CTU

CZECH TECHNICAL UNIVERSITY IN PRAGUE

F3

Faculty of Electrical Engineering
Department of Control Engineering

Master's Thesis

Open-source Motion Control on Mid-range and Small FPGAs

Jakub Janoušek
Cybernetics and Robotics

August 2024
Supervisor: Ing. Pavel Píša, Ph.D.

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Janoušek Jakub**	Personal ID number:	**483444**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Control Engineering**		
Study program:	**Cybernetics and Robotics**		

II. Master’s thesis details

Master’s thesis title in English:

Open-source Motion Control on Mid-range and Small FPGAs

Master’s thesis title in Czech:

Otevřené systémy řízení pohonů na středních a malých programovatelných obvodech

Guidelines:

Motion control is fundamental for a wide range of robotic, space, and industrial applications. The project aims to evaluate combinations of FPGA-based control electronics with GNU/Linux system and smaller RTOS as is NuttX system.
1) Familiarize with Zynq 7000 based MZ_APO board and microzed-mc-1 PMSM motor control FPGA design and power stages
2) Propose and evaluate the procedure for current sensor calibration and integrate current transformations into PXMC and pysimCoder control design
4) Test how the control PMSM system can be reproduced on a cheap entry and hobbyist-level ICE-V board and propose a combined design for ESP32C3 microcontroller-based electronics
5) Document developed software and hardware setup

Bibliography / sources:

[1] Prudek. M.: Brushless motor control with Raspberry Pi board and Linux; Bachelor Thesis; CTU; Prague 2015
[2] Jeřábek, M.: Open-source and Open-hardware CAN FD Protocol Support; Jeřábek Martin; 2018; Master Thesis; CTU; Prague 2016
[3] Belda, K.; Píša, P.: Explicit Model Predictive Control of PMSM Drives In: 2021 IEEE 30th International Symposium on Industrial Electronics (ISIE). Piscataway: IEEE Industrial Electronics Society, 2021. ISSN 2163-5145. ISBN 978-1-7281-9023-5.
[4] Jeřábek, M.; Ille, O.; Píša, P.; Novák, J.: CTU CAN FD Core integration to Zynq-7000 system [Prototype] 2018. online: https://canbus.pages.fel.cvut.cz/
[5] CTU GNU/Linux Simulink Target; online https://lintarget.sourceforge.net/

Name and workplace of master’s thesis supervisor:

Ing. Pavel Píša, Ph.D. Department of Control Engineering FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **05.02.2024** Deadline for master's thesis submission: **15.08.2024**

Assignment valid until: **21.09.2025**

Ing. Pavel Píša, Ph.D.	prof. Ing. Michael Šebek, DrSc.	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student’s signature

Acknowledgement

First and foremost, I would like to thank my supervisor, Dr. Píša, for his patience and countless hours spent teaching and helping me. I would also like to thank my mom, dad, and my brother for their support and help throughout my studies. And last but not least, my friends and especially my girlfriend for their patience and help.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, August 15, 2024

. . . .

Abstrakt / Abstract

Tato práce se zaměřuje na vývoj open source systému pro řízení pohybu pomocí FPGA střední a malé třídy, s důrazem na řízení synchronních motorů s permanentními magnety (PMSM). Projekt integruje řídicí elektroniku na bázi FPGA s operačním systémem reálného času NuttX a prostředím GNU/Linux a vytváří tak flexibilní a nákladově efektivní řešení pro aplikace v robotice nebo průmyslové automatizaci.

Mezi klíčové aspekty práce patří analýza principů řízení pohybu, vývoj a implementace metody kalibrace Hallova proudového senzoru a návrh a implementace nové architektury řídicího systému s využitím bezdrátové desky ICE-V, která kombinuje mikrokontrolér ESP32-C3 s FPGA iCE40. Navržený systém je pak otestován s knihovnou Portable, highly eXtendable Motion Control (PXMC) a grafickým nástrojem pro návrh řídicích systémů a generování kódu pysimCoder.

Klíčová slova: Řízení pohonů, PMSM, NuttX, otevřené systémy, FPGA, pysimCoder

Překlad titulu: Otevřené systémy řízení pohonů na středních a malých programovatelných obvodech

This thesis focuses on the development of an open source motion control system using mid-range and small FPGAs, with a particular emphasis on controlling Permanent Magnet Synchronous Motors (PMSM). The project integrates FPGA-based control electronics with the NuttX real-time operating system and the GNU/Linux environment to create a flexible and cost-effective solution for applications in robotics or industrial automation.

Key aspects of the work include the analysis of motion control principles, the development and implementation of Hall current sensor calibration method, and the design and implementation of a new control system architecture using the ICE-V Wireless board, which combines the ESP32-C3 microcontroller with the iCE40 FPGA. The designed system is then tested with the Portable, highly eXtendable Motion Control library (PXMC) and the graphical tool for control system design and code generation pysimCoder.

Keywords: Motion control, PMSM, NuttX, Open-source, FPGA, pysimCoder

Contents /

Tables / Figures

Chapter 1
Introduction

Motion control systems play a critical role in a wide range of applications, from robotics and industrial automation to aerospace technology. These systems enable precise control over the movement and positioning of machines and devices, which is essential to achieve high performance and accuracy in various tasks. Among the different types of motor used in these systems, Permanent Magnet Synchronous Motors (PMSM) are valued for their efficiency, reliability, and high power density. These characteristics make PMSM a preferred choice in scenarios where precise control of speed and position is important.

There is a growing interest in using open source technology to develop better and more affordable motion control systems. The combination of open-source software with Field Programmable Gate Arrays (FPGAs) creates a flexible platform for building custom motion control systems. FPGAs can be reprogrammed to handle specific tasks, making them ideal for implementing specialized control methods. Open-source software offers several advantages, including adaptability, cost-effectiveness, and a large community of developers who contribute to continuous improvement and support, making it easier to develop, deploy, and maintain these systems.

In this thesis, a motion control system for PMSM motors was developed using midrange and small FPGAs, specifically focusing on open-source solutions. The work involved the development of a current sensor calibration method to ensure accurate feedback, which is critical for effective motor control. The thesis also describes the design and implementation of a new control system architecture using the ICE-V Wireless board, integrating the ESP32-C3 microcontroller with the iCE40 FPGA. Finally, custom control software was developed and tested using the Portable, highly eXtendable Motion Control (PXMC) library and the open source graphical tool for control system design and code generation pysimCoder.

Chapter 2
Motion Control

Motion control is a system used to manage the movement of machines or mechanical systems. It involves controlling the speed, position, and acceleration of motors or actuators to achieve desired outcomes. The basic structure of a motion control system typically includes three main components:

- **Controller**: The controller receives input signals and generates outputs for the power stage. It processes information from sensors and received instructions to determine how the motors should move.
- **Power stage**: The power stage takes the low power signals from the controller and converts them to high power output directly for the motor. The exact input and output form depends on a specific implementation of the controller and the type of motor used.
- **Actuator**: This is the component responsible for producing mechanical motion. There are many different types of motors such as brushed DC motors, stepper motors, asynchronous motors, or permanent magnet synchronous motors (PMSM).

This thesis will focus on the PMSM motors.

2.1 PMS Motors

Permanent Magnet Synchronous Motor (PMSM) is a three-phase AC synchronous motor with permanent magnets attached to the rotor and windings going through the stator. They are known for their high efficiency, high reliability, and high power density. This makes them a good choice for many motion control applications, especially where there is a need for speed or position control.

PMSMs are operated by electronic commutation instead of mechanical brushes, unlike traditional brushed DC motors. This increases their reliability, as there are no parts that are subject to wear, such as brushes or parts of the commutator. The efficiency and power density are also improved. However, electronic commutation means additional requirements for the controller.

There is another similar variant of the three-phase synchronous motor with permanent magnets, the BLDC motor. The main difference is that the PMSM has sinusoidal back electromotive force (BEMF), but the BLDC has trapezoidal BEMF. PMSM has the advantage that the torque produced is constant throughout the turn, but BLDC is somewhat easier to control.

2.1.1 Construction of PMSM

The PMSM motor consists of a rotor with permanent magnets and a stator with coil windings of each of the three phases. The number of windings depends on the pole pair count of the permanent magnet inside the stator. One of the possible configurations is two pole pairs, as can be seen in the picture 2.1.

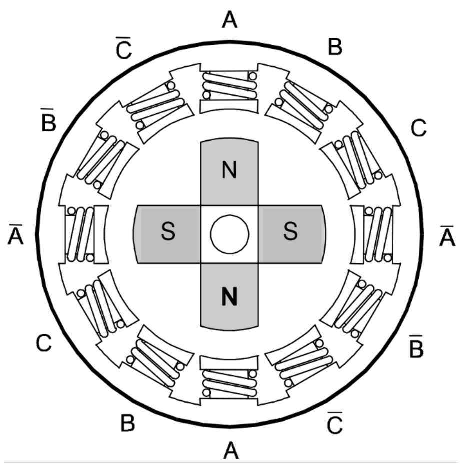

Figure 2.1. PMSM with two pole pairs [1]

2.1.2 PMSM Control

The control of PMSM requires managing the magnitude and direction of the stator magnetic field. The field needs to be closely controlled with regard to the position of the rotor to maximize efficiency. For this reason, it is useful to have mathematical transformations that relate the current flowing through each winding to the position of the rotor and back. There exist two transformations which are useful for this, Clarke Transformation and Park Transformation.

2.1.3 Clarke Transformation

The purpose of the Clarke transformation in PMSM control is to convert the currents from the three-phase system (a three-dimensional system) to a two-phase orthogonal system (a two-dimensional system). The two-phase system can also be interpreted as a vector in the complex plane. The image 2.2 shows the transformation in greater detail.

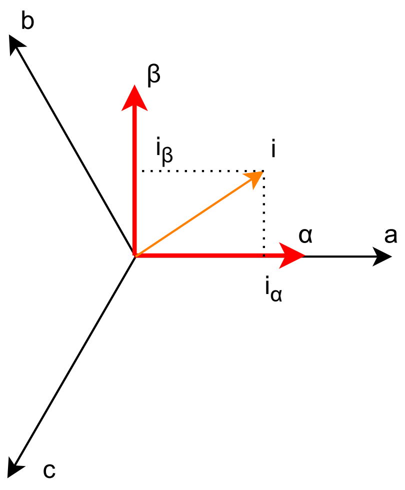

Figure 2.2. α and β currents in relation to a, b, c currents

We are able to write the Clarke transformation in matrix form as used in [2]

$$\begin{bmatrix} i_\alpha \\ i_\beta \\ i_\gamma \end{bmatrix} = \frac{2}{3} \begin{bmatrix} 1 & -\frac{1}{2} & -\frac{1}{2} \\ 0 & \frac{\sqrt{3}}{2} & -\frac{\sqrt{3}}{2} \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \end{bmatrix} \begin{bmatrix} i_a \\ i_b \\ i_c \end{bmatrix}, \tag{1}$$

where i_a, i_b and i_c are generic three-phase currents and i_α, i_β and i_γ are the corresponding currents given by the transformation.

We can also write the inverse Clarke transformation as follows.

$$\begin{bmatrix} i_a \\ i_b \\ i_c \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ -\frac{1}{2} & \frac{\sqrt{3}}{2} & 1 \\ -\frac{1}{2} & \frac{-\sqrt{3}}{2} & 1 \end{bmatrix} \begin{bmatrix} i_\alpha \\ i_\beta \\ i_\gamma \end{bmatrix} \tag{2}$$

For the currents flowing through the three windings of the motor stator i_a, i_b and i_c the following equation is true.

$$i_a + i_b + i_c = 0 \tag{3}$$

This means that in equation (2) the i_γ is zero and the transformation can be rewritten from equation (1) to

$$\begin{bmatrix} i_\alpha \\ i_\beta \end{bmatrix} = \sqrt{\frac{2}{3}} \begin{bmatrix} 1 & -\frac{1}{2} & -\frac{1}{2} \\ 0 & \frac{\sqrt{3}}{2} & -\frac{\sqrt{3}}{2} \end{bmatrix} \begin{bmatrix} i_a \\ i_b \\ i_c \end{bmatrix}. \tag{4}$$

where i_α and i_β are the transformed currents in the two-dimensional system.

2.1.4 Park Transformation

The primary purpose of the Park transformation is to convert the stationary magnetic field represented in the two-phase orthogonal system obtained by using Clarke transformation to a new set of axes aligned with the rotor's magnetic field.

The axes are chosen so that they are orthogonal, and called direct and quadrature axes, and the currents called i_d and i_q, respectively. The picture 2.3 shows the relation between i_α, i_β and i_d, i_q.

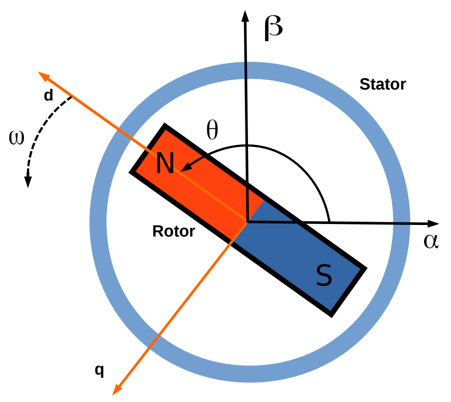

Figure 2.3. Axes d(direct) a q(quadrature) are chosen aligned to the rotor. [1]

The direct axis is aligned along the rotor's magnetic field direction. It primarily affects the magnetic flux within the motor. Control of i_d allows for the management of the motor flux, influencing how the motor handles magnetic saturation and efficiency.

The quadrature axis is orthogonal to the direct axis and primarily governs the production of torque in the motor. Controlling i_q directly influences the torque output of the motor.

We can write the Park transform in matrix form as follows [3].

$$\begin{bmatrix} i_d \\ i_q \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} i_\alpha \\ i_\beta \end{bmatrix} \tag{5}$$

We can also write the inverse Park transformation as

$$\begin{bmatrix} i_\alpha \\ i_\beta \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} i_d \\ i_q \end{bmatrix}. \tag{6}$$

2.2 Field Oriented Control

Field Oriented Control (FOC) is a control strategy for PMSM motors. It utilizes the Park and Clarke transforms, both in the normal and in their inverse forms. The primary goal of FOC is to control the magnetic field within the motor to achieve efficient torque production. This is done by aligning the stator current vector with the rotor's magnetic field. The process involves transforming the three-phase motor currents into two orthogonal components, the direct axis and the quadrature axis, as defined by the Park transform.

First, the currents i_a, i_b and i_c of each of the three phases of the motor are measured. These currents are used later for feedback control. Then these currents are transformed into a stationary two-phase reference frame (i_α, i_β) using the Clark transformation. Then i_α and i_β currents are transformed into a rotating reference frame aligned with the rotor magnetic field using the Park transformation. The currents are then expressed as i_d (d)and i_q (q) components, which are orthogonal. The d-axis current is aligned with the rotor's magnetic field, while the q-axis current is perpendicular to it.

The control algorithms are then applied to the i_d and i_q currents. The type of algorithm used depends on the application. For example, when controlling the speed of the motor, a Proportional-Integral (PI) controller might be implemented to adjust the i_q current for torque control, while the i_d current may be controlled to manage the magnetic flux, optimizing motor efficiency and preventing magnetic saturation.

The effective implementation of field oriented control requires precise feedback on rotor position, which is typically obtained using sensors such as encoders or Hall effect sensors or incremental rotary counters (IRC). This positional feedback is important as it determines the transformation angles used in the Park transformation, ensuring that the currents i_d and i_q are accurately aligned with the magnetic field axes of the rotor.

Chapter 3
Original System Setup

The initial phase of this thesis was to familiarize with the MZ_APO board based on the Zynq 7000 and microzed-mc-1 PMSM motor control FPGA design and power stages and then to design a procedure for Hall current sensor calibration.

The MZ_APO board was used for motor control and also during the calibration of the current sensors together with the `3p-motor-driver-1` power stage. Motor control was realized with both the Portable, highly eXtendable Motion Control library (PXMC) [4], as well as the open-source graphical tool for control system design and real-time code generation, pysimCoder [5], together with the microzed-mc-1 PMSM motor control FPGA design.

This chapter will introduce the components and designs used.

3.1 MZ_APO Board

The MZ_APO board is an educational kit built on the MicroZed board., which is based on the Zynq-7010 MicroZed SBC. The MZ_APO was created by the PiKRON company, and the full design can be found on the PiKRON gitlab [6].

This board provides numerous peripherals and interfaces, and is used in education and projects with GNU/Linux, RTEMS and VxWorks operating systems. The board is shown in Figure [7].

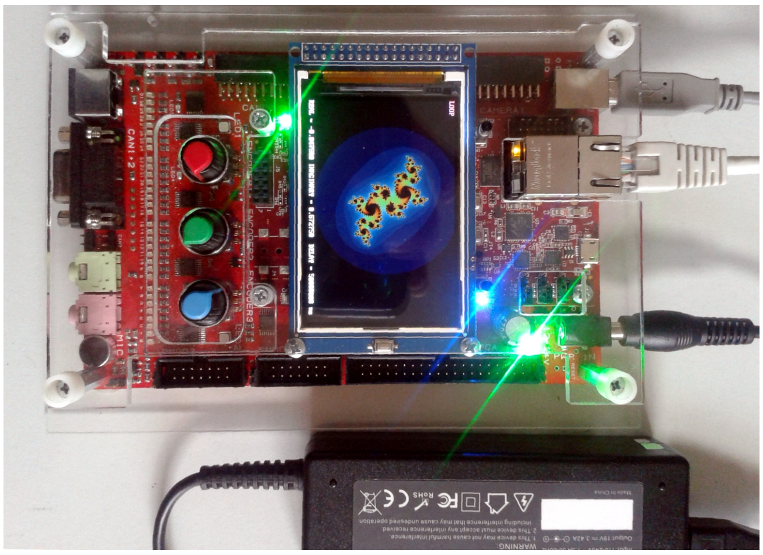

Figure 3.1. Picture of the MZ APO Board [8]

The more detailed board specification is documented on the B35APO course pages [8] and in the MZ-APO documentation [9]. The key properties are:

- Base Chip: Xilinx Zynq-7010 All Programmable SoC
- Type: Z-7010, part XC7Z010
- CPU: Dual ARM Cortex-A9 MPCore at 866 MHz (NEON &Single/Double Precision Floating Point)
- 2x L1 32 kB data + 32 kB instruction, L2 512 KB
- FPGA: 28K Logic Cells ($\sim$ 430K ASIC logic gates, 35 kbit)
- Computing units in FPGAs: 100 GMACs
- FPGA memory: 240 KB
- Memory on MicroZed board: 1GB

The MZ_APO board also has many communication interfaces, such as Ethernet, USB, Serial port, or CAN, which could be used for communication with the board.

3.2 Power Stage

The MZ_APO board is connected to the 3p-motor-driver-1 power stage board. This board was developed by PiKRON company for experimenting with motion control, and its design files can be found in the company's gitlab [10]. The schematic diagram is in Appendix D.

The board has multiple key parts, namely the motor power control, phase current measurement, and connection of the Hall sensors and the IRC to the main board.

3.2.1 Power Control

The power control of the motor is implemented by a trio of half H-bridges, each consisting of two power N-MOS transistors and controlled by a half-bridge N-Channel Power MOSFET Driver LT1158. The drivers are then controlled by PWM and Enable signals.

