Hochschule München

Projektbericht im Fach “Projekt Mechatronik“

Implementierung einer Steuerung eines Inverters

Gruppe: Software
Name: Martin Sölkner
Studiengruppe: ELM1

Übersicht

Der vorliegende Code hat das primäre Ziel, einen Inverter zu steuern. Zu Beginn wurde eine umfassende Analyse durchgeführt, um die spezifischen Anforderungen und Aufgaben der Software des Inverters zu bestimmen. Diese umfassten eine phasenkorrekte Sinus-Kommutierung sowie eine Zustandsmaschine (*State Machine*). Die anfängliche Implementierung basiert auf dem *Open-Loop*-Modus. Die Zustandsmaschine wurde so konzipiert, dass sie fünf Zustände aufweist.

Die Implementierungsstrategie folgte dem objektorientierten Ansatz, bei dem nahezu jede Aufgabe in einer eigenen Klasse gekapselt wurde. Dort, wo ein objektorientierter Ansatz nicht zielführend schien, wurde ein Namensraum (*Namespace*) verwendet.

Strukturierung des Programmcodes

Der Code wurde strukturiert, indem funktionell zueinander gehörende Elemente in separate Quellcodedateien (*Source Files*) aufgeteilt wurden. Die Struktur ist wie folgt gestaltet:

- conf.hpp: Diese Datei enthält sämtliche Makros, die sich auf Eingänge, Maschinenkennwerte und Businformationen beziehen.
- foc_pwm.hpp: Diese Datei umfasst die Funktionen der feldorientierten Regelung (*Field Oriented Control* - FOC). Zusätzlich beinhaltet sie eine Klasse für den Zeitgeber (*Timer*), der das Auslesen der Sensoren zeitlich steuert.
- interface.hpp: Die in dieser Header-Datei enthaltene Struktur dient dem Datenaustausch zwischen den Klassen. Sie enthält die Zeigerdeklarationen für die einzelnen Sensorklassen und Busse sowie wichtige Variablen.
- IState.hpp: Diese Interface-Klasse dient als Oberklasse für alle Zustandsinstanzen.
- sensor_func.hpp: Diese Header-Datei enthält die Methoden für die Sensoren, einschließlich des ADC und des Temperatursensors, sowie eine Funktion zum Auslesen des Potentiometers.

- Tstate_machine.hpp: Diese Klasse stellt die Oberklasse der Zustandsmaschine dar. Sie verwendet Templates, um eine Code-Wiederverwendbarkeit zu ermöglichen.
- state_machine.hpp: Diese Klasse erbt von der Klasse *TState_Machine* und spezifiziert die Zustandsmaschine durch spezifische Attribute sowie Methoden, die ausschließlich für diese bestimmte Zustandsmaschine entwickelt wurden. Darüber hinaus enthält sie die Zustandstabelle (*State-Transition Table*), welche die Kontrolle der einzelnen Zustands-übergänge (*State Transitions*) ermöglicht.
- states.hpp: In dieser Header-Datei sind die einzelnen Zustandsklassen implementiert. Diese Klassen erben von der Oberklasse *ISTate*, sodass alle Zustandsklassen dieselben Methoden besitzen.

Umsetzung der einzelnen Module

Aufbau der State Machine

Bei diesem Projekt war die Entwicklung einer Zustandsmaschine, oder *State Machine*, eine der zentralen Anforderungen. Im Rahmen des Betriebs der State Machine sind verschiedene Zustände (States) vorgesehen. Diese sind in Abbildung 1 dargestellt.

