

Echtzeitfähige Firmware für einen Motion-Controller mit CANopen Anbindung in C++

von

Aaron Pfeffer (pfaa 1011)

Lars Pföhler (pfla 1011)

Niruthan Seshendran (seni 1017)

Betreuer: Prof. Dr.-Ing. Tobias Baas

Koordinator: Prof. Dr.-Ing. Robert Weiß

Wintersemester 2023/24

Modul FZTB610 Entwicklungsprojekt

Projektcode: 23ws_BA_ EMoFi

19.04.2024

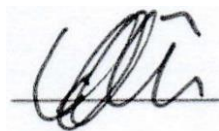
Erklärung

Hiermit erklären wir, dass wir die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt haben. Die Stellen des Berichtes, die andere Quellen im Wortlaut oder dem Sinn nach entnommen wurden, sind durch Angaben der Herkunft kenntlich gemacht. Dies gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen sowie für Quellen aus dem Internet.

Karlsruhe, den 19.04.2024



Aaron Pfeffer



Lars Pföhler



Niruthan Seshendran

Inhaltsverzeichnis

Abkürzungsverzeichnis.....	I
Abbildungsverzeichnis.....	II
1. Motivation.....	1
2. Aufgabenstellung.....	1
3. Vorgehensweise.....	2
3.1. Hardware.....	2
3.2. STM32 Cube IDE	7
3.3. Firmwarekonzept	8
3.4. Umsetzung der Module.....	14
3.4.1. Trajektoriengenerator.....	14
3.4.2. Strom- und Lageregelung	17
3.4.3. Output-Matrix	21
3.4.4. CANopen	23
3.4.5. Kommando-Interpreter	27
4. Test und Validierung.....	30
4.1. Funktionale Validierung	30
4.2. Performancetest	32
4.3. Kompatibilitätstest.....	35
5. Zusammenfassung und Ausblick	36
Literaturverzeichnis	37
Anhang.....	38
Klassendiagramm.....	38
Befehlsliste des Kommando-Interpreters	39

Abkürzungsverzeichnis

ADC	Analog Digital Converter
BRS	Bit Rate Switch
CAN	Controller Area Network
CPU	Central Processing Unit
DC-Motor	Direct Current-Motor
DMA	Direct Memory Access
FreeRTOS	Real-time operation system
GPIO	General Purpose Input Output
ID	Identification
IDE	Integrated Development Environment
MOSFET	metal-oxide-semiconductor field-effect transistor
OD	Object Dictionary
PWM	Pulsweitenmodulation
Shunt	Messwiderstand
RAM	Random Access Memory
UART	Universal Asynchronous Receiver Transmitter

Abbildungsverzeichnis

Abb. 1: STM32G474RET6[2]	2
Abb. 2: Pinout[2]	2
Abb. 3: Vollbrückenschaltung zur Ansteuerung eines Motors[3]	3
Abb. 4: Änderung der Drehrichtung durch unterschiedliche Beschaltung der MOSFETs[3]	3
Abb. 5: X-NUCLEO-IHM07M1 Motor-Treiberstufe[4]	4
Abb. 6: Micos PRS110, Rotationsmotor[5]	5
Abb. 7: Oszilloskopmessung der Lagesensorik bei konstanter Drehgeschwindigkeit.....	5
Abb. 8: Eingelötete Widerstände an Sensorleitungen	6
Abb. 9: Graphische Oberfläche in ioc-File	7
Abb. 10: BlackBox-Darstellung der Motion-Controller-Firmware	8
Abb. 11: Module der Motion-Controller-Firmware	8
Abb. 12: Komponenten des Systems, rot=echtzeitfähig[6].....	9
Abb. 13: Interface-Klasse "IMotorBase"	10
Abb. 14: Programmablaufplan main-Routine	11
Abb. 15: Programmablaufplan Timer-Callback-Funktion	12
Abb. 16: Trapezprofil für Geschwindigkeit[10]	14
Abb. 17: Programm-Code zur Trapez-Funktion der Klasse "TrajectoryGenerator"	15
Abb. 18: Schematischer Aufbau der Schaltung incl. shunts[4]	17
Abb. 19: Messung des ADC-Signals (gelb) und Shuntspannung (rot).....	18
Abb. 20: Messung von PWM-Ausgang (gelb) und Shuntspannung (rot)	19
Abb. 21: Messung von ADC-Konvertierung und PWM-Signalen	19
Abb. 22: Funktion für Lageregelung in PID-Klasse	20
Abb. 23: Funktionsprinzip der Output-Matrix.....	21
Abb. 24: Übersicht aller Funktionen in Output-Matrix-Klasse	22
Abb. 25: Beispielkonfiguration eines POD mit CANopenEditor	24
Abb. 26: Messung des PDO mit Oszilloskop.....	25
Abb. 27: Messung der Zykluszeit der PDO mit Oszilloskop	26
Abb. 28: Beispiel zur Veranschaulichung der Help-Funktion	28
Abb. 29: Eingabe zum Setzen und Lesen eines Reglerparameters	28
Abb. 30: Konfigurierter ioc-File	29
Abb. 31: Jitter-Messung über eine Periode.....	32
Abb. 32: Detaillierter Zoom der Jitter-Messung bei 50 ns/Div	32
Abb. 33: Stromregelung ohne Optimierungen.....	33
Abb. 34: Stromregelung im Release-Mode	33
Abb. 35: Prototyp für Aufruf einer Funktion aus RAM.....	34
Abb. 36: Stromregelung im Release-Mode nach Anpassung der Prioritäten	34

1. Motivation

Im Labor für informationstechnische Systeme soll zu einem eigens entwickelten Motion-Controller eine Echtzeitfirmware entstehen. Der Motion-Controller ist Teil eines Projektes, in dem eine Modellfertigungsstraße mit Roboterarmen entstehen soll. Der Motion-Controller soll die in den Roboterarmen verwendeten Elektromotoren ansteuern und die Position durch das Messen eines Positionssensors regeln. Die Modellfertigungsstraße soll in verschiedenen Lehrveranstaltungen als angewandtes Lehrmittel sowie in der Forschung dienen [1].

2. Aufgabenstellung

Für den Motion-Controller ist ein Firmwarekonzept zu entwickeln, welches eine Echtzeitsteuerung von Elektromotoren mit Strom- und Lageregelung ermöglicht. Das so entstehende sogenannte „Drive“ soll über einen CAN (Controller Area Network) -Bus mit CANopen Protokoll ansteuerbar sein. Weiterhin soll es einen Kommandointerpreter geben, der die Parametrisierung und Steuerung des Drives über USB und Serielle Datenverbindung ermöglicht. Die Servo-Bandbreite sollte 20 kHz betragen, wobei die Stromregelung auch bei 100 kHz liegen darf.[1]

Jeder Controller soll zwei getrennte Motor-Sensorkombinationen regeln können. Die Verknüpfung der einzelnen Regelkreise wird überlagert in einer separaten Steuerung realisiert, so dass der Drive über CANopen die Echtzeitdaten für jeden Regelkreis separat empfängt. [1]

Zu den Zielen der Projektarbeit gehören die Einarbeitung in die Tools der Firma „ST“ zum verwendeten STM32G4 Signalprozessor, die Recherche zu Möglichkeiten einer echtzeitfähigen Firmware für Steuer- und Regelungsaufgaben, die Erstellung eines Firmwarekonzeptes zur echtzeitfähigen Regelung, die Kapselung der Regelung, Ansteuerung und Echtzeitdatenverarbeitung, die Entwicklung eines Kommandointerpreters mit ersten Befehlen zum Kommandieren der Steuerung, sowie Test und Validierung des Konzeptes. [1]

3. Vorgehensweise

3.1. Hardware

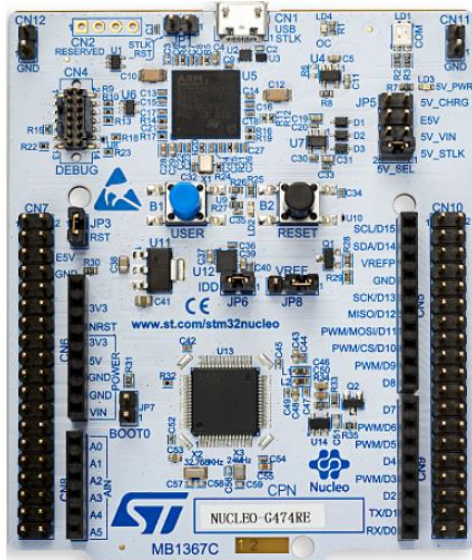


Abb. 1: STM32G474RET6[2]

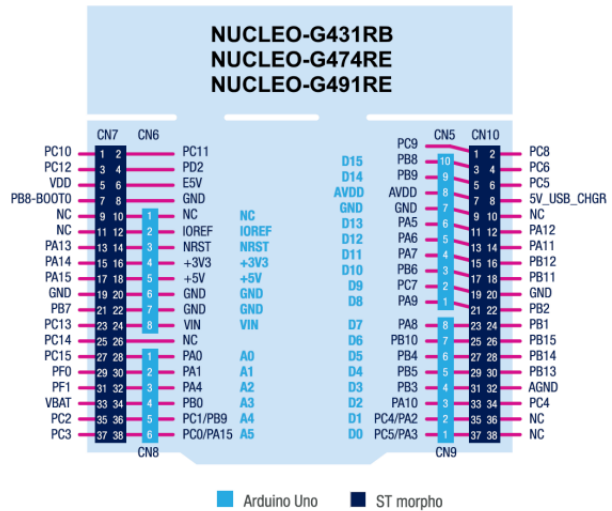


Abb. 2: Pinout[2]

Für die Projektarbeit soll die Firmware zunächst auf einem Nucleo-G474RE-Board (Abb. 1) mit einem STM32G474RET6 als Prozessor betrieben werden. Elemente des Boards, welche für diese Projektarbeit relevant sind, sind z.B. die ADC (Analog Digital Converter) -Schnittstellen, Timer, UART (Universal Asynchronous Receiver Transmitter) und CAN. Diese Komponenten werden unter anderem für das Lesen der Motorströme, die Realisierung einer echtzeitfähigen Regelung und die Kommunikation zu anderen Applikationen und Nutzern verwendet.

Bei den anzusteuernenden Motoren handelt es sich zunächst nur um DC (Direct Current) -Motoren. Insgesamt sollen zwei Stück in der späteren Anwendung geregelt werden. Anhand von Abb. 3 wird das grundlegende Konzept hinter der Bestromung eines DC-Motors veranschaulicht.

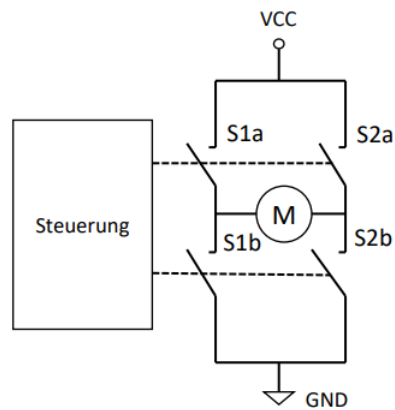


Abb. 3: Vollbrückenschaltung zur Ansteuerung eines Motors[3]

Die im Bild gezeigten Schalter repräsentieren MOSFETs (metal-oxide-semiconductor field-effect transistor), welche ab einer bestimmten Gate-Spannung den Stromfluss zwischen Source und Drain ermöglichen. Dabei wird die Gate-Spannung durch ein PWM (Pulsweitenmodulation)-Signal bereitgestellt. Die Verwendung eines PWM-Signals bietet den Vorteil einer schnellen und flexiblen Anpassung der Gate-Spannung, was im betrachteten System von besonderer Relevanz ist[3].

Grundsätzlich wird bei der Steuerung des DC-Motors zum einen die Stromstärke und zum anderen die Drehrichtung des Motors angepasst. Letzteres wird durch das Schalten von jeweils zwei überkreuzten MOSFETs erreicht. Wenn S1a und S2b schalten, fließt der Strom durch den Motor in eine Richtung. Entsprechend fließt der Strom bei umgekehrter Beschaltung entgegengesetzt und somit lässt sich die Drehrichtung des Motors einstellen (vgl. Abb. 4).