The motor power control part is completely galvanically isolated from the rest of the system to protect control electronics and suppress interference.

3.2.2 Current Measurement

The board measures the current flowing through all three motor windings. The sensors used for this measurements are analog Hall effect sensors Allegro ACS711 [11]. These sensors are mounted next to each other on the board. Unfortunately, the placement of the sensors results in them measuring not only the desired phase, but they are also affected by the currents flowing through the two other phases.

This is the reason for the design of the current measurement calibration described later in this thesis.

The analog outputs of the sensors are then measured by the onboard 12-bit analog-to-digital converter (ADC) ADS7841, which then allows reading of the ADC values via SPI.

3.3 Used Motor

The used motor in this thesis is a Permanent Magnet Synchronous Motor (PMSM). The exact type of motor used is BLWR233D-36V-4000 made by Aneheim Automation [12]. The motor can be seen in Figure 3.1.

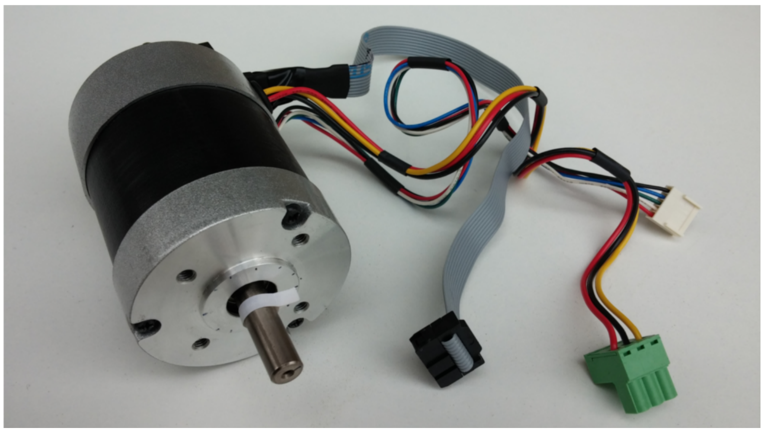
Figure 3.1. Picture of the used motor

The motor has a power rating of 92 Watts, its rated voltage is 36 V, and it also has two pole pairs. Its parameters are shown in the following list:

- **Rated Voltage (V)**: 36
- **Rated Power (Watts)**: 92
- **Rated Torque (oz-in)**: 31.2
- **Rated Speed (RPM)**: 4000
- **Torque Constant (oz-in/A)**: 8.5
- **Back EMF Voltage (V/kRPM)**: 4.45
- **Line-to-Line Resistance (ohm)**: 0.64
- **Line-to-Line Inductance (mH)**: 2.1
- **Rotor Inertia (oz-in-sec2)**: 0.0010600
- **"L" Length (in)**: 2.9
- **Weight (lbs)**: 1.65

3.3.1 Positional Sensors

To the motor are connected two position sensors, the Hall sensors, and an Incremental Rotary Counter (IRC).

The Hall sensors measure the motor position by detecting changes in the magnetic field as the rotor moves. There are three Hall sensors, which together provide the absolute position of the motor with a resolution of six sectors per one electrical rotation of the motor.

The Incremental Rotary Counter (IRC), also connected to the motor, offers much higher resolution in the measured position. The IRC works by producing a series of pulses as the motor shaft rotates, each pulse corresponding to a small increment in movement.

There are two signals from the IRC, typically named `A` and `B` that are out of phase (often referred to as quadrature output). The two signals together measure not only the change in position of the motor, but because of the phase shift, they also signalize the direction of the motor movement.

The IRC also has a third signal, the index, which indicates when the motor is in a specified angle. This index signal is used to obtain the absolute position of the motor, as the `A` and `B` signals would only provide relative information. For this reason, the IRC is often used together with the Hall position sensors.

3.4 Microzed-mc-1 PMSM Motor Montrol FPGA Design

The `microzed-mc-1` FPGA design was developed for the control of a PMSM motor as part of previous projects. The design can be found on the gitlab page[13] or it is used as part of the application of the rvapo project available at [14]. It is also described in the presentation [15] and also in the thesis by Martin Prudek[1].

The FPGA design is responsible for processing the outputs from the power stage board, namely the Hall position sensors, the IRC and the measured phase currents, and also generating the PWM and Enable signals for the half-bridges controlling the power MOSFETs.

The design communicates with the control system via SPI. The communication is two-way; at the same time, the control system tells the FPGA the instructions for the power control, and the FPGA reports the Hall position, IRC value and index position, and the information about measured currents.

3.4.1 Data Transfer

The data is transmitted in 128 bits. The following tables show the exact composition of the message. The table 3.1 shows the order of data transferred from the control system to the FPGA. The table 3.2 then shows the order of the data in the SPI transfer from the FPGA to the control system.

Bits	Function
127	ADC reset
126 .. 124	Enable PWM1 .. PWM3
123 .. 121	Half-Bridge Shutdown for PWM1 .. PWM3
120 .. 43	Unused
42 .. 32	PWM1 duty cycle
31 .. 27	Unused
26 .. 16	PWM2 duty cycle
15 .. 11	Unused
10 .. 0	PWM3 duty cycle

Table 3.1. Order of data in the SPI transfer from the control system to the FPGA

Bits	Function
127 ..96	IRC
95	Hall 1
94	Hall 2
93	Hall 3
92 .. 81	Index position
80 .. 72	Number of summed currents
71 .. 48	Sum of currents, channel 2
47 .. 24	Sum of currents, channel 0
23 .. 0	Sum of currents, channel 1

Table 3.2. Order of data in the SPI transfer from the FPGA to the control system

Chapter 4
Calibration of Current Sensors

The measurement of phase currents of a PMSM is important for current feedback control, for example, in field-oriented control or sensorless control of the motor. If the measurement is inaccurate, it can negatively affect the overall behavior and performance of the system.

This is the case of the used power stage board, the `3p-motor-driver-1`, where the analog current sensors ACS711 based on the Hall effect, described in the section 3.2, are too close to each other on the board. This results in each sensor being influenced not only by its corresponding phase current but also by the currents in the other two phases.

4.1 Theory

The sensors measure the current through the induced magnetic field. The output of the sensor is linear to the magnetic field (and to the measured current in the ideal case when not affected by anything else)[11]. The magnetic field measured by the sensor is caused by the current flowing through it and also by the other two currents in the neighboring sensors.

Let us assume that any other source of the magnetic field in the vicinity is negligible or at least static. This assumption is based on the fact that there are no other high current traces in the vicinity of the sensors and that the rest of the system is much less power demanding.

The linearity of the sensors means that there exists a linear transformation that correlates the measured current values with the true values. In fact, there can be some non-zero offset, so the transformation should be considered affine, but for the sake of simplicity, we can separate the problem of finding the offsets from the transformation.

The calibration can then be written as the following equation:

$$i_{\text{true}} = X i_{\text{measured}}, \tag{1}$$

where i_{measured} are the measured currents, X is the calibration matrix, and I_{true} are the true currents.

Finding the linear transformation is then the matter of finding the matrix X. Finding the calibration matrix X can be formulated as a Least Squares problem. The matrix X can then be calculated from the equation

$$XA = B, \tag{2}$$

where A are true currents measured externally in the form on $3 \times n$ (n is the number of measurements) and B are the values measured from Hall current sensors on the board.

Because we have three phases of the motor, we would ideally have the matrix X of size 3×3. To obtain a valid solution of the matrix X, the matrices A and B must have rank at least 3.

Because the motor windings are interconnected in a star pattern, the current flowing to the motor through one of the phases must flow out through the rest of phases. The sum of the currents must remain zero as there is no other path for the current to take.

The simplest way for the current to flow is that one of the phases is connected to the voltage source, one is connected to the ground, and the third is left disconnected. Then there are three possibilities for the configuration of the motor winding connections, as seen in table 4.1. The values in the table are normalized because their size does not matter for the rank of the matrix. The order of positive and negative currents is also arbitrary as long as the disconnected phase stays the same.

Winding A	Winding B	Winding C
1	-1	0
0	1	-1
1	0	-1

Table 4.1. Table of possible phase currents, normalized.

We can form any other possibility of the currents flowing through the phases as a linear combination of these vectors.

It is important to see that the rank of this matrix is 2, which means there is no solution to the least squares problem, and we need either more data to extend the rank or to use a different coordinate system.

4.1.1 Full Rank 3x3 Calibration Matrix

If we think of the table 4.1 as a matrix and calculate its kernel, we get the vector:

$$\ker(B) = \operatorname{span}([1, 1, -1])$$

The resulting vector shows us that it would suffice to extend the measurements by this vector or its linear combination with the rest. One such vector is

$$[1, 0, 0],$$

which would suffice to get the full rank of 3.

This would require a non-zero sum of the phase currents. To achieve this, we can rewire the motor connection and connect one of the phases directly to the ground via an external cable, so that the return current does not flow through the Hall sensor.

Another way of achieving the full rank of the matrices was tried. Unfortunately, even if we connect one of the phases to the power supply and the two remaining to ground through the sensors, so we would not have to modify the motor connection, we would still get linearly dependent vector on the measurements already conducted. This situation would result in the vector of

$$\left[\frac{4}{3}, \frac{2}{3}, \frac{2}{3}\right],$$

which does not increase the rank of the measurement matrix. This is to be expected, as it is still just a linear combination of the vectors from Table 4.1.

4.1.2 2x2 Calibration Matrix using the Clarke Transformation

The rank deficiency of the matrix shown in previous chapter is caused by the nature of the PMSM motor, that the sum of all currents must be equal to zero. If we used two perpendicular phases instead of the three in the PMSM, each 60° apart, it would solve the problem of the redundant dimension.

We can use the Clarke transformation for this purpose. In the new reduced coordinate system, the calibration matrix can be calculated without changing the motor wiring. From the control perspective, it does not matter if we are calibrating the currents before the Clarke transformation or after, the control is using the currents after the Park transformation.

The correction matrix in this case was then calculated as a least squares problem from the equation

$$C^{-1}XCA = B, \tag{3}$$

where C is the Clarke transform matrix, A is the matrix of true currents and B is the matrix of measured currents.

4.2 Implementation

The calibration procedures described previously were implemented and tested using the PXMC library. The PXMC is described later in Chapter 9, together with its usage in the ICE-V motion control system. The same principles are also applicable here.

4.2.1 Collecting Data with PXMC

The PXMC application was extended to allow measurement of the phase currents. For this purpose, the application command `currentcal` was used and extended.

The `currentcal` command can be called directly from the PXMC application using the command processor. The core of the command function is a loop that iterates through seven calibration cycles - one for the offset and two for each required vector (opposite current flow direction). For each cycle, the function retrieves a specific pattern of PWM settings, which define the duty cycles and enable states for three different phases. Before proceeding with each calibration cycle, the function waits for user confirmation. Once confirmed, the system is set up to perform current measurements, initially discarding the first set of measurements to allow the system to stabilize. Then it accumulates current readings over a specified number of samples.

After completing the measurements for each cycle, the function calculates the average current values for each of the three phases and stores these results in a matrix. The data are stored in the raw ADC counts, as this allows the function to be universal regardless of the type of the ADC or the sensors. This matrix is later saved as a CSV file.

The command measures the currents for different phase settings. The settings used are listed in table 4.1. This table is used by default in the configuration procedure, but it can be modified or extended according to the user's needs. For example, the table was extended for the 3×3 calibration matrix with the configuration shown in Table 4.2.

PWM1	Enable 1	PWM2	Enable 2	PWM3	Enable 3
0	0	0	0	0	0
1	1	0	1	0	0
0	1	1	1	0	0
0	0	1	1	0	1
0	0	0	1	1	1
0	1	0	0	1	1
1	1	0	0	0	1

Table 4.1. SPIMC current calibration pattern

PWM1	Enable 1	PWM2	Enable 2	PWM3	Enable 3
1	1	0	0	0	0

Table 4.2. SPIMC current calibration pattern extension for the 3×3 calibration matrix

The source code for the current calibration data acquisition from the PXMC is part of the ice_v_pmsm project[16] and it is implemented in the `appl_pxmccmds.c` file.

4.2.2 Calculating the Calibration Matrices

The measured phase current data must be processed to obtain the calibration matrices. This calculation was done using two methods, since the processing script was created in both Matlab and Python.

The Python script is divided into two main functions, `get_cal_matrix_clarke` and `get_cal_matrix_3x3`. Both of the functions expect as their input three parameters. The first parameter is the CSV file that contains the true currents measured externally. The second parameter is the CSV file that contains the currents measured by the PXMC from the on-board Hall sensors. The last parameter is the name of the output CSV file, to which the calculated calibration matrix will be written.

The `get_cal_matrix_3x3` function calculates the correction matrix in the 3×3 format. First, the measured currents are loaded into a matrix `B`. The data organization follows the pattern of Table 4.1 with the added line from Table 4.2. The first row contains the measured offsets, the rest then form the matrix B described in the equation (2). The true currents are then loaded into the matrix `A`, which were measured during the procedure using an external ammeter. Their format must also match the pattern.

The offsets are then subtracted from the `B` matrix and converted from the raw ADC values to Amperes. For this purpose, the function `corr_currents` was implemented.

After conversion, the calibration matrix X is calculated using the least-squares method with the NumPy function `np.linalg.lstsq` according to the equation (2) and saved in the desided CSV file.

The `get_cal_matrix_clarke` function calculates the correction matrix in the 2×2 format using the Clarke transformation. The procedure is similar to the `get_cal_matrix_3x3` function. The only difference is that the calibration matrix X needs to be calculated using the equation (3).

The same calculations are also performed in the Matlab scripts. The naming convention stays the same, the BA matrix contains the true, externally measured currents, and the B matrix contains the measured currents by the PXMC.

4.3 Calibration Results

4.3.1 Measurements

The measurements taken and used to calculate the calibration matrices are listed in the following tables. The table 4.1 shows the raw ADC values from the PXMC. The table 4.2 then shows the true values measured by external ammeter.

Winding A	Winding B	Winding C
2039.70	2067.93	2060.78
2045.37	2254.93	1886.63
2222.81	1886.33	2058.77
2227.91	2073.88	1883.68
2038.33	1905.48	2062.58

Table 4.1. Table of measured phase currents, in 12bit ADC values

Winding A [A]	Winding B [A]	Winding C [A]
0	1.30	-1.30
1.33	-1.33	0
1.30	0	-1.30
0	-1.130	0

Table 4.2. Table of true phase currents in Amperes

The last row in both tables was created by connecting one of the phases directly to the GND, bypassing one of the Hall current sensors.

4.3.2 Calibration Matrices

The following 3×3 matrix was calculated using the developed calibration procedure.

0.969059	-0.007795	0.029863
-0.027891	0.941421	0.000986
0.023964	0.011096	1.020851

Table 4.3. 3x3 Calibration matrix.

The same measurements without the last row were used to create the 2x2 calibration matrix using the Clarke transformation. The resulting matrix is then

0.968973	-0.001363
-0.002462	0.967489

Table 4.4. 2x2 Calibration matrix.

The calibration matrices are used in the pysimCoder models described in Section 10.6 and were also used in Ing. Damir Gruncl's Diploma Thesis[17], where he used them to control the PMSM motor with his developed RISC-V coprocessor.

Chapter 5
NuttX

In this section, we focus on the real-time operating system NuttX, designed specifically for use in embedded systems and microcontrollers. It will be used in the design of a custom motion control system based on the ICE-V board, described in a following chapter.

5.1 Overview of NuttX

NuttX is a real-time operating system (RTOS) specifically designed for use in embedded systems and microcontrollers with an emphasis on standards compliance and small footprint. NuttX was first introduced by Gregory Nutt in 2007. It has been under incubation at The Apache Software Foundation from 2019 to November 2022, when it became a top-level project. Development is managed under The Apache Software Foundation and follows the Apache License 2.0. [18]

NuttX is designed to comply with the POSIX (Portable Operating System Interface) and also implements ANSI standards and additional APIs from Unix systems and other RTOS such as VxWorks.[19] This conformity to standards ensures that software developed for other standard operating systems, such as Linux, can be easily ported to NuttX.

NuttX includes RTOS features such as fully preemptive scheduling, task priorities with priority inheritance, and symmetric multiprocessing. It also supports multithreading with semaphores, pthreads, or mutexes.

5.1.1 NuttX Advantages

The NuttX system is intended to be used in a wide variety of embedded applications, as it is scalable from small 8-bit to modern 64-bit microcontrollers. It can be used with many different architectures, for example ARM, RISC-V or Atmel AVR. The modularity of the system allows its use in low-cost microcontrollers with small memory (the final executable can then be run on as low as only 32 kB of total memory (code and data), although typical NuttX build usually requires about at least 64 kB memory). Larger systems, on the other hand, can benefit from extensive list of additional features and drivers.

NuttX is highly configurable, allowing projects to include only the necessary components, and uses the Kconfig system, borrowed from the Linux kernel. After configuring with Kconfig, the build system uses GNU makefiles. The small footprint is achieved by compiling and including only the required features in the executable file, along with other optimization techniques. Each peripheral or component can be added or removed from the build before compilation due to its modular design.

5.2 Basic Nuttx Organisation

NuttX employs a structured architecture that divides its system into two primary components, the kernel and the user space. This separation enhances modularity and security, allowing developers to modify the operating system to the specific needs of embedded systems.

5.2.1 NuttX Kernel

The NuttX kernel is the core component of the operating system, responsible for managing fundamental operations and resources. It includes several key subsystems. The scheduler handles task execution, ensuring tasks are scheduled based on priority and timing requirements, supporting policies like round-robin, priority-based, or rate-monotonic scheduling. Inter-Process Communication (IPC) mechanisms, such as message queues, semaphores, signals, and mutexes, allow tasks to communicate and synchronize with each other. Memory management handles the allocation and deallocation of memory resources, supporting both static and dynamic memory allocation, and features to manage memory fragmentation.

The kernel also includes low-level device drivers for interfacing with hardware peripherals such as GPIO, UART, I2C, SPI, ADC, CAN, and Ethernet, which abstract the hardware details from the application layer and provide a consistent interface for developers. The file system support in the kernel handles file operations, storage management, and file system mounting, with support for various file systems including FAT or ROMFS. The network stack implements networking protocols such as TCP/IP and UDP, managing network interfaces like Ethernet and Wi-Fi to enable network communication and connectivity.

5.2.2 Nuttx User Space

In NuttX, the user space operates separately from the kernel. This creates a secure and stable environment for application execution. This user space consists of several components, such as user applications, system applications, middleware, libraries, and configuration and initialization scripts.

Applications handle specific tasks within the system and utilize kernel services through well-defined APIs, enabling interaction with underlying system functions using drivers or libraries. The applications can be user-developed to handle specific tasks of the whole system, or they can be system applications that include built-in utilities and tools provided by NuttX, such as shell interfaces, diagnostic tools, and system management utilities. These applications are important for configuring, monitoring, and controlling the system, helping users and developers to manage and troubleshoot effectively.

There are also middleware and libraries responsible for higher-level functionalities, including graphical user interfaces, communication protocols, and application frameworks, or configuration and initialization scripts for setting up the runtime environment.

