerung :

- SENSe:APERture? - Abfrage der Apertur für den aktuellen Kanal
- SENSe:APERture? <MIN|MAX> - Rückgabe der Grenzwerte für die Apertur ("Oversampling"-Zeit), min oder max, für den aktuellen Kanal
- SENSe:APERture <Value|MIN|MAX> - Setzt die Apertur (Oversampling-Dauer) für den aktuellen Kanal auf einen benutzerdefinierten Wert oder einen der Grenzwerte

Spannungsmessung:

- MEASure:VOLTage? - Führt eine Messung durch und gibt die gemessene Spannung des aktuellen Kanals zurück.

Die SCPI-Befehle werden von der GUI-Anwendung generiert, wenn der Benutzer bestimmte Aktionen ausführt, wie z. B. die Auswahl eines Kanals oder die Änderung der Auflösung. Die Befehle werden dann über die UART-Schnittstelle an den STM32-Mikrocontroller gesendet. Der Mikrocontroller führt die entsprechenden Aktionen aus und sendet gegebenenfalls eine Antwort an die GUI-Anwendung zurück.

Um die Kommunikation zwischen der GUI und dem Board herzustellen, muss das Python-Paket serial importiert werden, um die UART-Kommunikation zu verwalten.

Durch die Verwendung von SCPI-Befehlen kann eine klare und konsistente Kommunikation zwischen der GUI-Anwendung und dem STM32-Mikrocontroller gewährleistet werden. Dies ermöglicht eine effektive Steuerung und Überwachung des Spannungsmesssystems und bietet dem Benutzer eine intuitive Bedienung der Anwendung.

4.Programmierung des Mikrocontrollers STM32

4.1 STM32CubeIDE

STM32CubeIDE ist eine umfassende Entwicklungsumgebung, die von STMicroelectronics für STM32-Mikrocontroller zur Verfügung gestellt wird. Diese integrierte Entwicklungsumgebung (IDE) enthält verschiedene Werkzeuge wie einen Code-Editor, einen Compiler, einen Debugger und ein Hardware-Konfigurationswerkzeug, die den Entwicklungsprozess für eingebettete Systeme auf STM32-Boards optimieren. In diesem Projekt wird STM32CubeIDE zur Programmierung des STM32L476RG Nucleo Boards verwendet.

4.1.1 Installation und Einrichtung

Um STM32CubeIDE zu installieren, wird das Installationsprogramm von der STMicroelectronics Webseite heruntergeladen. Nach der Überprüfung der Systemvoraussetzungen wird der Installationsprozess durchgeführt, wobei die Bildschirmanweisungen zur Vervollständigung des Setups verwendet werden. Nach der Installation wird die IDE gestartet und das Workspace-Verzeichnis konfiguriert, in dem alle Projektdateien gespeichert werden.

4.1.2 Projekterstellung und Konfiguration

Um ein neues Projekt in der STM32CubeIDE zu erstellen, klickt man auf "Datei" > "Neu" > "STM32 Projekt". Daraufhin erscheint das STM32 Target Selector Fenster, in dem das STM32L476RG Nucleo Board als Target ausgewählt wird. Sobald das Projekt erstellt ist, wird die .ioc-Datei im Projektordner geöffnet, um die Mikrocontroller-Einstellungen mit dem in der IDE integrierten grafischen Tool STM32CubeMX zu konfigurieren. Innerhalb von STM32CubeMX werden die für das Projekt benötigten Taktquellen, GPIO-Pins und Peripheriegeräte konfiguriert. Die ADC- und UART-Module werden aktiviert und ihre jeweiligen Parameter und Pinbelegungen so eingestellt, wie sie für die Spannungsmessung und die Kommunikation mit der GUI benötigt werden.

4.1.3 Code-Entwicklung

Der Code für den STM32-Mikrocontroller wird in der Programmiersprache C entwickelt. Die von STMicroelectronics bereitgestellten STM32Cube HAL-Bibliotheken (Hardware Abstraction Layer) ermöglichen das Schreiben von Code, der mit den Peripheriegeräten des Mikrocontrollers zusammenarbeitet, ohne sich mit Low-Level-Details befassen zu müssen.

Das ADC-Modul wird implementiert, um Spannungsmessungen durchzuführen, und das UART-Modul wird verwendet, um die Kommunikation mit der grafischen Benutzeroberfläche zu ermöglichen. Die SCPI-Befehle sind integriert, um das Spannungsmessgerät zu steuern, indem die über den UART empfangenen Befehle analysiert und die entsprechenden Aktionen ausgeführt werden, wie z. B. die Einstellung der ADC-Auflösung oder die Auswahl eines Kanals.

4.1.4 Debugging und Testen

Während des gesamten Entwicklungsprozesses werden die in der STM32CubeIDE verfügbaren Debugging-Funktionen verwendet, um Probleme oder Fehler im Code zu identifizieren und zu beheben. Die IDE erlaubt das Setzen von Breakpoints, das Durchlaufen des Codes und die Untersuchung von Variablen und Speicher zur Laufzeit, was eine effektive Diagnose und Behebung von Problemen ermöglicht.

Zum Testen der Implementierung wird der kompilierte Code auf das STM32L476RG Nucleo Board geladen und die Funktion überwacht, um den korrekten Betrieb des Spannungsmesssystems zu verifizieren.

4.2 ADC-Konfiguration

Der Analog-Digital-Wandler (ADC) ist eine entscheidende Komponente des Spannungsmesssystems, da er für die Umwandlung der analogen Spannungssignale der Eingangskanäle in digitale Werte verantwortlich ist, die von der GUI verarbeitet und

angezeigt werden können. Das STM32L476RG Nucleo Board verfügt über einen 12-Bit A/D-Wandler, der hochauflösende Messungen ermöglicht. In diesem Abschnitt werden die Schritte zur Konfiguration des ADCs für die spezifischen Anforderungen des Projekts beschrieben.

4.2.1 ADC Initialisierung

Der erste Schritt zur Konfiguration des ADCs ist die Initialisierung innerhalb des STM32CubeIDE Projektes. Im grafischen Tool STM32CubeMX wird das ADC-Peripheriegerät aktiviert, indem es aus der Liste der verfügbaren Peripheriegeräte ausgewählt wird.

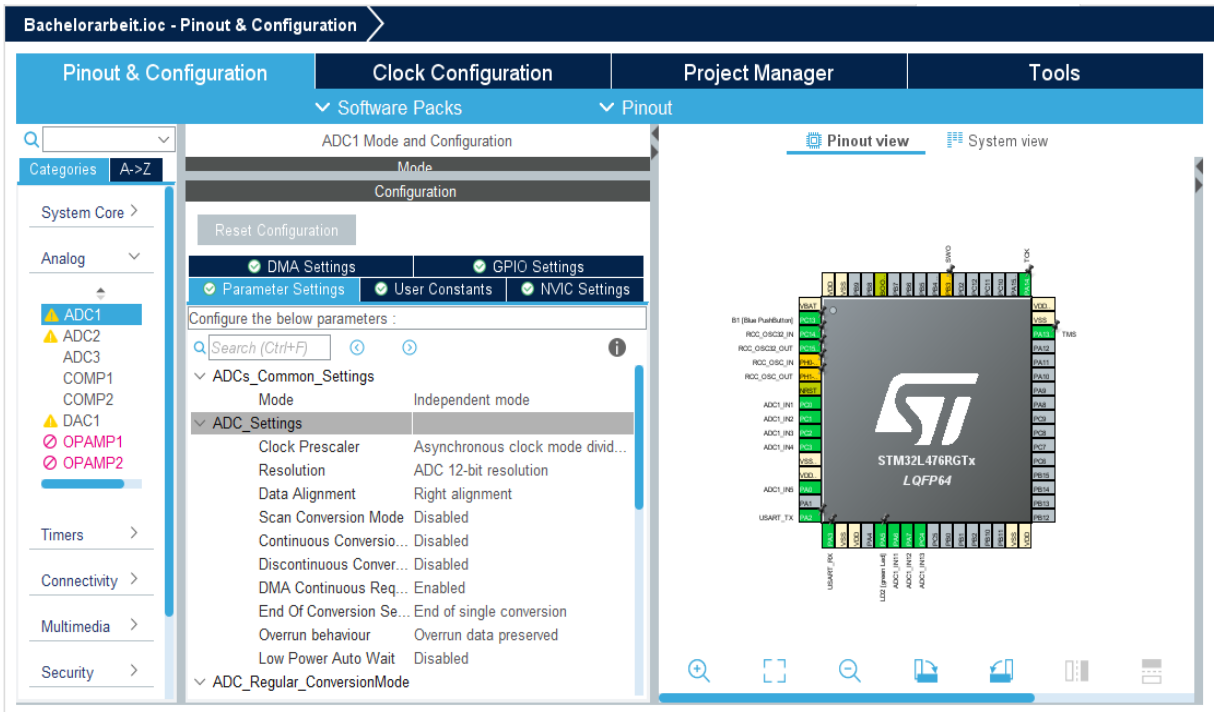

Abbildung 8: ADC-Konfiguration im STM32CubeMX

Anschließend werden die ADC-Einstellungen angepasst, wie z.B. die Auswahl der gewünschten Taktquelle und die Konfiguration der ADC-Auflösung (6-Bit, 8-Bit, 10-Bit oder 12-Bit). Der ADC-Betriebsmodus wird auf Einzelwandlung eingestellt, so dass bei Bedarf mehrere Wandlungen möglich sind.

4.2.2 Konfiguration der Eingangskanäle

Um die Spannung an den Eingangskanälen zu messen, müssen die entsprechenden ADC-Kanäle konfiguriert werden. Jeder Kanal entspricht einem bestimmten GPIO-Pin, der mit der Eingangsquelle verbunden ist. Im STM32CubeMX-Tool werden die gewünschten GPIO-Pins auf den Modus "ADC" gesetzt und die entsprechenden ADC-Kanäle ausgewählt. In diesem Projekt werden neun Kanäle verwendet, daher werden neun GPIO-Pins als ADC-Eingänge konfiguriert.

4.2.3 ADC-Abtastzeit

Die ADC-Abtastzeit ist ein wichtiger Parameter, der die Dauer der Erfassung des analogen Spannungssignals bestimmt. Eine längere Abtastzeit kann zu genaueren Messungen führen, aber auch die Wandlungszeit erhöhen. Im STM32CubeMX-Tool kann die Abtastzeit für jeden ADC-Kanal individuell eingestellt werden. Für dieses Projekt wurde eine Abtastzeit von 24,5 Taktzyklen gewählt, um einen Kompromiss zwischen Messgenauigkeit und Wandlungsgeschwindigkeit zu erreichen.

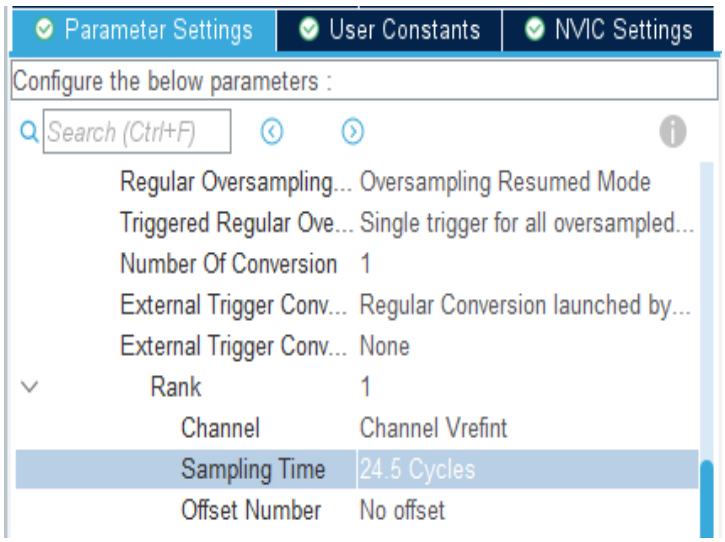

Abbildung 9: ADC-Abtastzeit-Konfiguration

4.2.4 DMA-Konfiguration

Um die ADC-Wandlungsergebnisse effizient auszulesen, ohne die CPU zu belasten, kann der Direct Memory Access (DMA) Controller eingesetzt werden. Der DMA-Controller ermöglicht den automatischen Datentransfer zwischen ADC und Speicher, wodurch die CPU für andere Aufgaben entlastet wird. Im STM32CubeMX-Tool wird der DMA-Controller aktiviert und mit der ADC-Peripherie verbunden. Der DMA-Modus wird auf zirkulär gesetzt, wodurch ein kontinuierlicher Datentransfer zwischen ADC und Speicher gewährleistet wird.

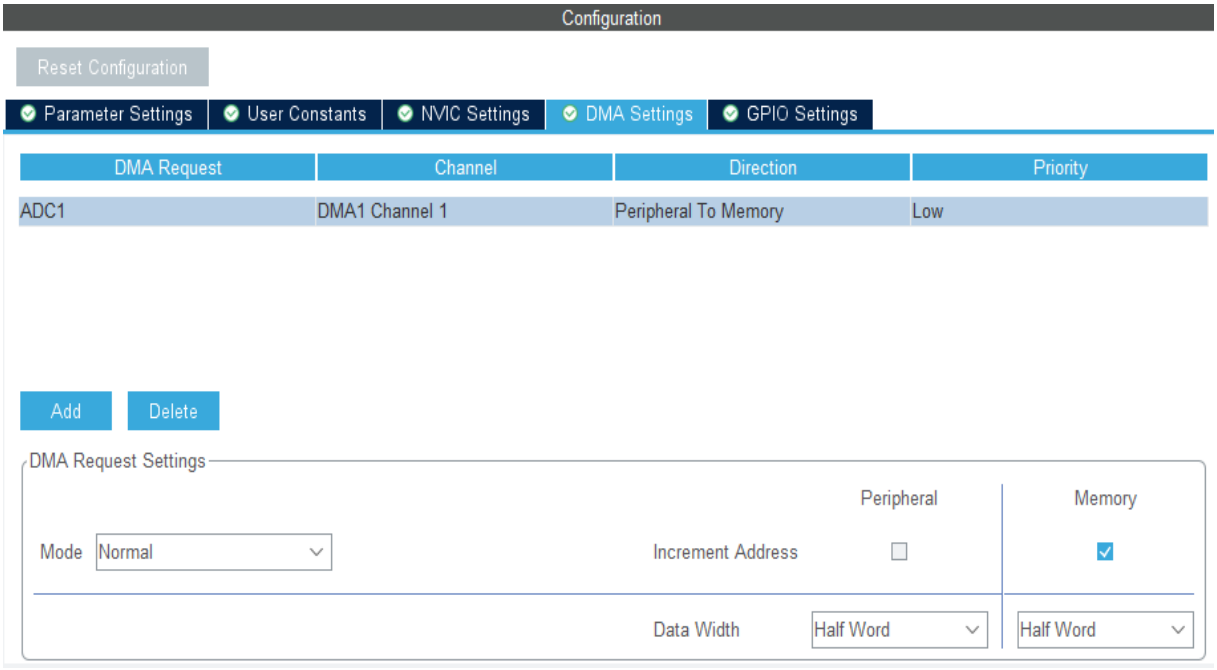

Abbildung 10: Konfiguration des DMA für den ADC

Nach Abschluss der ADC-Konfiguration werden die Einstellungen gespeichert und der notwendige Initialisierungscode vom STM32CubeMX generiert. Der ADC ist nun bereit für die Spannungsmessungen im Projekt.

4.3 Gesamtprogramm und Funktionsübersicht

4.3.1 Einführung in die Programmierung

Dieser Abschnitt befasst sich mit der Programmierung des STM32 Mikrocontrollers, die das Herzstück dieses Projekts darstellt. Die Programmierung ist der Prozess, bei dem der Mikrocontroller angewiesen wird, eine bestimmte Aufgabe auszuführen. Es ist im Wesentlichen das Gehirn des Systems, das sein Verhalten unter verschiedenen Bedingungen bestimmt.