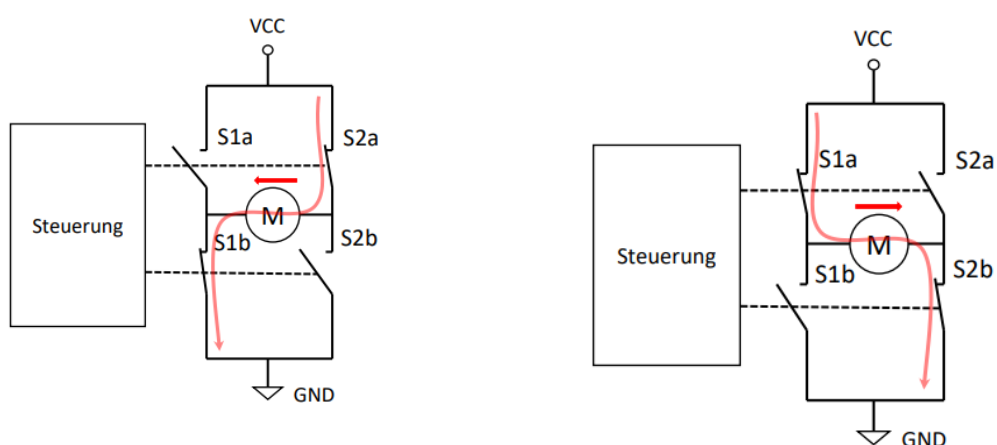


Abb. 4: Änderung der Drehrichtung durch unterschiedliche Beschaltung der MOSFETs[3]

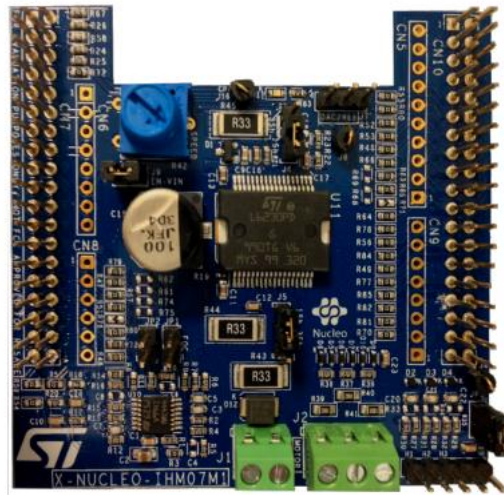


Abb. 5: X-NUCLEO-IHM07M1 Motor-Treiberstufe[4]

DC-Motor und Mikrocontroller sind über eine X-NUCLEO-IHM07M1 Motor-Treiberstufe (Abb. 5) miteinander verbunden. Dieses Bauteil sorgt dafür, dass aus zwei PWM-Signalen die Informationen zur Ansteuerung der vier MOSFETs gewonnen werden kann. Hierbei werden zwei MOSFETs aktiv mit den PWM-Signalen angesteuert, während die anderen beiden durch die interne Verschaltung der Treiberstufe mit den invertierten Signalen beschaltet werden[4]. Wenn beispielsweise am MOSFET S1a ein High-Pegel anliegt, wird der MOSFET S1b automatisch mit dem Low-Pegel bestromt. In der Firmware müssen demnach nur die oberen beiden MOSFETS angesteuert werden.

Die Treiberstufe verfügt zusätzlich über eine externe Spannungsversorgung und Messwiderstände (shunts)[4]. Die shunts dienen dazu den Ausgangsstrom der Endstufe zu bestimmen. Sie sind essenziell für die Funktion der Motion-Controller-Firmware, da sie eine Stromregelung ermöglichen.

Die Motortreiberplatine wird ab einer Spannung von 14 Volt funktionsfähig und ermöglicht es mit kleinen Steuerimpulsen bzw. Strömen der Steuerplatine einen Motorstrom von mehreren Ampere zu schalten.



Abb. 6: Micos PRS110, Rotationsmotor[5]

Der verwendete Motor ist ein Micos PRS-110 (Abb. 6). Hierbei handelt es sich um einen Präzisions-Rotations-Motor[5]. Der Motor verfügt über Positions- bzw. Lage-Sensoren, welche eine Rückmeldung über die Drehrichtung und Geschwindigkeit geben. Bei den Signalen handelt es sich um Rechteck-Signale (Abb. 7).

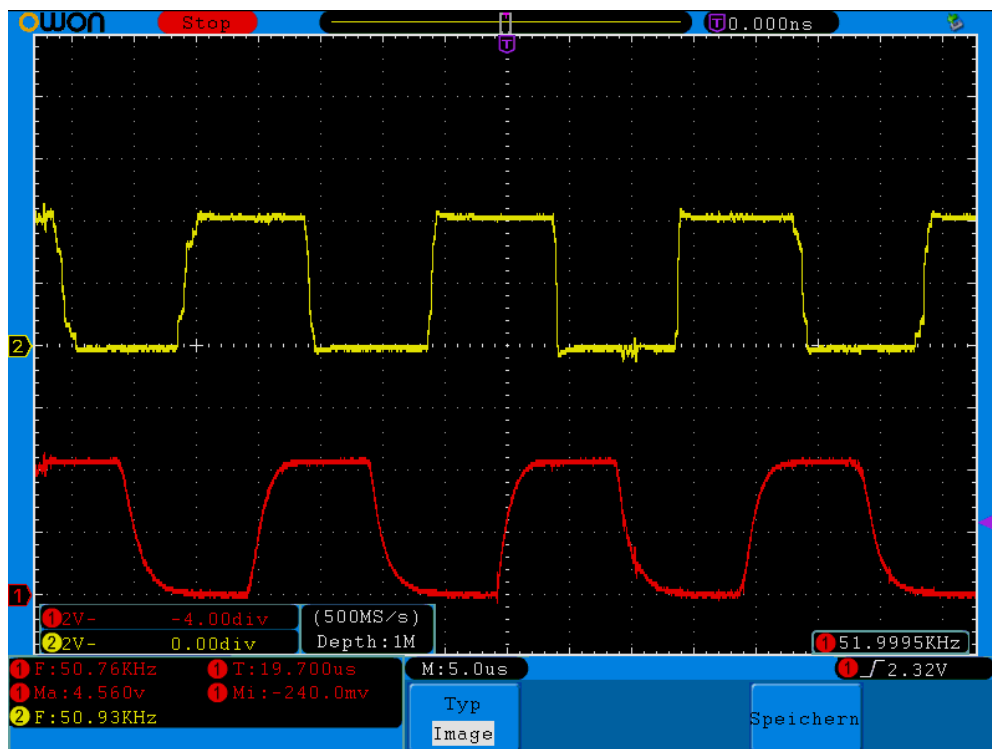


Abb. 7: Oszilloskopmessung der Lagesensorik bei konstanter Drehgeschwindigkeit

Um die Impulse bestmöglich nutzen zu können, werden in den Sensorleitungen Widerstände eingelötet (Abb. 8). Die Auswertung dieser Signale ermöglicht eine Positionsregelung des Motors. Das Funktionsprinzip des Rotationsmotors ist mit dem eines Linearmotors seitens der Ansteuerung sehr ähnlich, bietet jedoch den Vorteil keine Endanschläge zu besitzen. Somit wird bei Inbetriebnahme die Gefahr von Schäden oder Überlastungen durch ein Fahren gegen den Endanschlag vermieden.

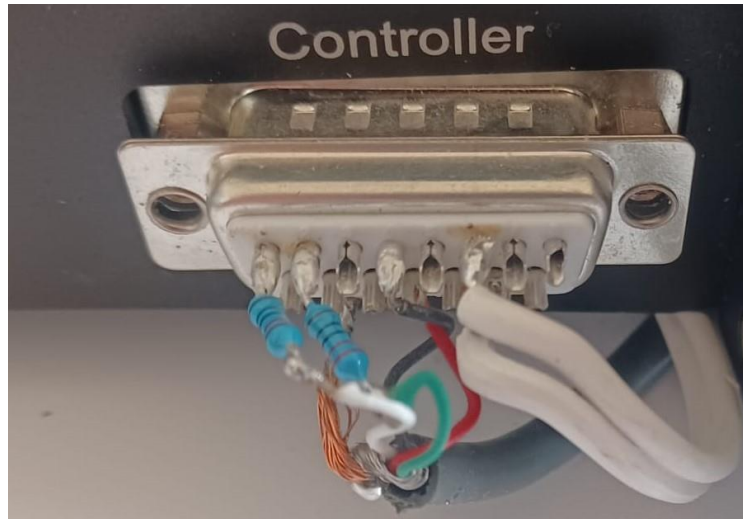


Abb. 8: Eingelötete Widerstände an Sensorleitungen

3.2. STM32 Cube IDE

Die Software wird in dem eigens für Mikrocontroller der Firma ST entwickeltem Programm „STM32Cube IDE“ (Integrated Development Environment) geschrieben. Ein großer Vorteil dieser Entwicklungsumgebung ist, dass die Konfiguration von Elementen wie Schnittstellen, Timer und Interrupts auf einer graphischen Oberfläche innerhalb eines ioc-Files durchgeführt werden kann (Abb. 9).

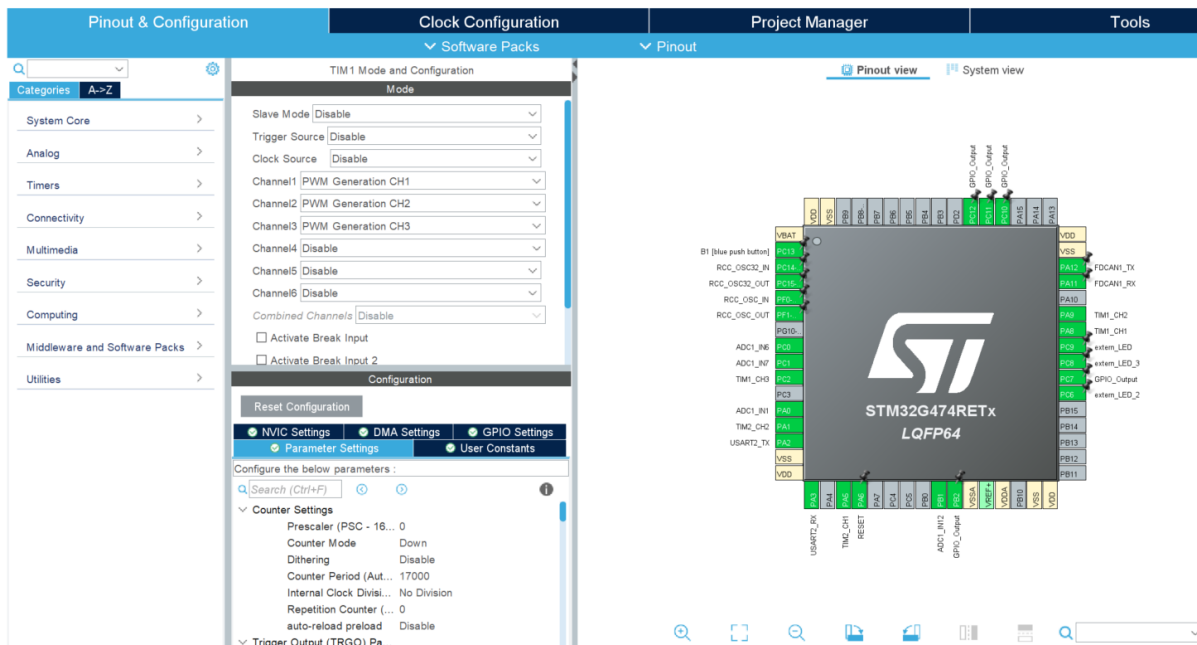


Abb. 9: Graphische Oberfläche in ioc-File

Sobald alle notwendigen Einstellungen durch den Nutzer festgelegt wurden, wird der zugehörige Code und die entsprechenden Files automatisch generiert und in die main.c-Datei eingefügt. Dies führt zu einer deutlichen Vereinfachung des Konfigurationsprozesses im Vergleich zu herkömmlichen, manuellen Konfiguration. Ein kleiner Nachteil ist jedoch, dass es sich bei den erzeugten Files um c-Dateien handelt. Die Software soll allerdings in C++ geschrieben, weshalb eigene cpp-Dateien angelegt werden, in welchen sich die Logik der Motion-Controller-Firmware befindet. Die Einarbeitung in die Entwicklungsumgebung wurde unter anderem durch zahlreiche, von ST selbst zur Verfügung gestellte, Tutorials und FAQs unterstützt.

3.3. Firmwarekonzept

Für die Auslegung des Firmwarekonzeptes müssen einige grundlegende Fragen geklärt werden. Zunächst stellt sich die Frage welche Eingangs- und Ausgangssignale das betrachtete System hat.

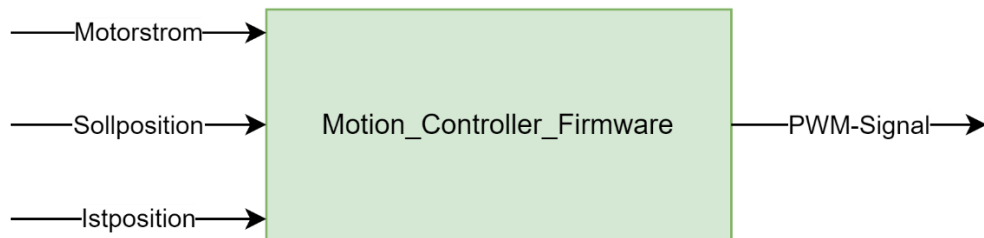


Abb. 10: BlackBox-Darstellung der Motion-Controller-Firmware

Aus Abb. 10 wird ersichtlich, dass die Firmware den Motorstrom, die Sollposition und die Istposition erhält und daraus ein PWM-Signal, genauer gesagt dessen Dutycycle, berechnen soll, mit welchem die Motoren angesteuert werden. Der Motorstrom wird über ADC-Schnittstellen mit Hilfe von Messshunts eingelesen. Die Sollposition wird in der späteren Anwendung über CANopen vorgegeben. Die Istposition wird über einen Positionssensor eingelesen. Für die konkrete Verarbeitung der Eingangssignale innerhalb der Firmware werden die jeweiligen Aufgaben in einzelne Module gekapselt. Abb. 11 veranschaulicht den Zusammenhang der Module zueinander.