The NuttX can divide the system kernel from applications on systems with Memory Management Unit (MMU). By dividing the system into kernel and user space, NuttX ensures that user applications run in a protected environment. This protects the system from application crashes or unexpected behavior, as it would not impact the kernel's operation. This separation enables applications to be platform independent and not focus on hardware specifics, allowing faster hardware changes if necessary. When using flat address space, the applications can even be loaded during the system execution.

5.3 Source Code Organization

NuttX source code organization is similar to that of the Linux kernel [20]. The key subdirectories for this thesis are `arch/`, `boards/`, and `drivers/`, which contain the necessary source code and headers for architectures, microcontrollers, boards, and device drivers. The NuttX core is not discussed as this is beyond the scope of this thesis.

5.3.1 Architecture Directory

This directory contains architecture-specific code. Each supported architecture such as ARM, x86 or RISC-V has its subdirectory with folders `include/` for architecture-specific definitions and `src/` for implementation files. The architecture-specific definitions are for example which peripherals are supported, external interrupts or peripheral identificators. The implementation files include all source files for hardware-specific implementation for supported drivers. These parts of the drivers are called in NuttX documentation as `lower half` drivers. [21]

5.3.2 Boards Directory

The `boards/` directory includes board-specific configurations and initialization code, organized by architecture and then by specific boards. This directory is important for initializing the boards using code from the `arch/` directory.

5.3.3 Drivers Directory

The `drivers/` directory contains the `upper half` device drivers for various hardware peripherals such as GPIO, UART, I2C, SPI, ADC, CAN and network interfaces. This part of the driver implements the high-level interface such as `read`, `write`, or `open`. The upper half of the driver connects to the lower half via callbacks.

This separation of drivers into `lower half` and `upper half` is helpful, ensuring that there is only one interface defined for applications or other drivers, such as SPI, and that it is in the upper half not dependent on architecture. The architecture-specific implementation is then left for the lower half of the driver.

5.4 NuttX Configuration and Compilation

NuttX configuration is managed through the Kconfig system, similar to the Linux kernel, and the build process uses the GNU make program to compile the source code. The configuration process involves selecting the desired features and components for the target system. This is done using the `menuconfig`, `qconfig`, or similar alternatives, which provide a user-friendly interface for NuttX configuration. Before the configuration itself, it is necessary to load an initial configuration depending on the board and chip used.

After the configuration, it is necessary to build the whole system. This is done using GNU makefiles, which create the final binary file to be uploaded to the board. This compilation not only compiles the kernel, but the final binary also includes the enabled applications.

This whole process is explained in detail in [22].

Chapter 6
Motion Control on the ICE-V Board

This chapter describes the design of a motion control system using the `ICE-V Wireless` board[23] as the controller, which is an open source board based on the microcontroller `ESP32-C3-MINI-1`[24] and the Lattice `iCE40UP5k-SG48` FPGA[25]. The new motion control system uses the same power stage as previously used with the MZ_APO board. A new board was designed to connect the power stage to the ICE–V board, providing a power source for the ICE–V from the input system voltage, and also provides a CAN interface for control and allows access to multiple features of both boards for future use.

The final design of the motion control system using the `ICE-V Wireless` is shown in Figure 6.1. The system for control of the connected PMSM is made up of the ICE-V Wireless board, the `3p-motor-driver-1` power stage, and the custom-designed adapter board.

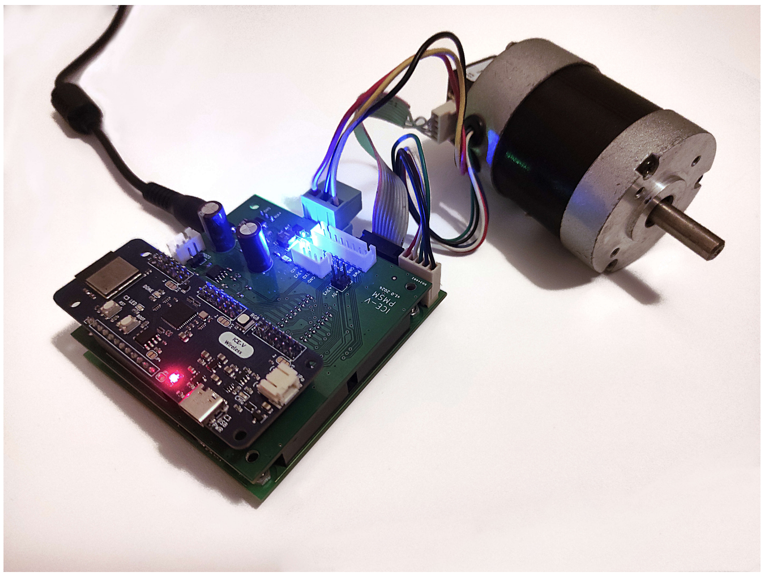

Figure 6.1. Picture of the entire motion control system using the ICE_V Wireless, the power stage and the designed adapter board

6.1 System Overview

The motion control system is structured into several logical blocks. This structure can be seen in figure 6.1. The first block depicts NuttX RTOS, in which one of the control applications is running, either PXMC or the code generated from pysimCoder. This application then communicates with the FPGA, which is responsible for providing the application with data from Hall sensors, IRC, and also measured currents. The FPGA is also responsible for generating the desired PWM signals for the H-Bridges on the

`3p-motor-driver-1` together with the Enable signals, which then drive the connected PMSM motor. The FPGA design is the `microzed-mc-1`, described in the Section 3.4.

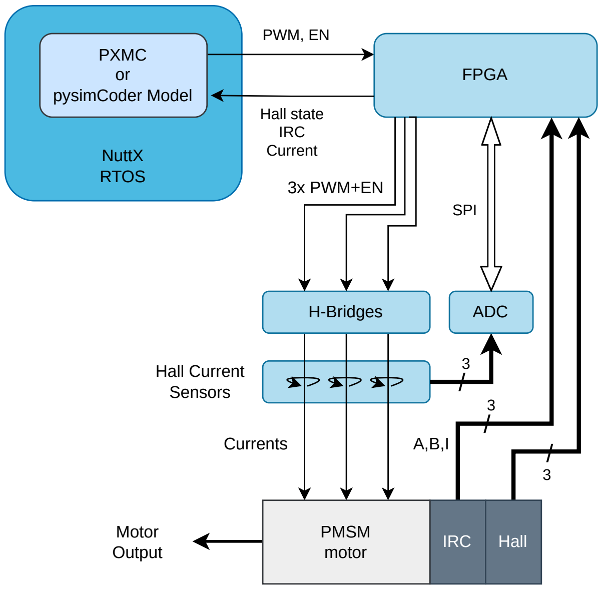

Figure 6.1. Overview of the motion control system.

The PXMC or pysimCoder applications are responsible for the speed or position control of the motor. The FPGA serves to alleviate some of the tasks of the ESP32–C3, which is to collect all the necessary information from the power stage and to keep generating the PWMs.

6.2 ICE-V Wireless Board

The `ICE-V-Wireless` is an open source development board designed for a variety of applications, including IoT devices and educational projects. It features the Lattice iCE40UP5k FPGA paired with an Espressif ESP32–C3 module based on the RISC–V architecture, providing Wi–Fi and Bluetooth Low Energy (BLE) capabilities.

The `ICE-V Wireless` GitHub project [23] has schematics and PCB layout available in KiCad 6.0 format, together with a PDF version of the schematic. The board is shown in figure 6.1.

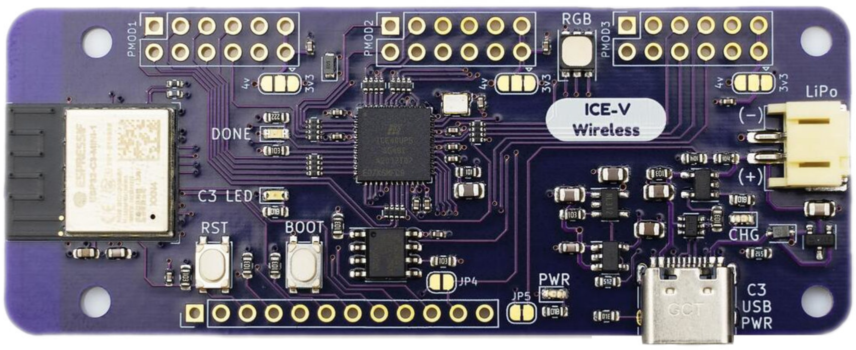

Figure 6.1. ICE-V Wireless board. [23]

The functional blocks and layout schematic diagram can be seen in Figure 6.2.

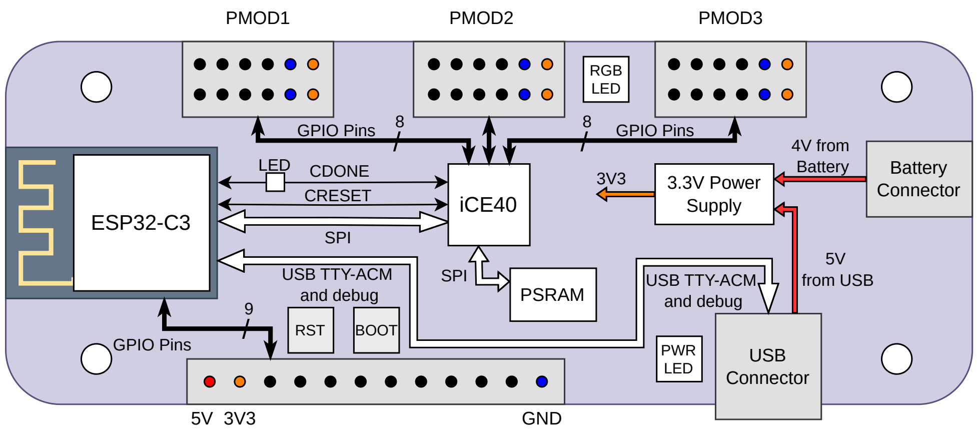

Figure 6.2. Schematic description of ICE-V Wireless board.

The key components of the board are:

- **ESP32-C3-MINI-1** module [24]:
  The ESP32-C3 is a microcontroller chip developed by Espressif Systems. It features a single-core RISC-V 32-bit processor which can run at up to 160 MHz. It comes with 4MB of embedded flash memory and supports 2.4 WiFi (802.11 b/g/n) and Bluetooth 5 (Low Energy)
- **ICE40UP5-SG48I** [25]:
  The ICE40UP5-SG48I is from the Lattice iCE40 UltraPlus family and features 5280 LUTs, 128 kbits of block RAM, 8 DSPs and 2 I2C and SPI cores. It also has low static power, which makes it suitable for battery-operated use cases.
- **LY68L6400** serial quad SPI Pseudo-SRAM (PSRAM):
  The PSRAM has size of 64Mb and is connected directly to the FPGA. This means that its contents must be loaded through the FPGA itself at the beginning.

The ESP32-C3 is connected to the iCE40 FPGA by SPI with two additional GPIO pins. It can be connected to by the USB connector, which provides a USB TTY-ACM and debug connection. The two buttons on the board, `RST` and `BOOT`, are connected to the ESP32-C3.

The iCE40 is apart from the ESP32-C3 also connected to the PSRAM via another SPI connection. The SPI connections between the iCE40 and the ESP32-C3 and the iCE40 and the PSRAM are not connected together. The pins of the iCE40 are accessible through the three PMOD connectors, with eight GPIO pins per connector.

The whole `ICE-V Wireless` board also has two power supplies, 3.3V and 1.2V. The 1.2V power supply is used only internally for the iCE40, but the 3.3V power supply provides power for the ESP32-C3 and also serves to power external peripherals connected to the PMOD connectors.

6.3 Adapter Board

The adapter board has been developed to replace the MZ_APO board with the `ICE-V Wireless`. It features connectors for both the `ICE-V Wireless` board as well as the power stage board `3p_motor_driver_1`, as well as a 5V power supply for the `ICE-V Wireless` from the system voltage supplied for the power stage and motor. It also allows access to unused pins of both the ESP32–C3 and the iCE40, and adds the possibility to communicate with the system via the CAN interface. The design is further described in Section 8.

6.4 Firmware

The whole motion control uses NuttX as its operating system. The NuttX is responsible for board initialization, such as loading the correct bitstream to the FPGA, and afterwards it handles all higher-level functionality of the entire motion control system. The lower level functionalities, such as PWM control and ADC calculations, are done internally by the bitstream inside the FPGA.

The control software itself is then running inside the NuttX. In this thesis, two different control software were used, PXMC[4] and pysimCoder[5].

6.4.1 Programming the ICE-V Wireless Board

The programming of the board is done in two steps, since both the ESP32–C3 and the iCE40 need to have the correct firmware or gateware loaded. The first is the programming of the ESP32–C3, which has a USB peripheral capable of programming and JTAG debugging directly accessible on the board. In the case of the iCE40, it is more complicated, as the only way to program it is through the ESP32-C3. The following diagram 6.1 shows how the ESP32–C3 and the FPGA are connected. The FLASH and PSRAM memories are also shown.

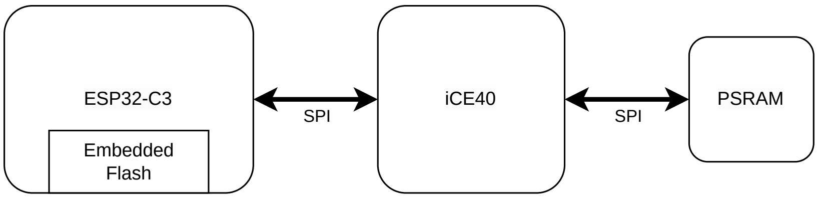

Figure 6.1. Block schematic description of the ESP32–C3 and the iCE40 on the `ICE-V Wireless` board.

The diagram shows how the devices are connected. The ESP32–C3 must load a bitstream to the FPGA during board initialization. Also, if there is a need to update the SPI PSRAM connected to the FPGA, it is only possible to do so through the FPGA. The authors of the board solve this by having a separate bitstream for the FPGA that allows the data to pass through and be loaded into the PSRAM.

In this thesis, we are using the NuttX RTOS, so we need to have the ability to load the bitstream configuration from the ESP32–C3 into the iCE40 FPGA directly inside the operating system. Unfortunately, no such driver or application was available. It was then decided to implement this functionality into the NuttX and try to make it part of the mainline.

This was successfully done and the pull request was successfully merged. The process of programming the FPGA through the NuttX ESP32—-C3 and the development of the driver is described in Section 7.

6.4.2 PXMC

The PXMC is Portable, highly eXtendable Motion Control library and system core, developed by PiKRON Ltd. [4] It is an open source library with the source code available at the company's gitlab [26]. It represents a ground-up rewrite of the company's previous motion control systems, initially implemented in MARS-2 units. Designed for portability and extensibility, PXMC can be adapted to a wide range of hardware platforms and is suitable for applications in robotics, laboratory, and medical instruments. It is compatible with multiple motor types, such as DC, PMSM, and stepper motors, with and without position feedback.

The library can be used from the NuttX operating system and connect to the iCE40 FPGA, and together they can control the PMSM motor. The usage of the PXMC is described in more detail in the chapter 9

6.4.3 PysimCoder

The software pysimCoder is an open source control application development tool designed by Professor Roberto Bucher of the University of Applied Sciences and Arts of Southern Switzerland [5]. The source code is available on the projects github [27]. It is a flexible tool that allows for the rapid development of control algorithms for embedded systems. PysimCoder provides a graphical user interface for designing control systems, which can then be automatically translated into executable code for various microcontrollers and operating systems, including NuttX.

In the context of the motion control system on the `ICE-V Wireless` board, pysimCoder can be used to create control algorithms that are then deployed to the board. The use of PysimCoder in the motion control system is further described in chapter 10.

Chapter 7
Programming of the iCE40 FPGA from NuttX

This chapter describes the implementation of an open source driver designed to load configuration bitstreams into an iCE40 FPGA from the NuttX operating system. The ICE-V Wireless board relies on a custom firmware for the ESP32-C3 to load the desired bitstream into the FPGA, which meant that a new driver must have been developed for NuttX to use the iCE40 FPGA. The driver code can be found in my repository [28]. A pull request [29] was created and later accepted, so it is now part of the NuttX mainline.

7.1 Overview of iCE40 FPGA Programming

The iCE40 UltraPlus FPGAs, developed by Lattice Semiconductor are SRAM-based FPGAs, which also have on-chip, one-time programmable NVCM (Non-Volatile Configuration Memory) to store configuration data. The SRAM memory cells are volatile, meaning that once power is removed from the device, its configuration is lost and must be reloaded on the next power-up. This means that the bitstream must be loaded at startup to configure its logic blocks and interconnections. The bitstream can be stored either in the on-chip, one-time programmable NVCM or in an external SPI Flash. As we want to use the iCE40 for prototyping and development, we will be using the second option, so we can modify the configuration later instead of only one permanent write. The bitstream can be loaded into the FPGA in either Master SPI Configuration Mode or Slave SPI Configuration Mode. [30]

The Master SPI Configuration Mode is the mode in which the FPGA operates in SPI Master mode and loads the bitstream directly from the connected SPI Flash PROM. The second possible mode is the Slave SPI Configuration Mode. In this mode, the FPGA operates in SPI Slave mode and waits for the external processor to load the bitstream into it from its memory or elsewhere. We need to utilize the Slave SPI Configuration Mode, as the FPGA is directly connected to the ESP32–C3, as shown in figure 6.1

In the context of the ICE-V Wireless board, bitstream loading is managed by the ESP32-C3 microcontroller, which runs the NuttX operating system. The NuttX RTOS did not have an available driver for bitstream loading, so we decided to create one as part of this thesis. First, it is necessary to describe the process of loading the bitstream into the iCE40 FPGA.

7.1.1 SPI Slave Configuration Interface

The ESP32–C3 serially writes a configuration image to an iCE40 FPGA Using the SPI slave configuration interface through the iCE40 SPI interface. In addition to the SPI connection, there are two configuration control signals, CDONE and CRESET, which are used during the writing of the configuration bitstream. The configuration interface is shown in figure 7.1. The Application Processor in the figure is, in our case, the ESP32-C3.

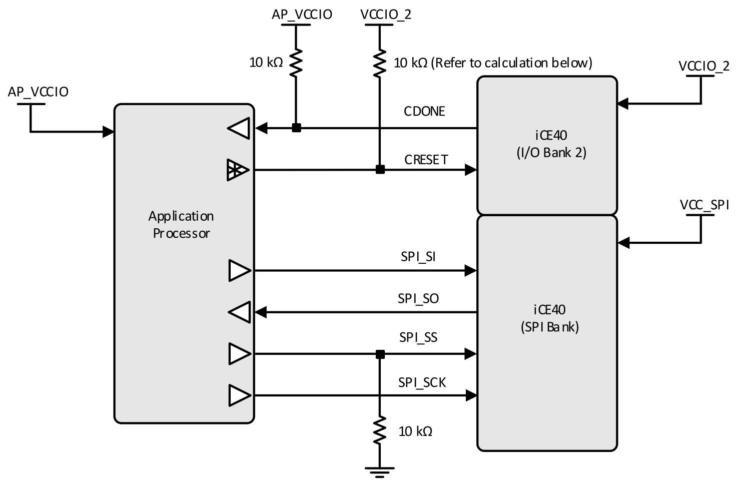

Figure 7.1. iCE40 SPI Slave Configuration Interface. [30]

SPI_SI, SPI_SO, SPI_SS, and SPI_SCK are SPI pins, where SI stands for Serial Input, SO for Serial Output, SS for Slave Select, and SCK for Slave Clock. The CRESET pin is the configuration reset input on the iCE40, active (reset) when low. The CDONE pin is the Configuration Done output from iCE40, which indicates the correct load of the configuration.