Zuerst werden notwendige Header-Dateien integriert.

```
20 #include "main.h"
21 #include "stm32l4xx_hal.h"
22 #include "stm32l4xx_hal_adc.h"
23 #include "string.h"
24 #include <stdio.h>
25 #include <stdlib.h>
26 #include <stdbool.h>
27 #include <ctype.h>
```

Diese Header-Dateien enthalten die für dieses Projekt benötigten Funktionen und Makros, einschließlich spezieller Dateien für den STM32L4xx Mikrocontroller, die Hardware Abstraction Layer (HAL) und den ADC. Darüber hinaus enthält es zusätzliche Header-Dateien für die Manipulation von Strings, die Manipulation von Standardein- und -ausgabedaten und Standard-Bibliotheksfunktionen. [13]+[15]

Im Folgenden werden einige globale Variablen und Puffer definiert, auf die im gesamten Programm zugegriffen werden kann.

```
33 #define RxBuf_SIZE   64                  //!< Größe des Rx-Puffers, der zum Empfang von Daten über UART verwendet wird.
34 #define MainBuf_SIZE 64                  //!< Größe des Hauptpuffers, der zum Speichern der über UART empfangenen Daten verwendet wird.
35 #define commandBuf_SIZE 64
36
37 uint8_t RxBuf[RxBuf_SIZE];               //!< Rx-Puffer für den Empfang von Daten über UART.
38 uint8_t MainBuf[MainBuf_SIZE];           //!< Hauptpuffer, der zum Speichern der über UART empfangenen Daten verwendet wird.
39 uint16_t adc_value;
40 uint8_t maxRes=12;
41 uint8_t minRes=6;
42
43 int maxAp=24.5*256;  //Vordefinierte Maximalapertur, hier bezogen auf eine Samplingzeit von 24,5 Zyklen und maximale Ratio
44 int minAp=24.5*2;   //Vordefinierte Minimalperture
45 int currentAp;
46 int division=4095;   //Initialisierung der Division für einen 12-Bit-ADC, die zur Umrechnung des ADC-Wertes in Volt verwendet wird.
```

Dazu gehören unter Anderem Puffer für den Empfang und die Speicherung von Daten über UART, ADC-Wert-Variablen, maximale und minimale Auflösung und Aperturwerte. [12]

Dann müssen Instanzen von Strukturen wie ADC_ChannelConfTypeDef erzeugt werden, um ADC-Kanalparameter und Handler-Strukturen für ADC-, DMA- und UART-Peripheriegeräte zu konfigurieren.

```
48 ADC_ChannelConfTypeDef sConfig = {0};
49 //Instanz der Struktur ADC_ChannelConfTypeDef für die Konfiguration von ADC-Kanalparametern erstellen
50
51 ADC_HandleTypeDef hadc1;
52 DMA_HandleTypeDef hdma_adc1;
53 UART_HandleTypeDef huart2;
54 DMA_HandleTypeDef hdma_usart2_rx;
```

Schließlich werden die Prototypen der Funktionen, die im Code verwendet werden, deklariert.

```
66 void SystemClock_Config(void);
67 static void MX_GPIO_Init(void);
68 static void MX_DMA_Init(void);
69 static void MX_USART2_UART_Init(void);
70 static void MX_ADC1_Init(void);
```

Diese Funktionen werden später im Programm definiert und umfassen Aufgaben wie die Konfiguration des Systemtaktes, die Initialisierung der GPIO-, DMA-, UART- und ADC-Peripherie.

Auf diese Weise werden die Grundlagen für die eigentliche Programmierung des Microcontrollers gelegt. In den folgenden Abschnitten wird jeder Teil der Programmierung genauer betrachtet, wobei erklärt wird, welche Aufgabe jede Funktion erfüllt und welche Bedeutung sie besitzt. Dadurch wird ein umfassendes Verständnis der gesamten Programmstruktur einschließlich der Interaktionen zwischen den verschiedenen Funktionen und dem Hauptprogramm ermöglicht.

4.3.2 UART Callback Funktion

Die Funktion HAL_UARTEx_RxEventCallback ist eine Callback-Funktion, die aufgerufen wird, wenn das im DMA-Modus arbeitende UART-Peripheriegerät Daten empfangen hat. Der Hauptzweck dieser Funktion besteht darin, die empfangenen Daten aus dem DMA-Puffer (RxBuf) in einen Hauptanwendungspuffer (MainBuf) zu übertragen und den Empfang und die Verarbeitung von Befehlen zu steuern.[12]

```
71 void HAL_UARTEx_RxEventCallback(UART_HandleTypeDef *huart, uint16_t Size)
72 {
73     uint8_t Counter = 0 ;   // Hilfsvariable zum Umkopieren von Daten aus dem RX-Puffer in den zentralen Puffer
74
75     if(huart -> Instance == USART2 )   // Überprüfung, ob der Interrupt von USART2 stammt, da wir aktuell USART 2 einsetzen
76     {
77
78         while(Counter <= Size)  // Durchlauf zur Übertragung der Daten vom Rx-Puffer in den zentralen Puffer
79         {
80             //Prüfen, ob der Hauptpuffer voll mit Daten ist. In diesem Fall werden die Daten von Anfang an ersetzt.
81             if(MainBufCounter > 64 )
82             {
83                 //Zurücksetzen des Hauptpufferzählers
84                 MainBufCounter = 0;
85             }
86             MainBuf[MainBufCounter] = RxBuf[Counter++] ; // Übertragung der Daten vom Rx-Puffer in den zentralen Puffer
87             if(MainBuf[MainBufCounter] == '\n')     // Überprüfung auf das Zeichen \n, das signalisiert, dass der vollständige Befehl eingegangen ist
88             {for(int i = 0 ; i < (MainBufCounter); i++)  // Einfügen des empfangenen Befehls in den Befehlspuffer
89                 {
90                     commandBuf[i] =MainBuf[i];
91                 }
92                 newCommandReceived = true;  //Statusflag, das signalisiert, dass ein neuer Befehl eingegangen ist
93             }
94
95             if(Counter - 1 != Size)    // Sicherstellen, dass nach jeder Datenkopie kein Nullzeichen hinzugefügt wird
96             {
97                 //Inkrementieren des Hauptpufferzählers
98                 MainBufCounter++;
99             }
100         }
101         //Diese Funktion wird verwendet, um Daten von UART mit DMA zu empfangen, bis der Datenpuffer voll ist oder die IDLE Line erkannt wird.
102         HAL_UARTEx_ReceiveToIdle_DMA(&huart2, RxBuf, RxBuf_SIZE);
103         //Deaktivieren Sie den Interrupt für die halb übertragenen Daten.
104         __HAL_DMA_DISABLE_IT(&hdma_usart2_rx, DMA_IT_HT);
105     }
106 }
```

Beim Empfang von Daten prüft die Callback-Funktion, ob der Interrupt von USART2, dem verwendeten UART-Peripheriegerät, stammt. Anschließend kopiert sie die empfangenen Daten aus dem DMA-Puffer (RxBuf) in den Hauptanwendungspuffer (MainBuf). Eine Hilfsvariable Counter wird verwendet, um durch die Daten zu navigieren.

MainBuf wird zurückgesetzt (d.h. MainBufCounter wird auf 0 gesetzt), wenn er voll ist, d.h. wenn MainBufCounter 64 überschreitet. Dieser Mechanismus verhindert einen Pufferüberlauf, der zu unvorhersehbarem Verhalten oder Datenbeschädigung führen könnte. Er bietet auch eine zirkuläre Pufferfunktionalität, bei der neu ankommende Daten alte Daten überschreiben, sobald der Puffer voll ist. Dies ist eine

übliche Strategie in Echtzeitsystemen, in denen die neuesten Daten von größter Bedeutung sind und der Verlust älterer Daten, die nicht rechtzeitig verarbeitet wurden, akzeptabel ist.

Während die empfangenen Daten in den MainBuf übertragen werden, prüft die Funktion ständig den Erhalt des Newline-Zeichens ('\n'). Die Erkennung des Newline-Zeichens dient in diesem Zusammenhang als Befehlstrennzeichen und zeigt an, dass ein vollständiger Befehl über den UART empfangen wurde. Dieses vollständig empfangene Kommando kann nun verarbeitet werden. Textbasierte Kommunikationsprotokolle wie SCPI verwenden in der Regel ein spezielles Zeichen, um das Ende eines Befehls zu markieren, und Newline ist eine übliche Wahl, da es auch das Standard-Zeilenendezeichen in vielen Textverarbeitungskontexten ist, was bedeutet, dass viele Tools und Bibliotheken es standardmäßig korrekt verarbeiten.

Wenn ein Zeilenumbruch erkannt wird, wird der empfangene Befehl zur Verarbeitung in den commandBuf kopiert. Gleichzeitig wird das Flag newCommandReceived auf true gesetzt, um anzuzeigen, dass ein neuer Befehl zur Verarbeitung bereit ist.

Die Funktion bereitet dann die UART-Peripherie für den nächsten Empfang vor, indem sie HAL_UARTEx_ReceiveToIdle_DMA() aufruft. Diese Funktion stellt den UART so ein, dass er Daten über DMA empfängt, bis der Datenpuffer voll ist oder eine leere Leitung erkannt wird.

Abschließend wird der Half-Transfer-Interrupt deaktiviert. Der Half-Transfer-Interrupt ist eine Funktion des DMA-Controllers (Direct Memory Access) und kann einen Interrupt auslösen, wenn die Hälfte der angeforderten Daten übertragen wurde. Dies kann zu einer vorzeitigen Abarbeitung von Befehlen führen. Durch die Deaktivierung des Half-Transfer-Interrupts und die ausschließliche Verwendung der Interrupts "Transfer Complete" und "Idle Line" wird sichergestellt, dass nur vollständige, korrekt abgegrenzte Befehle verarbeitet werden. Diese Strategie verhindert Szenarien, in denen ein Befehl verarbeitet wird, bevor er vollständig empfangen wurde, und vermeidet dadurch potenzielle Fehler oder Fehlverhalten.

4.3.3 UART-Übertragungsfunktion

Die Uprintf-Funktion ist eine benutzerdefinierte Funktion zur Übertragung von String-Daten über UART (Universal Asynchronous Receiver Transmitter). Diese Funktion ist wichtig, um die Kommunikation zwischen Mikrocontroller und Benutzer über Texte zu ermöglichen. [12]

```
114 void Uprintf(char *str)
115 {
116     //Funktion zur Übertragung von Daten auf UART.
117     HAL_UART_Transmit(&huart2 ,(uint8_t*) str, strlen(str),HAL_MAX_DELAY);
118 }
```

Die Uprintf-Funktion akzeptiert einen String vom Benutzer als Eingabeparameter. Sie verwendet die Funktion HAL_UART_Transmit aus der STM32-HAL-Bibliothek, um die bereitgestellten String-Daten über den UART zu übertragen. Die Parameter für HAL_UART_Transmit umfassen das UART-Handle (huart2), die zu übertragenden String-Daten, die Länge des Strings und die Timeout-Dauer (eingestellt auf HAL_MAX_DELAY).

Der Übertragungsprozess ist synchron, d.h. die Funktion blockiert, indem die Ausführung angehalten wird, bis die Übertragung abgeschlossen ist. Dies ermöglicht eine zuverlässige Datenübertragung, da die Funktion auf den Abschluss des Prozesses wartet, bevor sie fortfahren kann, um sicherzustellen, dass alle Daten gesendet wurden.

Im Zusammenhang mit dieser Anwendung ist die Uprintf-Funktion wichtig, um Befehle oder Antworten an den Benutzer zurückzugeben. Sie kann zum Beispiel verwendet werden, um den Status eines Prozesses zu senden, die erfolgreiche Ausführung eines Befehls zu bestätigen oder angeforderte Daten zurückzugeben. Dies macht sie zu einem wesentlichen Teil der Kommunikationsschnittstelle zwischen Benutzer und System.

4.3.4 Funktion zur Umwandlung von Großbuchstaben in Kleinbuchstaben

Die Funktion "lowercase" wird verwendet, um alle Großbuchstaben in einem String in Kleinbuchstaben umzuwandeln. Sie ist besonders nützlich in Situationen, in denen Operationen, die Groß- und Kleinschreibung berücksichtigen, unerwünscht sind, z. B. beim Vergleich von Zeichenketten, bei der Verarbeitung von Benutzereingaben und insbesondere bei der Verarbeitung von SCPI-Befehlen (Standard Commands for Programmable Instruments), die über UART (Universal Asynchronous Receiver/Transmitter) empfangen werden.

Dies bietet im Umgang mit SCPI-Befehlen erhebliche Vorteile. SCPI-Befehle unterscheiden nicht zwischen Groß- und Kleinschreibung und können daher in jeder beliebigen Kombination von Groß- und Kleinbuchstaben empfangen werden. Durch die Umwandlung aller Zeichen in Kleinbuchstaben wird die Befehlsverarbeitung rationalisiert, wodurch potenzielle Probleme beim Vergleich oder der Suche nach Groß- und Kleinbuchstaben vermieden werden.

Außerdem werden Fehler aufgrund von Tippfehlern vermieden. Beispielsweise könnte ein Benutzer versehentlich 'MEaSure' statt 'MEASure' eingeben. Mit der Kleinschreibfunktion werden beide Eingaben als "measure" behandelt, wodurch das Problem beseitigt wird.

```
120 char* lowercase(char* s)   // Diese Funktion wandelt alle Zeichen in einem String in Kleinbuchstaben um
121 {
122   for(char *p=s; *p; p++) *p=tolower(*p);   // Durchläuft jeden Charakter im String und wandelt ihn in Kleinbuchstaben um
123   return s;
124 }
```

Die Kleinschreibfunktion akzeptiert einen einzelnen String-Parameter. Sie durchläuft jedes Zeichen in der Zeichenkette und wandelt es, wenn es ein Großbuchstabe ist, in das entsprechende Kleinbuchstabenäquivalent um.

Auf diese Weise stellt die Funktion sicher, dass alle über UART empfangenen SCPI-Befehle einheitlich als Kleinbuchstaben behandelt werden, was die Zuverlässigkeit des Befehlsverarbeitungssystems erhöht.

4.3.5 Funktion zur Umwandlung von ADC-Kanalwerten in Strings

Die Funktion get_adc_channel_string dient der Konvertierung von Analog-Digital-Wandler (ADC)-Kanalwerten, die üblicherweise im unsigned integer (Uint) Format vorliegen, in das Stringformat. Dies ist besonders nützlich, wenn diese Werte über UART (Universal Asynchronous Receiver/Transmitter) an die grafische Benutzeroberfläche (GUI) übertragen werden.

In diesem System wird der ADC-Kanal vom Benutzer ausgewählt oder vom System festgelegt, und es ist von entscheidender Bedeutung, dass diese Auswahl zur Anzeige und für weitere Interaktionen an die grafische Benutzeroberfläche übertragen wird. Das UART-Kommunikationsprotokoll ist jedoch eher mit String-Daten kompatibel, und die direkte Übertragung von Uint-Werten kann zu Komplikationen oder Fehlinterpretationen führen.

```
126  const char* get_adc_channel_string(uint32_t channel) // Diese Funktion gibt einen String zurück, der dem übergebenen ADC-Kanal entspricht
127  {
128      switch(channel) {    // Switch-Case-Struktur zur Auswahl des korrekten Kanalstrings basierend auf dem eingegebenen channel-Wert
129          case ADC_CHANNEL_1:
130              return "CHANNEL_1";
131          case ADC_CHANNEL_2:
132              return "CHANNEL_2";
133          case ADC_CHANNEL_3:
134              return "CHANNEL_3";
135          case ADC_CHANNEL_4:
136              return "CHANNEL_4";
137          case ADC_CHANNEL_5:
138              return "CHANNEL_5";
139          case ADC_CHANNEL_11:
140              return "CHANNEL_11";
141          case ADC_CHANNEL_12:
142              return "CHANNEL_12";
143          case ADC_CHANNEL_13:
144              return "CHANNEL_13";
145          case ADC_CHANNEL_VREFINT:
146              return "CHANNEL_VREFINT";
147          default:
148              return "Channel not selected";
149      }
150  }
```

Die Funktion get_adc_channel_string nimmt einen Uint ADC Kanalwert als Eingabe und gibt eine entsprechende String-Repräsentation desselben Wertes zurück. Diese Konvertierung ermöglicht eine einfachere und zuverlässigere Übertragung von ADC-Kanalwerten über UART an die GUI.

Während des Konvertierungsprozesses ordnet die Funktion den ADC-Kanalwert Uint dem entsprechenden Stringwert zu, um eine genaue Darstellung zu gewährleisten. Diese Zuordnung kann anhand einer vordefinierten Konvertierungstabelle erfolgen, je nach Bereich und Variabilität der ADC-Kanalwerte.

Diese Funktion ermöglicht es dem System, ADC-Kanalwerte unabhängig von ihrem ursprünglichen Uint-Format effektiv an die grafische Benutzeroberfläche zu übertragen. Dies erhöht die Vielseitigkeit des Systems, ermöglicht eine benutzerfreundlichere Interaktion und verringert das Risiko von Fehlinterpretationen oder Datenverlusten während der Übertragung.

4.3.6 Funktion zur Extraktion von Ganzzahlen aus Befehlen

Die Funktion extract_integer_from_command extrahiert eine Ganzzahl aus dem an den Mikrocontroller gesendeten Befehl.

```
184 uint16_t extract_integer_from_command(const char* commandBuf, const char* command) {
185     // Extrahieren der ganzzahligen Teilzeichenkette aus dem Befehl
186     char integer_str[commandBuf_SIZE];
187     strlcpy(integer_str, lowercase((char *)commandBuf) + strlen(command), commandBuf_SIZE);
188     // Umwandlung der String in eine Ganzzahl
189     uint16_t value = atoi(integer_str);
190     return value;
191 }
```

Die Funktion hat zwei Parameter. Der Parameter commandBuf dem zu verarbeitenden Befehl und der Parameter command stellt den spezifischen Befehl dar, nach dem gesucht werden soll. Zuerst wird der Teilstring mit den Ganzzahlen aus dem Befehl extrahiert und in die Variable integer_str kopiert. Dazu wird eine Kopie von commandBuf erstellt, beginnend an einem Punkt, welcher der Länge des Befehls entspricht. Das Ergebnis ist ein String, der die Ganzzahlen enthält. Dann wird die Funktion atoi verwendet, um den String in eine Ganzzahl umzuwandeln. Schließlich gibt die Funktion diesen Integer-Wert zurück. Diese Funktion ist nützlich, um Parameter zu lesen, die in Mikrocontroller-Befehlen enthalten sind. Sie bietet eine effiziente Möglichkeit, ganzzahlige Werte wie Kanalnummer, Genauigkeit und Apertur aus den Befehlen zu extrahieren.