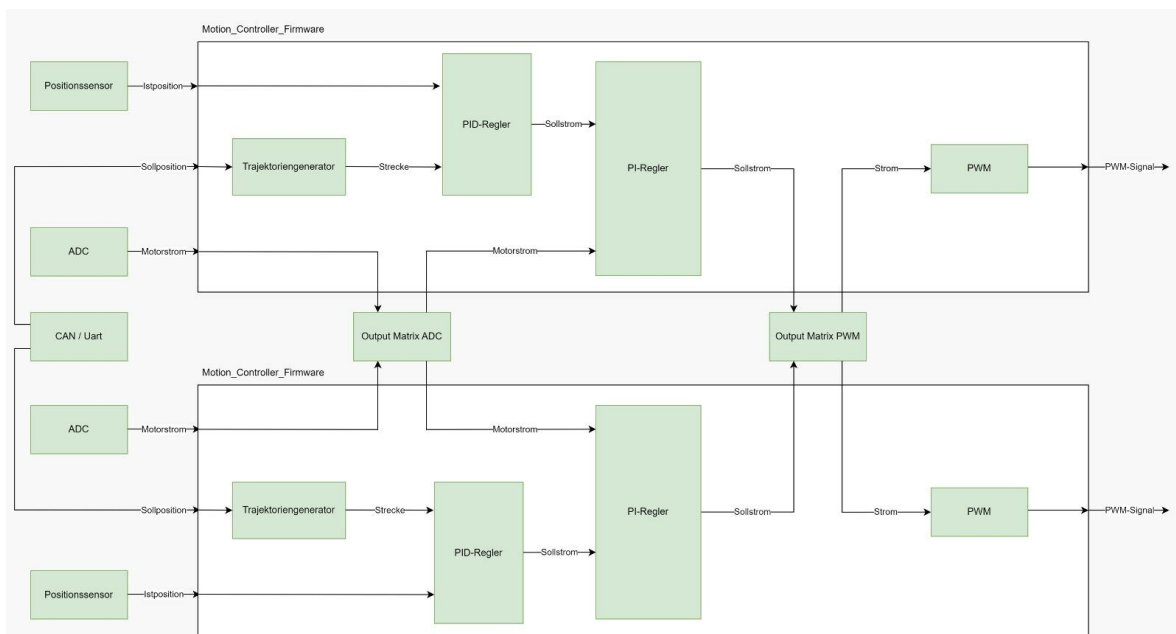


Abb. 11: Module der Motion-Controller-Firmware

Der Trajektoriengenerator erhält die Sollposition über CANopen bzw. UART und berechnet aus dieser die Wegstrecke bis zum Ziel. Mit dieser Information und der Istposition vom Positionssensor berechnet der Lage-Regler den notwendigen Sollstrom, um den Motor zu regeln. Dieser wird im Stromregler mit dem Iststrom aus dem ADC verglichen. Der resultierende Sollstrom wird anschließend über eine Output-Matrix, deren genauere Funktionsweise im Kapitel „3.4.3. Output-Matrix“ beschrieben ist, zu den PWM-Modulen geleitet, in welchen dann die Dutycycle gesetzt werden.

Die Strom- und Lageregelung (inklusive Auslesen der Sensordaten) sowie der Trajektoriengenerator und die PWM-Ansteuerung sind diejenigen Module, die eine Echtzeitfähigkeit verlangen (vgl. Abb. 12).

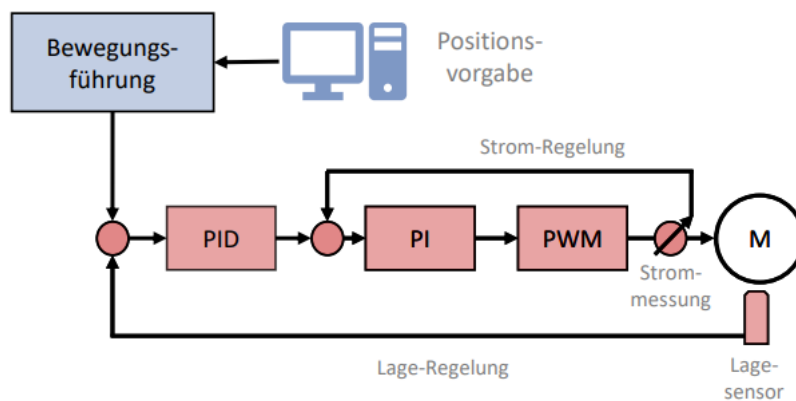


Abb. 12: Komponenten des Systems, rot=echtzeitfähig[6]

Die Echtzeitfähigkeit wird durch einen Timer-Interrupt umgesetzt. Der Interrupt wird immer dann ausgelöst, wenn es zu einem Überlauf des Timer-Counters kommt. Dadurch wird eine äquidistante Ausführung der Regelung und Ansteuerung ermöglicht. Der Kommando-Interpreter sowie das Auslesen der CANOpen-Schnittstelle sind nicht echtzeitkritisch. Für Module dieser Art wird „FreeRTOS“ (Free Realtime Operating System) verwendet. Dabei handelt es sich um ein Echtzeitbetriebssystem für eingebettete Systeme[7]. Es eignet sich besonders dafür interruptbasierte Tasks und Ressourcen zu verwalten[7]. Das Aufrufen und Verwalten der Tasks werden von einem sogenannten „Scheduler“ übernommen. In der STM32Cube-IDE gibt es die Möglichkeit FreeRTOS zu aktivieren und über die ioc-Datei Tasks zu erstellen und diesen eine Priorität zuzuweisen. Die Priorität entscheidet dabei, welche Tasks gegenüber anderen Tasks bevorzugt aufgerufen werden. Grundsätzlich gilt jedoch, dass kein FreeRTOS-Task eine höhere Priorität als ein Timer-Interrupt hat. Der Versuch auch die echtzeitkritischen Tasks in FreeRTOS umzusetzen ist nicht gelungen, da die FreeRTOS-Tasks mit einer maximalen Frequenz von einem kHz aufgerufen werden können. Dies genügt jedoch nicht den Anforderungen an die Regelung der Motion-Controller-Firmware.

Die Umsetzung der gekapselten Module erfolgt durch Klassen. Jede Klasse soll dabei jeweils einen Aufgabenteil bearbeiten. Die Klassen verfügen neben den notwendigen privaten Attributen über set- und get-Methoden, um beispielsweise auf berechnete Werte von anderen Klassen Zugriff zu erhalten. Durch die Kapselung der Module in Klassen arbeitet jede Klasse nur mit den Daten, die sie tatsächlich für ihre Funktionserfüllung benötigt. Dies macht den Code strukturierter und nachvollziehbarer. Die Klassen stehen aber auch in Zusammenhang zueinander. Zum einen werden Objekte einer Klasse in anderen Klassen als private Attribute verwendet (Komposition, Aggregation) und zum anderen gibt es Klassen, die von anderen Klassen Methoden übernehmen (Vererbung)[8]. Letzteres zeigt sich am Beispiel der Interface-Klasse „IMotorBase“ (Abb. 13).

```
#ifndef __cplusplus
class IMotorBase
{
public:
    virtual void CurrentRegulation()=0;
    virtual void PositionRegulation()=0;
    virtual void TrajectoryGenerator()=0;
    virtual void setPWMValue()=0;
};
#endif
```

Abb. 13: Interface-Klasse "IMotorBase"

Diese dient als eine Art Vorlage für alle Motorklassen, die zu einem späteren Zeitpunkt in die Firmware integriert werden sollen. Die IMotorBase-Klasse gibt dabei vor, welche Methoden jede Motorklasse haben muss. Beispielsweise benötigt jeder Motor eine Funktion für die Lageregelung und

Stromregelung. Durch das Schlüsselwort „virtual“ wird die jeweilige Methode in der Interface-Klasse deklariert und jede Klasse, die von dieser erbt, muss eine Implementierung für die Methode liefern[9]. Mit der Zuweisung „=0“ muss die virtual-Funktion nicht in der Interface-Klasse selbst implementiert werden[9]. Durch die Verwendung eines interface-basierten Firmwarekonzeptes ergeben sich zwei wesentliche Vorteile. Zunächst wird der Code übersichtlicher und strukturierter, da jede neue Motorklasse auf eine gemeinsame Elternklasse zurückgeführt werden kann und gleich aufgebaut ist. Außerdem kann man Funktionen, die ein Motorobjekt benötigen, einen Pointer der Klasse IMotorBase übergeben. Dadurch muss die aufrufende Funktion nicht mehr wissen, welche Motorklasse vorliegt, sondern kann jede Motorklasse, aufgrund des identischen Aufbaus, gleichbehandeln. Aus dieser Funktionalität lässt sich auch die Bezeichnung von Klassen wie IMotorBase ableiten. Sie fungieren als Schnittstellen (engl. „Interface“) z.B. bei Funktionsaufrufen mit Parametern. Das vollständige Klassendiagramm für die Motion-Controller-Firmware befindet sich im Anhang unter „Klassendiagramm“.

Insgesamt ergibt sich der grundlegende Ablauf für die main-routine entsprechend Abb. 14:

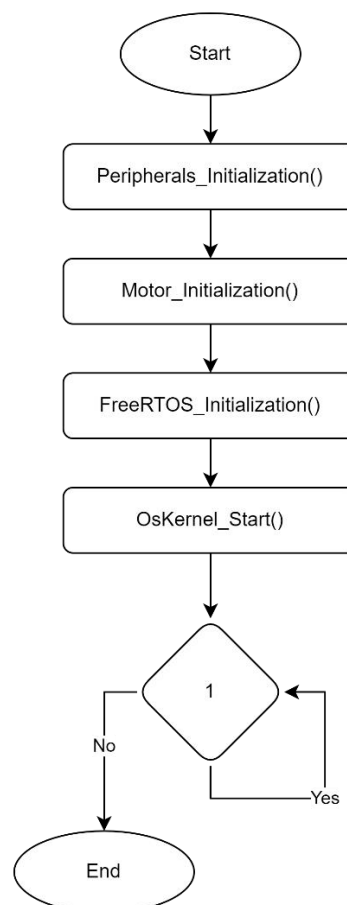


Abb. 14: Programmablaufplan main-Routine

Zunächst wird die Funktion zur Konfiguration der Peripherie ausgeführt. Dabei handelt es sich um eine vom Programm automatisch generierte Funktion auf Basis der Einstellungen im ioc-File. Hierzu gehören zum Beispiel die Initialisierungen für die Timer, Pins, CAN, UART, DMA (Direct Memory Access) und ADC. Anschließend erfolgt die Initialisierung der Achsen mit den notwendigen Parametern für die Regler, den Trajektoriengenerator und die Output-Matrix. Zuletzt wird die Initialisierung für FreeRTOS durchgeführt, bevor der zugehörige Kernel gestartet wird. Die while-Schleife ist für den Dauerbetrieb der Motion-Controller-Firmware notwendig. Ab diesem Zeitpunkt reagiert das System auf Ereignisse, die durch FreeRTOS oder einen Timer-Interrupt ausgelöst werden. Für den zuletzt genannten Fall wird eine Timer-Callback-Funktion aufgerufen, deren Ablauf in Abb. 15 zu sehen.

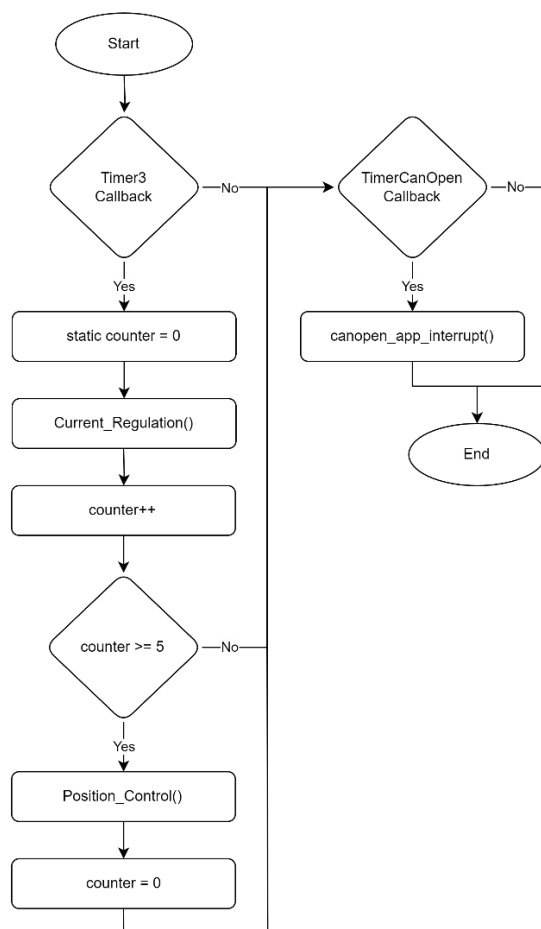


Abb. 15: Programmablaufplan Timer-Callback-Funktion

Die Callback-Funktion ist für die ausgelösten Ereignisse von zwei unterschiedlichen Interrupts verantwortlich. Zum einen von Timer 3, der für die Motorregelung verwendet wird und zum anderen vom Timer für den Aufruf der CANopen-Interrupt-Funktion. Die Motorregelung im Interrupt wird auf Basis eines 10µs-Timers umgesetzt. Dabei wird die Stromregelung bei jedem Aufruf und die Lageregelung bei jedem fünften Aufruf des Interrupts ausgeführt.