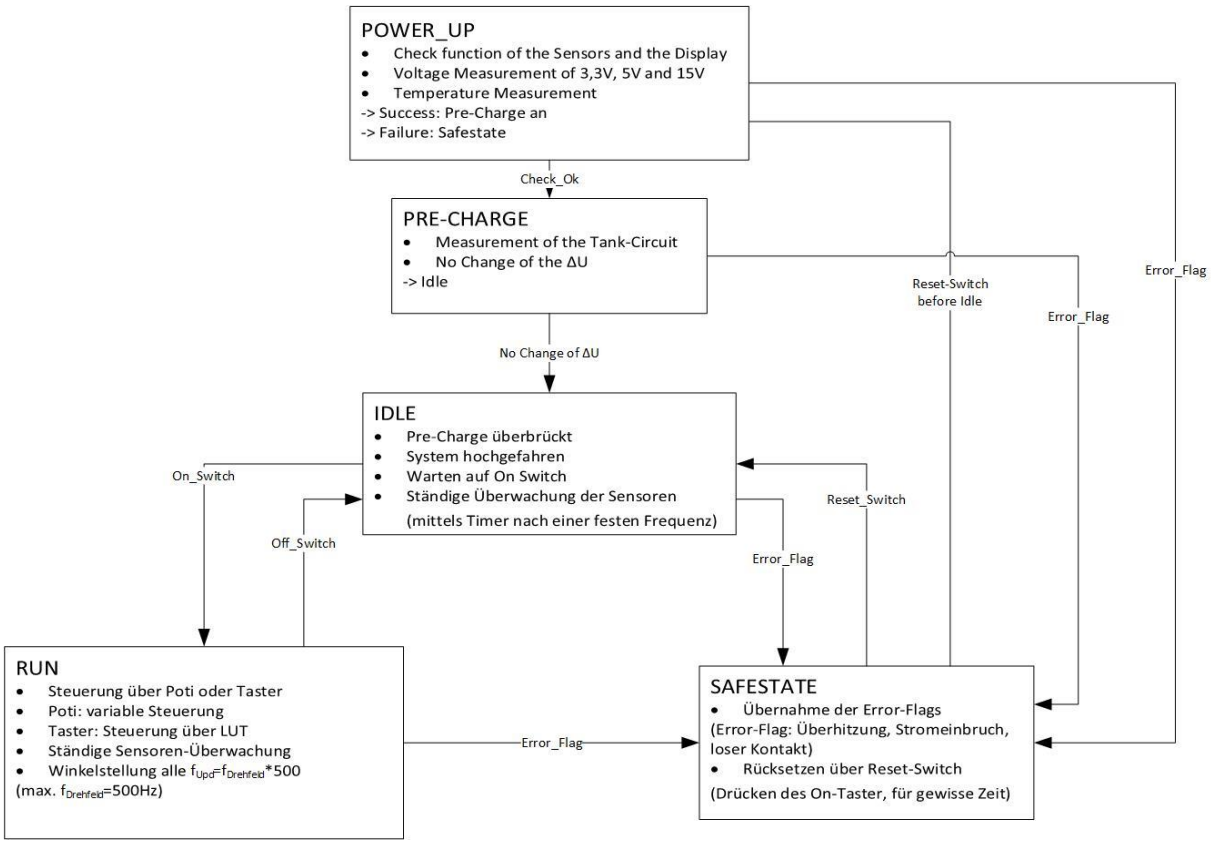

Abbildung 1 - Übersicht über das Zustandsdiagramm

Der erste Zustand ist der *Power_Up* State. In diesem Zustand werden zunächst die beiden Busse (*One Wire* und *Two Wire*) und anschließend die dazugehörigen Sensoren (ADC und

Temperatursensor) hochgefahren. Darüber hinaus finden in diesem Zustand bereits erste Messungen mit der Peripherie statt.

Nach erfolgreicher Durchführung aller Prüfungen erfolgt der Wechsel in den *Pre_Charge*. Hier wird die Zwischenkreisspannung U_{Tank} ständig gemessen. Sobald das Delta von ΔU_{Tank} nicht weiter ansteigt, wird der *Pre-Charge* (Vorlade) - Widerstand durch ein Relay überbrückt und es erfolgt ein Wechsel in den Standby Zustand.