7.1.2 SPI Slave Configuration Process

To write the bitstream of the configuration to the iCE40, the driver must follow a specific procedure, which is illustrated in the figure 7.2.

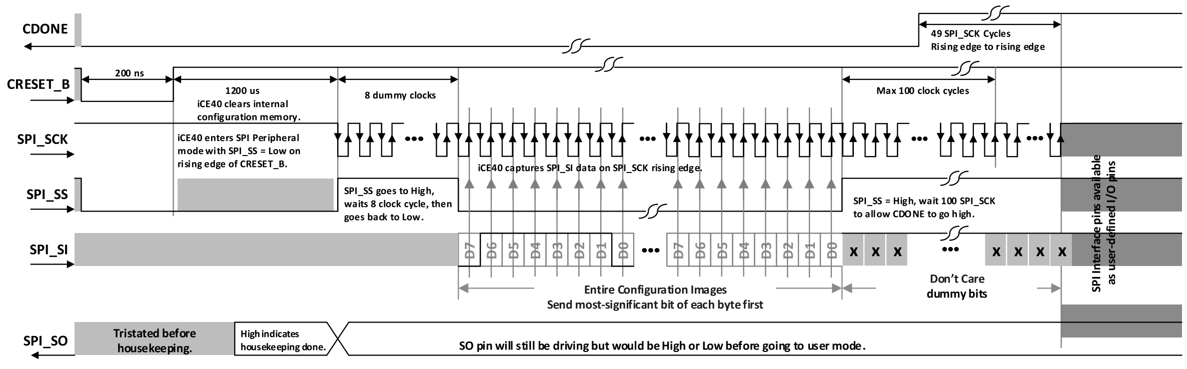
CDONE
CRESET_B
SPI_SCK
SPI_SS
SPI_SI
SPI_SO
200 ns
1200 us
iCE40 clears internal
configuration memory.
8 dummy clocks
49 SPI_SCK Cycles
Rising edge to rising edge
Max 100 clock cycles
iCE40 enters SPI Peripheral mode with SPI_SS = Low on rising edge of CRESET_B.
iCE40 captures SPI_SI data on SPI_SCK rising edge.
SPI_SS goes to High, waits 8 clock cycle, then goes back to Low.
SPI_SS = High, wait 100 SPI_SCK to allow CDONE to go high.
D7 D6 D5 D4 D3 D2 D1 D0
X X X
X X X X
Entire Configuration Images
Send most-significant bit of each byte first
Don't Care
dummy bits
SPI interface pins available as user-defined I/O pins
Tristated before housekeeping.
High indicates housekeeping done.
SO pin will still be driving but would be High or Low before going to user mode.

Figure 7.2. Application Processor Waveforms for SPI Peripheral Mode Configuration Process [30]

The driver begins by driving the iCE40 CRESET pin low and resetting the iCE40 FPGA. While in reset, the driver needs to hold the iCE40 SPI_SS pin low for at least 200 ns, and after that, while still holding the SPI_SS low, it can set the CRESET pin high again. When the iCE40 FPGA starts from reset with its SPI_SS low, it will enter the SPI peripheral mode, which is necessary for the upload of the bitstream by the driver.

After driving CRESET high, the driver must wait at least 1200 µs, to allow the iCE40 FPGA to clear its internal configuration memory. After this time, the SPI_SS is set to high, the driver sends 8 clock cycles with no payload, and then the SPI_SS goes back to low. Then the iCE40 is ready to receive the configuration bitstream. The driver then sends the entire configuration bitstream without interruption to the iCE40 SPI_SI input on the falling edge of the SPI_SCK clock input, with each byte being sent the most significant bit (msb) first. The SPI_SO output pin on the iCE40 is not used during SPI slave mode.

After sending the entire image, the iCE40 FPGA releases the CDONE output, allowing it to float high. If CDONE remains low, it means that an error occurred and the configuration was not uploaded to the FPGA. After the CDONE is high, the driver then must send at least 49 additional dummy bits, effectively 49 additional SPI_SCK clock cycles measured from rising-edge to rising-edge. This makes the SPI interface pins available to the user-application loaded in FPGA.

To upload another configuration, the driver just needs to restart the FPGA and start over.

The whole process can be summarized by the flow chart in Figure 7.3

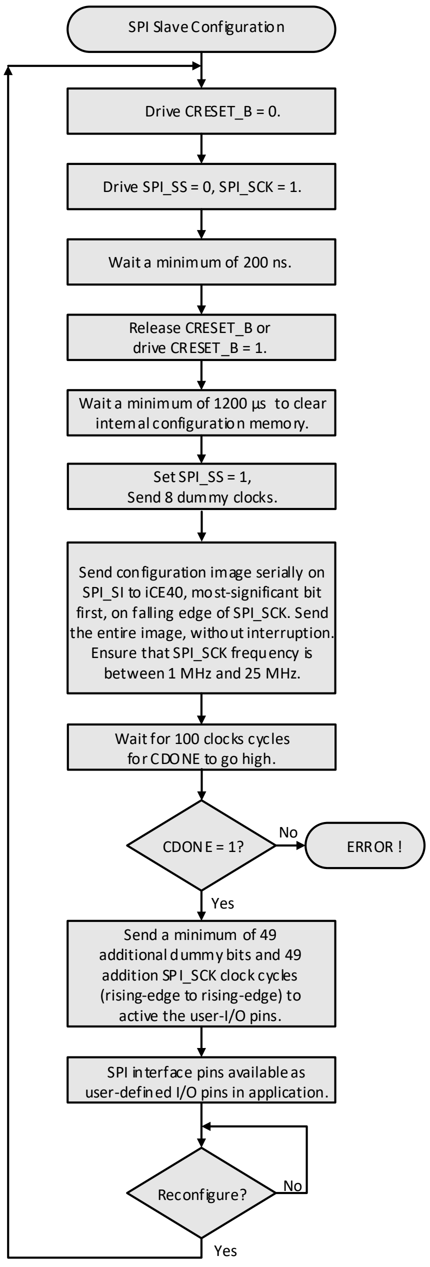
SPI Slave Configuration
Drive CRESET_B = 0.
Drive SPI_SS = 0, SPI_SCK = 1.
Wait a minimum of 200 ns.
Release CRESET_B or drive CRESET_B = 1.
Wait a minimum of 1200 µs to clear internal configuration memory.
Set SPI_SS = 1, Send 8 dummy clocks.
Send configuration image serially on SPI_SI to iCE40, most-significant bit first, on falling edge of SPI_SCK. Send the entire image, without interruption. Ensure that SPI_SCK frequency is between 1 MHz and 25 MHz.
Wait for 100 clocks cycles for CDONE to go high.
CDONE = 1?
No
ERROR !
Yes
Send a minimum of 49 additional dummy bits and 49 addition SPI_SCK clock cycles (rising-edge to rising-edge) to active the user-I/O pins.
SPI interface pins available as user-defined I/O pins in application.
Reconfigure?
No
Yes

Figure 7.3. SPI Slave Configuration Process Flowchart [30]

7.2 Driver Design and Architecture

The driver is designed for transfer of the bitstream file from the ESP32-C3's memory to the FPGA. The goal of the driver is to allow the loading of bitstream directly from the filesystem, either using ioctl or directly, using:

```
cp example_bitstream.bin /dev/ice40-0
```

The driver is developed as part of the SPI drivers and is divided into two parts, as discussed in Chapter 5, the upper half and the lower half. The upper half is created as the universal driver for the iCE40 FPGA, which is independent of the architecture or board used. The lower half is created for the ESP32-C3, namely esp32c3-legacy, as the NuttX code for ESP32-C3, which was used, was moved to the esp32c3-legacy and a new one was being written. For stability and continuity, we decided to continue the development for the old version and have the possibility to modify the code for the new version in the future, once it is stabilized.

Configuration options for enabling and setting the driver were also implemented using the Kconfig system.

7.2.1 Upper Half of the Driver

The upper half of the driver is divided into multiple files. The main part of the upper half of the driver is located in the drivers/spi/ice40.c and include/nuttx/spi/ice40.h.

The upper half of the driver must communicate with the lower half of the driver to execute architecture-specific commands and also needs to know status information about the state of the driver. For this purpose exists the ice40_dev_s structure, defined as follows:

```
struct ice40_dev_s
{
  FAR const struct ice40_ops_s *ops;
  FAR struct spi_dev_s *spi;
  bool is_open;
  bool in_progress;
};
```

This structure requires the lower half of the driver to provide two fields, ice40_ops_s and spi_dev_s. The structure ice40_ops_s stores the functions of the lower half of the driver. Then the structure spi_dev_s is for the driver to access the SPI driver in order to handle the SPI communication with the iCE40 FPGA.

Lastly, there are two booleans, is_open, which serves to check if the device was already opened, and in_progress, which is there to check if the FPGA was correctly initialized and is ready for SPI transfer of the bitstream.

The structure ice40_ops_s is then defined as

```
struct ice40_ops_s
{
  CODE void(*reset)(FAR struct ice40_dev_s *dev, FAR bool reset);
  CODE void(*select)(FAR struct ice40_dev_s *dev, FAR bool select);
  CODE bool(*get_status)(FAR struct ice40_dev_s *dev);
};
```

As seen in the code listing, communication with the lower part of the driver is done through the three functions: reset, select, and get_status. Their usage is explained later in context of the upper half functionality.

The upper half of the driver creates a character device driver in the `/dev` directory inside the NuttX operating system, by default named `/dev/ice40-0`. This initialization is performed by the ice40_register function called from board-level logic, supplying the `ice40_dev_s` structure.

The driver implements the standard POSIX file operations as follows:

```
static const struct file_operations g_ice40_fops =
{
  ice40_open,  /* open */
  ice40_close, /* close */
  ice40_read,  /* read */
  ice40_write, /* write */
  NULL,        /* seek */
  ice40_ioctl, /* ioctl */
};
```

The ice40_open serves to open the character device driver and to ensure that it is opened only once.

The ice40_read call is currently ignored, as it does not make sense in the context of bitstream loading into the FPGA.

The more interesting functions are ice40_write and ice40_ioctl, as both can be used to actually write the bitstream into the iCE40. The ice40_write can be used to load either the whole bitstream at once, or it can be called multiple times in a row to perform the whole upload. It first checks if the driver is in progress to load the bitstream after previous write or IOCTL. If the sequence has not yet been initiated, it starts with that sequence and sets the `in_progress` field in the state structure. The sequence is started by calling the private function `ice40_init_fpga`, and then by calling `ice40_writeblk` to actually write the bitstream into the iCE40. Then it returns the length of the buffer written. After the whole write sequence, the driver must be closed to finish the upload sequence.

The ice40_ioctl function has three implemented options:

- FPGAIOC_WRITE_INIT
- FPGAIOC_WRITE
- FPGAIOC_WRITE_COMPLETE

The FPGAIOC_WRITE_INIT initializes the FPGA by calling ice40_init_fpga, the same as in the case of ice40_write. FPGAIOC_WRITE calls the ice40_writeblk function to write the buffer to the FPGA, and lastly, the FPGAIOC_WRITE_COMPLETE finishes the write by calling ice40_endwrite.

The ice40_close then finally ends the write of the bitstream to the FPGA by calling ice40_endwrite, and closes the whole device.

The previously mentioned private functions ice40_init_fpga, ice40_writeblk, and ice40_endwrite, together with the rest, are implemented according to the iCE40 configuration procedure mentioned in Section 7.1.2.

With the driver's implementation, it was also necessary to add ioctl definitions for the FPGA inside the include/nuttx/fs/ioctl.h file, add the source files inside the drivers/spi/Make.defs, and also add the driver to the correct Kconfig file drivers/spi/Kconfig.

7.2.2 Lower Half of the Driver

The lower half of the driver is again divided into multiple files across two directories, arch/risc-v/src/esp32c3-legacy containing the architecture-specific functions called from the upper half driver, and the boards/risc-v/esp32c3-legacy/common, containing the board specifics and driver registration.

Inside the arch directory, there are the files esp32c3_ice40.c and esp32c3_ice40.h. They implement the functions of the structure ice40_ops_s: ice40_reset, ice40_select, and ice40_get_status. The ice40_reset and ice40_select are used to write to the GPIO pins of the ESP32–C3 to reset the iCE40 via the CRESET and toggle the chip select pin via SPI_SS, respectively. The ice40_get_status then is used to read the GPIO pin to get the iCE40 status from the CDONE pin.

There is also the function esp32c3_ice40_initialize, which is responsible for correct initialization of the GPIO pins, the SPI driver, and their initial configuration.

The source files also must have been added to the correct Make.defs. The following configurations of the board systems were also added to the `Kconfig` in `arch/risc-v/src/esp32c3-legacy`: `ESP32C3_ICE40_CSPIN` is the chip select pin, `ESP32C3_ICE40_CDONEPIN` is the CDONE pin, `ESP32C3_ICE40_CRSTPIN` is the CRESET pin, and ESP32C3_ICE40_SPI_PORT configures to which SPI port is the iCE40 connected.

In the boards directory, the main extension is the implementation of the function `esp32c3_ice40_setup`, implemented in the `esp32c3_board_ice40.c`, which is used to initialize the lower half of the driver using the `esp32c3_ice40_initialize` function and register the upper half of the driver using the `ice40_register` function.

This function is finally called from the `esp32c3_bringup.c` bringup file, and the whole driver is now registered.

Chapter 8
Design of the ICE-V PMSM Adapter Board

The `ICE-V PMSM` adapter board was designed as a bridge between the `ICE-V Wireless` board and the power stage board `3p_motor_driver_1`. The primary objective was to replace the MZ_APO board with a cheap entry-level ICE-V Wireless board while ensuring compatibility with the existing power stage. The adapter board integrates the ICE-V system into the motion control setup by providing the necessary power, communication interfaces, and additional connectivity options.

The design of the adapter board was done using KiCad, a widely used open source electronics design automation (EDA) software. The board design is open source, and its design files, including schematics and PCB layout, are available as a part of the `ice-v-pmsm` project in the Open Technologies Research Education and Exchange Services (otrees) gitlab[16]. The board render from KiCad can be seen in figure 8.1.

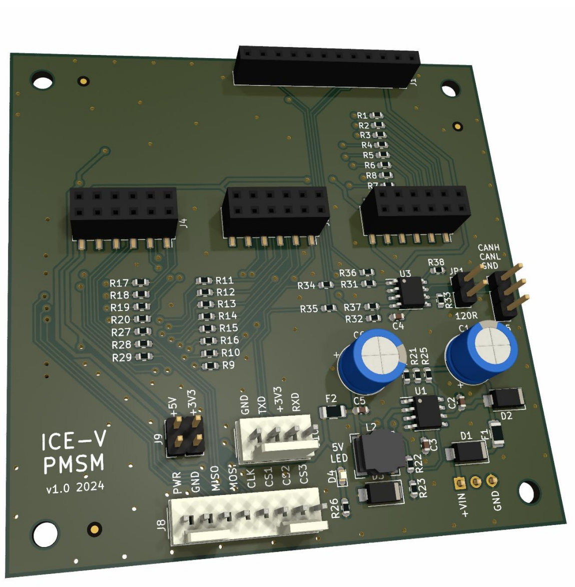

Figure 8.1. Picture of the designed adapter board for the ICE-V motion control system.

8.1 Purpose of the Board

The board has multiple purposes within the motion control system. The first is to connect the signals from both the ESP32–C3 and the iCE40 FPGA to the power stage. Next one is to provide power for the ICE–V board, without the need for multiple external power sources. Another is to provide connection for external peripherals and system, for example, to enable control of the motor from another higher-level system. The last is to mechanically connect the other two boards used in the system.

8.2 Design Considerations

The design of the board was constrained by the two connected boards, from both an electrical and a mechanical point of view. Meeting the requirements for the `ICE-V Wireless` board was easier, as its design [23] was also done in KiCad. In the case of the `3p-motor-driver-1` board it was a little more complicated, as it was developed in a different EDA software, the PEDA [31]

8.2.1 ICE–V Wireless Board

The `ICE-V Wireless` board interfaces with the `ICE-V PMSM` board via four connectors: three PMOD type connectors with FPGA signals and 3.3V power and one 12-pin connector that interfacing signals from ESP32–C3 and 5V power.

The FPGA signals from the three PMOD connectors are routed via zero ohm resistors to the `3p_motor_driver_1`, where they are responsible for controlling the transistors to drive the PMSM motor by three-phase voltage. They are also responsible for getting information from the Hall sensors, incremental rotary counters (IRC), and status of the MOSFET drivers, and for SPI connection to the analog-to-digital converter (ADC). The pin assignments of the FPGA pins to the PMODs are shown in figure 8.1.

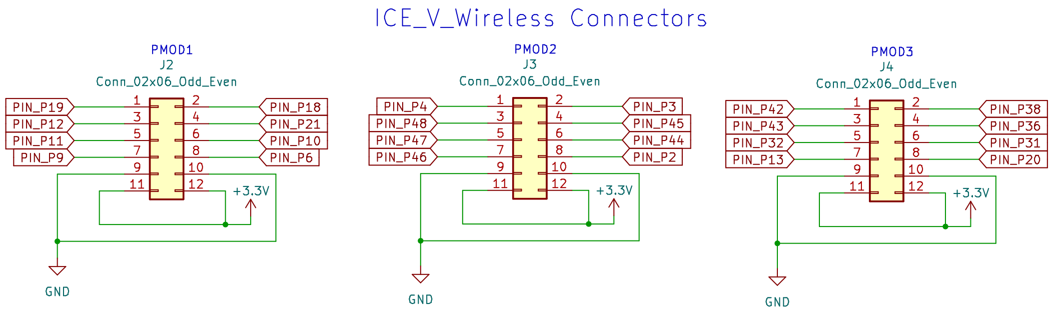

Figure 8.1. ICE-V PMSM PMOD pin mapping.

The exact pin assignments from FPGA pins to the `3p-motor-driver-1` are shown in figure 8.2. The reason for the separation of the signals via the zero-ohm resistors was to be able to disconnect some of the signals if necessary, and in the same time the pads can serve for measurement with an oscilloscope.

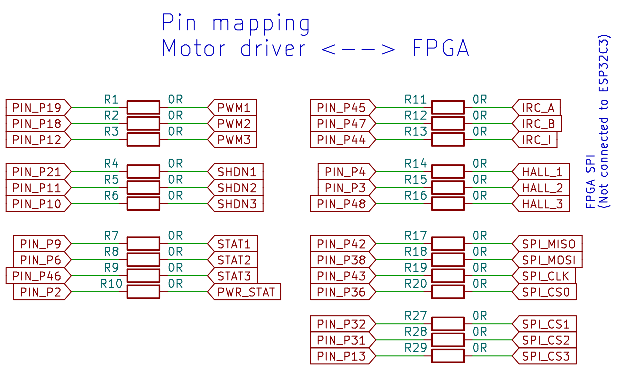

Figure 8.2. ICE-V PMSM FPGA pin mapping

The last connector on the `ICE-V Wireless` board is the 12-pin one with the signals from the ESP32–C3. Its pinout is shown in figure 8.3. The pins are used for the communication interfaces, which are later described in section 8.4.