Die beiden gezeigten Abläufe befinden sich in der main.c-Datei. Da in c-Dateien keine Klassenobjekte angelegt werden können, werden die Implementierungen aller selbstgeschriebenen Funktionen, d.h. „Motor_Initialization“, „Current_Regulation“, „Position_Control“ und „canopen_app_interrupt“, in eine separate cpp-Datei mit dem Namen „Process“ ausgelagert.

3.4. Umsetzung der Module

In diesem Kapitel wird die Umsetzung der einzelnen Module detailliert erläutert. Zum besseren Verständnis sind Ausschnitte des Firmware-Sourcecodes integriert. Diese Code-Snippets veranschaulichen das Vorgehen und sollen zudem ein Anhaltspunkt für nachfolgende Weiterentwicklungen an diesem Projekt sein.

3.4.1. Trajektoriengenerator

Die Hauptaufgabe des Trajektoriengenerators ist die Bewegungsführung für den Roboterarm. Ziel ist es eine flüssige Bewegung des Roboterarms zu erreichen. Hierfür wird auf Basis des aktuellen Bremswegs entschieden, ob das System gebremst werden muss (Wegstrecke bis zum Ziel kleiner als Bremsweg) oder weiter beschleunigen kann (Wegstrecke bis zum Ziel größer als Bremsweg). Ausgehend vom daraufhin festgelegten Beschleunigungswert kann nun die Geschwindigkeit und im nächsten Schritt die aktuelle Position rekursiv berechnet werden[6]. Der Zusammenhang zwischen diesen Größen wird graphisch in Abb. 16 veranschaulicht.

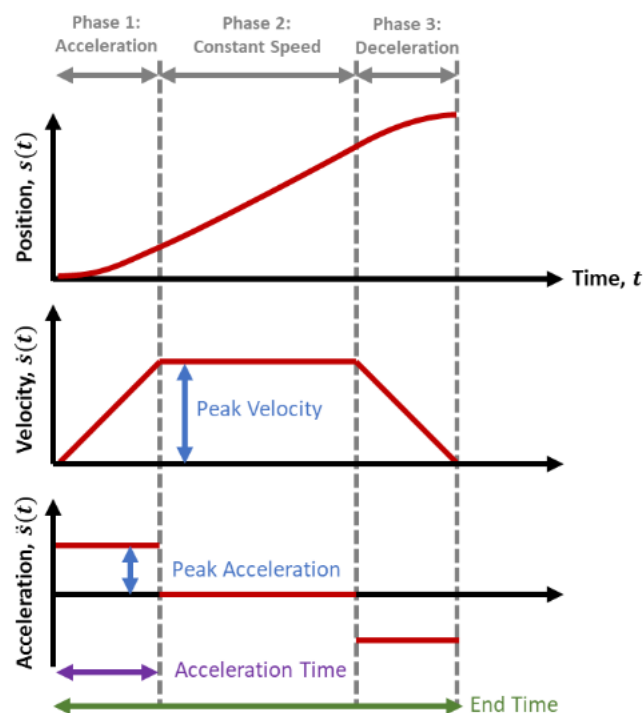


Abb. 16: Trapezprofil für Geschwindigkeit[10]

```

myParamTrajGen.s_delta = myParamTrajGen.setpoint - myParamTrajGen.s;

    if(myParamTrajGen.s_delta == 0.0)
    {
        myParamTrajGen.motion_on = false;
    }
    else
    {
        myParamTrajGen.motion_on = true;
        // aktueller Bremsweg
        myParamTrajGen.s_br = round(myParamTrajGen.d_s
/myParamTrajGen.dd_s_max_dec) * 0.5 * (myParamTrajGen.d_s);

        // Geschwindigkeitsberechnung
        if(myParamTrajGen.s_delta >= 0.0 )
        // positive Bewegungsrichtung
        {
            if(fabs(myParamTrajGen.s_delta) >
fabs(myParamTrajGen.s_br)) // Beschleunigung
            {
                myParamTrajGen.dd_s = myParamTrajGen.dd_s_max_acc;
            }
            else // Verzögerung
            {
                myParamTrajGen.dd_s = -1.0 *
myParamTrajGen.dd_s_max_dec;
            }
        }
        else // negative Bewegung
        {
            if(fabs(myParamTrajGen.s_delta) >
fabs(myParamTrajGen.s_br)) // Beschleunigung
            {
                myParamTrajGen.dd_s = -1.0 *
myParamTrajGen.dd_s_max_acc;
            }
            else // Verzögerung
            {
                myParamTrajGen.dd_s = myParamTrajGen.dd_s_max_dec;
            }
        }
        myParamTrajGen.d_s = myParamTrajGen.d_s + myParamTrajGen.dd_s;
        // Begrenzung der Geschwindigkeit
        if (myParamTrajGen.d_s > myParamTrajGen.d_s_max)
        {
            myParamTrajGen.d_s = myParamTrajGen.d_s_max;
            myParamTrajGen.dd_s = 0.0;
        }

        if(myParamTrajGen.d_s < -1.0 * myParamTrajGen.d_s_max )
        {
            myParamTrajGen.d_s = -1.0 * myParamTrajGen.d_s_max;
            myParamTrajGen.dd_s = 0.0;
        }
        if(fabs(myParamTrajGen.s_delta) <= fabs(myParamTrajGen.d_s))
        {
            myParamTrajGen.s = myParamTrajGen.setpoint;
            myParamTrajGen.dd_s = 0.0;
        }
        else
        {
            myParamTrajGen.s = myParamTrajGen.s + myParamTrajGen.d_s;
        }
    }

```

Abb. 17: Programm-Code zur Trapez-Funktion der Klasse "TrajectoryGenerator"

Abb. 17 zeigt die konkrete Umsetzung in der Software. Der Trajektoriengenerator erhält die vorgegebene Sollposition („setpoint“) und berechnet aus dieser die Wegstrecke bis zum Ziel („s“). Diese Größe entspricht dem Sollwert für die Lageregelung. Das Integrieren der Beschleunigung bzw. Geschwindigkeit wird durch Aufsummieren der jeweils berechneten Werte aus dem aktuellen Abtastschritt mit den vorherigen Werten erreicht.

3.4.2. Strom- und Lageregelung

Grundlegend werden lineare Standard-Regler bei der Umsetzung der Motorregelung verwendet, da sich diese verhältnismäßig einfach in der Software umsetzen lassen.

Bei der Auswahl der Regler für die jeweiligen Regelprozesse müssen einige Aspekte beachtet werden. Zum einen darf die Motorregelung keine bleibende Regelabweichung hinterlassen, zum anderen soll die Lageregelung eine möglichst hohe Dynamik aufweisen, um schnelles Ausregeln zu ermöglichen. Daher wird für die Stromregelung ein PI-Regler und für die Lageregelung ein PID-Regler verwendet. Durch den integrierenden Anteil (I-Term) wird die Problematik einer bleibenden Regelabweichung gelöst[11]. Der differenzierende Anteil (D-Term) ermöglicht die hohe Dynamik und das schnelle Ausregeln des Systems[11].

Die aktuelle Position des Motors wird aus dessen Positionssensoren ermittelt. Hierfür wird ein Timer des Motion-Controllers im Encoder-Mode betrieben. Die Inkrementierung des Timers erfolgt in diesem Fall nicht über die interne Clock des Prozessors, sondern wird über externe Signale angeregt. Zur Verwendung werden zwei GPIO (General Purpose Input Output) -Pins für den Timer konfiguriert, welche mit den Sensorleitungen des Motors verbunden werden. Aus den empfangenen Signalen bestimmt der Encoder des Timers die Drehrichtung und Anzahl der Übermittelten Impulse. Mit bekanntem Übersetzungsverhältnis (Impuls zu Drehwinkel) lässt sich bei Bewegung der Winkel bestimmen. Die Abfrage und Berechnung der Bewegung erfolgt vor der Lageregelung. Hierbei wird der Differenzwinkel zur letzten Iteration berechnet und mit der letzten Position summiert.

Der PI-Regler benötigt neben der Stellgröße, welche aus der Berechnung des PID-Reglers folgt, einen Istwert des Stroms. Zur Ermittlung des Stroms werden die shunts der Motor-Treiber-Platine verwendet. Die Schematische Aufbau ist Abb. 18 zu entnehmen.

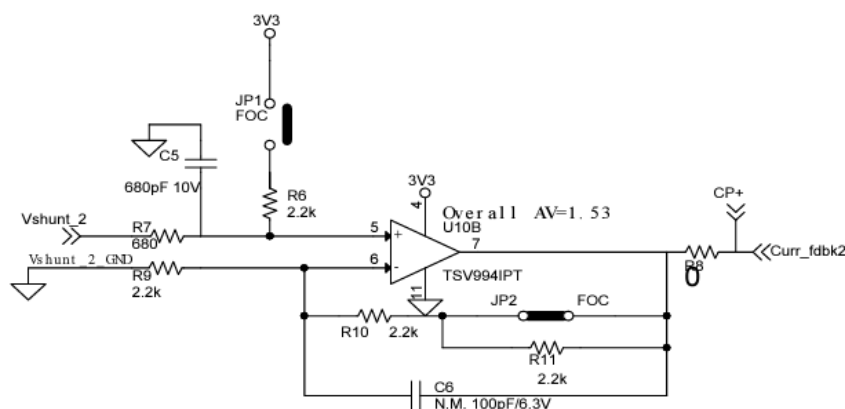


Abb. 18: Schematischer Aufbau der Schaltung incl. shunts[4]

Zur Messung der resultierenden Spannung mit dem Motion-Controller wird ein ADC mit 12 Bit Auflösung verwendet. Um die Spannung beider shunts in einem ADC auszuwerten, wird der verwendete ADC mit zwei Kanälen konfiguriert. Der Istwert des Stromes errechnet sich aus dem ADC-Wert x , der Auflösung n , der Referenzspannung U_{ref} und dem Widerstandswert R gemäß Formel (1).

$$I = \frac{x}{(2^n - 1)} * \frac{U_{ref}}{R} \quad (1)$$

Anhand einiger Messungen mit dem Oszilloskop wird das Abtastverhalten, sowie die Synchronität zwischen PWM und ADC-Konvertierung untersucht.

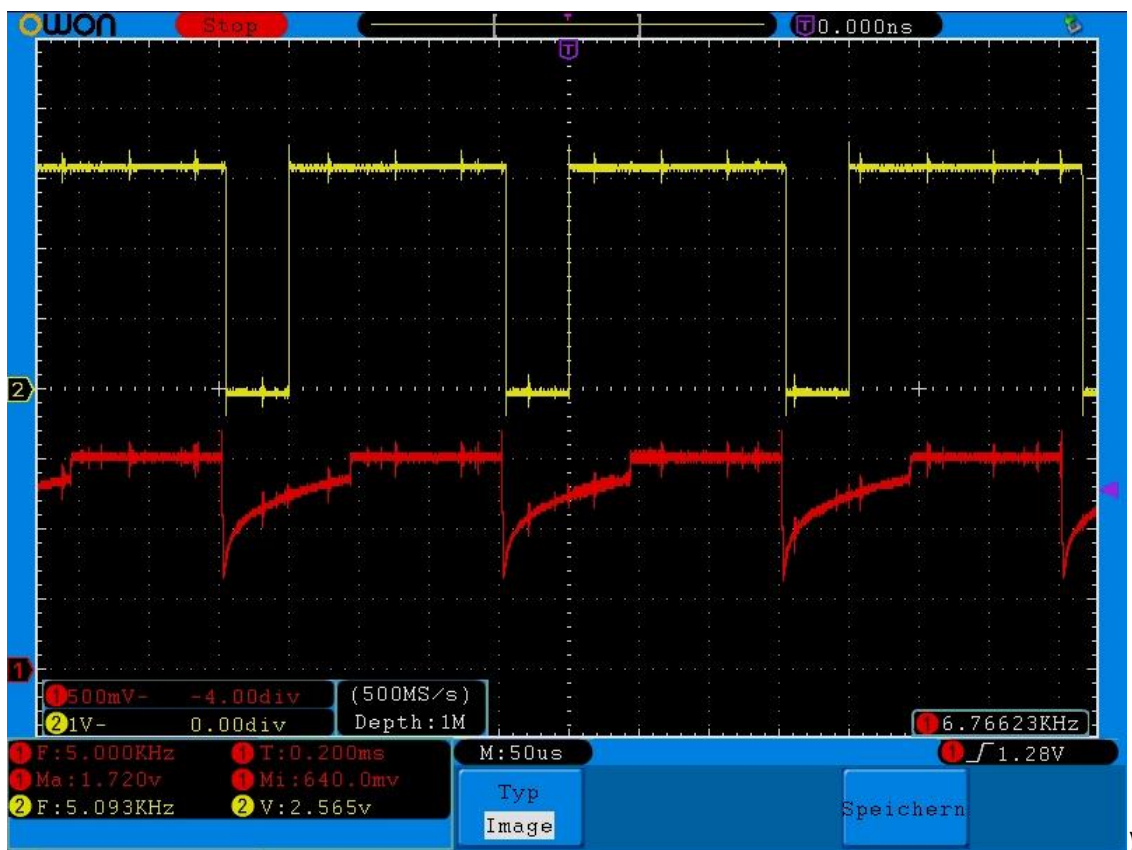


Abb. 19: Messung des ADC-Signals (gelb) und Shuntspannung (rot)

Aus Abb. 19 wird ersichtlich, dass die Synchronität zwischen der ADC-Konvertierung und den Strömen am Messshunt gewährleistet ist. Die fallende Flanke des ADC-Signals markiert eine erfolgreiche Konvertierung des Spannungswertes. Der Versatz zwischen der fallenden Flanke des shunt zum ADC-Signal lässt sich auf die benötigte Konvertierungszeit der Spannungsumwandlung zurückführen.