4.3.7 Funktion zum Setzen von ADC-Kanälen auf der Basis von String-Eingaben

Die Funktion convertStringToADCChannel spielt eine Schlüsselrolle bei der Steuerung der Konfiguration des ADC innerhalb des Systems. Ihr Hauptzweck besteht darin, eine String-Eingabe, die einen bestimmten ADC-Kanal identifiziert, zu verwenden, um den aktiven Kanal des ADC innerhalb des Programms einzustellen. Diese Funktionalität ist von entscheidender Bedeutung, wenn ein Benutzer die Auswahl eines bestimmten ADC-Kanals befiehlt und dieser Befehl als String über die UART-Schnittstelle empfangen wird.

```
152 void convertStringToADCChannel( char* str)   // Diese Funktion konvertiert einen String zu einem ADC-Kanal
153 {
154
155       if (strcmp(str, "ADC_CHANNEL_1") == 0) {  // Verwendet eine if-else-if-Struktur, um den String
156                                                  //zu überprüfen und den entsprechenden ADC-Kanal zuzuweisen
157           sConfig.Channel  = ADC_CHANNEL_1;
158       } else if (strcmp(str, "ADC_CHANNEL_2") == 0) {
159           sConfig.Channel  = ADC_CHANNEL_2;
160       } else if (strcmp(str, "ADC_CHANNEL_3") == 0) {
161           sConfig.Channel  = ADC_CHANNEL_3;
162       } else if (strcmp(str, "ADC_CHANNEL_4") == 0) {
163           sConfig.Channel  = ADC_CHANNEL_4;
164       } else if (strcmp(str, "ADC_CHANNEL_5") == 0) {
165           sConfig.Channel  = ADC_CHANNEL_5;
166       } else if (strcmp(str, "ADC_CHANNEL_11") == 0) {
167           sConfig.Channel  = ADC_CHANNEL_11;
168       } else if (strcmp(str, "ADC_CHANNEL_12") == 0) {
169           sConfig.Channel  = ADC_CHANNEL_12;
170       } else if (strcmp(str, "ADC_CHANNEL_13") == 0) {
171           sConfig.Channel  = ADC_CHANNEL_13;
172       } else if (strcmp(str, "ADC_CHANNEL_VREFINT") == 0) {
173           sConfig.Channel  = ADC_CHANNEL_VREFINT;
174       }
175
176       if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
177           {
178             Error_Handler();
179           }
180      HAL_Delay(10);
181   }
182
```

Wenn ein Benutzer einen Befehl zur Auswahl eines bestimmten ADC-Kanals sendet, wird dieser Befehl als String über die UART-Schnittstelle übertragen. Die Funktion convertStringToADCChannel empfängt diesen String und verarbeitet ihn, um den entsprechenden ADC-Kanal in der Software des eingebetteten Systems einzustellen.

Intern analysiert die Funktion die Stringeingabe, um die Kanalnummer zu extrahieren. Diese Kanalnummer wird dann dem entsprechenden ADC-Kanal im System zugeordnet. Es ist wichtig zu beachten, dass ADC-Kanäle im Hardware Abstraction Layer (HAL) in einem bestimmten Format dargestellt werden: ADC_CHANNEL_X, wobei X der Kanalnummer entspricht. Daher konstruiert die Funktion auch den entsprechenden String auf der Grundlage der geparsten Kanalnummer, um diesem Format zu entsprechen.

Sobald der entsprechende ADC-Kanalstring ermittelt wurde, verwendet die Funktion diese Information, um den Parameter sConfig.Channel im HAL zu aktualisieren. Dieser Parameter steuert den aktiven ADC-Kanal im System. Durch Setzen von sConfig.Channel auf den ADC-Kanal, der durch den Eingabestring repräsentiert wird, stellt die Funktion sicher, dass der ADC-Kanal im System mit der Auswahl des Benutzers übereinstimmt.

Diese Funktion ist für das System sehr nützlich, da sie eine wirksame Brücke zwischen den als String-Eingaben empfangenen Benutzerbefehlen und der tatsächlichen Hardwarekonfiguration bildet. Sie vereinfacht die Benutzerinteraktion, da der Benutzer die ADC-Kanäle mit einfachen Befehlen auswählen kann, und erhöht die Zuverlässigkeit des Systems, indem sie die Kompatibilität mit dem UART-Kommunikationsprotokoll und den HAL-Parameteranforderungen sicherstellt.

Im Kontext dieses Programms wird convertStringToADCChannel in Fällen verwendet, in denen der Benutzer einen bestimmten ADC-Kanal auswählen möchte. Wenn der Benutzer z. B. den Befehlsstring "sense:channel vrefint" oder "sens:chan vrefint" sendet, wird die Funktion mit dem Argument "ADC_CHANNEL_VREFINT" aufgerufen, um den ADC-Kanal entsprechend einzustellen.

4.3.8 Funktion zur Umwandlung von ADC-Auflösungswerten in Strings

Die Funktion get_adc_resolution_string dient dazu, den numerischen Auflösungswert eines ADC (Analog-Digital-Wandler) in ein für den Anwender lesbares Textformat zu übersetzen. Dies ist wichtig, um dem Benutzer den Auflösungsstatus des ADC in sinnvoller Weise mitzuteilen. Beispielsweise wird der Wert für die 6-Bit-Auflösung in den HAL-Konstanten (Hardware Abstraction Layer) als ADC_RESOLUTION_6B bezeichnet.

```
193 const char* get_adc_resolution_string(uint32_t resolution)  // Diese Funktion gibt einen String zurück, der der übergebenen ADC-Auflösung entspricht
194 {
195       switch(resolution) {   // Switch-Case-Struktur zur Auswahl des korrekten Auflösungsstrings basierend auf dem eingegebenen resolution-Wert
196           case ADC_RESOLUTION_6B:
197               return "6 Bits";
198           case ADC_RESOLUTION_8B:
199               return "8 Bits";
200           case ADC_RESOLUTION_10B:
201               return "10 Bits";
202           case ADC_RESOLUTION_12B:
203               return "12 Bits";
204
205
206           default:
207               return "Unbekannt";
208       }
209   }
```

Die Funktion get_adc_resolution_string erhält die aktuelle Auflösungseinstellung des ADC als Argument. Dieses Argument repräsentiert die Auflösung im HAL-Format.

Intern bildet die Funktion diese HAL-Konstanten auf entsprechende lesbare Texte ab. Beispielsweise würde die Konstante ADC_RESOLUTION_6B in die Zeichenkette "6 bits" übersetzt. Diese Zuordnung wird je nach Implementierung durch eine Reihe von bedingten Prüfungen oder durch eine Lookup-Tabelle innerhalb der Funktion erreicht.

Sobald der Auflösungswert in einen String umgewandelt wurde, wird dieser String an die aufrufende Funktion zurückgegeben. Dies erleichtert die Kommunikation des ADC-Auflösungsstatus an den Benutzer, da der String über die UART-Schnittstelle übertragen, auf einer grafischen Benutzeroberfläche (GUI) angezeigt oder für andere Formen der Rückmeldung an den Benutzer verwendet werden kann.

Diese Funktion erhöht die Benutzerfreundlichkeit des Systems erheblich. Durch die Umwandlung der Auflösungswerte in ein für den Benutzer leicht verständliches Format trägt sie zu einer klaren und transparenten Interaktion zwischen Benutzer und System bei. Im Rahmen des bereitgestellten Programms wird get_adc_resolution_string verwendet, um dem Benutzer bei Bedarf den aktuellen ADC-Auflösungsstatus mitzuteilen.

4.3.9 Funktion zum Einstellen der ADC-Auflösung

Die Funktion setADCResoltuion dient zur Einstellung der Auflösung des ADC. Durch Änderung der Auflösung ermöglicht diese Funktion dem System, ein Gleichgewicht zwischen Genauigkeit und Wandlungsgeschwindigkeit auf der Grundlage spezifischer Anwendungsanforderungen herzustellen.

```
210 void setADCResoltuion( uint8_t resolution)  // Diese Funktion stellt die adc-Auflösung ein, abhängig von dem angegebenen Parameter
211 {
212
213     switch(resolution) {   // Verwendet eine Switch-Case-Struktur, um die eingegebene Auflösung zu überprüfen und die entsprechende ADC-Auflösung zuzuweisen
214         case 6:
215           hadc1.Init.Resolution  = ADC_RESOLUTION_6B;
216          division=63;   //(2^6)-1 Division ist der Parameter, um den ADC-Wert als Volt in Abhängigkeit von der Auflösung zu erhalten
217         break;
218
219         case 8:
220           hadc1.Init.Resolution  = ADC_RESOLUTION_8B;
221          division=255;
222         break;
223
224         case 10:
225           hadc1.Init.Resolution  = ADC_RESOLUTION_10B;
226          division=1023;
227         break;
228
229         case 12:
230           hadc1.Init.Resolution  = ADC_RESOLUTION_12B;
231          division=4095;
232         break;
233     }
234
235       if (HAL_ADC_Init(&hadc1) != HAL_OK)
236        {
237          Error_Handler();
238        }
239
240       HAL_Delay(10);
241 }
```

Die Funktion akzeptiert einen Integerwert, der die gewünschte Auflösung in Bit darstellt. Dieser ganzzahlige Wert wird zur Einstellung der Auflösung des ADC verwendet.

Im Zusammenhang mit dem Befehl « sense:resolution min » wird die Funktion setADCResolution beispielsweise mit einem Argument aufgerufen, das der minimalen Auflösung entspricht. Diese minimale Auflösung wird durch minRes repräsentiert, was einer vordefinierte Ganzzahl mit dem Wert 6 entspricht. Dieser Befehl weist die Funktion an, den ADC auf seine minimale Auflösung, d. h. eine Auflösung von 6 Bit, einzustellen.

Wird dagegen der Befehl « sense:resolution max » empfangen, so wird die Funktion mit einem Argument aufgerufen, das der maximalen Auflösungsstufe entspricht. Dieses maximale Auflösungsniveau wird durch maxRes repräsentiert, eine vordefinierte Ganzzahl mit dem Wert 12. In diesem Fall setzt die Funktion den ADC auf seine höchstmögliche Auflösung, d. h. 12 Bit.

Wenn im Befehl ein bestimmter numerischer Wert angegeben ist, wird dieser Wert als Argument für die Funktion setADCResolution verwendet. Der Wert wird mit der Funktion atoi aus dem Befehlsstring geparst, nachdem er mit der Funktion strlcpy isoliert wurde.

Innerhalb der Funktion setADCResolution wird das Argument verwendet, um die Auflösungseinstellung des ADC anzupassen. Dies geschieht in der Regel durch Zuweisung des Arguments zu einer der vordefinierten Auflösungseinstellungen in der ADC-Konfiguration. Diese Anpassung ermöglicht es dem System, die Auflösung entsprechend der Benutzereingabe zu ändern, was eine vielseitigere und reaktionsschnellere Steuerung des ADC-Betriebs ermöglicht.

4.3.10 Funktion zur Rückgabe des Oversampling-Ratio als Zahl

Die Funktion getRatio wird verwendet, um das numerische Verhältnis einer gegebenen Überabtastrate eines ADC (Analog-Digital-Wandler) zu ermitteln. Mit dieser Funktion kann das System nachfolgende Berechnungen in Bezug auf die Oversampling Ratio des ADC durchführen, was insbesondere bei Berechnungen mit der Blendenzeit nützlich ist.

```
int getRatio(uint32_t ratio)  // Diese Funktion gibt das Verhältnis einer gegebenen ADC-Überabtastungsrate als Zahl zurück
{
      switch(ratio) {  // Switch-Case-Struktur zur Auswahl des korrekten Verhältnisses basierend auf dem eingegebenen ratio-Wert
          case ADC_OVERSAMPLING_RATIO_2:
              return 2;
          case ADC_OVERSAMPLING_RATIO_4:
              return 4;
          case ADC_OVERSAMPLING_RATIO_8:
              return 8;
          case ADC_OVERSAMPLING_RATIO_16:
              return 16;
          case ADC_OVERSAMPLING_RATIO_32:
              return 32;
          case ADC_OVERSAMPLING_RATIO_64:
              return 64;
          case ADC_OVERSAMPLING_RATIO_128:
              return 128;
          case ADC_OVERSAMPLING_RATIO_256:
              return 256;

          default:
              return 0;
      }
  }
```

Die ADC-Überabtastrate wird normalerweise durch eine Konstante in der ADC-Konfiguration repräsentiert. Die HAL-Bibliothek könnte z. B. ein Verhältnis von 2 durch eine Konstante wie ADC_OVERSAMPLING_RATIO_2 repräsentieren. Die Funktion getRatio wurde entwickelt, um den numerischen Wert aus solchen Konstanten zu extrahieren und ein nützlicheres Format für die weitere Verarbeitung bereitzustellen.

Die Funktion arbeitet, indem sie das Argument, das einer ADC-Überabtastrate entspricht, interpretiert und das numerische Verhältnis aus diesem Argument extrahiert. Dies wird normalerweise erreicht, indem die Konstante der Überabtastrate auf das entsprechende numerische Verhältnis abgebildet wird.

Wird die Funktion z. B. mit dem Argument ADC_OVERSAMPLING_RATIO_2 aufgerufen, gibt sie die ganze Zahl 2 zurück. Diese ganze Zahl kann dann in nachfolgenden Berechnungen verwendet werden. Sie ist besonders nützlich bei der Berechnung der Aperturzeit des ADC, die vom Oversampling-Verhältnis abhängen kann.

Auf diese Weise vereinfacht die Funktion getRatio den Prozess der Anpassung von ADC-Einstellungen und -Berechnungen, indem sie eine einfache Methode zur Extraktion des numerischen Verhältnisses der ADC-Konstante für die Überabtastrate bereitstellt.

4.3.11 Funktion zur Einstellung der ADC-Apertur

Die Funktion setADCAperture wird verwendet, um die Aperturzeit des Analog-Digital-Wandlers (ADC) basierend auf dem angegebenen Aperturwert einzustellen. Die ADC-Aperturzeit bezieht sich auf den Zeitraum, in dem der ADC ein analoges Signal aktiv abtastet, und die Manipulation dieses Wertes kann die resultierende digitale Darstellung des Signals erheblich beeinflussen.

```
267 void setADCAperture( uint16_t aperture)  // Diese Funktion setzt die Apertur des ADC basierend auf einer gegebenen Apertur
268 {
269
270     switch(aperture) {// Verwendet eine Switch-Case-Struktur, um die eingegebene Apertur zu überprüfen und die entsprechende ADC-Apertur zuzuweisen
271  // Aperture basierend auf Oversampling Time von 24.4 Cycles
272         case 49:
273           hadc1.Init.Oversampling.Ratio = ADC_OVERSAMPLING_RATIO_2;
274           hadc1.Init.Oversampling.RightBitShift = ADC_RIGHTBITSHIFT_1;
275         break;
276
277         case 98:
278           hadc1.Init.Oversampling.Ratio = ADC_OVERSAMPLING_RATIO_4;
279           hadc1.Init.Oversampling.RightBitShift = ADC_RIGHTBITSHIFT_2;
280         break;
281
282         case 196:
283           hadc1.Init.Oversampling.Ratio = ADC_OVERSAMPLING_RATIO_8;
284           hadc1.Init.Oversampling.RightBitShift = ADC_RIGHTBITSHIFT_3;
285         break;
286
287         case 392:
288           hadc1.Init.Oversampling.Ratio = ADC_OVERSAMPLING_RATIO_16;
289           hadc1.Init.Oversampling.RightBitShift = ADC_RIGHTBITSHIFT_4;
290         break;
291
292         case 784:
293           hadc1.Init.Oversampling.Ratio = ADC_OVERSAMPLING_RATIO_32;
294           hadc1.Init.Oversampling.RightBitShift = ADC_RIGHTBITSHIFT_5;
295         break;
296
297         case 1568:
298           hadc1.Init.Oversampling.Ratio = ADC_OVERSAMPLING_RATIO_64;
299           hadc1.Init.Oversampling.RightBitShift = ADC_RIGHTBITSHIFT_6;
300         break;
301
302         case 3136:
303           hadc1.Init.Oversampling.Ratio = ADC_OVERSAMPLING_RATIO_128;
304           hadc1.Init.Oversampling.RightBitShift = ADC_RIGHTBITSHIFT_7;
305         break;
306
307         case 6272:
308           hadc1.Init.Oversampling.Ratio = ADC_OVERSAMPLING_RATIO_256;
309           hadc1.Init.Oversampling.RightBitShift = ADC_RIGHTBITSHIFT_8;
310         break;
311     }
312
313       if (HAL_ADC_Init(&hadc1) != HAL_OK)
314        {
315          Error_Handler();
316        }
317
318       HAL_Delay(10);
319 }
```

Die Funktion setADCAperture interpretiert den beabsichtigten Aperturwert und passt die Oversampling-Parameter des ADC, d.h. Oversampling Ratio und Right Bit Shift, entsprechend an.