Im Standby Zustand wird ein *Timer* aktiviert, der alle 500 Hz eine Messung startet, um alle wichtigen Parameter zu überwachen. Dabei werden lediglich zwei Messwerte pro Zyklus ausgelesen, um Interrupts zeitlich nicht zu blockieren. Darüber hinaus, wird ein Taster ausgelesen, der im Falle der Betätigung in den *Run* Zustand wechselt.

Im *Run* Zustand kann entweder über einen Taster mit vordefinierten Werten oder über ein Potentiometer die Drehzahl eingestellt werden. Durch Auslösen des Stopp Tasters erfolgt der Wechsel zurück in den *Standby* Zustand.

Neben diesen Zuständen existiert ein Safety State. Dieser Zustand wird aktiviert, sobald ein ungewöhnlicher Wert auftritt und führt zum sofortigen Stopp der Maschine. Dieser Sicherheitszustand muss ausdrücklich vom Benutzer zurückgesetzt werden. Nach der Rücksetzung wird die Maschine in den *Pre_Charge* Zustand versetzt und beginnt den Hochfahrprozess erneut.

Implementierung der State Machine

Der ursprüngliche Ansatz basierte auf einer einfachen Switch-Case-Funktion, doch dieser Ansatz stellte sich als problematisch heraus. Insbesondere die hintereinander Reihung von mehreren Zuständen und deren Logiken beeinträchtigten die Wartbarkeit des Programms. Darüber hinaus war es schwierig, die Kontrolle über die Zustandsübergänge, den sogenannten *State Transitions*, zu behalten. Ein Beispiel ist die Überprüfung, ob ein direkter Übergang von *Power_Up* zu *Run* zulässig ist.

Zur Lösung dieser Herausforderungen wurde eine gekapselte Zustandsmaschine implementiert. Diese Methode bietet den Vorteil, dass alle Anweisungen gekapselt sind und eine integrierte Funktion zur Überprüfung der Legitimität der Zustandsübergänge vorhanden ist. Die Abbildung 2 soll einen Überblick über die Klassenstruktur, der State Machine liefern.