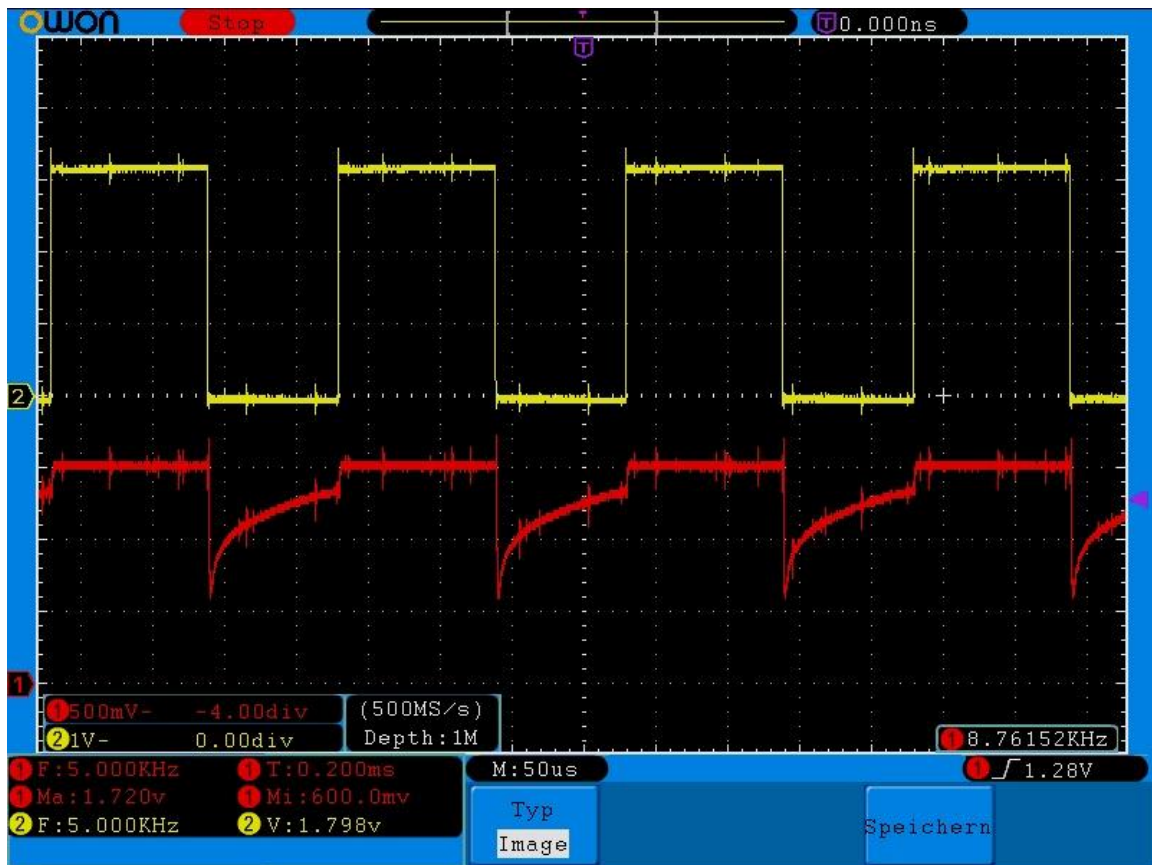


Abb. 20: Messung von PWM-Ausgang (gelb) und Shuntspannung (rot)

Außerdem muss darauf geachtet werden, dass das Messen des Stroms immer dann erfolgen soll, wenn die PWM-Signale auf dem Low-Pegel sind. Nur dann können sinnvolle Werte ausgelesen werden (vgl. Abb. 20).

Zusätzlich ist zu beachten, dass die shunt-Spannung einen Offset von 1.53 Volt hat und die Differenz hierzu eine Aussage über den Strom ermöglicht.

In Abb. 21 ist die Synchronisierung zwischen den PWM-Signalen und der ADC-Konvertierung zu sehen. Letztere wird dabei durch einen Pin repräsentiert, welcher immer dann getoggelt wird, wenn die Ströme eingelesen werden.

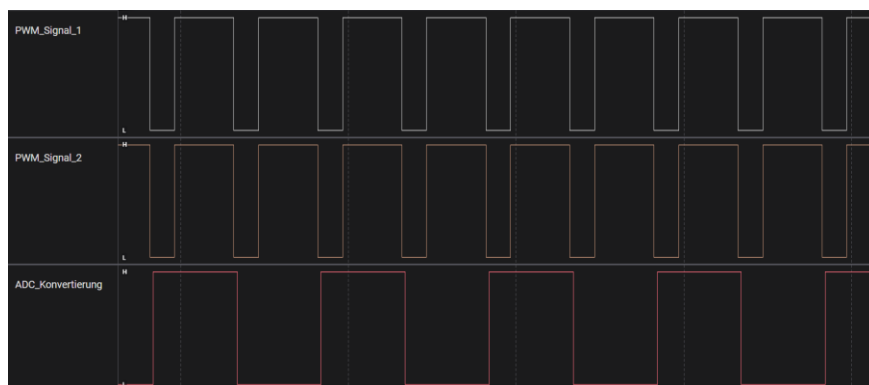


Abb. 21: Messung von ADC-Konvertierung und PWM-Signalen

Abb. 22 zeigt die Umsetzung der Regelung am Beispiel des PID-Reglers für die Lageregelung

```
double P;
double I;
double D;

//Regelfehler
PID_error = s - m_pos;

PID_I_MEM = PID_I_MEM + PID_error;

// P-Glied
//P = PID_error * PID_P * 0.000001;
P = PID_error * mP_fkt;
// I-Glied
//if(PID_Tn > 0.0001)
if(mTn > 0.0001)
{
    //I = 1/PID Tn * PID I MEM;
    I = mI_fkt * PID_I_MEM;
}
else
{
    I = 0.0;
}

//D-Glied
//D = (PID_error - PID_error_1) * PID Tv * 0.000001;
D = (PID_error - PID_error_1) * mD_fkt;
// speichere den Wert für nächste Iteration...
PID_error_1 = PID_error;

// Ausgang
PID_Y = P + I + D;

//Limiter
if(PID_Y > PID_lim)
{
    PID_Y = PID_lim;
}
else if(PID_Y < -1 * PID_lim)
{
    PID_Y = -1 * PID_lim;
}

//Limiter
if(PID_Y > PID_lim * 0.3)
{
    PID_I_MEM -= PID_error;    // anti Windup
    PID_Y = PID_Y - I;
}
else if(PID_Y < -0.3 * PID_lim)
{
    PID_I_MEM -= PID_error;    // anti Windup
    PID_Y = PID_Y - I;
}
```

Abb. 22: Funktion für Lageregelung in PID-Klasse

3.4.3. Output-Matrix

Um die Flexibilität des Systems zu erhöhen und für bestimmte Situationen eine optimalere Sensorauswertung zu gewährleisten, wird eine Output-Matrix-Klasse in die Motion-Controller-Firmware integriert. Diese Klasse hat die Aufgabe die Zuweisung der Sensorwerte, welche über die ADC-Schnittstellen geliefert werden, an die jeweiligen Achsen zu steuern. Die Idee ist dabei für jede Achse Parameter zu übergeben, welche die Relevanz eines Sensors bzw. dessen Sensorwerte für die jeweilige Achse gewichten. In Abb. 23 ist der schematische Aufbau einer Output-Matrix dargestellt.

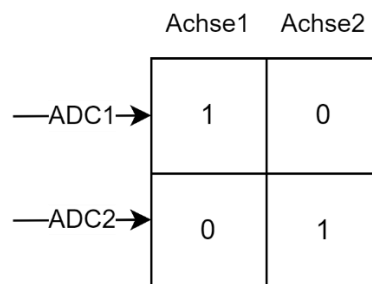


Abb. 23: Funktionsprinzip der Output-Matrix

Im gezeigten Beispiel würde Achse 1 ausschließlich die Werte von Sensor 1 und Achse 2 ausschließlich die Werte von Sensor 2 berücksichtigen. Die Parameterwerte können zwischen 0 und 1 frei gewählt werden, sodass auch Kombinationen wie z.B. Achse 1 mit den Gewichten 0,25 für Sensor 1 und 0,75 für Sensor 2 möglich sind. Die Gewichte werden jeweils mit den Sensorwerten multipliziert, addiert und können anschließend von der entsprechenden Achse durch eine get-Funktion gelesen und weiterverarbeitet werden. Gleichzeitig muss beachtet werden, dass die Gewichtung auch die Berechnung der PWM-Dutycycle beeinflusst. Daher muss in der Output-Matrix auch deren Berechnung erfolgen. Hierfür müssen die Ausgänge der Stromregelung der Achsen in der Output-Matrix-Klasse, in gleicher Weise wie zuvor beschrieben, mit den Gewichten verrechnet werden. Insgesamt umfasst die Output-Matrix Funktionen zur Einstellung der Gewichte, sowie Berechnung der ADC- bzw. PWM-Werte (vgl. Abb. 24)

```

#ifndef INC_CPLUSPLUS_OUTPUTMATRIX_HPP_
#define INC_CPLUSPLUS_OUTPUTMATRIX_HPP_

#include "map"

class DCMotor;

#ifdef __cplusplus
class OutputMatrix
{
public:
    struct axisMatrix {
        double sensor_matrix_1[2] = {1, 0};
        double sensor_matrix_2[2] = {0, 1};
        double pwm_axis_value;
        double adc_axis_value;
    };
    OutputMatrix();
    void setMatrix(axisMatrix &matrix, DCMotor* motor);
    void setMatrix(DCMotor* motor); //default matrix
    void calcPWMAxisValue();
    double getADCAxisValue(DCMotor* motor_ptr);
    double getPWMAxisValue(DCMotor* motor_ptr);
    void calcADCAxisValue();

private:
    std::map<DCMotor*, axisMatrix> m_matrixes;
};
#endif

#endif /* INC_CPLUSPLUS_OUTPUTMATRIX_HPP */

```

Abb. 24: Übersicht aller Funktionen in Output-Matrix-Klasse

3.4.4. CANopen

Die Verbindung des Motion-Controllers zur übergelagerten Steuerung erfolgt über CANopen. CANopen ist ein auf CAN basierendes Kommunikationsprotokoll[12]. Mit dem CANopen Standard wird der Austausch von Daten in einem CAN basierten Netzwerk definiert und in einer höheren Ebene umgesetzt.

Im Gegensatz zur Abitrierungshierarchie beim „klassischen“ CAN arbeitet CANopen mit Client-Server-Diensten und dem Producer-Consumer-Konzept. CANopen ähnelt also einem Master-Slave-System. Das „object dictionary“ (OD) in CANopen definiert die Struktur, den Datenzugriff und die Struktur des Datenaustausches im Netzwerk.[13]

Das OD kann entsprechend der Anforderungen angepasst und konfiguriert werden.

Für die Implementierung von CANopen in der Firmware wird ein CANopen-Stack mit dem Namen „CANopenNode STM32“ verwendet, welcher auf dem CANOpenNode-Stack basiert[14]. Dieser Stack ist für die Verwendung in der STM32-Umgebung und den Betrieb auf dem nucleo-board optimiert. Durch diese Einbindung sind bereits low-level Treiber und die Basisfunktionen von CANopen vorhanden. Für die Nutzung muss nach Einbindung die entsprechende CAN-Schnittstelle konfiguriert werden. Die Konfiguration umfasst im Wesentlichen die Baudrate, die für das Senden und Empfangen verwendeten Pins und das Speichermanagement der gesendeten und empfangenen Nachrichten. Zusätzlich besteht die Möglichkeit zwei Status-LEDs über GPIO-Pins einzubinden, welche den aktuellen Status visualisieren.