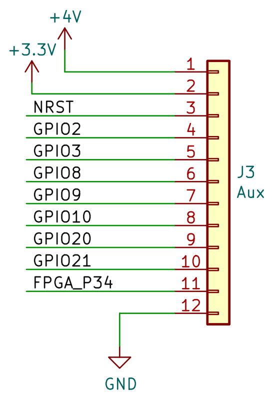

Figure 8.3. ICE-V PMSM EPS32–C3 pin mapping

8.2.2 3p-motor-driver-1 Board

The `3p-motor-driver-1` is connected to the board via one data connector shown in figure 8.4 and also one power connector, which attaches the main power supply from the `3p-motor-driver-1` board, where it is used to power the motor and transistors.

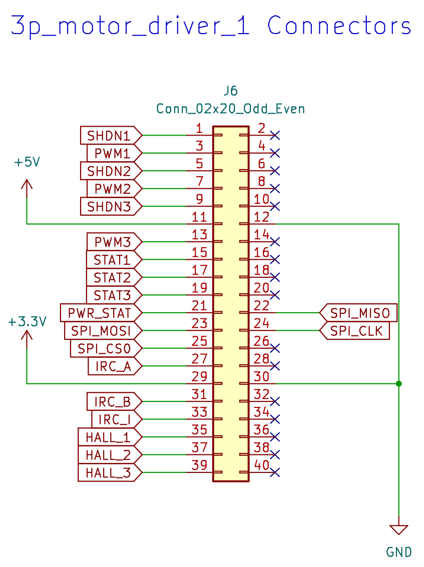

Figure 8.4. ICE-V PMSM 3p-motor-driver-1 board connector

Figures 8.1, 8.2, and 8.4 contain all the information about how the FPGA is connected to the power stage, except for power.

The `3p-motor-driver-1` is powered via an external power supply and its input voltage is typically 24V, but can range from 9V up to 30V maximum. This voltage is also supplied to the ICE–V PMSM board via an additional connector.

8.3 Power Supply

The ICE–V Wireless board is normally powered through its USB port, or it has the possibility of being powered with a LiPo battery. Internally, it then has to convert the input 5V voltage to 3.3V for the ESP32–C3 and also for the PMOD connectors, as well as to 1.2V for the iCE40 FPGA. Both the 3.3V and the 1.2V are created directly from the 5V.

This means that if the designed motion control system is to be operated standalone, the ICE-V PMSM board must provide 5V instead of the USB, created from the input voltage from the `3p-motor-driver-1` board.

A power supply was designed using the MC33063AD, which is a monolithic control circuit containing the primary functions required for DC—to—DC converters [32]. The MC33063AD was used to create a step-down converter. The circuit is shown in figure 8.1.

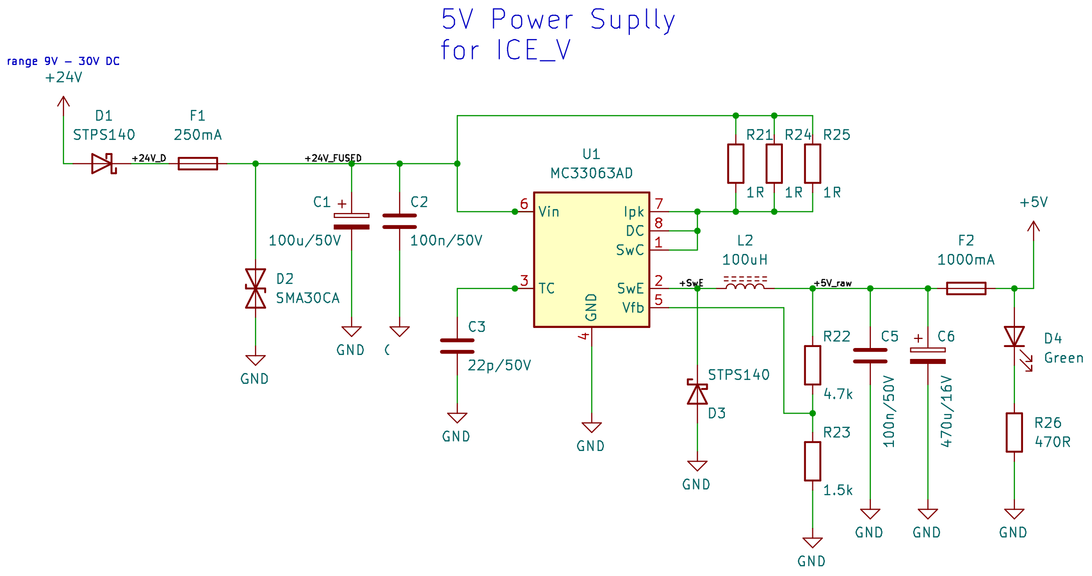

Figure 8.1. ICE-V PMSM 5V Power supply using the MC33063AD

The resistors R22 and R23 serve to set the output voltage, which is calculated with equation (1) and with $R22 = 4700$ and $R23 = 1500$ the output voltage is 5.16, within the tolerated margins for all components.

$$V_{out} = 1.25\left(1 + \frac{R22}{R23}\right), \tag{1}$$

The design of the power supply also includes fuses to protect the boards from overcurrent and also LED indication of functionality.

The `3p-motor-driver-1` also needs to have 5V delivered to the on-board sensors, together with 3.3V. The 5V is already created by the mentioned power supply, but the 3.3V is not. However, it is not necessary to add another power supply for the 3.3V, as the ICE–V Wireless board has the 3.3V power supply for the ESP32–C3 and also for external peripherals via the PMODs. The power supply mentioned uses an XC6222B331MR-G regulator, which according to its datasheet can deliver over 700mA of current, more than enough for the internal needs of the ICE-V Wireless and the sensors on the `3p-motor-driver-1`.

8.4 Communication Interfaces

The ICE-V PMSM board has also added additional communication interfaces. Their schematic can be seen in figure 8.1. The first added interface is the connector J8, which enables additional devices to be connected to the FPGA, such as different types of encoders or sensors. There is the possibility of connecting multiple devices, as there are in total 3 additional chip select pins provided.

The power output on the J8 connector has a selectable voltage output, either 5V or 3.3V, which extends the range of peripherals that can be connected. This can be done via the JP2 jumper.

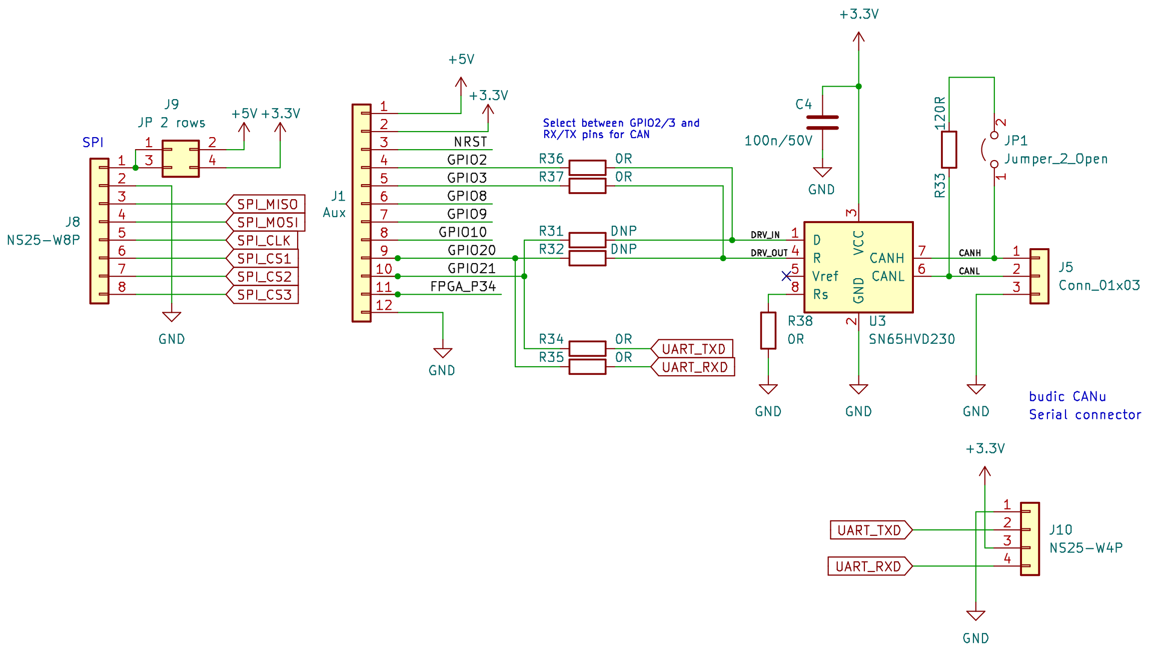

Figure 8.1. ICE-V PMSM ICE-V PMSM Communication Interfaces

Next, there is the possibility of connecting to the board via CAN interface, for which the SN65HVD230 CAN driver was selected. It operates on 3.3V, which is required for the ESP32–C3, and is compatible with the specifications of the ISO 11898-2 High-Speed CAN Physical Layer standard with speeds up to 1 Mbps according to the datasheet. The user of the board can choose which pins of ESP32–C3 will be used for the CAN interface by selecting whether the resistors R36 and R37 (default) or R31 and R32 are to be populated. In the default configuration, the main serial interface of the ESP32–C3 is left as the UART and is directed to connector J10.

The board also has an optional CAN termination of 120 Ohms, which can be enabled by shorting the JP1 jumper.

Chapter 9
PXMC Library on the ICE-V

9.1 Library Description

The open source Portable, highly eXtendable Motion Control library (PXMC) [4] developed by PiKRON Ltd. and available from the company's gitlab [26], is used in this project to control the movement of the PMSM motor. There is also available documentation on the project's webpage[33]. The structure of the PXMC library is shown in Figure 9.1

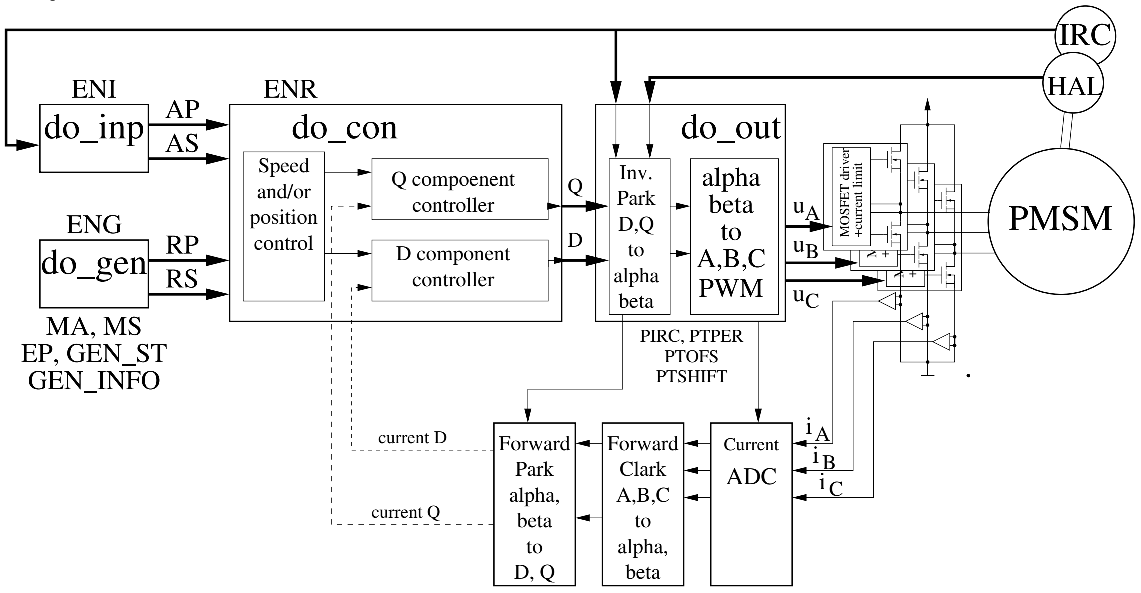

Figure 9.1. Overview of the structure of the PXMC library [33].

On the right side there is the PMSM motor together with its coupled Hall sensors and the incremental encoder sensor (IRC). The position of the IRC and partially the Hall sensors is taken to the function `do_inp` (if the enable input flag ENI is set), which calculates the actual position (AP) and the actual speed (AS). These two values are then given to the controller implemented under the function `do_con`.

During the same time, if the Enable generator flag ENG is set, the requested position (RP) and the requested speed (RS) are calculated and also given to the controller.

The controller then based on the selected mode calculates the quadrature (Q) and direct (D) components of the currents or voltages. The controller is implemented as two PID regulators for each of the two components of the currents. In our case, the controller is working in a voltage mode, which means that only the speed and position difference are used to calculate the D and Q currents, and the real measured D and Q currents are not used to compensate, as is depicted by the dashed lines from the forward Park and Clarke transformations.

Another possible operating mode of the PXMC library is the current mode, in which the measured D and Q currents can be used to improve the performance, although it has not been implemented for our setup, mainly due to higher computational power and higher sampling frequency required for current control loop than achievable on the ESP32–C3, as it does not contain FPU and all floating point arithmetic calculations are emulated in software and are very slow.

From the controller the calculated D and Q currents are passed into the `do_out` function, which is responsible for calculating the final voltages for the 3-phase PMSM motor using the inverse Park and Clarke transformations.

9.2 The Command Processor

The PXMC library implements a command processor, which can be used as an alternative to the C functions call interface to interact with the system. The command processor allows the user to run various functions of the PXMC library and also change various settings and parameters while controlling the motor. This can be useful for testing and debugging, as well as fine-tuning the parameters, for example of the PID regulators.

The generic command format is as follows:

```
<COMMAND>[<PARAM1>][<OPCHAR>][<VALUE>]
```

The `COMMAND` is the name of the command, `PARAM1` is the first parameter, `OPCHAR` is for the operation character, which is either `":"` character for write or `"?"` character for read, and lastly the `VALUE` is the value you want to set. The value is only used with the set opchar `":"` character, for reading the value does not make sense.

There is the second type of `OPCHAR` characters, which are used in responses to commands. The first response `OPCHAR` is the `=`, which is a response to the previous command with `":"` The original command is also repeated, so there is the possibility of sending batch commands.

The second response `OPCHAR` is the `"!"`. It is used for the asynchronous output, which is not directly the result of a command, but rather is printed later. An example of this can be the `RA:` command, which is used to get confirmation that the command is finished. This command sends `RA!` upon successful completion of the currently running motor operation, or `FAIL!` in case of failure during execution.

The commands also follow a convention, where the commands written as upper case are meant for full control of the motor via, for example, serial interface, and the lower case commands are used for debugging or other internal usage.

It is best to show the commands in an example. Let us say that we want to set the speed of the motor connected to the ICE-V to 100 units (this is not done in revolutions per second, but fractions 1/256 of IRC per sampling period). The command would be `SPD` and we want to control the axis `A` as PXMC is capable of running multiple axes at the same time and we have only one motor connected. This would result in the following command:

```
SPDA:100
```

To show how to use a command to get a value, we can look at how to ask for the current position of the motor. For this purpose, there is the `AP` command to get the actual position, again for the `A` axis. The resulting command is then:

```
APA?
```

The last example of commands is how to stop the motor on the `A` axis. This can be done using the command `STOP`. In this case, we are giving the command and not asking, but the value is here omitted as it would not make sense. The result is then:

```
STOPA:
```

If we had multiple axes and wanted to for example stop the second, `B` axis, the command would simply be:

```
STOPB:
```

9.2.1 Available Commands

Here are some of the most useful commands. The full list can be found in the documentation or related source files, namely inside the `pxmc_basecmds.c` inside the PXMC library or inside our application in the `appl_pxmccmds.c` file. In addition, all available commands can be printed using the available `help` command. This command lists all the commands implemented from both the library and the application. It is also possible to add or modify the commands, which has been done in this thesis as well to implement some debugging functionality.

- **Speed control**:
  Motor speed control is performed with the `SPD` or `SPDFG` (Fine grained speed) commands. They are able to set the parameters only; you cannot ask for current speed using them. According to documentation, the `SPD` must be in the range of $\langle 1, 1500 \rangle$ and the `SPDFG` in the range of $\langle 100000, 98500000 \rangle$.
- **Position control**:
  Position control have multiple commands. The user can obtain the actual position using `AP`, or go to the position by either `GA` to go to the absolute position from the start, or `GR` to go relative to the current position. The starting position (position of zero) can be set by the command `ZERO`, such as `ZEROA:`.
- **Controller parameters**:
  The command processor allows the user to set parameters of regulators and other functions inside PXMC during execution. Each change is applied immediately after the command execution, even during axis movement.
  For example, the PID constants of the controller can be set by `REGP`, `REGI` and `REGD` respectively. The rest of the available commands are listed in the source code.
- **General debugging commands**:
  There are two useful commands for debugging the application, the `AXERR` and the `PURGE`. The `AXERR` returns the last axis error code. The axis errors can be found in the `pxmc.h` file in the library, where they are defined in hexadecimal.
  The three most common are the `AXERRA=262` (0x106), which is named in the header file `PXMS_E_MAXPD`, and it means that the difference of actual and requested position is above the limit, the `AXERRA=263` (0x107), named PXMS_E_OVERL, which means that an overload error has occurred, and the `AXERRA=264` (0x108), named PXMS_E_HAL, which means there was a problem with the Hall sensors.
  The complete list of errors is in Appendix C.

9.2.2 Application Specific Commands

It is possible to add custom commands to the PXMC command processor at the application level. The application commands are defined in the `sw/nuttx-omk/ice_v_pmsm/appl_pxmccmds.c` file.

For our application, a command was added to measure the slack of the control loop, which is useful to measure if the application is keeping up and the control is on time. Here, we use this added command to show how to implement similar custom commands as well on the application level.

The command implemented for our application was REGSLACK. The REGSLACK? returns the maximal measured timing slack of our application, which is the difference between the requested time of the next execution and the actual time. The command REGSLACK: resets the measured value, which is done by writing the command without the value parameter.

The command needs to be registered in the appl_pxmccmds.c file, so that the command processor knows about it. The registration is done with the following line, where it is registered with the command name and also the help message.

```
cmd_des_t const cmd_des_slack_max =
{
    0,
    CDESM_OPCHR|CDESM_RW,
    "REGSLACK",
    "maximum timing slack of sampling period",
    cmd_do_slack_max,
    {}
};
```

The structure cmd_des_t stores all the necessary parameters for the command creation and is defined as:

```
typedef struct cmd_des{
  int code;
  int mode;
  char *name;
  char *help;
  int (*fnc)(cmd_io_t *cmd_io, const struct cmd_des *des, char *param[]);
  char *info[];
} cmd_des_t;
```

The most important fields are the name, which determines how the command will be called from the command processor, the mode, which determines if the command is read, write, or both, and fcn, which is the function that will be called.

In our case, the function cmd_do_slack_max checks if we want to read or write and calls the corresponding function from the appl_utils.c, which handles the rest.

9.3 Application Description

The application used in this thesis extends the code developed for tests performed on the Raspberry Pi with FPGA, which is available in the gitlab repository [34].

The application is developed as a NuttX application, which is to be run inside the NuttX shell. The user can then control the motor with this application through a simple command processor implemented by the PXMC library, or develop another application that can utilize the PXMC library directly, for example to control the motor from external peripherals such as CAN interface added to the ICE-V PMSM board.