Oversampling Ratio ist ein Parameter, der bestimmt, wie oft der ADC das Eingangssignal über die Nyquist-Rate hinaus abtastet. Right Bit Shift ist ein weiterer Parameter, der die digitale Auflösung des ADC beeinflusst.

Die Funktion nimmt den angegebenen Blendenwert und dividiert ihn durch eine Samplingzeit von 24,5 Zyklen, um das Überabtastverhältnis zu berechnen. Zum Beispiel ergibt ein Aperturwert von 784 geteilt durch 24,5 ein Oversampling-Ratio von ungefähr 32.

Der Parameter Right Bit Shift wird ebenfalls innerhalb der Funktion angepasst, um sicherzustellen, dass die gesamte ADC-Umwandlung korrekt gelesen werden kann, insbesondere wenn die Auflösung erhöht wird.

Es ist wichtig zu beachten, dass die Funktion setADCAperture und die abgeleiteten Blendenwerte auf einer Samplingzeit von 24,5 Zyklen basieren. Daher sind die berechneten Aperturwerte nicht genau, wenn sie mit einer anderen Messzeit verwendet werden.

Die Funktion setADCAperture bietet eine benutzerfreundliche Schnittstelle zur Einstellung dieser kritischen Parameter und ermöglicht so eine präzise und effiziente Steuerung des ADC-Betriebs.

4.3.12 Funktion zur Überprüfung des Befehlstyps

Die Funktion checkCommandType ist für die Steuerung des Anwendungsablaufs von entscheidender Bedeutung. Sie ist dafür verantwortlich, den Typ des über UART empfangenen Benutzerbefehls zu identifizieren und entsprechend zu verarbeiten. Die Ausgabe der Funktion bestimmt im Wesentlichen das Verhalten des Hauptprogramms, da jeder Befehl eine andere Reihe von Ereignissen oder Funktionen auslöst.

```
int checkCommandType(char* commandBuf)  // Diese Funktion überprüft den Befehlstyp eines gegebenen Befehls
{

    // Verwendet eine Switch-Case-Struktur, um das erste Zeichen des Befehls zu prüfen und dann den einzelnen Befehl zu bestimmen.

    switch (commandBuf[0]) {
    case 'm':
        if (strncmp(commandBuf, "measure:voltage?", 16) == 0 || strncmp(commandBuf, "meas:volt?", 10) == 0) {
            return 1;
        }
        break;
    case 's':
        if (strncmp(commandBuf, "sense:channel?", 14) == 0 || strncmp(commandBuf, "sens:chan?", 10) == 0) {
            return 2;

        } else if (strncmp(commandBuf, "sense:channel ", 14) == 0|| strncmp(commandBuf, "sens:chan ", 10) == 0) {
            return 3;

        } else if (strncmp(commandBuf, "sense:resolution? max", 21) == 0 || strncmp(commandBuf, "sens:reso? max", 14) == 0) {
            return 4;

        } else if (strncmp(commandBuf, "sense:resolution? min", 21) == 0 || strncmp(commandBuf, "sens:reso? min", 14) == 0) {
            return 5;

        } else if (strncmp(commandBuf, "sense:resolution?", 17) == 0 || strncmp(commandBuf, "sens:reso?", 10) == 0) {
            return 6;

        } else if (strncmp(commandBuf, "sense:resolution ", 17) == 0|| strncmp(commandBuf, "sens:reso ", 10) == 0) {
            return 7;

        } else if (strncmp(commandBuf, "sense:aperture? max", 19) == 0 || strncmp(commandBuf, "sens:aper? max", 14) == 0) {
            return 8;

        } else if (strncmp(commandBuf, "sense:aperture? min", 19) == 0 || strncmp(commandBuf, "sens:aper? min", 14) == 0) {
            return 9;

        } else if (strncmp(commandBuf, "sense:aperture?", 15) == 0 || strncmp(commandBuf, "sens:aper?", 10) == 0) {
            return 10;

        } else if (strncmp(commandBuf, "sense:aperture ", 15) == 0 || strncmp(commandBuf, "sens:aper ", 10)== 0){
            return 11;
        }
        break;
        case 'e':
            if (strncmp(commandBuf, "enable_oversampling", 19) == 0) {
                return 12;
            }
            break;
        case 'd':
            if (strncmp(commandBuf, "disable_oversampling", 20) == 0) {
                return 13;
            }
            break;
    }


    return 0;

}
```

Die Funktion checkCommandType erhält den über UART empfangenen Befehlsstring als Eingabe. Die Funktion analysiert diesen String, indem sie ihn mit einer Liste vordefinierter Befehlsmuster vergleicht. Jedes dieser Muster entspricht einem unterschiedlichen Befehlstyp.

Die Funktion normalisiert den Befehlsstring, durch Entfernen von Leerzeichen, durch Umwandlung aller Zeichen in Kleinbuchstaben mit der Funktion "lowercase" oder durch andere ähnliche Operationen. Dadurch wird sichergestellt, dass die Befehlszeile in einem zuverlässigen und standardisierten Format vorliegt, bevor der Vergleichsprozess beginnt.

Sobald der Befehl aufbereitet worden ist, vergleicht die Funktion ihn mit einer Liste der Befehlsmuster. Dies kann auf verschiedene Weise geschehen, z. B. mit Hilfe von Stringvergleichsfunktionen wie strcmp oder regulären Ausdrücken. Die Funktion sucht nach einer Übereinstimmung zwischen der Eingabezeichenkette und den Befehlsmustern.

Wenn eine Übereinstimmung gefunden wird, identifiziert die Funktion den entsprechenden Befehlstyp und gibt ihn als Ausgabe zurück. Beispielsweise kann eine Ganzzahl oder ein Enum-Wert zurückgegeben werden, der den Befehlstyp repräsentiert. Wenn keine Übereinstimmung gefunden wird, kann die Funktion einen speziellen Wert zurückgeben, der einen nicht erkannten oder ungültigen Befehl anzeigt.

Das Hauptprogramm verwendet die Ausgabe der Funktion checkCommandType, um zu bestimmen, welche Maßnahmen zu ergreifen sind. Gibt die Funktion beispielsweise einen Wert zurück, der auf einen "Set Channel"-Befehl hinweist, würde das Hauptprogramm die Funktion convertStringToADCChannel aufrufen, um den ADC-Kanal entsprechend einzustellen.

Zusammenfassend kann gesagt werden, dass die Funktion checkCommandType als eine Art Dispatch-Center für das Hauptprogramm dient. Indem sie die Art des Benutzerbefehls identifiziert, hilft sie, den Programmfluss zu kontrollieren und auf jeden Benutzerbefehl entsprechend zu reagieren.

4.3.13 Hauptprogramm

Das Hauptprogramm des entwickelten Systems wird durch die Funktion int main(void) repräsentiert, die als zentrale Steuerungseinheit fungiert. Sie koordiniert die Initialisierung der Peripherie und steuert die kontinuierliche Abarbeitung der eingehenden Befehle.

Wie im Programmcode zu erkennen ist, werden während der Initialisierungsphase des Programms verschiedene Systemkomponenten initialisiert.

```
379 int main(void)
380 {
381
382   HAL_Init();
383
384   /* Configure the system clock */
385   SystemClock_Config();
386
387
388   hadc1.Init.Resolution = ADC_RESOLUTION_12B;    //HAL-Funktion zur Initialisierung der ADC-Auflösung,
389   hadc1.Init.OversamplingMode = DISABLE;          // Oversampling deaktiviert als Startpunkt
390   hadc1.Init.Oversampling.Ratio = ADC_OVERSAMPLING_RATIO_256;   // Oversampling Ratio
391   hadc1.Init.Oversampling.RightBitShift = ADC_RIGHTBITSHIFT_8;   //Oversampling RightBitshift, abhängig von Ratio
```

HAL_Init(); Initialisiert die Bibliothek der Hardware Abstraction Layer (HAL), die den Prozessor, den Systemtimer und den eingebetteten Flash-Speicher enthält. Der Systemtakt des Mikrocontrollers wird dann durch Aufruf von SystemClock_Config() initialisiert. In dieser Phase wird auch die Voreinstellung für den ADC auf eine Auflösung von 12 Bit und deaktiviertes Oversampling festgelegt.

GPIO, DMA, UART und ADC1 werden dann mit den Funktionen MX_GPIO_Init(), MX_DMA_Init(), MX_USART2_UART_Init() und MX_ADC1_Init() initialisiert.

```
393   /* Initialize all configured peripherals */
394   MX_GPIO_Init();
395   MX_DMA_Init();
396   MX_USART2_UART_Init();
397   MX_ADC1_Init();
398   /* USER CODE BEGIN 2 */
399   HAL_UARTEx_ReceiveToIdle_DMA(&huart2, RxBuf, RxBuf_SIZE);
400   //Diese Funktion wird zum Empfangen von Daten vom UART mit DMA verwendet, bis der Datenpuffer voll ist oder eine Leerzeile erkannt wird.
401   __HAL_DMA_DISABLE_IT(&hdma_usart2_rx, DMA_IT_HT);
402   //Deaktivieren des Interrupts
```

Die Funktion HAL_UARTEx_ReceiveToIdle_DMA(&huart2, RxBuf, RxBuf_SIZE) wird verwendet, um Daten vom UART über DMA zu empfangen, bis der Datenpuffer voll ist oder eine leere Queue erkannt wird. Zusätzlich können Interrupts durch Aufruf von __HAL_DMA_DISABLE_IT(&hdma_usart2_rx, DMA_IT_HT) deaktiviert werden.

In der Hauptschleife des Programms wird zunächst geprüft, ob ein neuer Befehl empfangen wurde.

```
409   while (1)
410   {
411
412       if(newCommandReceived == true)                  //Prüfen, ob der neue Befehl empfangen wurde.
413                       {
414
415           char uart_buffer[50]; // Puffer für UART-Übertragung
416
417           switch (checkCommandType(lowercase((char *)commandBuf))) // Befehlstyp prüfen
418           {
419
```

Beim Eintreffen eines neuen Befehls identifiziert die Funktion checkCommandType automatisch den Typ des Befehls mit Hilfe einer switch-Anweisung. Das Programm führt dann die entsprechenden Aktionen aus. Insgesamt kann das System auf 13 verschiedene Befehlstypen individuell reagieren.

- **Fall 1 :** SpannungsMessung.

```
case 1: // measure:voltage?


    HAL_ADCEx_Calibration_Start(&hadc1, ADC_SINGLE_ENDED);  // ADC-Kalibrierung starten
    HAL_ADC_Start_DMA(&hadc1, (uint32_t *)&adc_value, 1);  //Startet den ADC-DMA-Modus für den ADC "hadc1"
    HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);   //Wartet, bis die ADC-Konvertierung abgeschlossen ist.
    HAL_Delay(100);  //Eine Pause von 100 Millisekunden, um sicherzustellen, dass der ADC-Wert stabil ist.

      float voltage = (3.3f * adc_value) / (division) ; // ADC-Wert in Spannung umrechnen


      sprintf(uart_buffer, "Voltage: %.3f V", voltage); //Schreibt den Spannungswert formatiert in den uart_buffer, um ihn später über UART zu übertragen.


    break;
```

Der Code startet die ADC-Kalibrierung mit HAL_ADCEx_Calibration_Start, startet den ADC-DMA-Modus für ADC "hadc1" mit HAL_ADC_Start_DMA, wartet auf den Abschluss der ADC-Konvertierung mit HAL_ADC_PollForConversion und konvertiert den ADC-Wert in eine Spannung.

- **Fall 2 :**Rückgabe des aktuellen ADC-Kanals.

```
case 2:
      sprintf(uart_buffer, "Current Channel: %s", get_adc_channel_string(sConfig.Channel));

      //Schreibt den formatierten Kanalnamen in den uart_buffer zur späteren Übertragung über UART.
    break;
```

Der Kanalname wird formatiert und mit sprintf in den uart_buffer geschrieben.

- **Fall 3 :** Setzen des ADC-Kanals entsprechend des empfangenen Befehls.

```
case 3:


                if(strcmp(lowercase((char *)commandBuf), "sense:channel vrefint")==0||strcmp(lowercase((char *)commandBuf), "sens:chan vrefint")==0)
                {
                    //Überprüft, ob der empfangene Befehl "sense:channel vrefint" oder "sens:chan vrefint" ist.

                convertStringToADCChannel("ADC_CHANNEL_VREFINT"); //ADC-Kanal auf "ADC_CHANNEL_VREFINT" setzen.

                }

                else if(strncmp(lowercase((char *)commandBuf), "sens:chan ", 10) == 0)
                {
                uint16_t channel= extract_integer_from_command(((char *)commandBuf),"sens:chan ");
                // Extrahiert die Kanalnummer aus dem Befehl "sense:channel x" und speichert sie in der Variable channel.
                char adc_channel_temp[20];
                snprintf(adc_channel_temp, sizeof(adc_channel_temp), "ADC_CHANNEL_%u", channel);
                //Erstellt einen temporären Puffer adc_channel_temp und formatiert den Kanalwert darin.
                convertStringToADCChannel(adc_channel_temp); //ADC-Kanal entsprechend adc_channel_temp  setzen.
                }
                else {

                uint16_t channel= extract_integer_from_command(((char *)commandBuf),"sense:channel ");
                // Extrahiert die Kanalnummer aus dem Befehl "sense:channel x" und speichert sie in der Variable channel.
                char adc_channel_temp[20];
                snprintf(adc_channel_temp, sizeof(adc_channel_temp), "ADC_CHANNEL_%u", channel);
                //Erstellt einen temporären Puffer adc_channel_temp und formatiert den Kanalwert darin.
                convertStringToADCChannel(adc_channel_temp); //ADC-Kanal entsprechend adc_channel_temp  setzen.

                }

                sprintf(uart_buffer, "Selected Channel: %s", get_adc_channel_string(sConfig.Channel));
                // Schreibt den ausgewählten ADC-Kanal in den uart_buffer für die spätere Übertragung über UART.


            break;
```

Der ADC-Kanal wird auf "ADC_CHANNEL_VREFINT" oder einen anderen Kanal gesetzt, basierend auf der Kanalnummer im Befehl, der mit der Funktion convertStringToADCChannel ermittelt wurde.

- **Fälle 4,5 und 6 :** Ausgabe der ADC-Auflösung

```
case 4: // Maximale Auflösung erhalten
        sprintf(uart_buffer, "Max ADC resolution: %d bits", maxRes); //Schreibt die maximale Auflösung des ADCs in den uart_buffer.


    break;
case 5: // Mindestauflösung erhalten

        sprintf(uart_buffer, "Min ADC resolution: %d bits", minRes); //Schreibt die minimale Auflösung des ADCs in den uart_buffer.

    break;


case 6: // aktuelle Auflösung erhalten
{

    sprintf(uart_buffer, "Current resolution: %s ", get_adc_resolution_string(hadc1.Init.Resolution));
    //Schreibt die aktuelle Auflösung des ADCs in den uart_buffer unter Verwendung der Funktion get_adc_resolution_string.


    break;
}
```

In diesen Fällen wird die Auflösung des ADC ausgegeben. Fall 4 liefert die maximale Auflösung, Fall 5 die minimale Auflösung und Fall 6 die aktuelle Auflösung mit Hilfe der Funktion "get_adc_resolution_string". Alle Werte werden korrekt formatiert und in den uart_buffer geschrieben.

- **Fall 7 :** Einstellung der ADC-Auflösung.

```
case 7:  //Set Resolution
{
    if(strncmp(lowercase((char *)commandBuf), "sense:resolution min",20)==0 || strncmp(lowercase((char *)commandBuf), "sens:reso min",13)==0 ) // Befeh Überprüfung
    {
        setADCResoltuion(minRes);  //ADC-Auflösung entsprechend der angegebenen Auflösung setzen, hier Min Res

    }
    else if(strncmp(lowercase((char *)commandBuf), "sense:resolution max",20)==0 || strncmp(lowercase((char *)commandBuf), "sens:reso max",13)==0)
    {
        setADCResoltuion(maxRes); // ADC-Auflösung entsprechend der angegebenen Auflösung setzen, hier Max Res

                }
    else if (strncmp(lowercase((char *)commandBuf), "sens:reso ",10)==0){

        uint16_t resolution = extract_integer_from_command(((char *)commandBuf),"sens:reso ");
        //Extrahiert die Auflösungszahl aus dem Befehl "sens:reso x" und speichert sie in der Variable resolution.
        setADCResoltuion(resolution);
    }
    else{

        uint16_t resolution = extract_integer_from_command(((char *)commandBuf),"sense:resolution ");
        setADCResoltuion(resolution);
    }


    sprintf(uart_buffer, "Selected resolution: %s ", get_adc_resolution_string(hadc1.Init.Resolution));
    break;
}
```

Abhängig vom empfangenen Befehl wird die Auflösung auf einen minimalen, maximalen oder festen Wert gesetzt. Dies wird mit der Funktion setADCResoltuion erreicht.