Durch das zugrundeliegende CAN-Protokoll mit CAN-FD-Fähigkeit besteht die Möglichkeit einen Baudrate-Switch (BRS) zu verwenden, was Datenraten bis 3 Mbit/s ermöglicht. Da die primäre Anwendung des Motion-Controllers das nicht erfordert, wird die CAN-Schnittstelle mit einer Datenrate von 500 kBit/s und ohne BRS konfiguriert. Diese Datenrate ist für die zugrundeliegenden Funktionen ausreichend und reduziert die Störanfälligkeit. Der Samplepunkt wird entsprechend des gängigen Standards auf 80 Prozent gesetzt.[15]

Um die CPU (Central Processing Unit) -Last zu reduzieren, wird DMA verwendet. Hierbei erfolgt der RAM (Random Access Memory) -Zugriff des CAN-Controllers beim Senden und Empfangen ohne Umweg über die CPU.

Im Verbund mit der übergeordneten Steuerungseinheit agiert der Motion-Controller als Slave und reagiert entsprechend.

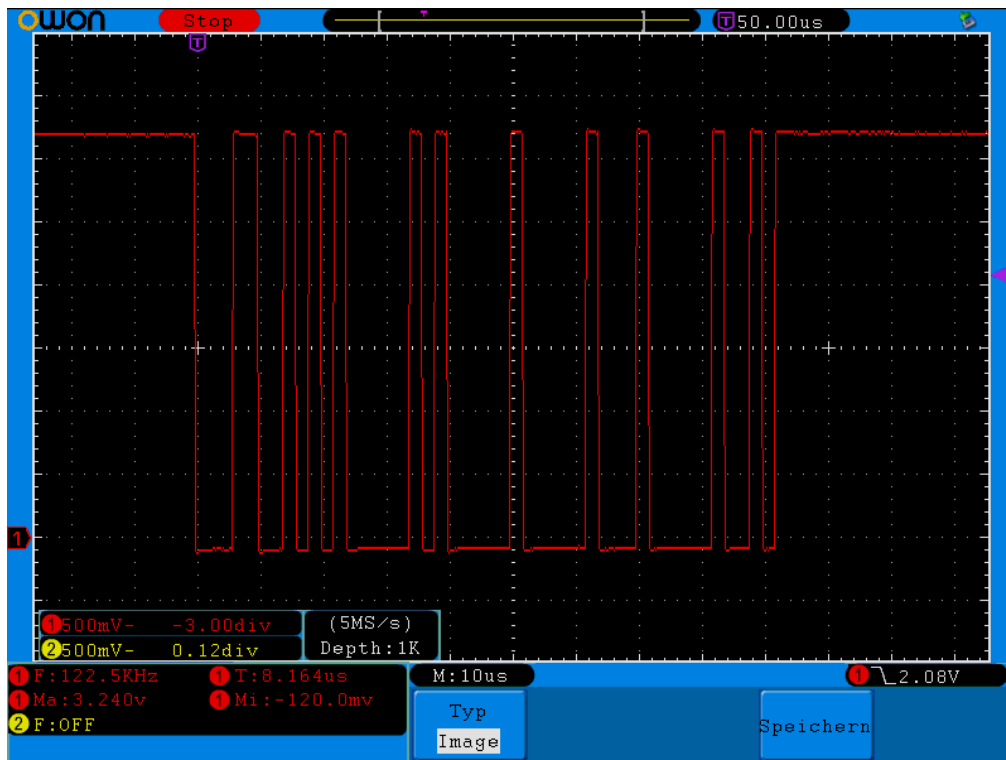


Abb. 26: Messung des PDO mit Oszilloskop

Durch die Auslegung als Slave-CANopenNode und das Fehlen einer übergeordneten Steuereinheit wird die Prüfung der Funktionsfähigkeit ohne Peripheriegeräte durchgeführt. Zum einen wird das Verhalten der Status-LEDs verifiziert und zum anderen der Ausgang der CAN-Schnittstelle mit einem Oszilloskop ausgewertet. Hierbei wird der CANopenNode so konfiguriert, dass er regelmäßig eine Status-Nachricht (Heartbeat) über den Bus sendet. Die Messung des Ausgangssignales ist in Abb. 26 dargestellt.

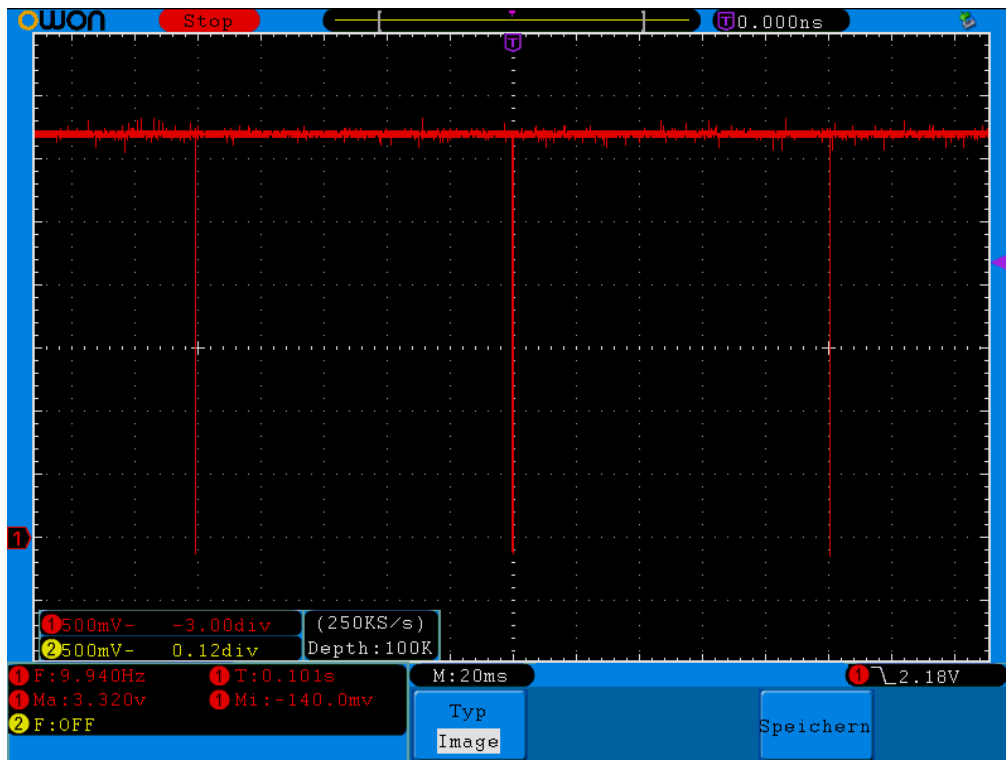


Abb. 27: Messung der Zykluszeit der PDO mit Oszilloskop

Zur Überprüfung der Zykluszeit der versendeten Statusnachrichten wird eine weitere Messung durchgeführt. Die Ergebnisse, welche Abb. 27 zu entnehmen sind, verifizieren die mit dem CANopenEditor eingestellte Zykluszeit von 100ms. Eine Abweichung von drei Millisekunden ist dennoch vorhanden. Diese Abweichung ist im Vergleich zur Periodendauer vernachlässigbar und schränkt die Funktion nicht ein.

Aufbau, Baudrate und Funktion können somit auch ohne andere Busteilnehmer sichergestellt werden. Bei Verbindung mit einer übergeordneten Steuerungseinrichtung können noch Änderungen der Baudrate oder des OD erforderlich sein, welche aber ohne Eingriffe in die bestehende Software umsetzbar sind. Für die Verwendung mehrerer CANopen-Schnittstellen auf einem Controller-Board müssen zusätzliche Änderungen der Software und des verwendeten Stacks vorgenommen werden.

3.4.5. Kommando-Interpreter

Um eine Kommunikation während des Betriebs vornehmen zu können, ist eine Schnittstelle mit dem Motion-Controller erforderlich. In diesem Projekt wird die serielle Schnittstelle mithilfe der UART realisiert. Über die UART sollen aktuelle Parameter und Zustände abgefragt und auch gesetzt werden können. Das ist für die Parametrisierung und auch für Tests der Software unerlässlich.

Die UART wird bidirektional konfiguriert. Somit ist eine UART-Schnittstelle ausreichend. Die Verbindung erfolgt in diesem Projekt über die Micro-USB-B-Schnittstelle des Nucleo-Boards. Diese dient als Spannungsversorgung und gleichzeitig auch als Datenverbindung.

Wie auch beim CANopen wird hier DMA verwendet, um die CPU-Auslastung so gering wie möglich zu halten. Eine weitere verwendete Optimierung stellt der IDLE-Interrupt beim Empfangen von Daten dar. Hierbei wird die Interrupt-Routine erst durchlaufen, wenn die UART einen Zeittakt lang keine Daten mehr erhalten hat. Für die Verwendung dieser Art von Interrupt muss sichergestellt sein, dass der zu beschreibende Speicher groß genug für empfangene Daten ist. Andernfalls gehen Informationen verloren und die Funktion ist nichtmehr sichergestellt.

Für eine übersichtliche und einheitliche Implementierung wird eine Wrapper-Klasse für die Funktionen der UART implementiert. Diese wird mit den entsprechenden Parametern und Hardware-Verweisen initialisiert und bündelt die Funktionen der UART. Dieses Vorgehen ermöglicht effiziente Erweiterungen der Funktion. Zusätzlich ist die Einführung einer zweiten UART-Schnittstelle unter Wiederverwendung der Klasse einfach möglich.

Der Empfang von Daten über die UART startet mit Auslösen des IDLE-Interrupts. Im Interrupt werden die empfangenen Daten zur Auswertung aus dem DMA-Buffer in einen zur Verarbeitung verwendeten Buffer kopiert. Das ermöglicht ein erneutes Empfangen ohne Überschreiben der vorherigen Informationen. Wird im Interrupt ein Zeilenumbruch erkannt, beginnt im Command-Interpreter Task die Auswertung des Kommandos.

Pro Eingabe wird ein Kommando ausgewertet. Es ist möglich einen Wert zu setzen oder einen Wert anzufragen. Über eine ID (Identification) nach dem set- oder get-Kommando lässt sich der gewünschte Parameter adressieren. Eine Liste aller Befehle für den Kommando-Interpreter inklusive Erklärung ist im Anhang unter „Befehlsliste des Kommando-Interpreters“ beigefügt. Bei einer Werteanfrage wird der geforderte Parameter über die UART zurückgesendet. Sollte ein fehlerhaftes Kommando eingegeben worden sein, wird eine Fehlermeldung mit Verweis auf ein Hilfefkommando zurückgesendet und die Eingabe verworfen. Das Hilfefkommando gibt alle verfügbaren Kommandos und IDs aus. Zusätzlich wird ein Beispielbefehl ausgegeben, welcher verdeutlicht, wie der Kommando-Interpreter zu verwenden ist. (vgl. Abb. 28).


```

blabla
invalid command try again :(
For help use: get 0006
get 0006
cmd-example: set/get xxxx Value
1. digit 0=Sys,1=Axis
Case sys: 2.digit 0
Case axis: 2.digit 1=Axis1, 2=Axis2
Case sys: 3.digit 0
Case axis: 3.digit 0=PI,1=PID,2=Pos,3=Vel,4=Acc,5=Dec,6=PWM-DutyCycle
Case sys: 4.digit 0=Version,1=Status,2=Ready,3=Operation,4=Status,5=Control,6=Cmd List
Case axis:
Controller command:4.digit 0=Kp,1=Tn,2=Tv
Pos,Vel,Acc,Dec,PWM:4.digit 0=valid,else: invalid

```

Abb. 28: Beispiel zur Veranschaulichung der Help-Funktion

Auch das Setzen einer Position ist mithilfe eines Kommandos inkl. ID möglich. Dies befähigt den Nutzer auch ohne CANopen-Anbindung die Basis-Funktion eines Motors zu Testen und die Funktion zu verifizieren.

Durch die Auswahl der Parameter mithilfe einer ID lässt sich der Kommando-Interpreter effizient um weitere Parameter erweitern, wenn diese erforderlich werden. Bei Bedarf kann der Motion-Controller auch um eine Debug-Ausgabe erweitert werden, welche während des Betriebs gewünschte Parameter ausgibt. Das kann für kommende Optimierungen oder Fehlersuchen einen Vorteil bieten. Die UART könnte für zukünftige Anwendungsfälle auch als Ersatz für die CANopen-Anbindung dienen, diese ergänzen oder deren Funktion übernehmen. Abb. 29 zeigt beispielhaft die Eingabe eines Befehls zum Setzen und Auslesen des Kp-Wertes eines PI-Reglers im Terminal. Die Klasse „SerialConnection“ übernimmt dabei die zuvor genannten Hauptaufgaben.

```

set 1100 4
Command valid, good job! :)
get 1100
read: 4.000

```

Abb. 29: Eingabe zum Setzen und Lesen eines Reglerparameters

Auf Basis der Erklärungen aus den vorangegangenen Kapiteln ergibt sich die in Abb. 30 dargestellte Konfiguration des ioc-Files für die Motion-Controller-Firmware.