9.3.1 Application Structure

The application with PXMC developed for NuttX RTOS does not use the standard `apps` directory, but instead uses exported NuttX. The applications are then built against the exported files outside the source tree.

The application used is registered inside NuttX under the name `ice_v_pmsm`, and its source code is part of the ice_v_pmsm project on the Open Technologies Research Education and Exchange Services (otrees) gitlab [16].

The application is compiled as part of the entire NuttX, so the result of its compilation is the entire NuttX binary. When building the binary for NuttX with PXMC, the OMK build system is used.

Ocera Make System (OMK) is an advanced make system written entirely in GNU make. Its use requires only GNU Make and standard UNIX utilities and aims to simplify the whole build process. [35]

The application was based on the NuttX OMK template. The template creates the NuttX application and registers it, so it can be run through the NSH shell. The template also mounts a ROMFS filesystem at /etc and provides a system init script at /etc/init.d/rc.sysinit and a startup script at /etc/init.d/rcS. Inside the rcS startup script file there is the command to load the created bitstream to the iCE40:

```
cp /etc/fpga/ice_v_ice40_pmsm.bin /dev/ice40-0
```

This command is using the newly implemented NuttX driver. There is also the possibility to run the `ice_v_pmsm` at the start, but currently the user has the ability to choose what to do with the system, as the NuttX shell (NSH) is run at the start.

To run the application at startup, the command, in our case name of the application

```
ice_v_pmsm
```

can be added at the end of the `rcS` file. The same is true for any other command that the user or the developer wants to perform at startup.

9.3.2 Running the Application

The application `ice_v_pmsm` that can be started from the NSH command line interface. When started, the PXMC application initializes and then runs the earlier described command processor waiting for input on the system console. The first is the initialization of the application, in which the background task of the PXMC is started. Then the application runs a command processor, so the user can interact with the PXMC task via commands on the terminal and not just another application.

The PXMC base thread is spawned as a task with a high priority, in our case 200 (out of 256), to ensure it will not be delayed by the scheduler and other tasks. The base thread is the center of the PXMC and it runs a control loop in every sampling period.

Each sampling period then calculates the position of the motor by using the function `pxmc_sfi_input`, then calls the controller, and sets the output PWM to the motor via the function `pxmc_sfi_controller_and_output`. After that, it transfers the output to the FPGA to generate the PWMs and receives the data from Hall sensors, IRC value, and currents. Then another position setpoint is generated with the `pxmc_sfi_generator`, and lastly, there is the debug function `pxmc_sfi_dbg`.

The sampling frequency can be set from the command processor with the command `REGSFRQ`. The command is able to read the current frequency and set a new one, even during the execution of the application.

Using the previously described and implemented command `REGSLACK`, we measured the slack of the control loop (and added ways to measure it). The NuttX maximal measured latency from the planned sampling time is about 1.5 ms with the setup and configuration used. For this reason, we set the control loop frequency to 500 Hz. The precise reason for this is unclear, but it is due to the NuttX scheduler, and an important contribution is caused by the need to read program code from SPI connected program Flash, and it needs to be studied more in the future.

Chapter 10
PysimCoder Models on the ICE-V

PysimCoder is an open-source graphical tool for control system design and real-time code generation, developed by Professor Roberto Bucher from the University of Applied Sciences and Arts of Southern Switzerland. The source code is available on the project github [27] and there is also extensive documentation in Professor Bucher's book Python for Control Purposes[5].

PysimCoder serves dual purposes: It can be used as a simulation tool for simple control schemes and as a rapid prototyping application for real-time control systems. The tool is compatible with various target operating systems, including GNU/Linux (with or without a preemptive RT kernel) and the NuttX RTOS, which is used in this thesis.

The application consists of an extended Python control library and a graphical block editor with a code generator. It allows for the rapid development of control algorithms for embedded systems, where the user can design the control system via a graphical block editor. This lets the user not focus on implementation details but rather on the control system itself.

The designed block diagram then can be either simulated, or compiled for the target hardware and then deployed.

10.1 Block Editor

PysimCoder's graphical block diagram editor provides a user interface that allows users to construct control systems visually, similar to how they would do so in other tools like Simulink. The editor consists of a diagram window for building control systems and a library window that contains a collection of prebuilt blocks categorized into different libraries, such as Input, Output, Linear, or NuttX.

Users can easily drag and drop these blocks into the diagram window to create their control systems. Each block within the editor is customizable, with specific parameters that can be adjusted by double-clicking on the block. This allows for fine-tuning of control algorithms, such as setting controller gains or specifying device names. Many blocks also support multiple inputs and outputs, which can be configured through a right-click menu.

The block editor and the library can be seen in Figure 10.1, which shows a simple test model in pysimCoder.

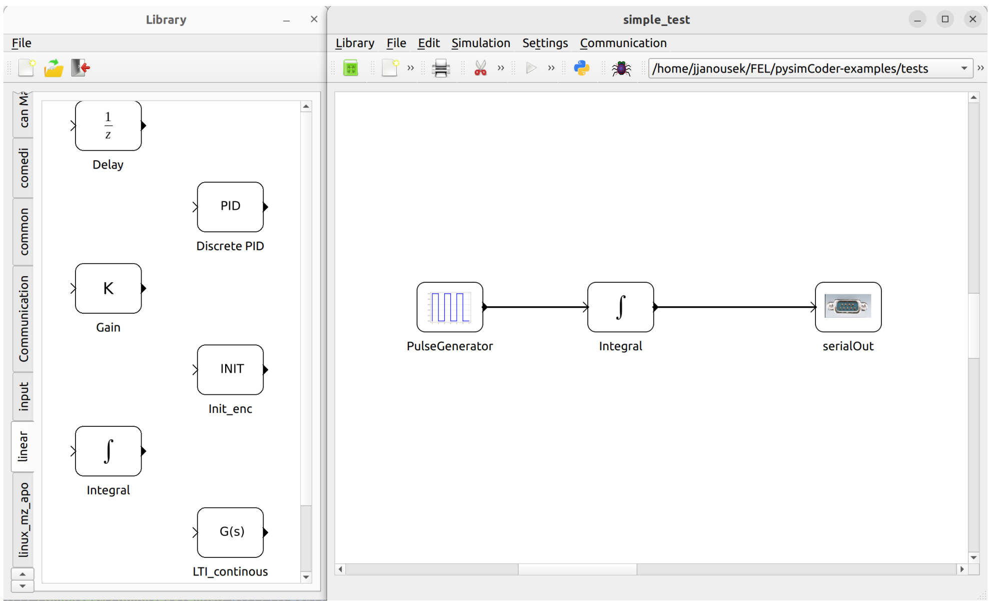

Figure 10.1. PysimCoder's library (left) and diagram window (right)

10.2 Source Code Organization

The source code of the pysimCoder is organized into several directories, with the two most relevant for this thesis being "resources" and "CodeGen". The "resources" directory contains JSON declarations for blocks and the associated Python code, while the "CodeGen" directory includes Makefile templates and C code of the blocks for supported targets.

Each block has three main components: the declaration of the block, its Python file, and the C code.

10.2.1 Resources

The `resources` directory contains the upper part of the pysimCoder blocks, the declarations, and the Python code. The declaration of the block is inside its parent library folder in `resources/blocks/blocks/`. Each of the library folders contain `.xblk` files in JSON format and provides a declaration of the block parameters, such as input, output, or its icon.

The `.xblk` JSON file contains the following list of parameters:

- **lib** – Name of the parent library
- **name** – Name of the block
- **ip** – Number of inputs
- **op** – Number of outputs
- **stin** – 1 if number of inputs can be set by user, 0 otherwise
- **stout** – 1 if number of outputs can be set by user, 0 otherwise
- **icon** – Name of the icon
- **params** – Name of Python function with list of parameters
- **help** – User help to be displayed in the block

The Python file is located inside the library folder at `resources/blocks/rcpBlk/`. The main part is the function called from the `.xblk` file, and its input parameters must match the parameters defined inside the `.xblk` file. The most important part of the Python function is then to call the `RCPblk` function of the pysimCoder, which is a procedural interface to the RCPblk library. The call contains the name of the corresponding C function, as well as other parameters of the block.

10.2.2 CodeGen

The last part of the pysimCoder block is inside of the `CodeGen` directory. Inside, there is the C code responsible for executing the block's functionality on the target.

The C function, which was referenced in the Python part of the block, is responsible for the correct execution of the block during the whole model run. Typically, the function has the following form:

```
void example_function(int flag, python_block *block)
{
  if (flag==CG_OUT){          /* input / output */
    inout(block);
  }
  else if (flag==CG_END){     /* termination */
    end(block);
  }
  else if (flag ==CG_INIT){    /* initialization */
    init(block);
  }
}
```

The `example_function` receives a flag, which indicates the state of the execution, and a pointer to the block structure. The state of execution can be one of the following three states:

- **CG_OUT**:
  Performs the functionality of the block from input to output.
- **CG_END**:
  Stops the output of the block and correctly terminates everything needed.
- **CG_INIT**:
  Initializes the block, the underlying drivers and everything else required for block operation.

10.3 Using PysimCoder with NuttX

The pysimCoder models can be run on the NuttX RTOS, which was used in this thesis for the motion control system on the ICE-V board. Here are the necessary steps to run the pysimCoder models on the NuttX RTOS. The pysimCoder does not add the generated application to the NuttX `apps` directory, but rather uses the exported NuttX, similarly to the PXMC application described in Section 9.3.

The exported NuttX files must be copied or linked into the `\nuttx` directory under the `pysimCoder/CodeGen/`, as they are needed for model compilation. With the exported NuttX files in the correct pysimCoder directory, the C files must be compiled. This can be done by going to the `CodeGen/nuttx/devices/` directory and running the `make` command.

After the compilation of the NuttX blocks, the user can compile the pysimCoder model. However, it is necessary to select the correct Template Makefile `nuttx.tmf` in the pysimCoder block editor.

The generated loadable executable is then a standard NuttX operating system built according to its configuration prior to export. The binary file can be loaded onto the chip depending on its type and programming interface.

Inside NuttX, the generated code from the pysimCoder model is available as `main` application, and it can be run with the following command:

```
nsh> main
```

10.4 Created Models for the ICE-V

To test the pysimCoder on the ICE-V, two models were created. The first is the `nuttx_spi_pmsm_align_check`, and its purpose is to check the alignment of the Hall sensor position to the incremental rotary counter. The second is the `nuttx_spi_pmsm_cl_pid_q`, which implements closed loop control of the motor.

Both implemented models use the `Phase3Motor` implemented for communication with the FPGA subsystem via SPI. This block was implemented as part of the `rpi-mc-1` project [10] and currently is in the mainline of pysimCoder.

10.5 Model for PMSM Align Check

The Permanent Magnet Synchronous Motor used in our motion control has coupled Hall sensors and an incremental rotary counter with index, both of which give information about the motor's absolute position. The created model can be used to determine how these two positions are related to the motor phases.

To help with the process, the `PMSM Align` block was used. It computes the electric angle of the PMSM form the Hall sector, encoder position, last position of encoder's index, and index count. It also has configurable offsets for both the Hall sensor and the IRC. The block uses the Hall sensors until the encoder hits its index for the first time, after that it uses the more precise IRC.

The Hall sensor and IRC offsets must be correctly determined for the correct commutation of the motor. For this purpose, a pysimCoder model was designed. The model is shown in Figure 10.1.

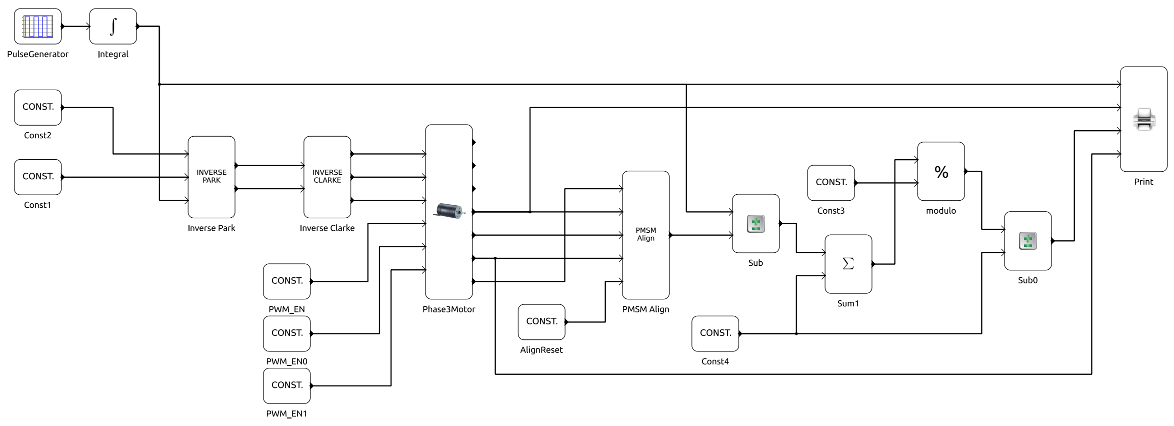

Figure 10.1. PysimCoder Model for PMSM Align Check

10.5.1 Structure of the Model

Inside the model, there are multiple important sections. The first section is the input of the `Phase3Motor`. The input of the block is divided into three PWM signals and the corresponding three PWM Enable signals. The PWM Enable signals can be set to either 1 to enable the PWM output to the motor, or 0, to disable it. In the model, all three constants are set to 1.

The PWM signals are calculated from direct (D) and quadrature (Q) values using inverse Park and Clarke transformations. The Inverse Park transformation block has three inputs used. The first two from the top correspond to the quantities D and Q, and the third one corresponds to the electric rotation angle of the motor.

The next part is the output of the `Phase3Motor` block and the `PMSM Align` block. The `Phase3Motor` has in total seven outputs. The first three are the measured currents that flow through the motor as they are measured. That means they are 12 bit unsigned numbers (0 - 4098) and the zero current corresponds to half of the range - 2048.

The next three outputs are related to IRC. The fourth from the top is the IRC position, the fifth is the IRC index position, and the sixth is the IRC index count. The last, the 7th output, is the Hall sector (0-5).

The `PMSM Align` block then takes the IRC and Hall outputs from the `Phase3Motor` and calculates the resulting electrical angle of the PMSM. The block also has an align reset input, which when active forces the block to use Hall position and not to switch to encoder position. For our purposes, we connect a constant to this block with either 1 or 0, depending on the need.

The last section of the model is responsible for calculating the position difference between the requested position and the calculated position and cleaning it of the periodicity. The difference between the two angles ϕ_{diff} is then calculated according to the formula (1). The formula is calculated using the provided pysimcoder blocks.

$$\phi_{\text{diff}} = \left(\left(\phi_{\text{requested}} - \phi_{\text{calculated}}\right) + \pi\right) \pmod{2\pi} - \pi \tag{1}$$

10.5.2 Model Usage

For the purpose of finding the correct Hall and IRC offsets, the Q value is set to zero, and the D value is set to an arbitrary low value to induce magnetic field in the windings, enough to make the motor shaft spin during model execution, but not high enough to burn the motor. The rotation of the motor is then controlled via the electric angle input to the inverse Park transformation. In the model, the angle is generated as a triangle signal. The signal was created by integrating a square signal of amplitude 4, with a period of 8π and 50% duty cycle, which means that the motor should perform four electric rotations clockwise and return to zero, on repeat.

By setting the non-zero value only to the D axis and then changing the electric rotation of the motor, we are trying to control the motor in a feedforward manner. The motor needs to have no load connected to its shaft for this to work. The motor then should follow the angle input to the inverse Park transformation, after some initial jump in position to match the magnets position to the induced magnetic field.

With the motor following the generated angle, we can proceed with offset calibration. The first offset that needs to be calibrated is the Hall position offset. We do not want to use the IRC position, so we need to set the `AlignReset` constant to 1. Then the calculated position will be only from the Hall sensors.

We need to set the Hall offset so that the resulting difference in position will be zero, which can be seen on the printed output to the NSH inside the NuttX. In the case of our motor, this was equal to -2.

After the Hall offset is configured correctly, we can calibrate the IRC offset. First, we need to set the `AlignReset` constant to 0 to enable the use of IRC. We need to ensure that the generated angle trajectory allows the motor to turn multiple rotations, so the motor will certainly go over the IRC index position, which allows the `PMSM Align` block to switch from using the Hall sensor position to the IRC position.

The IRC offset can be calculated as the mean value of the position difference over multiple full periods of the generated and requested position. In the case of our motor, the calculated mean difference was -0.15, which is then the configured IRC angle offset.

The measured data from after the calibration procedure was successfully performed are shown in Figure 10.2.

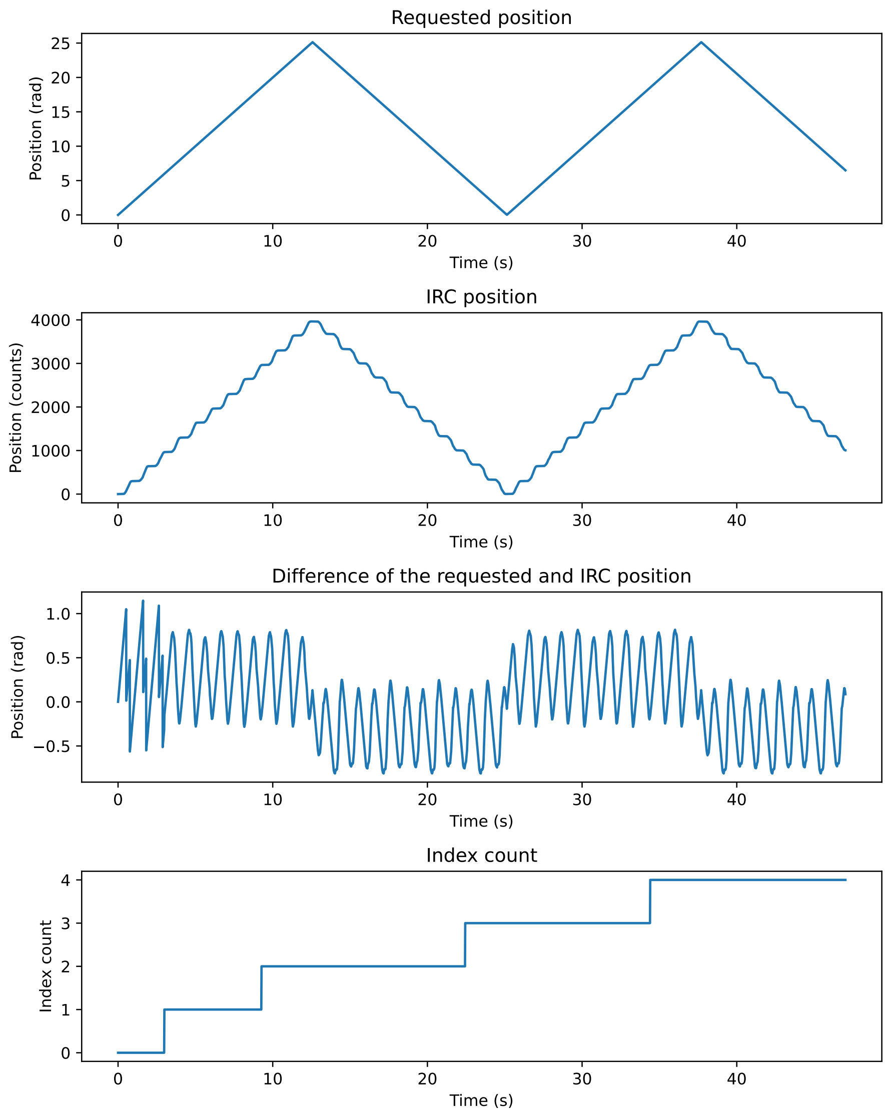

Figure 10.2. Graphs of the measured values from PMSM Align Check

In the first graph, we can see the requested position in radians. The second graph shows the position of the motor as seen from the incremental rotary counter. In the third graph, we can see the measured difference between the requested position and the calculated one. Here we can see that the difference oscillates around zero and there is some lag visible. That is the reason for calculating the IRC offset as a median over multiple periods.