- **Fälle 8,9 und 10 :** Aperturwert ausgeben.

```
case 8: // maximale Apertur erhalten

        sprintf(uart_buffer, "Max Aperture: %d Cycles", maxAp);

    break;
case 9: // minimale Apertur erhalten

        sprintf(uart_buffer, "Min Aperture: %d Cycles", minAp);
    break;

case 10: // aktuelle Apertur ermitteln

        currentAp=24.5*getRatio(hadc1.Init.Oversampling.Ratio); // Berechnet die aktuelle Apertur des ADCs basierend auf dem Oversampling-Verhältnis
        sprintf(uart_buffer, "Current Aperture: %d Cycles", currentAp);
    break;
```

Hier wird der Wert der Apertur zurückgegeben, entweder max, min oder der aktuelle Wert unter Verwendung der Formel

$$Aperture = (Oversampling\ Ratio * Sampling\ Time)$$

wobei Sampling Time 24,5 gewählt wurde.

- **Fall 11 :** Einstellen der ADC-Apertur.

```
case 11: // Apertur einstellen

    if(strncmp(lowercase((char *)commandBuf), "sense:aperture min",18)==0 || strncmp(lowercase((char *)commandBuf), "sens:aper min",13)==0)
    {
        setADCAperture(minAp);

    }
    else if(strncmp(lowercase((char *)commandBuf), "sense:aperture max",18)==0 || strncmp(lowercase((char *)commandBuf), "sens:aper max",13)==0)
    {
        setADCAperture(maxAp);


    }else if(strncmp(lowercase((char *)commandBuf), "sens:aper ",10)==0) {

        uint16_t aperture = extract_integer_from_command(((char *)commandBuf),"sens:aper ");
        //Extrahiert den Aperturwert aus dem Befehl "sens:aper x" und speichert ihn in der Variable aperture.
        setADCAperture(aperture);
        //die Apertur des ADCs entsprechend dem angegebenen Wert setzen
                                        }
    else{

        uint16_t aperture = extract_integer_from_command(((char *)commandBuf),"sense:aperture ");

        setADCAperture(aperture);

                                        }

    int aperture=24.5*getRatio(hadc1.Init.Oversampling.Ratio);

    sprintf(uart_buffer, "Selected aperture: %d Cycles",aperture);


    break;
```

Bei Empfang des Befehls "sense:aperture min" oder "sens:aper min" wird die Apertur auf den minimalen Wert gesetzt (setADCAperture(minAp);). Im Fall von "sense:aperture max" oder "sens:aper max" wird die Apertur auf den maximalen Wert gesetzt (setADCAperture(maxAp);).

Wenn jedoch "sens:aper x" oder "sense:aperture x" empfangen wurde, wird die Apertur auf den Wert x gesetzt. Der entsprechende Wert wird aus dem Befehl extrahiert und mit setADCAperture(aperture); gesetzt. Nach dem Setzen wird die Apertur formatiert und in den uart_buffer geschrieben.

- **Fälle 12 und 13 :** Oversampling ein-/ausschalten

```
case 12: // Enable Oversampling

        hadc1.Init.OversamplingMode = ENABLE; //Aktiviert das Oversampling für den ADC.
        HAL_ADC_Init(&hadc1);  //nitialisiert den ADC nach der Aktivierung des Oversamplings.
        sprintf(uart_buffer, "Oversampling enabled");

    break;

case 13: // Disable Oversampling

        hadc1.Init.OversamplingMode = DISABLE; ////Deaktiviert das Oversampling für den ADC.
        HAL_ADC_Init(&hadc1);
        sprintf(uart_buffer, "Oversampling disabled");
    break;
```

Oversampling wird mit "HADC1.Init.OversamplingMode " aktiviert oder deaktiviert und der ADC wird mit HAL_ADC_Init initialisiert, um die letzte Veränderung zu implementieren.

4.4 UART-Kommunikation

Die UART-Kommunikation ist eine wichtige Komponente des Programms und dient als Kommunikationsweg für den Empfang von Benutzerbefehlen. Über eine grafische Benutzeroberfläche (GUI), die über USB übertragen wird, kann der Benutzer Befehle eingeben, die vom Programm interpretiert und ausgeführt werden.

In diesem Programm wird USART2 für die UART-Kommunikation verwendet. USART2 ist eine der seriellen Hardware-Schnittstellen, die auf dem Mikrocontroller zur Verfügung stehen. Die USART2-Instanz wird mit einer bestimmten Konfiguration initialisiert, um die Kommunikationsanforderungen optimal zu erfüllen.

In der UART-Konfiguration wird eine Baudrate von 115200 verwendet. Dies entspricht einer etablierten Standardgeschwindigkeit in der seriellen Datenkommunikation, wobei 115200 Baud im Wesentlichen bedeutet, dass 115200 Bits pro Sekunde übertragen werden. Außerdem ist die Datenwortlänge auf 8 Bit festgelegt. Das bedeutet, dass jedes gesendete oder empfangene Datenpaket aus 8 Bit besteht, eine Größe, die im allgemeinen Sprachgebrauch als Byte bezeichnet wird.

Schließlich wird die Anzahl der Stoppbits auf eins festgelegt. Stoppbits spielen in der seriellen Kommunikation eine wichtige Rolle, da zur Synchronisation des Datenstroms dienen. Sie signalisieren das Ende der Übertragung eines Bytes, indem nach jedem gesendeten Byte ein einzelnes Stoppbit gesendet wird. Auf diese Weise kann die Empfangsseite ein Byte vom nächsten unterscheiden, was die Integrität der übertragenen Daten sicherstellt.

In der Praxis wird ein Befehl, der von der grafischen Benutzeroberfläche über USB gesendet wird, von der USART2-Instanz der UART-Schnittstelle empfangen und in einem Puffer gespeichert. Diese Daten werden dann verarbeitet und die entsprechenden Aktionen auf der Grundlage des empfangenen Befehls ausgeführt. Beispielsweise veranlasst ein Befehl zur Spannungsmessung das Programm, die für die Durchführung der Messung erforderlichen Funktionen zu starten und die Ergebnisse über denselben UART-Kommunikationskanal an den Benutzer zurückzusenden.

5.GUI-Entwicklung mit Qt Designer und Python

5.1 GUI-Design

Die Gestaltung der grafischen Benutzeroberfläche (GUI) ist ein wesentlicher Bestandteil der ADC-Softwareanwendung, da sie eine intuitive und benutzerfreundliche Möglichkeit zur Interaktion mit dem System bietet. Eine GUI kapselt die zugrundeliegenden komplexen Operationen und bietet dem Benutzer einfach zu bedienende grafische Elemente, mit denen er Aufgaben ausführen kann, ohne den zugrundeliegenden Code verstehen zu müssen. Das Design ist klar und übersichtlich und bietet dem Benutzer eine einfache Bedienung. Die Benutzeroberfläche ist in verschiedene Abschnitte unterteilt, von denen jeder einer bestimmten ADC-Funktion zugewiesen ist. Sie enthält Abschnitte für das Verbinden und Trennen des Systems, die Kanalauswahl, die Auflösungseinstellungen und die Apertureinstellungen.

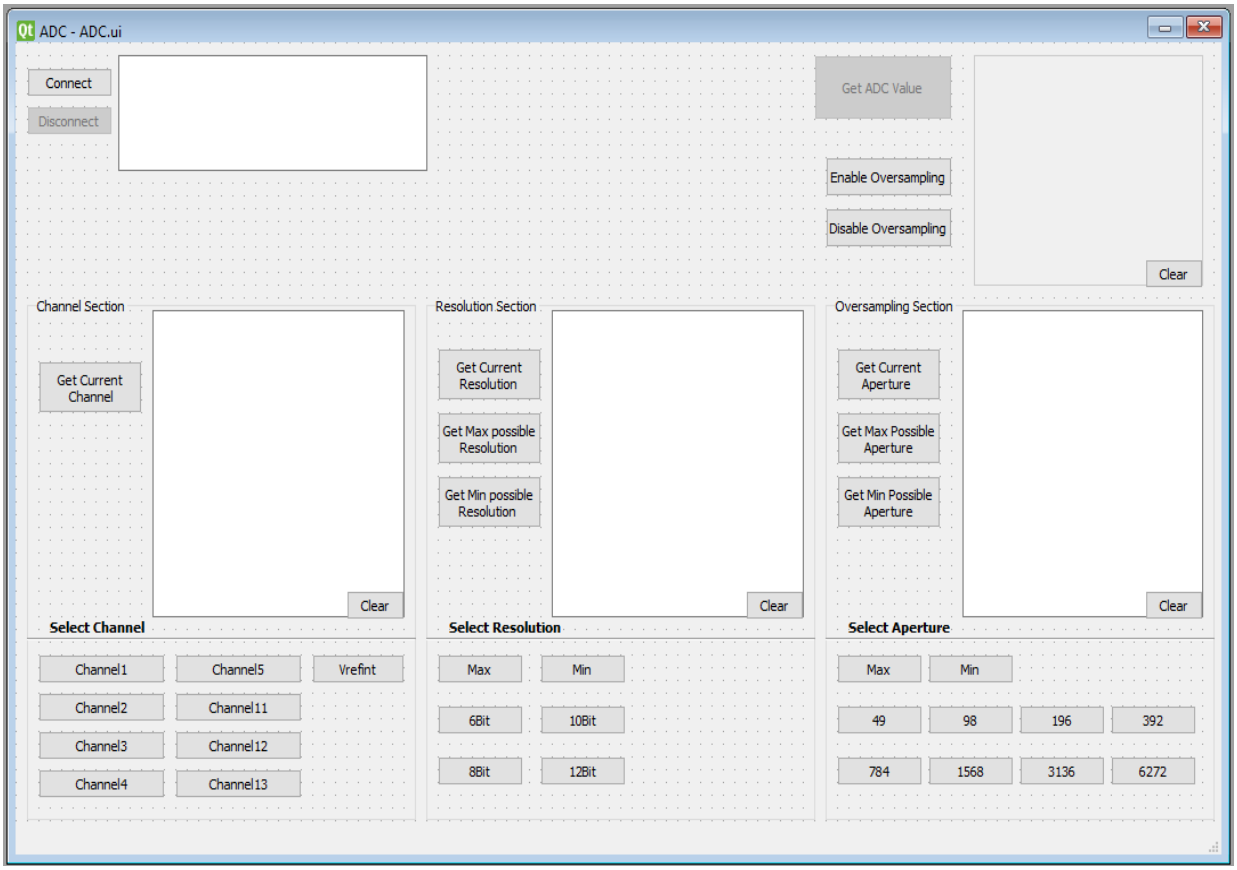

Abbildung 11: Übersicht über die Benutzeroberfläche

Für die verschiedenen Befehle werden im gesamten Design zahlreiche Schaltflächen verwendet, die dem Benutzer ein vertrautes Interaktionsmuster bieten. Jeder Taste ist ein bestimmter Befehl zugeordnet, der eine Funktion aktiviert, um die entsprechende Aktion auszuführen. Beispielsweise kann der Benutzer mit einem Klick Kanäle auswählen, die Auflösung einstellen oder den Messvorgang starten. Ein weiteres wichtiges Element des GUI-Designs sind Textfelder. Sie liefern Echtzeit-Feedback und zeigen die Ergebnisse der Befehlsausführung an. Sie helfen auch bei der Anzeige von Fehlermeldungen und erleichtern dem Benutzer die Diagnose und Behebung von Problemen.

5.2 Kanalauswahlbereich

Der Kanalauswahlbereich der GUI dient zur Verwaltung und Steuerung der Eingangskanäle des ADC-Systems. Dieser Teil bietet die Möglichkeit, verschiedene Kanäle je nach den Bedürfnissen des Benutzers auszuwählen, was ein hohes Maß an Anpassung und Flexibilität für verschiedene Anwendungsfälle bietet.

Der Kanalauswahlbereich ist benutzerfreundlich und einfach gestaltet. Die grafische Benutzeroberfläche stellt für jeden Kanal eine eigene Schaltfläche mit der entsprechenden Kanalnummer zur Verfügung, so dass die Auswahl des gewünschten Kanals klar und einfach ist. Wenn der Benutzer auf einen Button klickt, wird die entsprechende Kanalnummer extrahiert und in einer Variablen gespeichert.

Um den ausgewählten Kanal an das ADC-System zu senden, wird ein Befehl generiert, der die entsprechende Kanalnummer enthält. Dieser Befehl wird über die serielle Schnittstelle an den Mikrocontroller gesendet. Nachfolgend ein Codeausschnitt zur Illustration der Kanalauswahl:

```
81    def SelectChannelFunction(self):
82        Channel = self.sender().objectName()
83        channel_number = Channel[7:]
84        Command = f"SENS:CHAN {channel_number}" + "\n"
85        ser.write(Command.encode())
86        receivedLine = str(ser.readline())
87        N = len(receivedLine)
88        self.TextChannel.append(">>" + receivedLine[2:N])
89
```

In diesem Code ist die Methode SelectChannelFunction dargestellt, welche aufgerufen wird, wenn eine Kanalschaltfläche angeklickt wird. Der Name der Schaltfläche wird erfasst und die Kanalnummer aus dem Namen extrahiert. Anschließend wird ein Befehl mit der Kanalnummer generiert und über die serielle Schnittstelle an den Mikrocontroller gesendet. Die Antwort des Boards wird gelesen und in einem Textfeld in der GUI angezeigt, um zu bestätigen, dass die Kanalauswahl erfolgreich war.

Zusätzlich zur manuellen Kanalauswahl bietet die grafische Benutzeroberfläche die Möglichkeit, eine interne Spannungsreferenz (VREFINT) als Kanal auszuwählen. Diese Funktion ist besonders für Kalibrierungs- und Diagnosezwecke nützlich. Durch die Auswahl dieses Kanals kann der Benutzer die interne Referenzspannung überwachen, was zum Verständnis der Gesamtfunktion des ADC-Systems beiträgt.

5.3 Bereich ADC Auflösung

Der Bereich ADC Resolution der Benutzeroberfläche dient als umfassendes Werkzeug zur Steuerung der Auflösung des Analog-Digital-Wandlers (ADC). Dieser Bereich ermöglicht es dem Benutzer, die Auflösungseinstellung nach Bedarf anzupassen und bietet eine dynamische Schnittstelle, welche die Genauigkeit und Effizienz des ADC-Betriebs erleichtert. Die ADC-Auflösungssektion wurde für eine einfache Bedienung konzipiert.

Die Abfrage der aktuellen Auflösung wird durch die Funktion ADCGetResolutionFunction realisiert.

```
111     def ADCGetResolutionFunction(self):
112         Command = "SENSe:RESOlution?"+ "\n"
113         ser.write(Command.encode())
114         receivedLine2 = str(ser.readline())
115         N = len(receivedLine2)
116         self.TextResolution.append(">>" + receivedLine2[2:N])
117
```

Diese Funktion sendet den Befehl "SENSe:RESOlution?" an den Mikrocontroller und erwartet eine Antwort, welche die aktuelle Auflösung des ADCs enthält. Die erhaltene Antwort wird in einem Textfeld der GUI angezeigt, um dem Benutzer eine sofortige Rückmeldung zu geben.

Die Einstellung der Auflösung erfolgt mit der Funktion ADCSETResolutionFunction.

```
132    def ADCSETResolutionFunction(self):
133        Resolution = self.sender().objectName()
134        Resolution_bits = Resolution[3:]
135        Command = f"SENSe:RESOlution {Resolution_bits}" + "\n"
136        ser.write(Command.encode())
137        receivedLine2 = str(ser.readline())
138        N = len(receivedLine2)
139        self.TextResolution.append(">>" + receivedLine2[2:N])
```

Diese Funktion extrahiert die gewünschte Auflösung aus dem Namen des auslösenden Buttons. Der entsprechende Befehl "SENSe:RESOlution {Resolution_bits}" wird an den Mikrocontroller gesendet, um die Auflösung entsprechend einzustellen. Die erhaltene Antwort wird ebenfalls im Textfeld angezeigt.