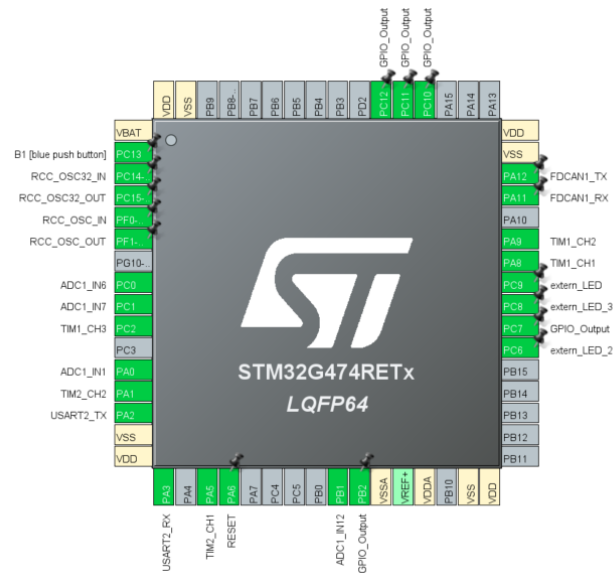


Abb. 30: Konfigurierter ioc-File

4. Test und Validierung

Die Validierung der Motion-Controller-Firmware ist von entscheidender Bedeutung, um eine korrekte Funktion und die Erfüllung der Anforderungen sicherzustellen. Eine umfassende Validierung gewährleistet die reibungslose Funktionsweise und bietet im Betrieb Zuverlässigkeit, sowie Sicherheit. Die Validierung kann in verschiedene Kategorien gegliedert werden, welche im Folgenden näher erläutert werden.

Für die Validierung der Basisfunktionen kommen das Nucleo-G474RE-Board (Abb. 1), die Motor-Treiberstufe X-NUCLEO-IHM07M1 (Abb. 5) und der Micos PRS-110 (Abb. 6) zum Einsatz. Somit wird hier lediglich die Grundlegende Ansteuerung eines DC-Motors, entsprechend den Anforderungen, getestet und validiert. Diese Validierung und das zugehörige Firmwarekonzept bilden eine sichere Grundlage für folgende Projekte.

4.1. Funktionale Validierung

Im Rahmen der funktionalen Validierung werden zunächst die grundlegenden Funktionen überprüft. Dazu gehören Bewegungsbefehle, wie eine Drehung vorwärts oder rückwärts mit konstanter Geschwindigkeit. Der Test erfolgt durch Setzen der Sollposition im Kommando-Interpreter. Der Drehwinkel wird hierbei mit zwei Umdrehungen (720 Grad) angegeben. Das ermöglicht eine konstante Drehgeschwindigkeit zwischen Beschleunigung und Abbremsen. Der Motor fährt wie erforderlich mit konstanter Geschwindigkeit. Die konstante Fahrt wird außerdem über die Frequenz der Lage Sensorik verifiziert. Zusätzlich wird hier das Erreichen der Sollposition geprüft, indem der gefahrene mit dem vorgegebenen Winkel verglichen wird. Die Abweichung beträgt unter einem Grad, was die Funktion der Lageregelung bestätigt.

Nach der konstanten Drehbewegung werden die Beschleunigungs- und Geschwindigkeits-einstellungen bei entsprechender Bewegung miteinbezogen. Hierzu werden erneut die Rahmenbedingungen mit dem Kommando-Interpreter gesetzt. Die Prüfung der Sollgeschwindigkeit erfolgt durch eine visuelle Verifikation und zusätzlich durch eine Frequenzmessung der Lage Sensorik. Beim Beschleunigungstest wird das Richtungswechselverhalten untersucht. Hierbei ist eine hohe Drehgeschwindigkeit erforderlich, um die Resultate erkenntlich zu machen. Auch die Variation der Beschleunigung führt zum geforderten Ergebnis.

Des Weiteren wird die Stromregelung genauer betrachtet. Hierbei wird der Motor bei Drehung gebremst. Bedingt durch die Stromregelung bleibt die Drehgeschwindigkeit konstant und der erhöhte Widerstand wird durch eine äquivalente Stromerhöhung ausgeglichen. Die Stromerhöhung ist

entsprechend den Regler Parametern limitiert, was bei Überschreiten eines Grenzwertes zum Abbremsen und zum Stillstand führt. Bis zum Erreichen des Grenzbereichs folgt die Stromregelung den Vorgaben.

Ist der Betrieb mit zuvor genannten Faktoren gewährleistet, können essenzielle Randfälle getestet werden. Hierzu gehören beispielsweise die Bewegung entlang der minimalen und maximalen Vorgabewerte. In diesem Fall sind das eine Bewegung mit maximaler Geschwindigkeit, Winkeländerungen kleiner ein Grad und mehrere zeitnahe Richtungswechsel zum Test der Maximalbeschleunigung. Durch dieses Vorgehen wird gewährleistet, dass die Firmware auch in Extremsituationen zuverlässig agiert. Auch beim Validieren des Randbedingungsverhaltens entspricht die Funktion der Anforderung.

Der letzte Testfall der funktionalen Validierung stellt die Fehlerbehandlung dar. Hierbei wird geprüft, wie die Firmware auf Fehler, wie beispielsweise Blockaden oder fehlerhafte Vorgabewerte, reagiert.

Insgesamt ist das Ziel einen fehlerfreien und zuverlässigen Prozess sicherzustellen. Bei Vorgabe falscher Werte, beispielsweise negativer Werte für die Maximalbeschleunigung, wird der Wert verworfen und eine Rückmeldung über die Kommando-Schnittstelle ausgegeben. Bei Blockieren des Rotationskörpers vor Erreichen der Sollposition, setzt dieser nach Freigabe seine Bewegung zum Sollwinkel fort.

Zusammenfassend lässt sich sagen, dass die funktionale Validierung erfolgreich ist und ein sicherer und zuverlässiger Betrieb gewährleistet ist.

4.2. Performancetest

Beim Performancetest wird die Leistungsfähigkeit der Software evaluiert. Zunächst wird der Jitter beim Auslösen der Interrupt-Routine gemessen. Abb. 31 und Abb. 32 zeigen das Ergebnis der Jittermessung.

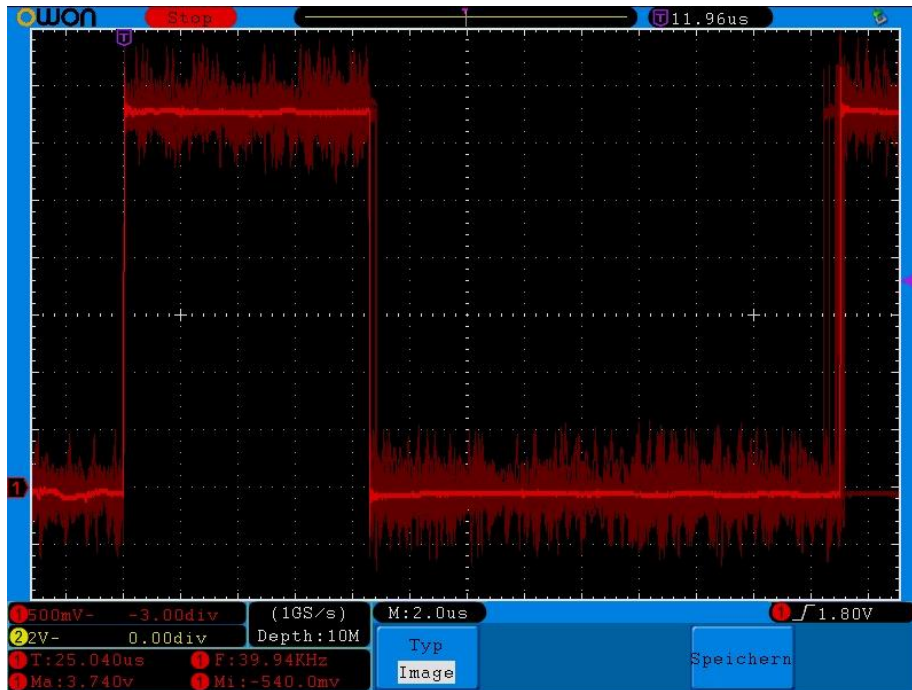


Abb. 31: Jitter-Messung über eine Periode

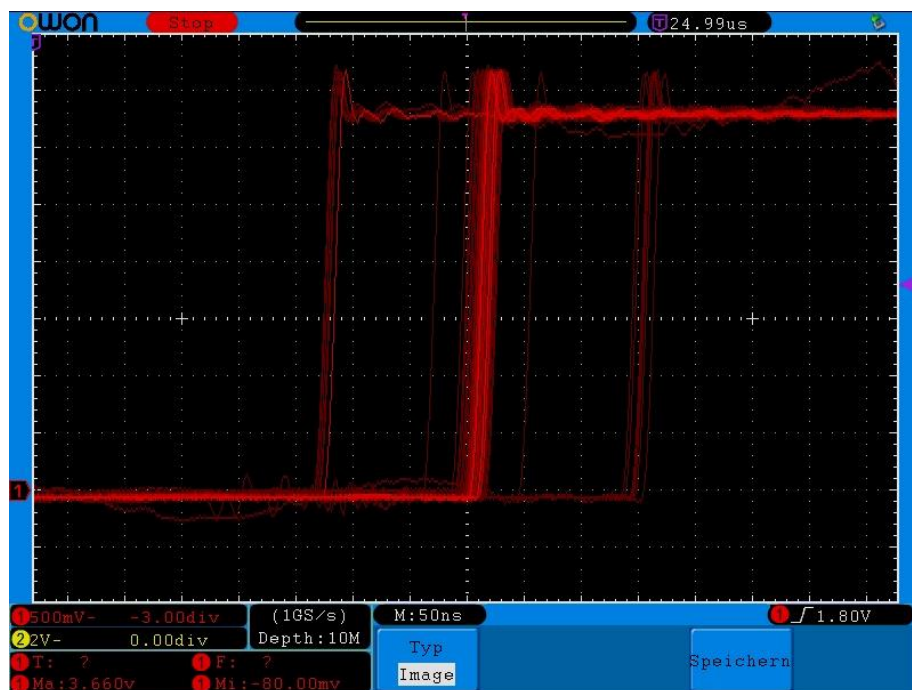


Abb. 32: Detaillierter Zoom der Jitter-Messung bei 50 ns/Div

Vor allem aus Abb. 32 lässt sich erkennen, dass der Jitter, gemessen von der zeitlich frühesten steigenden Flanke bis zur spätesten Flanke, bei ca. 275 ns liegt. Bei einer Regelfrequenz im kHz-Bereich kann dieser Wert als vertretbar eingeordnet werden.

Im nächsten Schritt wird das Erreichen der Regelfrequenz geprüft, um zu gewährleisten, dass die Rechnungen äquidistant und schnell genug ausgeführt werden. Die Vorgabe der Regelfrequenz für die Stromregelung des Motion-Controllers ist 100 kHz. Die tatsächlich erreichte Frequenz der ersten Validierungsphase sind 10 kHz (Abb. 33).

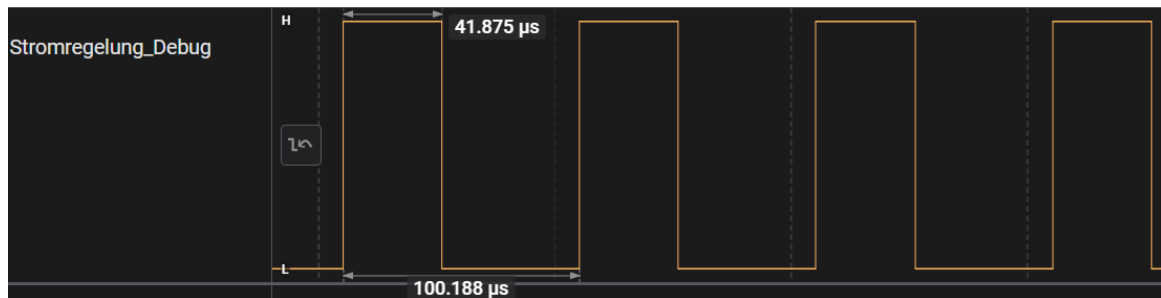


Abb. 33: Stromregelung ohne Optimierungen

Die Dauer der Stromregelung wird durch die Anzeit des Signals repräsentiert. Da diese Frequenz weit von der Vorgabe abweicht und die Funktion gefährdet, wird mit diversen Optimierungen versucht sich der Vorgabe anzunähern. Zu den umgesetzten Optimierungen zählen zum einen Compiler-Optimierungen, welche die Ausführung des Codes auf dem Microcontroller schneller ablaufen lassen. Hierfür wird das Programm, statt im „Debug“-Modus, im „Release“-Modus ausgeführt, was zur Folge hat, dass insgesamt weniger Code erzeugt wird. In Abb. 34 wird das Ergebnis der Optimierung dargestellt. Es wird eine Frequenz von 20 kHz erreicht.