At the beginning of the third graph there is also visible change of the shape of the difference curve. The reason is that initially, the position is calculated from Hall sensors, which is much more coarse, and after the first IRC index is found, the position starts to be calculated from the IRC position.

The moment of the first IRC index is clearly visible on the fourth graph and corresponds to the change in the third one.

10.6 Model for PMSM Closed Loop Control

After the Hall sensor and IRC alignment was calibrated, we can use the pysimCoder to actually control the motor. As part of this thesis, an example model was created to control the position of the motor.

The designed pysimCoder model can be seen in Figure 10.1.

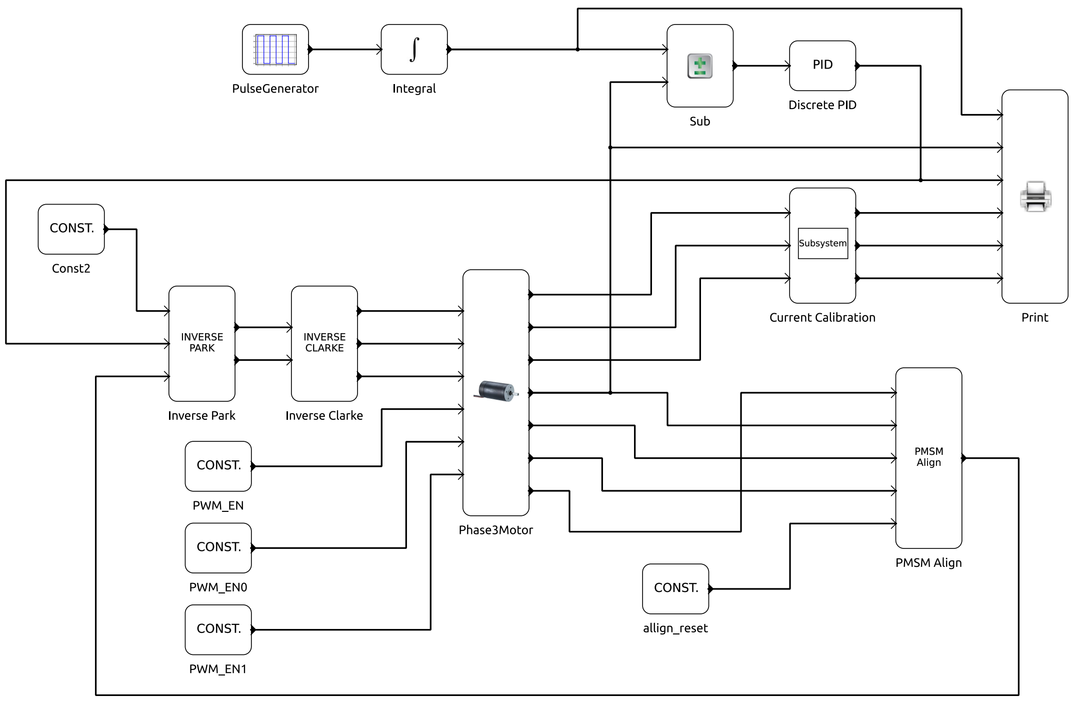

Figure 10.1. PysimCoder Model for PMSM Closed Loop Control

The model uses the calculated position from the previously used and configured `PMSM Align` block as the angle input to the inverse Park transformation. This time, the D component is set to zero and the Q component is used to control the motor.

Position control is achieved by using the PID controller, where the error value at its input is calculated from the difference of the desired setpoint and the measured position of the motor from the IRC.

The desired setpoint in this example was generated using a setup similar to the previous model used for alignment 10.5. The desired trajectory was generated by integrating the output from the `PulseGenerator` block, this time with arbitrary parameters, in our case amplitude of 2000, period of 10 seconds, 50% duty cycle and symmetric around zero.

This example model also measures the motor currents, although they are not used for control in this setup, mainly due to the limited control loop frequency. Both of these examples are running at 200 Hz, at higher frequencies is occurring a timing overrun, and the control loop is not keeping up.

10.6.1 Current Calibration Subsystem

At the beginning of this thesis, a Hall current sensor calibration was designed and conducted. In the designed model, a pysimCoder subsystem is implemented that calibrates the measured currents. The detail of the subsystem is shown in Figure 10.2.

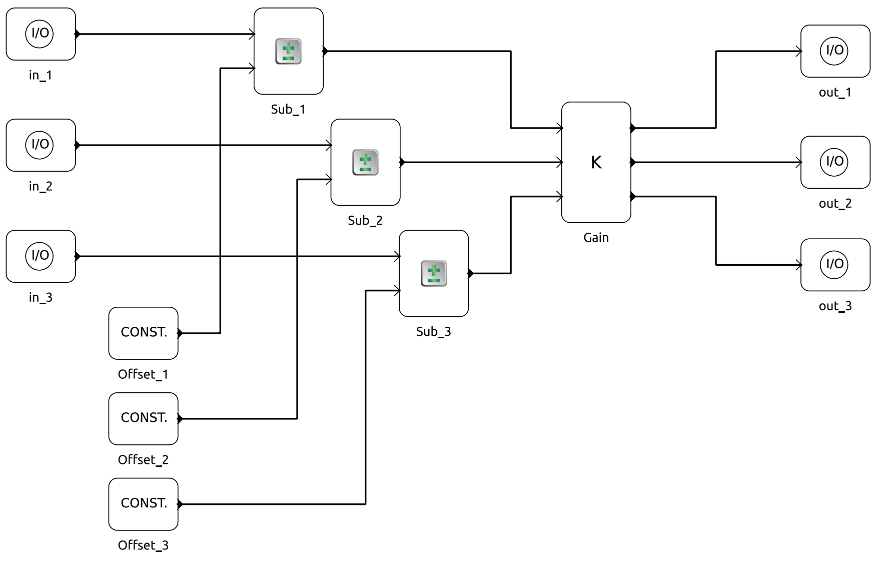

Figure 10.2. PysimCoder Subsystem Model for PMSM Current Calibration

The subsystem is using the created 3x3 matrix calibration. The first step is to subtract the current offsets, which were measured while no current was flowing through the sensors. Afterwards, the `Gain` block is used to perform the matrix multiplication of the three input currents with the calibration matrix.

Here we take advantage of the pysimCoder's ability to use python syntax inside the block's parameters, as the matrix can be inserted in a format of 2D python array. The current matrix and offsets used correspond to the ones calculated in Section 4.3.

The motor position control using the designed block was successfully performed, and the results are shown in Figure 10.3.

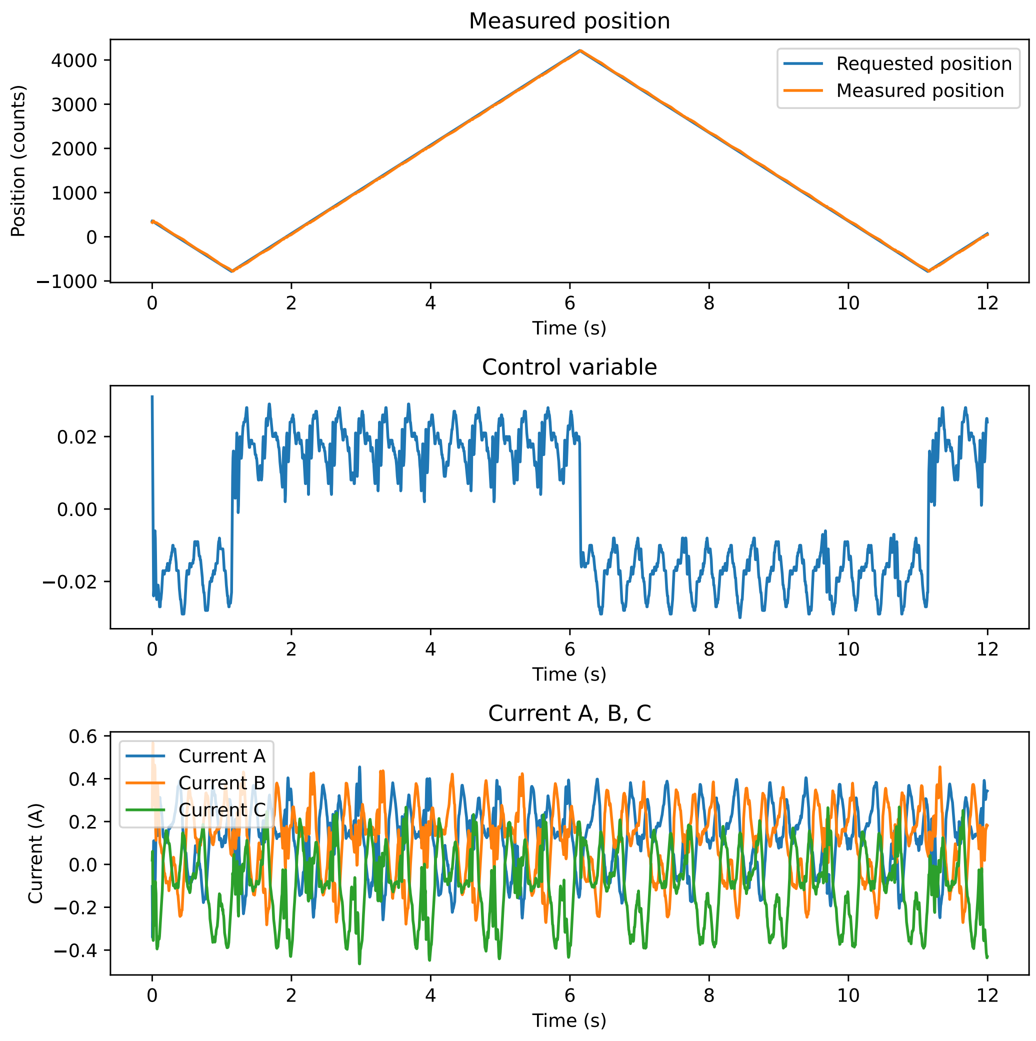

Figure 10.3. Graphs of the measured values from PMSM Closed Loop Control

We can see that the motor successfully followed the requested position. The measured currents are also calibrated by the model design.

Chapter 11
Conclusions

This project successfully demonstrated the integration and functionality of motion control systems on the MZ_APO board and the ICE-V Wireless board. The focus of this thesis was on the calibration of the current sensors used by the power stage and development of a new system using the ICE-V board.

11.1 Current Sensor Calibration

The calibration of current sensors was an important part of the thesis, improving the precision of current measurements in the motor control system. The calibration process began with the collection of raw current data through the PXMC library, which was then processed to generate a calibration matrix using both 3x3 and 2x2 matrix formats, where the second one uses the Clarke transformation to reduce the system to the orthogonal vector base.

The created calibration matrix compensated for sensor crosstalk and other systematic errors. This was later used in the implementation of pysimCoder models for motor position control and also used in the motor control system done by Ing. Damir Gruncl in his diploma thesis[17].

11.2 Development of the Motion Control System using the ICE-V Board

The development of the motion control system on the ICE-V board represented a significant portion of the thesis, involving several steps:

- **System Design and Integration**:
  The ICE-V board was selected as a cost-effective alternative to the MZ_APO board. It is an inexpensive board with the interesting combination of a RISC-V based ESP32-C3 microprocessor with the small iCE40 FPGA. The new system was designed to use the existing`p-motor-driver-1` power stage, utilized already with the MZ_APO. This meant that a new adapter board had to be designed to connect the ICE-V board with the power stage, which was described in the Chapter 8.
- **Firmware and Driver Development**:
  NuttX RTOS was selected for the ESP32-C3, which would then run the control software. An important part of the development involved programming a custom NuttX driver for the ESP32-C3 to handle the FPGA's configuration. This driver was responsible for loading the FPGA with the necessary bitstreams at startup to be later utilized by the control software.
- **Control Software Implementation**:
  The control software was developed using the PXMC library and also using the pysimCoder, which were adapted to run on the NuttX operating system on the

ESP32C3. The control software with the PXMC library was used to control the PMSM position and speed, and it can also be used to perform the current calibration.

The pysimCoder was used to create models for the PMSM control and also to perform the alignment of the Hall position sensors and the IRC to the motor electrical angle.

11.3 Further Development Options

There are several options for the further development of the motion control system presented in this thesis. One significant enhancement would be replacing the current ESP32-C3 microcontroller with the newer ESP32-C6. The ESP32-C6 features a dual-core RISC-V architecture, which provides an opportunity to offload motor control tasks to the second core. Specifically, the second core could be dedicated to running the control loop for the PMSM, potentially allowing for an increase in the control loop frequency and allowing the use of the full field-oriented control using current feedback.

The ESP32-C6 has already been used with NuttX RTOS to create small robotic platforms during teaching the Microcomputer Engineering with Space Applications course at the Luleå University of Technology in Kiruna, Sweden, where I assisted Dr. Píša with the practical aspects of the course. The experience gained from these sessions proved that the ESP32-C6 can be successfully used with the NuttX RTOS and it would be interesting to utilize it further.

Another area of potential development is the modification of the FPGA bitstream to incorporate the Park and Clarke transformations directly within the FPGA. Currently, these transformations are performed by the main processor, which increases its computational load. By moving these calculations into the FPGA, the processing burden on the main processor could be significantly reduced, freeing up resources, and potentially improving overall system performance.

Appendix A
Schematics and Fabrication Outputs of the ICE-V PMSM Board

Below are the schematic diagram and fabrication outputs for the ICE-V PMSM board, as designed in KiCad and available at the project's gitlab.

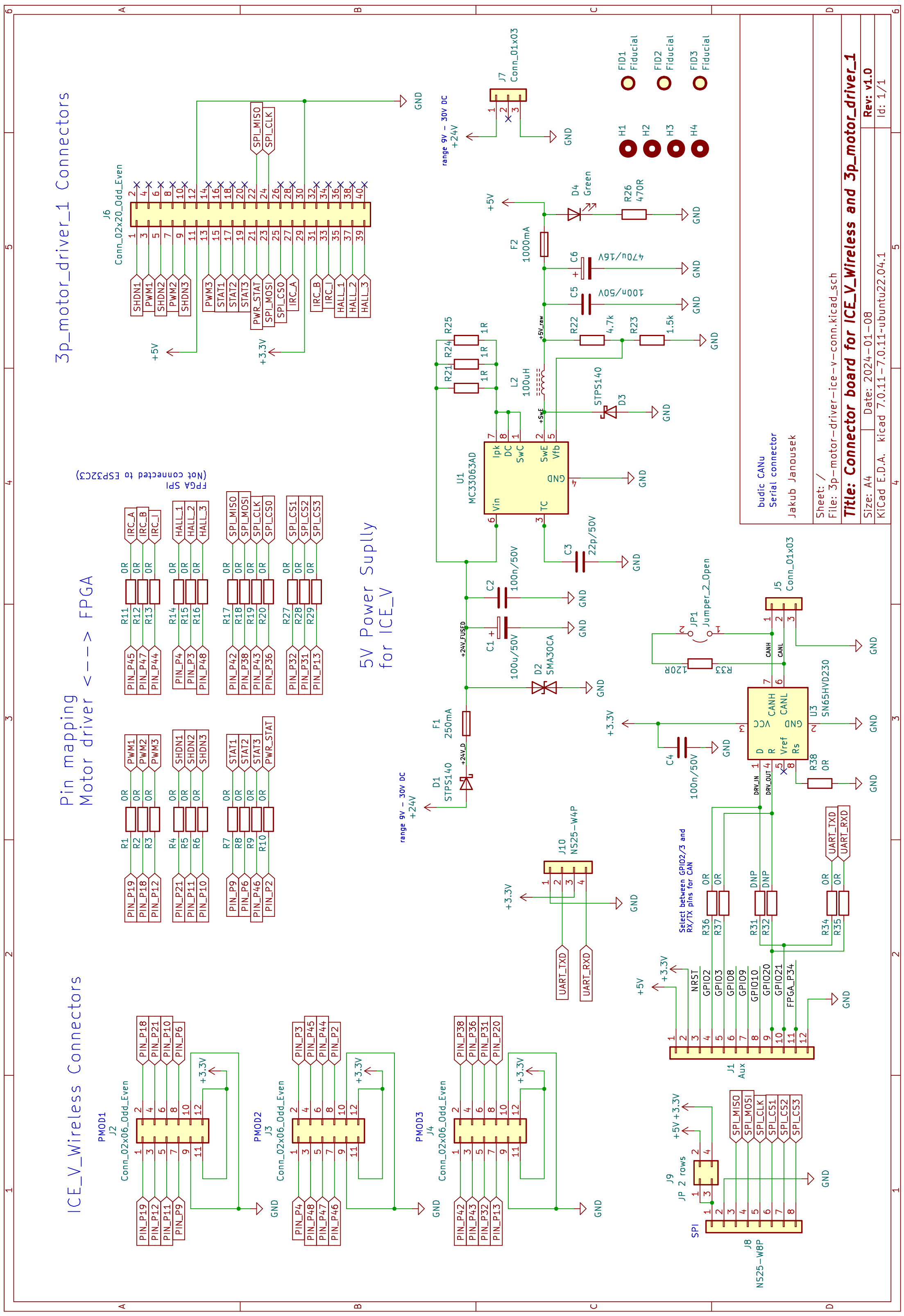
3p_motor_driver_1 Connectors
Pin mapping Motor driver <---> FPGA
FPGA SPI (Not connected to ESP32C3)
5V Power Suplly for ICE_V
ICE_V_Wireless Connectors
Select between GPIO2/3 and RX/TX pins for CAN
budic CANu Serial connector
Jakub Janousek
Sheet: /
File: 3p-motor-driver-ice-v-conn.kicad_sch
Title: Connector board for ICE_V_Wireless and 3p_motor_driver_1
Size: A4
Date: 2024-01-08
Rev: v1.0
KiCad E.D.A. kicad 7.0.11-7.0.11-ubuntu22.04.1
Id: 1/1