5.4 Oversampling und Apertur-Bereich

Der Bereich "Oversampling und Apertur" der grafischen Benutzeroberfläche ist ein wichtiger Teil, der es dem Benutzer ermöglicht, die erweiterten Funktionen des ADC zu verwalten. Der Zweck dieses Abschnitts besteht darin, die Leistung des ADU durch die Manipulation der Oversampling-Funktion und die Einstellung der Aperturzeit zu optimieren. Oversampling ist eine digitale Signalverarbeitungstechnik, welche die Auflösung des ADC-Ausgangs erhöht. In diesem Teil der grafischen Benutzeroberfläche kann der Benutzer Oversampling über eine einfache Schnittstelle aktivieren oder deaktivieren. Zwei verschiedene Schaltflächen mit den Bezeichnungen Enable Oversampling und Disable Oversampling machen es einfach, diese Funktion mit einem Mausklick umzuschalten. Diese Flexibilität ermöglicht es dem Benutzer, die Konfiguration zu wählen, die seinen spezifischen Anforderungen am besten entspricht, und so die ADC-Leistung in einer Vielzahl von Situationen zu verbessern. Wenn Oversampling aktiviert ist, wird der gesamte Oversampling- und Aperturbereich aktiv und zeigt Funktionen an, die von der Apertur abhängig sind. Die Aperturzeit ist das Zeitintervall, in dem der ADC das Eingangssignal abtastet. Die Einstellung der Zeitdauer beeinflusst die Wandlungsgeschwindigkeit und die Genauigkeit des ADCs und ermöglicht dem Benutzer eine Feinabstimmung der Funktion

des ADCs. Die Benutzeroberfläche bietet mehrere vordefinierte Einstellungen für die Aperturzeit, die jeweils einer bestimmten Taste zugeordnet sind.

```
186     def SETApertureFunction(self):
187         Aperture = self.sender().objectName()
188         Aperture_number = Aperture[2:]
189         Command = f"SENS:APER {Aperture_number}" + "\n"
190         ser.write(Command.encode())
191         receivedLine2 = str(ser.readline())
192         N = len(receivedLine2)
193         self.TextAperture.append(">>" + receivedLine2[2:N])
```

Die Funktion "SETApertureFunction" ermöglicht dem Benutzer die Einstellung der gewünschten Aperturzeit. Wenn der Benutzer eine bestimmte Schaltfläche für die Aperturzeit auswählt, wird die Funktion ausgeführt. Der Button wird identifiziert und die entsprechende Aperturzeit extrahiert. Anschließend wird der Befehl "SENS:APER <Aperturzeit>" über die serielle Verbindung an den Mikrocontroller gesendet, um die Einstellung vorzunehmen. Die empfangene Antwort über die erfolgreiche Parametrierung wird in einem Textfeld angezeigt.

5.5 Anzeige der ADC-Werte

In der GUI der Anwendung bietet ein Textfeld in Kombination mit der Schaltfläche "Get ADC Value" dem Benutzer die Möglichkeit, die Funktion des Analog-Digital-Wandlers (ADC) im laufenden Betrieb zu beobachten und zu analysieren. Die Schaltfläche " Get ADC Value " ist so programmiert, dass sie bei jedem Klick eine ADC-Konvertierung auslöst. Das Ergebnis dieser Konvertierung wird unmittelbar nach der Durchführung in dem dafür vorgesehenen Textfeld in der grafischen Benutzeroberfläche angezeigt. Die dahinterliegende Funktion, die diese Interaktion ermöglicht, ist ADCValuefunction(self).

```
104     def ADCValuefunction(self):
105         Command = "MEAS:VOLT?"+ "\n"
106         ser.write(Command.encode())
107         receivedLine2 = str(ser.readline())
108         N = len(receivedLine2)
109         self.textADC.append(">>" + receivedLine2[2:N])
```

Diese Funktion dient dazu, mit dem Analog-Digital-Wandler (ADC) zu kommunizieren und besteht aus mehreren Schritten. Zuerst wird der Befehl "MEAS:VOLT?" erzeugt,

um eine Spannungsmessung anzufordern. Mit der Methode ser.write(Command.encode()) wird dieser Befehl in ein Byteformat umgewandelt und an das serielle Gerät gesendet. Die Antwort des seriellen Geräts, ein Byte-String, der die Spannungsmessung repräsentiert, wird dann mit ser.readline() ausgelesen und in einen String umgewandelt, der in receivedLine2 gespeichert wird. Schließlich wird die Antwort an das Textfeld self.textADC hinzugefügt.

5.6 Kommunikation mit dem Mikrocontroller

Die Kommunikation zwischen der GUI-Anwendung und dem Mikrocontroller wird über die Python-Bibliothek "pySerial" abgewickelt. Diese Bibliothek bietet leistungsfähige serielle Kommunikationsfunktionen und wird durch den Aufruf "import serial" eingebunden. Das Skript verwendet die Bibliothek zur Konfiguration der UART-Parameter, einschließlich COM-Port-Nummer, Baudrate, Parität, Stoppbit und Bytegröße, welche die Konfiguration des Mikrocontrollers widerspiegeln.

```
 8   ser=serial.Serial()
 9   ser.port = 'COM3'
10   ser.baudrate = 115200
11   ser.parity   = serial.PARITY_NONE
12   ser.stopbits = serial.STOPBITS_ONE
13   ser.bytesize = serial.EIGHTBITS
```

Diese Synchronisierung gewährleistet eine nahtlose Kommunikation, die ein wesentlicher Aspekt der Echtzeit-Datenverarbeitung durch Mikrocontroller ist. Jede Benutzerinteraktion, wie z. B. das Drücken einer GUI-Schaltfläche, löst eine spezielle Funktion aus, die den entsprechenden Befehl mit der Funktion "ser.write()" an den Mikrocontroller sendet. In der Methode hinter der Schaltfläche zum Abrufen des ADC-Werts wird beispielsweise der Befehl "MEASure:VOLTage?" gesendet, um den ADC-Spannungswert abzufragen. Nach dem Senden wartet die Anwendung mit der Funktion "ser.readline()" auf eine Antwort. Die empfangenen Bytes werden in einen String umgewandelt und in der GUI angezeigt. Das Design bietet dem Benutzer ein dynamisches Feedback und verbessert die Kontrolle und Interaktion mit dem ADC.

6.Software und Hardware Tests

Die Inbetriebnahme der entwickelten Umgebung umfasst umfangreiche Tests, um die Genauigkeit und Leistungsfähigkeit der programmierten Spannungsmessgeräte sicherzustellen.

Die erste Testphase stellt ein Funktionstest dar. Dabei wird die interne Spannungsreferenz des Mikrocontrollers (VREFINT) verwendet und die GUI-Tasten und der ADC-Ausgangswert durch Anklicken gesteuert.

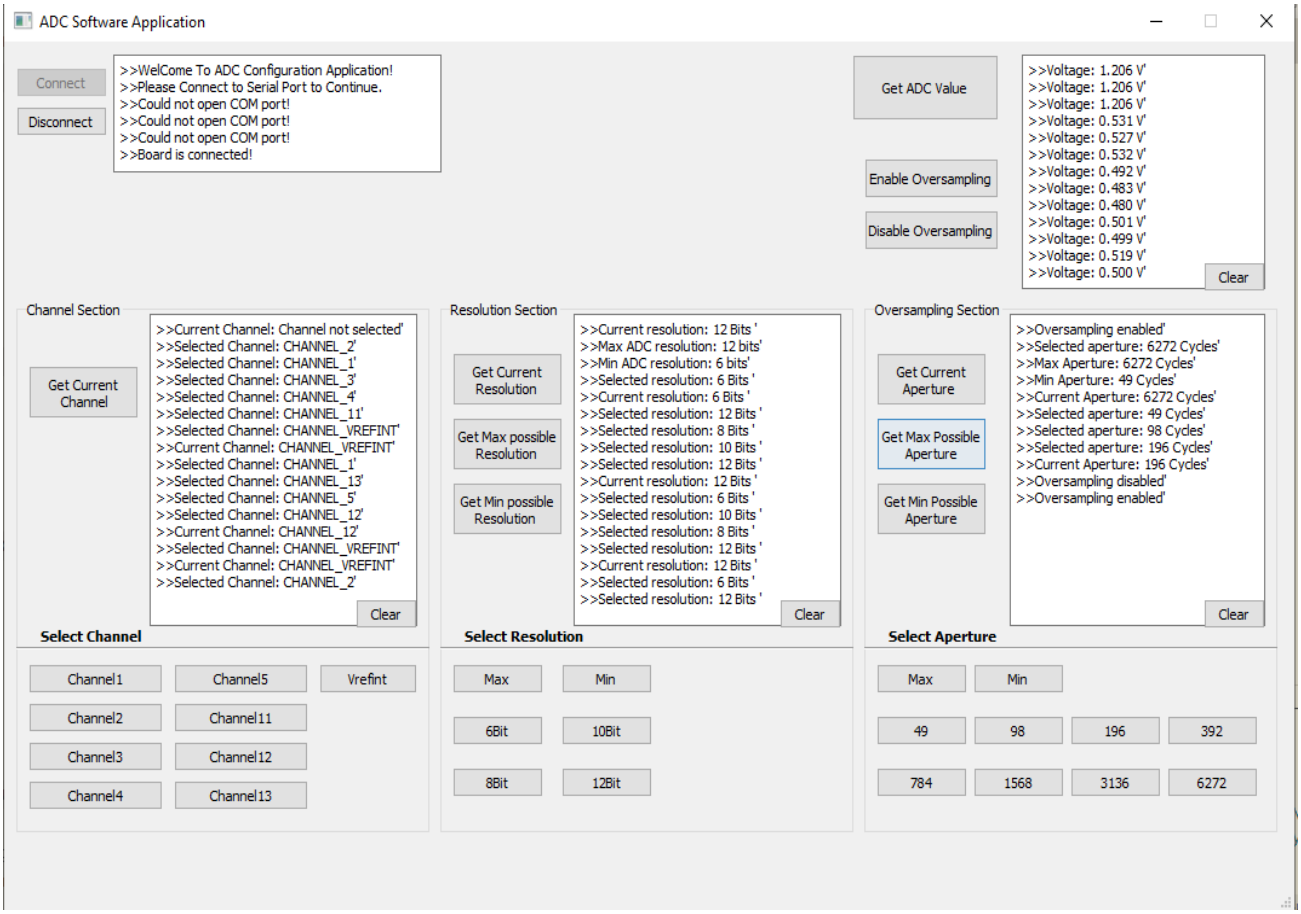

Abbildung 12: Test der GUI und der ADC-Software

Mit diesem Test wird überprüft, ob die grafische Benutzeroberfläche und der Mikrocontroller korrekt programmiert sind und zusammenarbeiten. Die Funktionsfähigkeit konnte erfolgreich nachgewiesen werden, da die UART-Werte direkt aus der ADC-Konfiguration extrahiert werden.

Nach diesem ersten Test wurden umfangreiche Labortests durchgeführt. Der Mikrocontroller wurde mit einer externen Spannungsquelle und Masse (GND) verbunden und der Programmkanal über Kabel angeschlossen. Die Quellenspannung wurde von 0 bis 5 V eingestellt und die ADC-Ausgangswerte wurden in verschiedenen Konfigurationen aufgezeichnet. Es wurde festgestellt, dass jeder Tastendruck eine Änderung der ADC-Auflösung und der Kanalauswahl bewirkt und dass sich diese Änderungen direkt auf den ADC-Ausgangswert auswirken.

Da der ADC mit einer Referenzspannung von 3,3 Volt arbeitet, ist es wichtig zu wissen, wie der Mikrocontroller mit einer analogen Eingangsspannung über 3,3 Volt umgeht. Wenn Spannungen eingegeben werden, die diesen Wert von 3,3 Volt überschreiten, interpretiert der ADC diese als 3,3 Volt und wandelt sie in den entsprechenden digitalen Wert um. Daher werden alle Eingangsspannungen, die größer als 3,3 Volt sind, in der digitalen Darstellung auf den maximalen Wert abgebildet, der einer analogen Eingangsspannung von 3,3 Volt entspricht. Das bedeutet, dass Spannungsspitzen über 3,3 Volt nicht zu einer Änderung des digitalen Ausgangswertes des ADC führen, da diese Werte auf 3,3 Volt begrenzt sind.

Die ADC-Abtastzeit variiert ebenfalls zwischen 2,5 Zyklen und 640,5 Zyklen. Interessanterweise wurde, wie aus der Tabelle ersichtlich, kein signifikanter Einfluss auf den ADC-Wert über 24,5 Zyklen festgestellt, was darauf hindeutet, dass die optimale Abtastzeit für diese spezielle Anwendung 24,5 Zyklen beträgt.

Tabelle 1 12-Bit-ADC-Messungen mit unterschiedlichen Abtastzeiten

ADC-Abtastzeit	Eingangspannung	ADC-Spannung	Abweichung
2.5 Cycles	3 V	2,858 V	4.73 %
12.5 Cycles	3 V	2,966 V	1.13 %
24.5 Cycles	3 V	2,987 V	0.43 %
47.5 Cycles	3 V	2,988 V	0.40 %
92.5 Cycles	3 V	2,984 V	0.53 %
247.5 Cycles	3 V	2,986 V	0.46 %
640.5 Cycles	3 V	2,987 V	0.43 %

In der Tabelle sind nur die Werte von 12-Bit-Messungen mit eingeschaltetem Oversampling bei maximaler Apertur aufgeführt. Auflösungstests zwischen 6-Bit- und 12-

Bit-Auflösung bei allen Abtastzeiten zeigen vernachlässigbare Unterschiede bei aktivierter Überabtastung. Dies ist darauf zurückzuführen, dass die Gesamtauflösung höher ist und die Überabtastung auch bei der niedrigsten Auflösung zu genauen Messwerten führt.

Ein bedeutender Aspekt der Testphase war der Vergleich der Genauigkeit der ADC-Messwerte mit und ohne Oversampling. Oversampling erhöht die Auflösung der ADC-Messwerte durch Reduzierung des Quantisierungsrauschens und erhöht damit die Genauigkeit der Messwerte.

Tabelle 2: 12-Bit-ADC-Messungen mit/ohne Oversampling

Eingangspannung	ADC-Wert ohne Oversampling	Abweichung	ADC-Wert mit Oversampling	Abweichung
1 V	0.955-1.1 V	Bis zu 4.5%	0.995 V	0.50 %
1.5 V	1.48-1.515 V	Bis zu 1.3%	1.493 V	0.46 %
2 V	1.970-2.02 V	Bis zu 1.5%	1.992 V	0.40 %

Mit Oversampling betragen die ADC-Messwerte bei 2 V, 1,5 V und 1 V beispielsweise 1,992, 1,493 bzw. 0,994. Im Vergleich dazu betragen die ADC-Messwerte ohne Oversampling bei gleicher Eingangsspannung 1,972-2,02, 1,485-1,515 bzw. 0,955-1,1, was auf einen größeren Wertebereich und damit auf eine geringere Genauigkeit hinweist.

Schließlich hat die Testphase erfolgreich die robuste Funktion des programmierbaren Spannungsmessgerätes mit STM32 Mikrocontroller und GUI demonstriert und seine Effektivität in verschiedenen Konfigurationen und Bedingungen bestätigt.

7. Zusammenfassung und Schlussbemerkung

Die Abschlussarbeit beschäftigt sich mit der Programmierung eines STM32 Mikrocontrollers als steuerbares Voltmeter mit SCPI Schnittstelle. Die Möglichkeiten von Qt Designer und Python werden genutzt, um eine grafische Benutzeroberfläche (GUI) zu entwickeln, die eine nahtlose Kommunikation zwischen dem Benutzer und dem Entwicklungsboard ermöglicht. Die GUI beinhaltet Funktionen wie Kanalauswahl, Suche und Änderung der ADC-Auflösung, Steuerung und Auslesen der ADC-Werte aus dem ausgewählten Kanal, der Auflösung und der Apertur.

Mit Blick auf die Zukunft gibt es potentielle Entwicklungsmöglichkeiten. Ein Bereich, der von zukünftigen Entwicklungen profitieren könnte, ist die Integration einer dynamischen Abtastzeitsteuerung in die grafische Benutzeroberfläche. Durch Hinzufügen einer Funktion zur Anpassung der Abtastzeit direkt über die grafische Benutzeroberfläche könnte der Benutzer die ADC-Konvertierung an seine spezifischen Anforderungen anpassen. Hierzu wäre die Integration dynamischer Aperturwerte erforderlich, die sich in Abhängigkeit von der gewählten Abtastzeit anpassen.

Außerdem könnte die Möglichkeit in Betracht gezogen werden, zwischen verschiedenen ADC-Konvertierungsmodi zu wechseln. Beispielsweise könnte eine neue Funktion in die grafische Benutzeroberfläche integriert werden, die es dem Benutzer ermöglicht, kontinuierliche Wandlungen mit einem einzigen Tastendruck zu aktivieren. Diese Funktion würde die Vielseitigkeit des Spannungsmessgeräts erhöhen, so dass es für ein breiteres Spektrum von Anwendungen eingesetzt werden könnte.

Zusammenfassend beweist diese Arbeit, dass es möglich ist, mit einem sorgfältigen Entwurf, gründlichen Tests, einem STM32 Mikrocontroller und einer gut gestalteten Benutzerschnittstelle ein multifunktionales Spannungsmessgerät zu implementieren. Das Projekt hat nicht nur seine ursprünglichen Ziele erreicht, sondern auch Verbesserungsmöglichkeiten für zukünftige Arbeiten aufgezeigt, die dem Benutzer mehr Kontrolle über den Wandlungsprozess geben und somit die Bandbreite möglicher Anwendungen für dieses Gerät erweitern könnten.