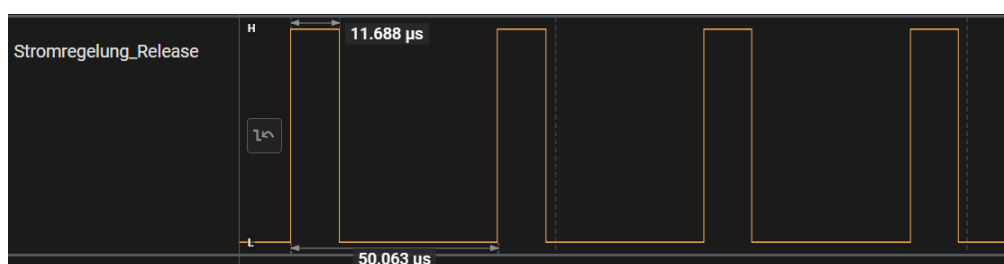


Abb. 34: Stromregelung im Release-Mode

Zum anderen wird der Code optimiert, indem Zustandsabfragen zusammengefasst bzw. umverteilt werden. Zyklisch identische Rechnungen werden, statt in jeder Iteration, nur noch einmal ausgeführt. Dies wird hauptsächlich in den Regelfunktionen bei der Berechnung der Reglerwerte angewendet. Allerdings bewirkt diese Maßnahme nur eine geringfügige Optimierung der Zykluszeit. Eine weitere Möglichkeit die Gesamtgeschwindigkeit des Regelkreises zu erhöhen ist es, alle mit der Regelung zusammenhängenden Funktionen aus dem RAM, statt aus dem Flash-Speicher aufzurufen. Im Code wird dies entsprechend Abb. 35 umgesetzt.

```
void __attribute__((section(".RamFunc"))) Stromregelung(void)
```

Abb. 35: Prototyp für Aufruf einer Funktion aus RAM

Leider trägt auch diese Maßnahme nur geringfügig zu einer Verringerung der Zykluszeit bei. Als letzte Maßnahme werden die Prioritäten der verschiedenen Tasks so angepasst, dass die Stromregelung und die ADC-Konvertierung die höchste Priorität haben. Alle anderen Tasks, wie z.B. die CAN- und UART-Task, besitzen eine geringere Priorität, sind aber untereinander gleichgestellt. Diese Veränderung bewirkt eine deutliche Verbesserung der möglichen Maximalfrequenz, welche nun bei 40 kHz liegt (vgl. Abb. 36)

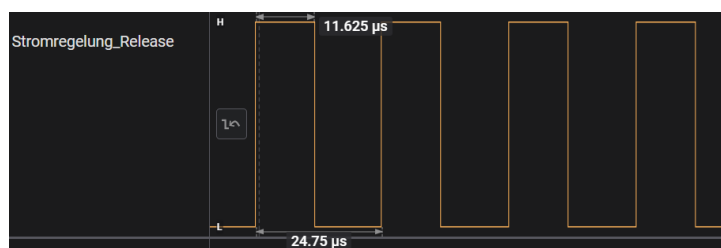


Abb. 36: Stromregelung im Release-Mode nach Anpassung der Prioritäten

Wie sich die Performance bei der Steuerung und Regelung mehrerer Motoren synchron verhält, muss getestet werden, sobald die entsprechende Hardware verfügbar ist. Gegebenenfalls sind dann weitere Optimierungen oder eine geringfügige Änderung des Firmware-Konzeptes erforderlich.

Auch die Funktion der seriellen Schnittstelle und der CANopen Anbindung müssen im Betrieb innerhalb der Vorgabezeit erfolgen. Da keine übergeordnete Schnittstelle mit CANopen-Anbindung verfügbar ist, wird für den Test das zyklische Senden eines Heartbeat als Testsignal herangezogen. Näheres wurde bereits im Kapitel „3.4.4. CANopen“ erläutert. Auch die serielle Schnittstelle wird mit Kommandos belastet und die Reaktion auf hochfrequente Eingaben wird geprüft. Hochfrequent bedeutet in diesem Fall eine Frequenz von einem Hz, da bisher nur von einer manuellen, humanen Interaktion mit der Schnittstelle ausgegangen wird und somit eine Kommandoeingabe pro Sekunde als Limit angesetzt werden kann.

Da sowohl CAN- als auch UART-Schnittstelle gemäß den Erwartungen agieren, kann deren Funktion als validiert angenommen werden.

4.3. Kompatibilitätstest

Der Kompatibilitätstest stellt sicher, dass die Firmware reibungslos mit verschiedenen Hardware- und Software-Komponenten funktioniert. Zu den Hardwarekomponenten zählen Sensoren und Aktuatoren. Durch den Modularen Aufbau und die Konfigurationsmöglichkeiten der Schnittstellen des Nucleo-Boards lassen sich effizient Anpassungen vornehmen und umsetzen. Da bisher nur der im Kapitel „3.1. Hardware“ beschriebene Umfang zu Verfügung steht, ist eine eindeutige Aussage nicht möglich. Es kann davon ausgegangen werden, dass eine Adaption und Erweiterung von Sensoren durch ausreichend verfügbare GPIO-Pins und Konfigurationsalternativen gegeben sind. Der Anschluss und die Ansteuerung eines AC (Alternating Current) -Motors erfordern einige Änderungen, da das aktuelle Konzept nur die Verwendung eines DC-Motors voraussetzt.

Die Verwendung einer anderen Steuereinheit ist nach Anpassung der Pins auf das entsprechende Layout möglich. Zudem können Änderungen der ADCs und der zugehörigen Implementierung anstehen, falls andere shunts oder Referenzwerte Verwendung finden.

Die Entwicklungsumgebung ist für alle gängigen Betriebssysteme verfügbar und bietet bei gleicher Versionierung äquivalente Möglichkeiten der Entwicklung. Auch die Funktion der seriellen Schnittstelle ist nach Konfiguration der Datenrate und des Formats problemlos umsetzbar.

5. Zusammenfassung und Ausblick

In der hier beschriebenen Entwicklung der Motion-Controller Software war es möglich, die im Studium erlernten Grundlagen aus Gebieten der Signalverarbeitung, Regelungstechnik und der objektorientierten Programmierung anzuwenden und zu vertiefen. Die breit gefächerten Themengebiete dieses Projektes haben es möglich gemacht vorhandenes Wissen zu festigen und weitere neue Fertigkeiten und Fähigkeiten in diesen Bereichen zu erlernen.

Dies war in erster Linie durch die Forderung und Förderung des Betreuers möglich, bei welchem wir uns an dieser Stelle bedanken wollen.

Auch die Arbeit als Projektgruppe und die Interaktion mit anderen Projekten war interessant und lehrreich. Zudem hat es motiviert ein bestmögliches Ergebnis zu erreichen.

Das Ziel des Projektes, eine lauffähige Motion-Controller-Firmware für einen DC-Motor zu entwickeln, ist erreicht worden. Die Regelkreise, sowie der Kommando-Interpreter konnten erfolgreich umgesetzt werden. Die Implementierung von CANopen, sowie Test der Basisfunktionen, waren erfolgreich. Die Steuerung über CANopen konnte jedoch nicht getestet und final implementiert werden, da die übergeordnete Steuerung noch nicht verfügbar war. Die zu Beginn angestrebte Regelfrequenz von 100 kHz für die Stromregelung konnte nicht erreicht werden, da ein zuverlässiger Betrieb bei Frequenzen über 40 kHz nicht gewährleistet ist. Da ein Hauptziel der Entwicklung die Echtzeitfähigkeit im Betrieb ist, wird dieses gegenüber der Regelfrequenz priorisiert. Zudem bietet eine Bandbreite von 40 kHz bereits das Potential einer hohen Dynamik.

Dieses Projekt bietet die Basis für zukünftige Projekte, die sich mit der Weiterentwicklung des Motion-Controllers beschäftigen. Durch die Anfertigung dieses Berichtes und ein GitHub-Repository für das Projekt, wird der Einstieg in die Thematik für nachfolgende Projektgruppen erleichtert. Der Code wird zusätzlich an diversen Stellen durch Kommentare im Sourcecode und Readme-Files erklärt. Zudem sind die wichtigsten Informationen zur Firmware im Wiki des Ilias-Kurses „Informationstechnische Systeme“ von Herr Baas zusammengetragen. Wünschenswerte Erweiterungen wären z.B. eine getestete und validierte Steuerung über CANopen mit der oben genannten übergeordneten Steuerung oder die Implementierung eines Datenrecorders in die vorhandene Firmware. Außerdem besteht die Möglichkeit durch Anpassungen und Optimierungen noch höhere Regelfrequenzen bei gleichbleibend hoher Zuverlässigkeit zu erreichen.

Literaturverzeichnis

- [1] T. Baas, „Aufgabenstellung FW MotionController[250]“. 27. Februar 2024.
- [2] STMicroelectronics, „UM2505 STM32 Nucleo-64 boards (MB1367)“. 2. März 2024. [Online]. Verfügbar unter: <https://www.st.com/en/evaluation-tools/nucleo-g474re.html#documentation>
- [3] T. Baas, „PWM-Ansteuerung“. 2. März 2024.
- [4] STMicroelectronics, „Getting started with the X-NUCLEO-IHM07M1 motor driver expansion board based on the L6230 for STM32 Nucleo“. 30. März 2024. [Online]. Verfügbar unter: <https://www.st.com/en/ecosystems/x-nucleo-ihm07m1.html#documentation>
- [5] Micos, „Precision Rotation Stage PRS-110“. [Online]. Verfügbar unter: https://www.micosusa.com/old/PRS_110.html
- [6] T. Baas, „Echtzeitansteuerung“. 28. März 2024.
- [7] S. Luber und N. Litzel, „Was ist FreeRTOS?“, 22. März 2024. [Online]. Verfügbar unter: <https://www.bigdata-insider.de/was-ist-freertos-a-1043163/>
- [8] TU Darmstadt, „Assoziation, Aggregation, Komposition“. 19. März 2024. [Online]. Verfügbar unter: https://www.iim.maschinenbau.tu-darmstadt.de/kursunterlagen_archiv/ikt_ws1415/05/Theorie/assoziation_aggregation_komposition.html
- [9] Tutorialspoint, „Interfaces in C++ (Abstract Classes)“. [Online]. Verfügbar unter: https://www.tutorialspoint.com/cplusplus/cpp_interfaces.htm
- [10] „Design Trajectory with Velocity Limits Using Trapezoidal Velocity Profile“. [Online]. Verfügbar unter: <https://de.mathworks.com/help/robotics/ug/design-a-trajectory-with-velocity-limits-using-a-trapezoidal-velocity-profile.html>
- [11] T. Baas, „Implementierung Regelung“. 8. März 2024.
- [12] „CANopen“. 16. Dezember 2023. [Online]. Verfügbar unter: <https://de.wikipedia.org/wiki/CANopen>
- [13] CiA, „CAN FD - The basic idea“, Dez. 2024. [Online]. Verfügbar unter: <https://www.can-cia.org/can-knowledge/can/can-fd/>
- [14] T. Majerle und H. Jafarzadeh, „CanOpenSTM32“. 22. Dezember 2023. [Online]. Verfügbar unter: <https://github.com/CANopenNode/CanOpenSTM32>
- [15] P. Decker, „Wege vom klassischen CAN zum verbesserten CAN FD“. 18. Dezember 2023. [Online]. Verfügbar unter: https://cdn.vector.com/cms/content/know-how/_technical-articles/CAN_FD_ElektronikAutomotive_201304_PressArticle_DE.pdf
- [16] R. Cornelius, „CANopenEditor“. 22. Dezember 2023. [Online]. Verfügbar unter: <https://github.com/CANopenNode/CANopenEditor>

Anhang

Klassendiagramm



Befehlsliste des Kommando-Interpreters

Befehlsaufbau: set/get ID Wert

Beispiel: set 1100 4

→ Setze Kp Wert von PI-Regler der Achse 1 auf 4

Zusammensetzung der ID:

10³ (erste Stelle)	10² (zweite Stelle)	10¹ (dritte Stelle)	10⁰ (vierte Stelle)
0: System-Befehl	0: gültig 1 – 9: ungültig	0: gültig 1 – 9: ungültig	0: Firmwareversion 1: Switch On 2: Ready 3: Operation 4: Status 5: Control 6: Befehlsliste 7 – 9: ungültig
1: Achsen-Befehl	0: ungültig 1: Achse 1 2: Achse 2 3 – 9: ungültig	0: PI-Regler	0: Kp
			1: Tn
			2 – 9: ungültig
		1: PID-Regler	0: Kp
			1: Tn
			2: Tv
			3 – 9: ungültig
		2: Position 3: Geschwindigkeit 4: Beschleunigung 5: Verzögerung 6: PWM 7 – 9: ungültig	0: gültig 1 – 9: ungültig