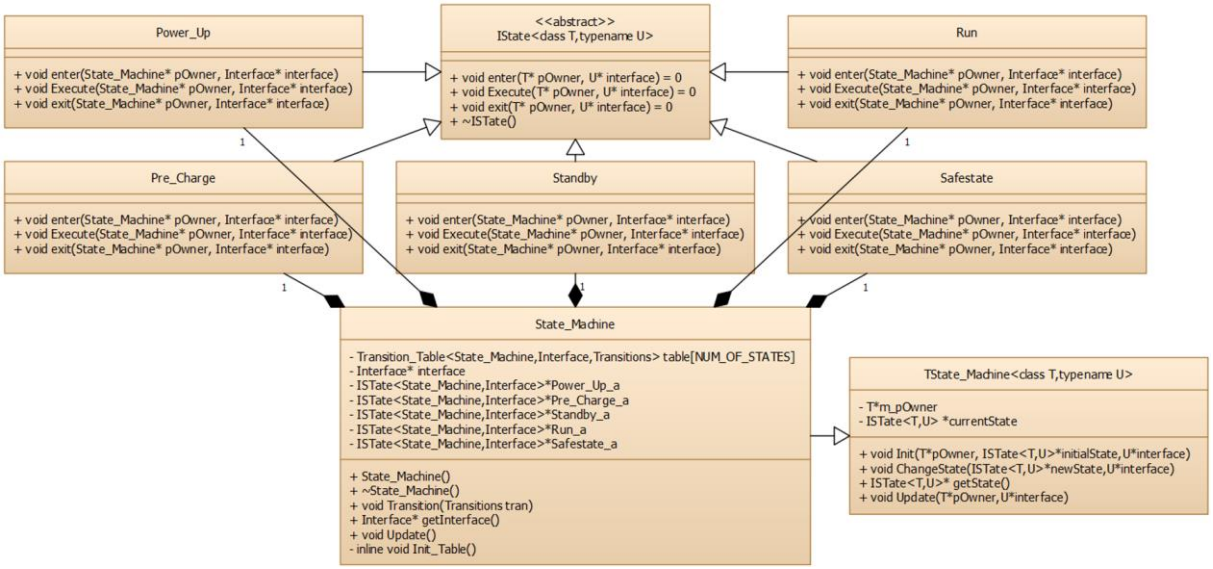

Abbildung 2 - Übersicht über das Klassendiagramm

Jeder Zustand ist in seiner eigenen Klasse definiert. Jede Klasse hat eine *Enter*-Methode, die aufgerufen wird, wenn der Zustand betreten wird, eine *Execute*-Methode, die bei jedem Aufruf von der *State_Machine* Klasse ausgeführt wird, und eine *Exit*-Methode, die aufgerufen wird, wenn der Zustand verlassen wird. Um nicht jede Klasse einzeln aufrufen zu müssen, nutzen wir das Prinzip der Polymorphie. Das bedeutet, wir behandeln Objekte verschiedener Klassen, als wären sie vom gleichen Typ. Die spezifische Implementierung einer Methode wird zur Laufzeit bestimmt. Hierzu haben wir eine Interface-Klasse mit den drei rein virtuellen Methoden (*Enter, Execute* und *Exit*) erstellt. Diese Methoden sind im Compiler definiert, ihre Implementierung erfolgt jedoch in den jeweiligen Zustandsklassen.

Die Zustandsmaschine basiert auf der selbst implementierten Klasse *TState_Machine*. Diese Klasse beinhaltet zwei Templates: eines für die Spezialisierung der Zustandsmaschine und eines für die Datenstruktur, die Daten zwischen den Zuständen austauscht.

Des Weiteren besitzt die Klasse zusätzliche Attribute wie *currentState*, welches den aktuellen Zustand speichert, und *m_pOwner*, welcher als Zeiger auf die spezialisierten State Machine-Instanzen dient. Die Klasse verfügt über Methoden wie *Init*, um einen Anfangszustand festzulegen und *m_pOwner* eine Instanz zuzuweisen, *ChangeState*, um den Zustand zu ändern und dabei die *exit*- und *enter*-Methoden aufzurufen, und *Update*, um die *Execute*-Methode der aktuellen State-Klasse auszuführen.

In der Klasse *State_Machine*, die von *TState_Machine* erbt, werden nun die exakten Parameter des Zustandsautomaten festgelegt. Dafür verfügt diese Klasse über Attribute, wie Zeiger auf jede einzelne Zustandsklasse und eine Zustandstabelle. Der Konstruktor *State_Machine()* erstellt neue Instanzen der einzelnen State-Klassen. Die *Transition*-Methode ruft die *ChangeState*-Methode der *TState_Machine* Klasse auf und überprüft anhand der Zustandstabelle, ob der Zustandsübergang legitim ist.

Darstellung eines Objektaufrufs

Anhand der folgenden Ausführungen soll beispielhaft die Steuerung einer Zustandsmaschine mittels einer Instanz der Klasse State_Machine beschrieben werden.

Durch den Befehl State_Machine fsm; wird eine Instanz dieser Klasse erzeugt und der Zustand der Maschine wird auf den im Konstruktor spezifizierten Anfangszustand gesetzt.

Der Wechsel von einem Zustand zu einem anderen erfolgt durch den Aufruf der Methode *Transition(transition)*. In Abbildung 3 wird die Abfolge dieses Zustandswechsels visualisiert.

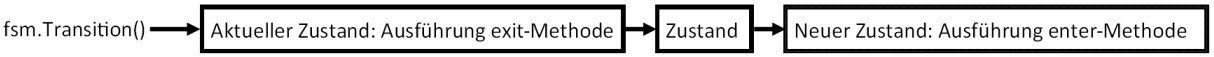

Abbildung 3 - Illustration des Zustandswechsels

Die Methode leitet den Übergang von einem Zustand zum nächsten ein. Zunächst wird die im Argumentwert übergebene *Transition* geprüft und bei Erfolg, die *exit()*-Methode des aktuellen Zustands ausgeführt, woraufhin der Wechsel zum neuen Zustand stattfindet. Anschließend wird die *enter()*-Methode des neuen Zustands aufgerufen.