8. Literaturverzeichnis

[1] https://www.mikrocontroller.net/articles/STM32

[2] STMicroelectronics.: https://www.st.com/en/evaluation-tools/nucleo-l476rg.html

[3] Arm Keil: https://www.keil.com/boards2/stmicroelectronics/nucleo_l476rg/

[4] Grant Maloy Smith (8.09.2022): https://dewesoft.com/de/blog/was-ist-ein-analog-digital-wandler#sar-a-d-wandler-sukzessive-approximation

[5] Nourane Gamal(12.2015): https://www.researchgate.net/figure/SAR-ADC-block-diagram-6_fig10_304347043

[6] Wikipedia(31.01.2023): https://en.wikipedia.org/wiki/Standard_Commands_for_Programmable_Instruments

[7] JPA Consulting Ltd.: https://jpacsoft.com/scpi_commands.php

[8] EDI(14.11.2020): https://edistechlab.com/wie-funktioniert-uart/?v=3a52f3c22ed6

[9] Institut for Technical Optics: https://itom.bitbucket.io/v1-0-14/docs/06_extended_gui/qtdesigner.html

[10] stmicroelectronics. (08.2020): https://www.st.com/resource/en/user_manual/dm00105823-stm32-nucleo-64-boards-mb1136-stmicroelectronics.pdf

[11] ShawnHymel : https://www.digikey.de/en/maker/projects/getting-started-with-stm32-working-with-adc-and-dma/f5009db3a3ed4370acaf545a3370c30c

[12] Omar Ben Hamouda. (13.08.2022): *Konfiguration eines STM32-Mikrocontrollers als einstellbare Referenzspannungsquelle mit SCPI-Schnittstelle.*

[13] STMicroelectronics: Description of STM32L4/L4+ HAL and low-layer drivers: https://www.st.com/content/ccc/resource/technical/document/user_manual/63/a8/8f/e3/ca/a1/4c/84/DM00173145.pdf/files/DM00173145.pdf/jcr:content/translations/en.DM00173145.pdf

[14] The Qt Company Ltd.: https://doc.qt.io/qt-6/qtdesigner-manual.html

[15] Prof. Dr.-Ing.Otto Parzhuber Einführung in C für STM32 (15.11.2018): Kapitel 10: https://docplayer.org/106456811-Einfuehrung-in-c-fuer-stm32-einfuehrung-in-die-c-programmierung-fuer-mikrocontroller-kapitel-1-9-schwerpunkt-stm32-ab-kapitel-10.html