Figure A.1. Schematic diagram of the ICE-V PMSM Board

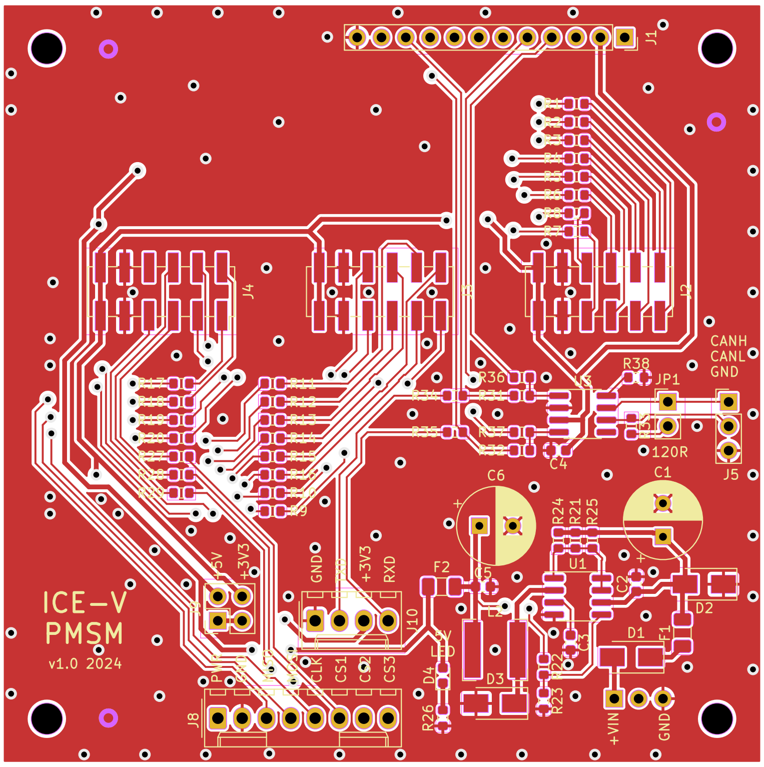

Figure A.2. Board layout of the ICE-V PMSM Board - Front

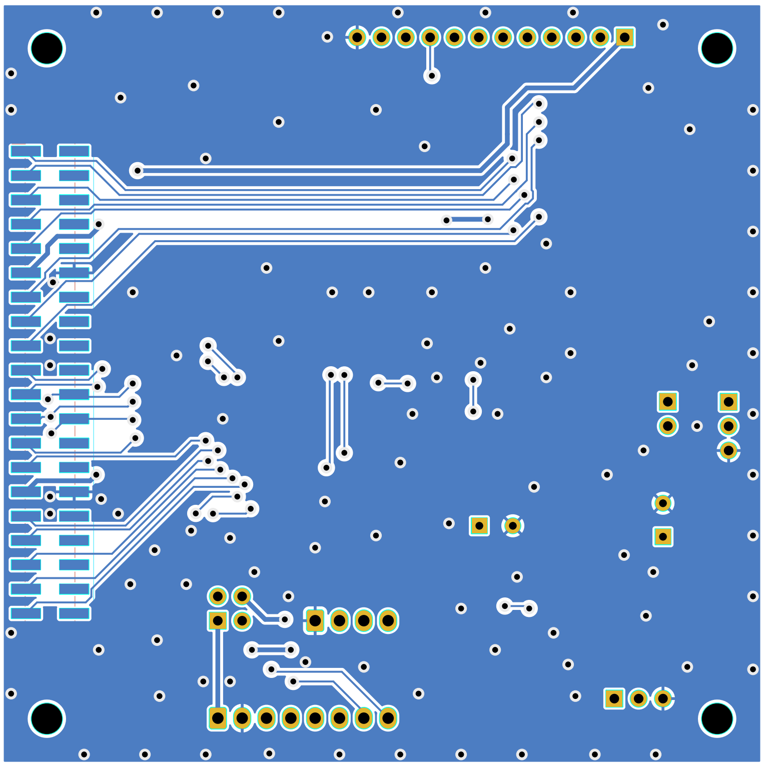

Figure A.3. Board layout of the ICE-V PMSM Board - Back

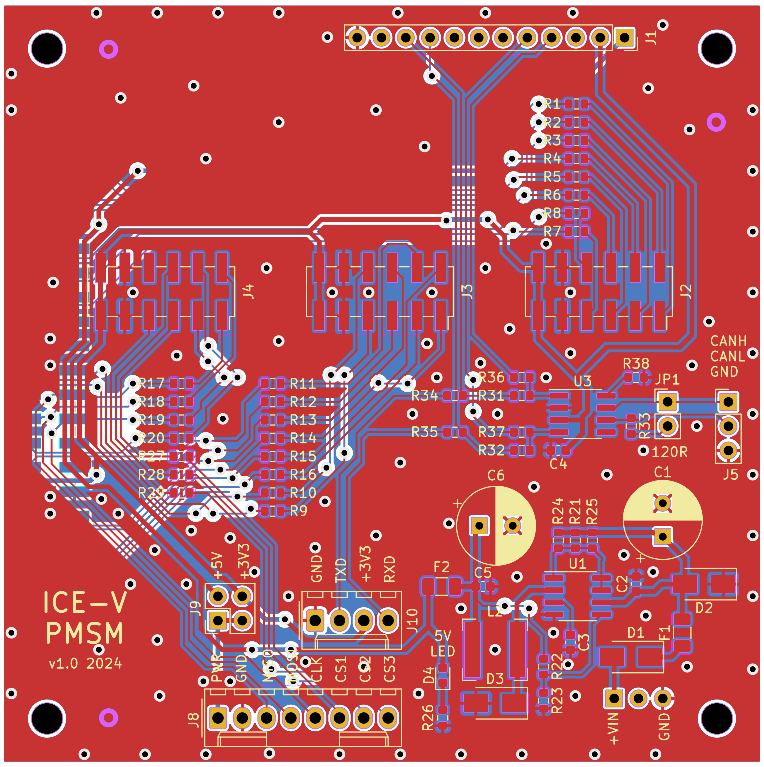

Figure A.4. Board layout of the ICE-V PMSM Board - Whole

Appendix B
List of all PXMC Commands in the ice_v_pmsm Application

Here is the complete list of possible commands in the `ice_v_pmsm` application, which can be called inside the PXMC command processor from the NSH. The list is also available by calling the `help` command from inside the application.

- help - prints help for commands
- G? - go to target position
- GR? - go relative
- PWM? - direct axis PWM output
- HH? - hard home request for axis
- SPD? - speed request for axis
- SPDT? - speed request with timeout
- SPDFG? - fine grained speed request for axis
- SPDFGT? - fine grained speed request for axis with timeout
- STOP? - stop motion of requested axis
- RELEASE? - releases axis closed loop control
- ZERO? - zero actual position
- PURGE? - clear 'axis in error state' flag
- AP? - actual position
- ST? - axis status bits encoded in number
- AXERR? - last axis error code
- REGP? - controller proportional gain
- REGI? - controller integral gain
- REGD? - controller derivative gain
- REGS1? - controller S1
- REGS2? - controller S2
- REGMD? - maximal allowed position error
- REGMS? - maximal speed
- REGACC? - maximal acceleration
- REGME? - maximal PWM energy or voltage for axis
- REGCFG? - hard home and profile configuration
- REGPTIRC? - number of irc pulses per phase table
- REGPTPER? - number of elmag. revolutions per phase table
- REGPTMARK? - phase index at encoder index mark in irc pulses
- REGPTSHIFT? - shift (in irc) of generated phase curves
- REGPTVANG? - angle (in irc) between rotor and stator mag. fld.
- REGPWM1COR? - PWM1 correction
- REGPWM2COR? - PWM2 correction
- REGPWM3COR? - PWM3 correction
- REGPTHALPH? - hal input phase
- R? - send R?! or FAIL?! at axis finish

- R - send R! or FAIL! at finish
- COORDMV - initiate coordinated movement to point f1,f2,...
- COORDMVT - coord. movement with time to point mintime,f1,f2,...
- COORDRELMVT - coord. relative movement with time to point mintime,f1,f2,...
- COORDSPLINET - coord. spline movement with time to point mintime,order,a11,a12,...,a21,..
- COORDGRP - group axes for COORDMV, for ex. C,D,F
- COORDDISCONT - max relative discontinuity between segs
- REGCURDP? - current controller d component p parameter
- REGCURDI? - current controller d component i parameter
- REGCURQP? - current controller q component p parameter
- REGCURQI? - current controller q component i parameter
- REGCURHOLD? - current steady hold value for stepper
- REGMODE? - axis working mode
- REGSFRQ - set new sampling frequency
- REGSLACK - maximum timing slack of sampling period
- logcurrent - log current history
- currentcal - current calibration

Appendix C
PXMC Errors

Here is the complete list of PXMC Error codes as defined in the file `pxmc.h` in the `submodule/pxmc/libs4c/pxmc_core/` directory of the `ice_v_pmsm` project.

- **PXMS_E_COMM**: 0x105 (261)
  Offset of commutation error
- **PXMS_E_MAXPD**: 0x106 (262)
  Difference of position over limit
- **PXMS_E_OVERL**: 0x107 (263)
  Overload error
- **PXMS_E_HAL**: 0x108 (264)
  Hall sensor error
- **PXMS_E_POWER_STAGE**: 0x109 (265)
  Power stage fault signal
- **PXMS_E_I2PT_TOOBIG**: 0x10A (266)
  Too big difference between Hall and index phase offset
- **PXMS_E_WINDCURRENT**: 0x10B (267)
  Winding current too high
- **PXMS_E_WINDCURADC**: 0x10C (268)
  Winding current sensing ADC failure
- **PXMS_E_MCC_FAULT**: 0x10D (269)
  Motion control coprocesor failure
- **PXMS_E_UV_PROT**: 0x10E (270)
  Undervoltage protection

Appendix D
Schematic Diagram of the 3p-motor-driver-1 Board

Here is the complete schematic diagram of the `3p-motor-driver-1` board. It was designed in PEDA software and the source can be found in the PiKRON Gitlab project, the `rpi-mc-1` [10]

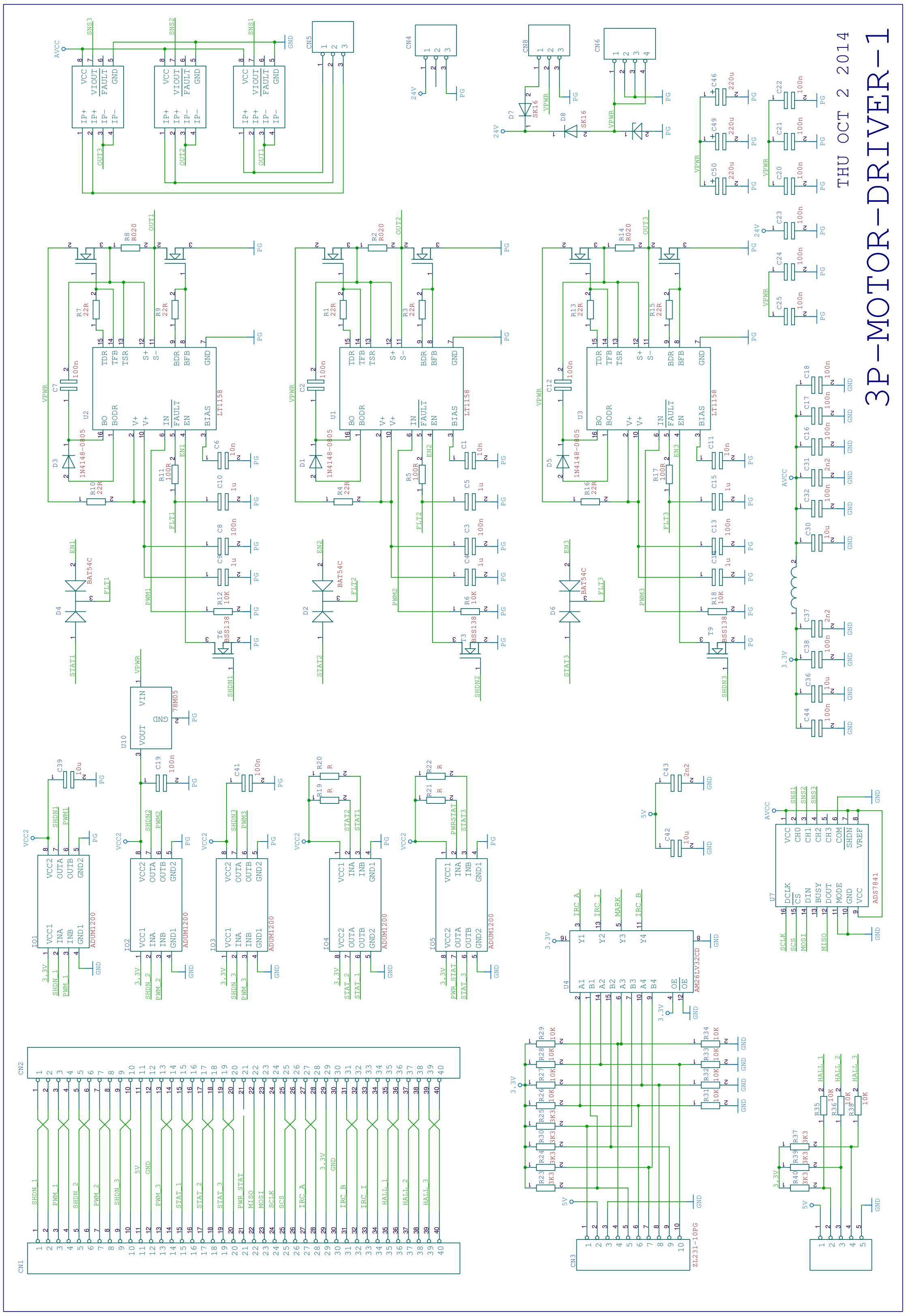

Figure D.5. Schematic diagram of the 3p-motor-driver-1 Board

References

[1] Prudek Martin. *Brushless motor control with Raspberry Pi board and Linux.* 2015.
https://dspace.cvut.cz/bitstream/handle/10467/62036/F3-BP-2015-Prudek-Martin-Bp_2015_prudek_martin.pdf?sequence=41&isAllowed=y.

[2] W. C. Duesterhoeft, Max W. Schulz, and Edith Clarke. Determination of Instantaneous Currents and Voltages by Means of Alpha, Beta, and Zero Components. *Transactions of the American Institute of Electrical Engineers.* 1951, 70 (2), 1248-1255. DOI 10.1109/T-AIEE.1951.5060554.

[3] Colm J. O'Rourke, Mohammad M. Qasim, Matthew R. Overlin, and James L. Kirtley. A Geometric Interpretation of Reference Frames and Transformations. *IEEE Transactions on Energy Conversion.* 2019, 34 (4), 2070-2083. DOI 10.1109/TEC.2019.2941175.

[4] PiKRON. *PXMC - Portable, highly eXtendable Motion Control library.* 2024.
https://pxmc.org/.

[5] Roberto Bucher. *Python for control purposes.* 2019.
https://robertobucher.dti.supsi.ch/wp-content/uploads/2017/03/BookPythonForControl.pdf.

[6] PiKRON. *microzed_apo repository.* 2024.
https://gitlab.com/pikron/projects/mz_apo/microzed_apo.

[7]

[8] CTU FEE. *B35APO course pages.* 2024.
https://cw.fel.cvut.cz/wiki/courses/b35apo/documentation/mz_apo-howto/start.

[9] CTU FEE. *Education Kit MicroZed APO.* 2024.
https://cw.fel.cvut.cz/wiki/_media/courses/b35apo/en/semestral/mz_apo-datasheet-en.pdf.

[10] PiKRON. *rpi-mc-1 gitlab repository.* 2024.
https://gitlab.com/pikron/projects/rpi/rpi-mc-1.

[11] Allegro Microsystems. *ACS711 Datasheet.* 2023.
https://www.allegromicro.com/-/media/files/datasheets/acs711-datasheet.pdf.

[12] Aneheim Automation. *Brushless motor BLWR233D-36V-4000.* 2024.
https://anaheimautomation.com/blwr233d-36v-4000.html.

[13] Pavel Pisa. *microzed-mc-1 gitlab.* 2024.
https://gitlab.fel.cvut.cz/canbus/zynq/zynq-can-sja1000-top/-/tree/master/system/ip/pmsm_3pmdrv1_1.0.

[14] Open Technologies Research Education, and Exchange Services. *rvapo-apps gitlab.* 2024.
https://gitlab.fel.cvut.cz/otrees/fpga/rvapo-apps.

[15] Pavel Pisa. *GNU/Linux and FPGA in Real-time Control Applications.* Installfest presentation. 2017.
https://installfest.cz/if17/slides/so_t2_pisa_realtime.pdf. PDF - PMSM and DC motor control on Raspberry Pi and Xilinx Zynq MZ_APO.

[16] Open Technologies Research Education, and Exchange Services. *ice-v-pmsm gitlab repository.* 2024.
https://gitlab.fel.cvut.cz/otrees/risc-v-esp32/ice-v-pmsm.

[17] Damir Gruncl. *Pipelined RISC-V processor design in VHDL for education and FPGA demonstration.* 2024.
https://dspace.cvut.cz/bitstream/handle/10467/114963/F3-DP-2024-Gruncl-Damir-samproj_v2-3.pdf.

[18] The Apache Foundation. *About Apache NuttX.* 2024.
https://nuttx.apache.org/docs/latest/introduction/about.html.

[19] The Apache Foundation. *The Inviolable Principles of NuttX.* 2024.
https://nuttx.apache.org/docs/latest/introduction/inviolables.html.

[20] The Apache Foundation. *Directory Structures.* 2024.
https://nuttx.apache.org/docs/10.0.0/quickstart/organization.html.

[21] The Apache Foundation. *Device Drivers.* 2024.
https://nuttx.apache.org/docs/latest/components/drivers/index.html.

[22] The Apache Foundation. *Configuring NuttX.* 2024.
https://nuttx.apache.org/docs/latest/quickstart/configuring.html.

[23] QWERTY Embedded Design. *ICE-V Wireless.* 2024.
https://github.com/ICE-V-Wireless/ICE-V-Wireless.

[24] Espressif. *ESP32 C3.* 2024.
https://www.espressif.com/en/products/socs/esp32-c3.

[25] Lattice Semiconductor. *Lattice iCE40 UltraPlus FPGA.* 2024.
https://www.latticesemi.com/en/Products/FPGAandCPLD/iCE40UltraPlus.

[26] PiKRON. *PXMC - Portable, highly eXtendable Motion Control library gitlab repository.* 2024.
https://gitlab.com/pikron/sw-base/pxmc/.

[27] Roberto Bucher. *pysimCoder Github repository.* 2024.
https://github.com/robertobucher/pysimCoder.

[28] Jakub Janousek. *NuttX fork with support for FPGA iCE40 bitstream loading.* 2024.
https://github.com/janouja/nuttx/tree/ice40-driver.

[29] Jakub Janousek. *NuttX pull request: drivers/spi: Add support for FPGA iCE40 bitstream loading.* 2024.
https://github.com/apache/nuttx/pull/12012.

[30] Lattice Semiconductor. *iCE40 Programming and Configuration.* 2022.
https://www.latticesemi.com/view_document?document_id=46502.

[31] Petr Porazil Pavel Pisa. *PEDA - electronic design automation software.* 2024.
https://sourceforge.net/p/peda/wiki/Home/.

[32] Texas Instruments. *MC3x063A 1.5-A Peak Boost/Buck/Inverting Switching Regulators*. 2024.
https://www.ti.com/lit/gpn/MC33063A.

[33] PiKRON. *PXMC Documentation*. 2024.
https://pxmc.org/files/pxmc.pdf.

[34] Open Technologies Research Education, and Exchange Services. *PXMC-Linux gitlab repository*. 2024.
https://gitlab.fel.cvut.cz/otrees/motion/pxmc-linux.

[35] Pavel Pisa Michal Sojka. *OMK: Ocera Make System*. 2024.
https://rtime.felk.cvut.cz/omk/omk-manual.html.