Die *Update()*-Methode wird aufgerufen, um die spezifischen Logiken und Algorithmen, die in jedem Zustand definiert sind, auszuführen. Abbildung 4 stellt diesen Prozess anhand des *Pre_Charge*-Zustands dar, aber dieselbe Methode gilt auch für die anderen Zustände.

Abbildung 4 - Veranschaulichung der Ausführung der Execute-Methode

Zu Beginn des Prozesses wird das im currentState gespeicherte Objekt untersucht. Abhängig vom dynamischen Typ des Objekts wird eine entsprechende Methode aufgerufen - in diesem Beispiel handelt es sich um den *Pre_Charge*-Zustand. Innerhalb der Execute-Methode wird dann die für den jeweiligen Zustand definierte Logik ausgeführt.

Erzeugung eines PWM-Ausgangssignals

Ein zentraler Bestandteil dieses Projekts war die Implementierung der sinusförmigen Kommutierung für den Asynchronmotor. Zur Erzeugung der erforderlichen Pulsweiten-modulation (PWM) wurde die SimpleFOC - Bibliothek verwendet, die einen Algorithmus für die feldorientierte Steuerung bereitstellt. Für ein tiefergehendes Verständnis der Nutzung und Funktionsweise dieser Bibliothek wird auf die zugehörige Dokumentation verwiesen, die umfassende Erläuterungen zu den bereitgestellten Methoden und Funktionen bietet.

In diesem Projekt wurde die Open-Loop-Steuerung eingesetzt. Außerdem wird die Dreiphasen-Halbbrücke mittels sechs separater PWM-Signale gesteuert. Die beiden Methoden zur Erzeugung dieser Signale wurden in einem eigenen Namenspace zusammengefasst. Die Init()-Funktion innerhalb dieses Namensraums nimmt die Initialisierung aller relevanten Motorparameter und Treiber vor.

Um den Motor zu steuern, wird die Methode *move()* verwendet. Sie nimmt ein Argument namens *new_target* entgegen, welches die Winkelgeschwindigkeit repräsentiert, mit der der Motor bewegt werden soll. Die Methode befindet sich in der selbst implementierten Funktion *PWM_Generation(BLDCMotor* motor, float freq)*, welche als Parameter einen Zeiger auf die Instanz des Motorobjekts und eine Frequenz benötigt.

Verwendete Sensoren

Im Zuge des Projekts kamen sowohl der MAX31820 Temperatursensor als auch der TLA2528 Analog-Digital-Wandler zum Einsatz.

Der MAX31820 ist ein digitaler Temperatursensor, der präzise Messungen ermöglicht und über einen *One-Wire-Bus* gesteuert wird. Zur Steuerung dieses Sensors wurde die DallasTemperature-Bibliothek verwendet, die eine Vielzahl an Methoden zur Vereinfachung des Temperaturauslesens bereitstellt. Die dabei verwendeten Methoden wurden in der Funktion *readTemperature* zusammengefasst.

Der TLA2528 wiederum ist ein 12-Bit-Analog-Digital-Wandler (ADC) mit acht Kanälen, der über eine I2C-Schnittstelle gesteuert wird. Mithilfe des TLA2528 werden diverse Strom- und Spannungsmessungen durchgeführt, die in Abbildung 5 dargestellt sind.

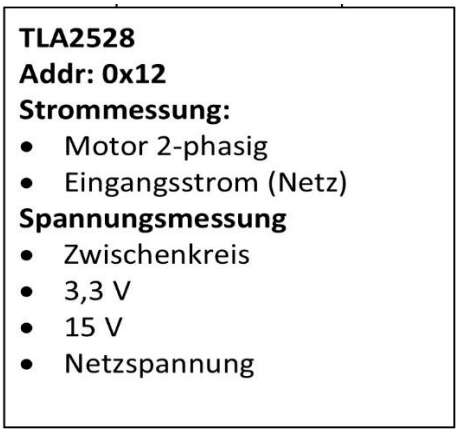

Abbildung 5 - Übersicht über die Aufgaben des TLA2528

Zu den durchgeführten Messungen zählen unter anderem die des Stroms in zwei Phasen, wobei der Strom in der dritten Phase berechnet wird. Zudem wird der Eingangsstrom des Stromnetzes erfasst. Des Weiteren überwacht der TLA2528 durch vier separate Messungen die Spannung am Zwischenkreis, am Netz, die IGBT-Gate-Spannung sowie die digitale Spannung.