9. STM32-Board-Firmware

```
/* Includes ------------------------------------------------------------------*/
#include "main.h"
#include "stm32l4xx_hal.h"
#include "stm32l4xx_hal_adc.h"
#include "string.h"
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <ctype.h>



/* Private variables ---------------------------------------------------------*/
#define RxBuf_SIZE   64                                          //!< Größe des Rx-Puffers,
der zum Empfang von Daten über UART verwendet wird.
#define MainBuf_SIZE 64                    //!< Größe des Hauptpuffers, der zum Speichern
der über UART empfangenen Daten verwendet wird.
#define commandBuf_SIZE 64

uint8_t RxBuf[RxBuf_SIZE];                 //!< Rx-Puffer für den Empfang von Daten über
UART.
uint8_t MainBuf[MainBuf_SIZE];                          //!< Hauptpuffer, der zum Spei-
chern der über UART empfangenen Daten verwendet wird.
uint16_t adc_value;
uint8_t maxRes=12;
uint8_t minRes=6;

int maxAp=24.5*256;  //Vordefinierte Maximalapertur, hier bezogen auf eine Samplingzeit von
24,5 Zyklen und maximale Ratio
int minAp=24.5*2;   //Vordefinierte Minimalperture
int currentAp;
int division=4095;   //Initialisierung der Division für einen 12-Bit-ADC, die zur Umrech-
nung des ADC-Wertes in Volt verwendet wird.

ADC_ChannelConfTypeDef sConfig = {0};
//Instanz der Struktur ADC_ChannelConfTypeDef für die Konfiguration von ADC-Kanalparametern
erstellen

ADC_HandleTypeDef hadc1;
DMA_HandleTypeDef hdma_adc1;
UART_HandleTypeDef huart2;
DMA_HandleTypeDef hdma_usart2_rx;


uint8_t commandBuf[commandBuf_SIZE];                        //In diesem Puffer wird der
empfangene Befehl gespeichert.
bool newCommandReceived = false;                            //Dieses Flag wird verwendet,
um zu prüfen, ob der neue Befehl empfangen wurde.
static uint16_t MainBufCounter ;           //!< Statische Variable, die den aktuellen Ein-
trag des Main-Buffers enthält.



/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_DMA_Init(void);
static void MX_USART2_UART_Init(void);
static void MX_ADC1_Init(void);
/* USER CODE BEGIN PFP */
```

```c
void HAL_UARTEx_RxEventCallback(UART_HandleTypeDef *huart, uint16_t Size)
{
	uint8_t Counter = 0 ;	// Hilfsvariable zum Umkopieren von Daten aus dem RX-Puffer in den zentralen Puffer

	if(huart -> Instance == USART2 )   // Überprüfung, ob der Interrupt von USART2 stammt, da wir aktuell USART 2 einsetzen
	{

		while(Counter <= Size)  // Durchlauf zur Übertragung der Daten vom Rx-Puffer in den zentralen Puffer
		{
			//Prüfen, ob der Hauptpuffer voll mit Daten ist. In diesem Fall werden die Daten von Anfang an ersetzt.
			if(MainBufCounter > 64 )
		    {
				//Zurücksetzen des Hauptpufferzählers
			MainBufCounter = 0;
		    }
			MainBuf[MainBufCounter] = RxBuf[Counter++] ; // Übertragung der Daten vom Rx-Puffer in den zentralen Puffer
			if(MainBuf[MainBufCounter] == '\n')     // Überprüfung auf das Zeichen \n, das signalisiert, dass der vollständige Befehl eingegangen ist
			{for(int i = 0 ; i < (MainBufCounter); i++)  // Einfügen des empfangenen Befehls in den Befehlspuffer
				{
					commandBuf[i] =MainBuf[i];
				}
				newCommandReceived = true;  //Statusflag, das signalisiert, dass ein neuer Befehl eingegangen ist
			}

			if(Counter - 1 != Size)    // Sicherstellen, dass nach jeder Datenkopie kein Nullzeichen hinzugefügt wird
			{
				//Inkrementieren des Hauptpufferzählers
				MainBufCounter++;
			}
		}
		//Diese Funktion wird verwendet, um Daten von UART mit DMA zu empfangen, bis der Datenpuffer voll ist oder die IDLE Line erkannt wird.
		HAL_UARTEx_ReceiveToIdle_DMA(&huart2, RxBuf, RxBuf_SIZE);
		//Deaktivieren Sie den Interrupt für die halb übertragenen Daten.
		__HAL_DMA_DISABLE_IT(&hdma_usart2_rx, DMA_IT_HT);
	}
}
/*

 * */



void Uprintf(char *str)
{
	//Funktion zur Übertragung von Daten auf UART.
	HAL_UART_Transmit(&huart2 ,(uint8_t*) str, strlen(str),HAL_MAX_DELAY);
}

char* lowercase(char* s)   // Diese Funktion wandelt alle Zeichen in einem String in Kleinbuchstaben um
{
  for(char *p=s; *p; p++) *p=tolower(*p);   // Durchläuft jeden Charakter im String und wandelt ihn in Kleinbuchstaben um
  return s;
}
```

```
const char* get_adc_channel_string(uint32_t channel) // Diese Funktion gibt einen String zurück, der dem übergebenen ADC-Kanal entspricht
{
      switch(channel) {    // Switch-Case-Struktur zur Auswahl des korrekten Kanalstrings basierend auf dem eingegebenen channel-Wert
          case ADC_CHANNEL_1:
              return "CHANNEL_1";
          case ADC_CHANNEL_2:
              return "CHANNEL_2";
          case ADC_CHANNEL_3:
              return "CHANNEL_3";
          case ADC_CHANNEL_4:
              return "CHANNEL_4";
          case ADC_CHANNEL_5:
              return "CHANNEL_5";
          case ADC_CHANNEL_11:
              return "CHANNEL_11";
          case ADC_CHANNEL_12:
              return "CHANNEL_12";
          case ADC_CHANNEL_13:
              return "CHANNEL_13";
          case ADC_CHANNEL_VREFINT:
              return "CHANNEL_VREFINT";
          default:
              return "Channel not selected";
      }
  }

void convertStringToADCChannel( char* str)   // Diese Funktion konvertiert einen String zu einem ADC-Kanal
{

      if (strcmp(str, "ADC_CHANNEL_1") == 0) {  // Verwendet eine if-else-if-Struktur, um den String
                                                //zu überprüfen und den entsprechenden ADC-Kanal zuzuweisen
          sConfig.Channel  = ADC_CHANNEL_1;
      } else if (strcmp(str, "ADC_CHANNEL_2") == 0) {
          sConfig.Channel  = ADC_CHANNEL_2;
      } else if (strcmp(str, "ADC_CHANNEL_3") == 0) {
          sConfig.Channel  = ADC_CHANNEL_3;
      } else if (strcmp(str, "ADC_CHANNEL_4") == 0) {
          sConfig.Channel  = ADC_CHANNEL_4;
      } else if (strcmp(str, "ADC_CHANNEL_5") == 0) {
          sConfig.Channel  = ADC_CHANNEL_5;
      } else if (strcmp(str, "ADC_CHANNEL_11") == 0) {
          sConfig.Channel  = ADC_CHANNEL_11;
      } else if (strcmp(str, "ADC_CHANNEL_12") == 0) {
          sConfig.Channel  = ADC_CHANNEL_12;
      } else if (strcmp(str, "ADC_CHANNEL_13") == 0) {
          sConfig.Channel  = ADC_CHANNEL_13;
      } else if (strcmp(str, "ADC_CHANNEL_VREFINT") == 0) {
          sConfig.Channel  = ADC_CHANNEL_VREFINT;
      }

          if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
                {
                  Error_Handler();
                }
        HAL_Delay(10);
  }


uint16_t extract_integer_from_command(const char* commandBuf, const char* command) {
       // Extrahieren der ganzzahligen Teilzeichenkette aus dem Befehl
```

```
        char integer_str[commandBuf_SIZE];
        strlcpy(integer_str, lowercase((char *)commandBuf) + strlen(command), command-
Buf_SIZE);
        // Umwandlung der String in eine Ganzzahl
        uint16_t value = atoi(integer_str);
    return value;
}

const char* get_adc_resolution_string(uint32_t resolution)  // Diese Funktion gibt einen
String zurück, der der übergebenen ADC-Auflösung entspricht
{
      switch(resolution) {   // Switch-Case-Struktur zur Auswahl des korrekten
Auflösungsstrings basierend auf dem eingegebenen resolution-Wert
          case ADC_RESOLUTION_6B:
              return "6 Bits";
          case ADC_RESOLUTION_8B:
              return "8 Bits";
          case ADC_RESOLUTION_10B:
              return "10 Bits";
          case ADC_RESOLUTION_12B:
              return "12 Bits";


          default:
              return "Unbekannt";
      }
  }
void setADCResoltuion( uint8_t resolution)  // Diese Funktion stellt die adc-Auflösung ein,
abhängig von dem angegebenen Parameter
{

    switch(resolution) {   // Verwendet eine Switch-Case-Struktur, um die eingegebene
Auflösung zu überprüfen und die entsprechende ADC-Auflösung zuzuweisen
        case 6:
         hadc1.Init.Resolution  = ADC_RESOLUTION_6B;
        division=63;   //(2^6)-1 Division ist der Parameter, um den ADC-Wert als Volt in
Abhängigkeit von der Auflösung zu erhalten
       break;

        case 8:
         hadc1.Init.Resolution  = ADC_RESOLUTION_8B;
        division=255;
       break;

        case 10:
         hadc1.Init.Resolution  = ADC_RESOLUTION_10B;
        division=1023;
       break;

        case 12:
         hadc1.Init.Resolution  = ADC_RESOLUTION_12B;
        division=4095;
       break;
    }

         if (HAL_ADC_Init(&hadc1) != HAL_OK)
         {
           Error_Handler();
         }

         HAL_Delay(10);
}
int getRatio(uint32_t ratio)  // Diese Funktion gibt das Verhältnis einer gegebenen ADC-
Überabtastungsrate als Zahl zurück
{
```

```
      switch(ratio) {  // Switch-Case-Struktur zur Auswahl des korrekten Verhältnisses ba-
sierend auf dem eingegebenen ratio-Wert
          case ADC_OVERSAMPLING_RATIO_2:
              return 2;
          case ADC_OVERSAMPLING_RATIO_4:
              return 4;
          case ADC_OVERSAMPLING_RATIO_8:
              return 8;
          case ADC_OVERSAMPLING_RATIO_16:
              return 16;
          case ADC_OVERSAMPLING_RATIO_32:
              return 32;
          case ADC_OVERSAMPLING_RATIO_64:
              return 64;
          case ADC_OVERSAMPLING_RATIO_128:
              return 128;
          case ADC_OVERSAMPLING_RATIO_256:
              return 256;

          default:
              return 0;
      }
  }

void setADCAperture( uint16_t aperture)  // Diese Funktion setzt die Apertur des ADC basie-
rend auf einer gegebenen Apertur
{

    switch(aperture) {// Verwendet eine Switch-Case-Struktur, um die eingegebene Apertur zu
überprüfen und die entsprechende ADC-Apertur zuzuweisen
 // Aperture basierend auf Oversampling Time von 24.4 Cycles
        case 49:
         hadc1.Init.Oversampling.Ratio = ADC_OVERSAMPLING_RATIO_2;
         hadc1.Init.Oversampling.RightBitShift = ADC_RIGHTBITSHIFT_1;
       break;

        case 98:
         hadc1.Init.Oversampling.Ratio = ADC_OVERSAMPLING_RATIO_4;
         hadc1.Init.Oversampling.RightBitShift = ADC_RIGHTBITSHIFT_2;
       break;

        case 196:
         hadc1.Init.Oversampling.Ratio = ADC_OVERSAMPLING_RATIO_8;
         hadc1.Init.Oversampling.RightBitShift = ADC_RIGHTBITSHIFT_3;
       break;

        case 392:
         hadc1.Init.Oversampling.Ratio = ADC_OVERSAMPLING_RATIO_16;
         hadc1.Init.Oversampling.RightBitShift = ADC_RIGHTBITSHIFT_4;
       break;

        case 784:
         hadc1.Init.Oversampling.Ratio = ADC_OVERSAMPLING_RATIO_32;
         hadc1.Init.Oversampling.RightBitShift = ADC_RIGHTBITSHIFT_5;
       break;

        case 1568:
         hadc1.Init.Oversampling.Ratio = ADC_OVERSAMPLING_RATIO_64;
         hadc1.Init.Oversampling.RightBitShift = ADC_RIGHTBITSHIFT_6;
       break;

        case 3136:
         hadc1.Init.Oversampling.Ratio = ADC_OVERSAMPLING_RATIO_128;
         hadc1.Init.Oversampling.RightBitShift = ADC_RIGHTBITSHIFT_7;
       break;
```

```
        case 6272:
         hadc1.Init.Oversampling.Ratio = ADC_OVERSAMPLING_RATIO_256;
         hadc1.Init.Oversampling.RightBitShift = ADC_RIGHTBITSHIFT_8;
        break;
    }

         if (HAL_ADC_Init(&hadc1) != HAL_OK)
          {
            Error_Handler();
          }

         HAL_Delay(10);
}
int checkCommandType(char* commandBuf)  // Diese Funktion überprüft den Befehlstyp eines gegebenen Befehls
{

       // Verwendet eine Switch-Case-Struktur, um das erste Zeichen des Befehls zu prüfen und dann den einzelnen Befehl zu bestimmen.

    switch (commandBuf[0]) {
    case 'm':
        if (strncmp(commandBuf, "measure:voltage?", 16) == 0 || strncmp(commandBuf, "meas:volt?", 10) == 0) {
            return 1;
        }
        break;
    case 's':
        if (strncmp(commandBuf, "sense:channel?", 14) == 0 || strncmp(commandBuf, "sens:chan?", 10) == 0) {
            return 2;

        } else if (strncmp(commandBuf, "sense:channel ", 14) == 0|| strncmp(commandBuf, "sens:chan ", 10) == 0) {
            return 3;

        } else if (strncmp(commandBuf, "sense:resolution? max", 21) == 0 || strncmp(commandBuf, "sens:reso? max", 14) == 0) {
            return 4;

        } else if (strncmp(commandBuf, "sense:resolution? min", 21) == 0 || strncmp(commandBuf, "sens:reso? min", 14) == 0) {
            return 5;

        } else if (strncmp(commandBuf, "sense:resolution?", 17) == 0 || strncmp(commandBuf, "sens:reso?", 10) == 0) {
            return 6;

        } else if (strncmp(commandBuf, "sense:resolution ", 17) == 0|| strncmp(commandBuf, "sens:reso ", 10) == 0) {
            return 7;

        } else if (strncmp(commandBuf, "sense:aperture? max", 19) == 0 || strncmp(commandBuf, "sens:aper? max", 14) == 0) {
            return 8;

        } else if (strncmp(commandBuf, "sense:aperture? min", 19) == 0 || strncmp(commandBuf, "sens:aper? min", 14) == 0) {
            return 9;

        } else if (strncmp(commandBuf, "sense:aperture?", 15) == 0 || strncmp(commandBuf, "sens:aper?", 10) == 0) {
            return 10;

        } else if (strncmp(commandBuf, "sense:aperture ", 15) == 0 || strncmp(commandBuf, "sens:aper ", 10)== 0){
```

```
            return 11;
        }
        break;
        case 'e':
            if (strncmp(commandBuf, "enable_oversampling", 19) == 0) {
                return 12;
            }
            break;
        case 'd':
            if (strncmp(commandBuf, "disable_oversampling", 20) == 0) {
                return 13;
            }
            break;
    }


    return 0;

}
int main(void)
{

  HAL_Init();

  /* Configure the system clock */
  SystemClock_Config();


  hadc1.Init.Resolution = ADC_RESOLUTION_12B;    //HAL-Funktion zur Initialisierung der
ADC-Auflösung,
  hadc1.Init.OversamplingMode = DISABLE;          // Oversampling deaktiviert als
Startpunkt
  hadc1.Init.Oversampling.Ratio = ADC_OVERSAMPLING_RATIO_256;   // Oversampling Ratio
  hadc1.Init.Oversampling.RightBitShift = ADC_RIGHTBITSHIFT_8;   //Oversampling
RightBitshift, abhängig von Ratio

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_DMA_Init();
  MX_USART2_UART_Init();
  MX_ADC1_Init();
  /* USER CODE BEGIN 2 */
  HAL_UARTEx_ReceiveToIdle_DMA(&huart2, RxBuf, RxBuf_SIZE);
  //Diese Funktion wird zum Empfangen von Daten vom UART mit DMA verwendet, bis der Daten-
puffer voll ist oder eine Leerzeile erkannt wird.
  __HAL_DMA_DISABLE_IT(&hdma_usart2_rx, DMA_IT_HT);
  //Deaktivieren des Interrupts



  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {

        if(newCommandReceived == true)                  //Prüfen, ob der neue Befehl
empfangen wurde.
                                    {

                char uart_buffer[50]; // Puffer für UART-Übertragung

                switch (checkCommandType(lowercase((char *)commandBuf))) // Befehlstyp
prüfen
                {
```

```
                case 1: // measure:voltage?


                    HAL_ADCEx_Calibration_Start(&hadc1, ADC_SINGLE_ENDED);  // ADC-
Kalibrierung starten
                    HAL_ADC_Start_DMA(&hadc1, (uint32_t *)&adc_value, 1);  //Startet den
ADC-DMA-Modus für den ADC "hadc1"
                    HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);   //Wartet, bis
die ADC-Konvertierung abgeschlossen ist.
                    HAL_Delay(100);  //Eine Pause von 100 Millisekunden, um sicherzus-
tellen, dass der ADC-Wert stabil ist.

                                float voltage = (3.3f * adc_value) / (division) ; // ADC-
Wert in Spannung umrechnen


                                sprintf(uart_buffer, "Voltage: %.3f V", voltage);
//Schreibt den Spannungswert formatiert in den uart_buffer, um ihn später über UART zu
übertragen.



                    break;
                case 2:
                                sprintf(uart_buffer, "Current Channel: %s",
get_adc_channel_string(sConfig.Channel));

                                //Schreibt den formatierten Kanalnamen in den
uart_buffer zur späteren Übertragung über UART.
                    break;
                case 3:


                                                if(strcmp(lowercase((char *)commandBuf),
"sense:channel vrefint")==0||strcmp(lowercase((char *)commandBuf), "sens:chan vrefint")==0)
                                                {
                                                        //Überprüft, ob der empfangene
Befehl "sense:channel vrefint" oder "sens:chan vrefint" ist.

                                                convertStringToADCChan-
nel("ADC_CHANNEL_VREFINT"); //ADC-Kanal auf "ADC_CHANNEL_VREFINT" setzen.

                                                }

                                                else if(strncmp(lowercase((char
*)commandBuf), "sens:chan ", 10) == 0)
                                                {
                                                uint16_t channel= ex-
tract_integer_from_command(((char *)commandBuf),"sens:chan ");
                                                // Extrahiert die Kanalnummer aus dem
Befehl "sense:channel x" und speichert sie in der Variable channel.
                                            char adc_channel_temp[20];
                                                snprintf(adc_channel_temp,
sizeof(adc_channel_temp), "ADC_CHANNEL_%u", channel);
                                                //Erstellt einen temporären Puffer
adc_channel_temp und formatiert den Kanalwert darin.
                                                convertStringToADCChannel(adc_channel_temp);
//ADC-Kanal entsprechend adc_channel_temp  setzen.
                                                }
                                                else {

                                                uint16_t channel= ex-
tract_integer_from_command(((char *)commandBuf),"sense:channel ");
                                // Extrahiert die Kanalnummer aus dem Befehl "sense:channel
x" und speichert sie in der Variable channel.
                                            char adc_channel_temp[20];
```

```
                                                snprintf(adc_channel_temp,
sizeof(adc_channel_temp), "ADC_CHANNEL_%u", channel);
                                                //Erstellt einen temporären Puffer
adc_channel_temp und formatiert den Kanalwert darin.
                                                convertStringToADCChannel(adc_channel_temp);
//ADC-Kanal entsprechend adc_channel_temp  setzen.

                                                    }

                                                    sprintf(uart_buffer, "Selected Channel:
%s", get_adc_channel_string(sConfig.Channel));
                                                    // Schreibt den ausgewählten ADC-Kanal in
den uart_buffer für die spätere Übertragung über UART.


                                                break;

                    case 4: // Maximale Auflösung erhalten
                            sprintf(uart_buffer, "Max ADC resolution: %d bits", maxRes);
//Schreibt die maximale Auflösung des ADCs in den uart_buffer.


                        break;
                    case 5: // Mindestauflösung erhalten

                            sprintf(uart_buffer, "Min ADC resolution: %d bits", minRes);
//Schreibt die minimale Auflösung des ADCs in den uart_buffer.

                        break;


                    case 6: // aktuelle Auflösung erhalten
                    {

                        sprintf(uart_buffer, "Current resolution: %s ",
get_adc_resolution_string(hadc1.Init.Resolution));
                        //Schreibt die aktuelle Auflösung des ADCs in den uart_buffer unter
Verwendung der Funktion get_adc_resolution_string.


                        break;
                    }

                    case 7:  //Set Resolution
                    {
                       if(strncmp(lowercase((char *)commandBuf), "sense:resolution
min",20)==0 || strncmp(lowercase((char *)commandBuf), "sens:reso min",13)==0 ) // Befeh
Überprüfung
                        {
                                setADCResoltuion(minRes);  //ADC-Auflösung entsprechend der
angegebenen Auflösung setzen, hier Min Res

                        }
                        else if(strncmp(lowercase((char *)commandBuf), "sense:resolution
max",20)==0 || strncmp(lowercase((char *)commandBuf), "sens:reso max",13)==0)
                                {
                                setADCResoltuion(maxRes); // ADC-Auflösung entsprechend der
angegebenen Auflösung setzen, hier Max Res

                                                    }
                        else if (strncmp(lowercase((char *)commandBuf), "sens:reso
",10)==0){

                                uint16_t resolution = extract_integer_from_command(((char
*)commandBuf),"sens:reso ");
```

```
                            //Extrahiert die Auflösungszahl aus dem Befehl "sens:reso x"
und speichert sie in der Variable resolution.
                             setADCResoltuion(resolution);
                        }
                        else{

                            uint16_t resolution = extract_integer_from_command(((char
*)commandBuf),"sense:resolution ");
                             setADCResoltuion(resolution);
                        }


                         sprintf(uart_buffer, "Selected resolution: %s ",
get_adc_resolution_string(hadc1.Init.Resolution));
                        break;
                    }

                    case 8: // maximale Apertur erhalten

                        sprintf(uart_buffer, "Max Aperture: %d Cycles", maxAp);

                    break;
                    case 9: // minimale Apertur erhalten

                        sprintf(uart_buffer, "Min Aperture: %d Cycles", minAp);
                    break;

                    case 10: // aktuelle Apertur ermitteln

                       currentAp=24.5*getRatio(hadc1.Init.Oversampling.Ratio); // Be-
rechnet die aktuelle Apertur des ADCs basierend auf dem Oversampling-Verhältnis
                        sprintf(uart_buffer, "Current Aperture: %d Cycles", currentAp);
                    break;

                    case 11: // Apertur einstellen


                       if(strncmp(lowercase((char *)commandBuf), "sense:aperture
min",18)==0 || strncmp(lowercase((char *)commandBuf), "sens:aper min",13)==0)
                        {
                            setADCAperture(minAp);

                        }
                        else if(strncmp(lowercase((char *)commandBuf), "sense:aperture
max",18)==0 || strncmp(lowercase((char *)commandBuf), "sens:aper max",13)==0)
                            {
                            setADCAperture(maxAp);


                            }else if(strncmp(lowercase((char *)commandBuf), "sens:aper
",10)==0) {

                                uint16_t aperture = ex-
tract_integer_from_command(((char *)commandBuf),"sens:aper ");
                                //Extrahiert den Aperturwert aus dem Befehl
"sens:aper x" und speichert ihn in der Variable aperture.
                            setADCAperture(aperture);
                            //die Apertur des ADCs entsprechend dem angegebenen Wert
setzen
                                                                }
                        else{

                            uint16_t aperture = extract_integer_from_command(((char
*)commandBuf),"sense:aperture ");

                            setADCAperture(aperture);
```

```
                                                    }
                        int aperture=24.5*getRatio(hadc1.Init.Oversampling.Ratio);
                        sprintf(uart_buffer, "Selected aperture: %d Cycles",aperture);

                         break;


                    case 12: // Enable Oversampling

                             hadc1.Init.OversamplingMode = ENABLE; //Aktiviert das Oversam-
pling für den ADC.
                             HAL_ADC_Init(&hadc1);  //nitialisiert den ADC nach der Aktivie-
rung des Oversamplings.
                             sprintf(uart_buffer, "Oversampling enabled");
                        break;
                    case 13: // Disable Oversampling

                            hadc1.Init.OversamplingMode = DISABLE; ////Deaktiviert das Over-
sampling für den ADC.
                            HAL_ADC_Init(&hadc1);
                               sprintf(uart_buffer, "Oversampling disabled");
                        break;


                }//switch end


                Uprintf(uart_buffer);  // Überträgt den Inhalt von uart_buffer über UART.
                memset(uart_buffer, 0, sizeof(uart_buffer)); //Löscht den Inhalt von
uart_buffer, indem alle Bytes auf den Wert 0 gesetzt werden.
          newCommandReceived = false;            //Setzt das Flag newCommandReceived
zurück, um anzuzeigen, dass kein neuer Befehl empfangen wurde.
          memset(commandBuf , 0 ,commandBuf_SIZE*(sizeof(commandBuf[0]))); //Löscht den
Inhalt von commandBuf, indem alle Bytes auf den Wert 0 gesetzt werden. Dadurch wird der
Befehlspuffer zurückgesetzt.
          MainBufCounter = 0 ; // Setzt den Zähler MainBufCounter auf 0, um den Hauptspei-
cherzähler zurückzusetzen.
          memset(MainBuf , 0 ,MainBuf_SIZE*(sizeof(MainBuf[0]))); //Löscht den Inhalt von
MainBuf, indem alle Bytes auf den Wert 0 gesetzt werden.
}
}
  }
/**
  * @brief System Clock Configuration
  * @retval None
  */
void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

  /** Configure the main internal regulator output voltage
  */
  if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
  {
    Error_Handler();
```

```
  }

  /** Initializes the RCC Oscillators according to the specified parameters
  * in the RCC_OscInitTypeDef structure.
  */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
  RCC_OscInitStruct.HSIState = RCC_HSI_ON;
  RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
  RCC_OscInitStruct.PLL.PLLM = 1;
  RCC_OscInitStruct.PLL.PLLN = 10;
  RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
  RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
  RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }

  /** Initializes the CPU, AHB and APB buses clocks
  */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;
  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
  {
    Error_Handler();
  }
}

/**
  * @brief ADC1 Initialization Function
  * @param None
  * @retval None
  */
static void MX_ADC1_Init(void)
{

  /* USER CODE BEGIN ADC1_Init 0 */

  /* USER CODE END ADC1_Init 0 */

  ADC_MultiModeTypeDef multimode = {0};


  /* USER CODE BEGIN ADC1_Init 1 */
  /* USER CODE END ADC1_Init 1 */

  /** Common config
  */
  hadc1.Instance = ADC1;
  hadc1.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;

  hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
  hadc1.Init.ScanConvMode = ADC_SCAN_DISABLE;
  hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
  hadc1.Init.LowPowerAutoWait = DISABLE;
  hadc1.Init.ContinuousConvMode = DISABLE;
  hadc1.Init.NbrOfConversion = 1;
  hadc1.Init.DiscontinuousConvMode = DISABLE;
  hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
  hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
```

```
  hadc1.Init.DMAContinuousRequests = ENABLE;
  hadc1.Init.Overrun = ADC_OVR_DATA_PRESERVED;
  hadc1.Init.Oversampling.TriggeredMode = ADC_TRIGGEREDMODE_SINGLE_TRIGGER;
  hadc1.Init.Oversampling.OversamplingStopReset = ADC_REGOVERSAMPLING_RESUMED_MODE;
  if (HAL_ADC_Init(&hadc1) != HAL_OK)
  {
    Error_Handler();
  }

  /** Configure the ADC multi-mode
  */
  multimode.Mode = ADC_MODE_INDEPENDENT;
  if (HAL_ADCEx_MultiModeConfigChannel(&hadc1, &multimode) != HAL_OK)
  {
    Error_Handler();
  }

  /* Configure Regular Channel*/

  sConfig.Rank = ADC_REGULAR_RANK_1;
  sConfig.SamplingTime = ADC_SAMPLETIME_24CYCLES_5;
  sConfig.SingleDiff = ADC_SINGLE_ENDED;
  sConfig.OffsetNumber = ADC_OFFSET_NONE;
  sConfig.Offset = 0;
  if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
  {
    Error_Handler();
  }

  /* USER CODE BEGIN ADC1_Init 2 */

  /* USER CODE END ADC1_Init 2 */

}

/**
  * @brief USART2 Initialization Function
  * @param None
  * @retval None
  */
static void MX_USART2_UART_Init(void)
{

  /* USER CODE BEGIN USART2_Init 0 */

  /* USER CODE END USART2_Init 0 */

  /* USER CODE BEGIN USART2_Init 1 */

  /* USER CODE END USART2_Init 1 */
  huart2.Instance = USART2;
  huart2.Init.BaudRate = 115200;
  huart2.Init.WordLength = UART_WORDLENGTH_8B;
  huart2.Init.StopBits = UART_STOPBITS_1;
  huart2.Init.Parity = UART_PARITY_NONE;
  huart2.Init.Mode = UART_MODE_TX_RX;
  huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart2.Init.OverSampling = UART_OVERSAMPLING_16;
  huart2.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
  huart2.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
  if (HAL_UART_Init(&huart2) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN USART2_Init 2 */
```

```
  /* USER CODE END USART2_Init 2 */

}

/**
  * Enable DMA controller clock
  */
static void MX_DMA_Init(void)
{

  /* DMA controller clock enable */
  __HAL_RCC_DMA1_CLK_ENABLE();

  /* DMA interrupt init */
  /* DMA1_Channel1_IRQn interrupt configuration */
  HAL_NVIC_SetPriority(DMA1_Channel1_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(DMA1_Channel1_IRQn);
  /* DMA1_Channel6_IRQn interrupt configuration */
  HAL_NVIC_SetPriority(DMA1_Channel6_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(DMA1_Channel6_IRQn);

}

/**
  * @brief GPIO Initialization Function
  * @param None
  * @retval None
  */
static void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_GPIOH_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_GPIOB_CLK_ENABLE();

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);

  /*Configure GPIO pin : B1_Pin */
  GPIO_InitStruct.Pin = B1_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
  GPIO_InitStruct.Pull = GPIO_PULLUP;
  HAL_GPIO_Init(B1_GPIO_Port, &GPIO_InitStruct);

  /*Configure GPIO pin : LD2_Pin */
  GPIO_InitStruct.Pin = LD2_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(LD2_GPIO_Port, &GPIO_InitStruct);

}

/* USER CODE BEGIN 4 */

/* USER CODE END 4 */

/**
  * @brief  This function is executed in case of error occurrence.
  * @retval None
  */
void Error_Handler(void)
{
```

```
  /* USER CODE BEGIN Error_Handler_Debug */
  /* User can add his own implementation to report the HAL error return state */
  __disable_irq();
  while (1)
  {
  }
  /* USER CODE END Error_Handler_Debug */
}

#ifdef  USE_FULL_ASSERT
/**
  * @brief  Reports the name of the source file and the source line number
  *         where the assert_param error has occurred.
  * @param  file: pointer to the source file name
  * @param  line: assert_param error line source number
  * @retval None
  */
void assert_failed(uint8_t *file, uint32_t line)
{
  /* USER CODE BEGIN 6 */
  /* User can add his own implementation to report the file name and line number,
     ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
  /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */


/************************ (C) COPYRIGHT STMicroelectronics *****END OF FILE****/
```