Für den TLA2528 war keine vorgefertigte Bibliothek verfügbar, weshalb die notwendigen Funktionen zur Steuerung des TLA2528 eigenständig entwickelt wurden.

Dies umfasst den Startprozess des ADCs, der das Einstellen eines *Prescaler* und das Aktivieren eines Mittelwert-Filters für genauere Messergebnisse beinhaltet. Anschließend erfolgt die interne Kalibrierung des Geräts, um mögliche Offset Fehler zu eliminieren. Nach dem erfolgreichen Abschluss dieser Startsequenz ist der ADC bereit für die erste Messung.

Die Messung wird initiiert, indem die ID des auszulesenden Kanal über den I2C gesendet wird. Anschließend wird ein Lesevorgang des Registers angefordert, in welches der ADC-Wert nach der Messung gespeichert wird. Diese Vorgang startet den Messprozess und schickt nach dessen Abschluss die Daten über den I2C Bus zurück.

Ausgabe über den Display

Bei der Darstellung aller relevanten Werte kommt ein 1,2-Zoll-Display zum Einsatz. Dieses wird durch die Verwendung der Bibliotheken Adafruit_SSD1306.h und Adafruit_GFX.h angesteuert. Dank vorhandener Dokumentationen zu diesen Bibliotheken können Nutzer detaillierte Informationen über die verwendeten Methoden und Funktionen erhalten. Aus diesem Grund wird in diesem Projektbericht nicht weiter darauf eingegangen.

Ausblick des Projekts

Das Software hat bereits wichtige Meilensteine erreicht, darunter die erfolgreiche Implementierung der State Machine, die eine zentrale Rolle im Betrieb des Inverters spielt, sowie die Feldorientierte Regelung (FOC) im Open-Loop-Modus mit der Unterstützung der SimpleFOC-Bibliothek. Es ist jedoch zu beachten, dass es zu Kompatibilitätsproblemen zwischen dieser Bibliothek und der verwendeten Hardware gekommen ist.

Blickt man auf die weitere Entwicklung des Projekts, stehen noch einige Aufgaben und Ziele an. Eine herausragende Priorität ist dabei die Implementierung der Sensorlogik. Die korrekte und effiziente Erfassung und Verarbeitung der Sensordaten ist von entscheidender Bedeutung für die zuverlässige Funktion des Systems.

Ein zusätzliches, wünschenswertes Feature wäre die Implementierung einer temperaturgesteuerten Kühlung. Hier könnten Lüfter aktiviert werden, wenn die Temperatur einen bestimmten Schwellenwert überschreitet. Dies würde nicht nur zur Zuverlässigkeit, sondern auch zur Langlebigkeit des Systems beitragen.

Ein weiteres wichtiges Ziel ist die Implementierung der *Safestate*-Logik. Dafür müssen noch diverse Fehlerfälle einprogrammiert werden, die den *Safestate* auslösen und ein Mechanismus, um diesen wieder zurückzusetzen.

Außerdem muss noch der abschließende Test des Gesamtsystems stattfinden.

Weiterführende Links

Simple FOC

- https://docs.simplefoc.com/
- https://github.com/simplefoc

Dallas Temperature

- https://github.com/milesburton/Arduino-Temperature-Control-Library

TwoWire

- https://docs.arduino.cc/learn/communication/wire#wire-library

Adafruit Display

- https://github.com/adafruit/Adafruit_SSD1306

Github des Projekts

- https://github.com/Eg0st/MecProInverter