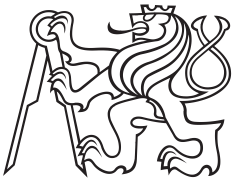
Bachelor Project



Czech Technical University in Prague

Faculty of Electical Engineering
Department of Control Engineering

Open Hardware Motion Controller for Model-Based Rapid Prototyping with NuttX RTOS

Štěpán Pressl

Supervisor: Ing. Pavel Píša, Ph.D.
Field of study: Cybernetics and Robotics
May 2024

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Pressl Štěpán** Personal ID number: **507291**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Control Engineering**

Study program: **Cybernetics and Robotics**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Open Hardware Motion Controller for Model-Based Rapid Prototyping with NuttX RTOS

Bachelor’s thesis title in Czech:

Otevřená jednotka pro řízení motorů využitelná pro modelový návrh pod systémem NuttX

Guidelines:

Many motion controllers exist, but this project targets the design of open and reasonably priced solutions suitable for high-level model-based control designs even with open tools like pysimCoder. Such a target requires double precision floating point support, and POSIX-based RTOS (i.e., NuttX) is also a significant simplification for porting.
1) Familiarize with referenced projects and prepare a short overview of available options
2) Design hardware platform - expected is the use of ARM Cortex-M7 based micro-controller
3) Adapt NuttX BSP for the developed platform
4) Prepare demonstration with pysimCoder or PXMC PMSM motor control
5) Document results and prepare requests to submit changes to upstream projects

Bibliography / sources:

[1] Patterson, D. A., and J. L.: Computer Organization and Design RISC-V Edition, The Hardware Software Interface 2nd ed. Morgan Kaufman, 2021, ISBN: 9780128203316
[2] NuttX, GitHub https://github.com/apache/nuttx
[3] NuttX, Documentation https://nuttx.apache.org/docs/latest/
[4] Lenc, M., Píša, P., Bucher, R.: pysimCoder – Open-Source Rapid Control Prototyping for GNU/Linux and NuttX, In Process Control 2023, Slovakia
[5] SAM E70/S70/V70/V71 - 32-bit Arm Cortex-M7 MCUs with FPU, Audio and Graphics Interfaces, High-Speed USB, Ethernet, and Advanced Analog, Micochip, 2023, available online

Name and workplace of bachelor’s thesis supervisor:

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

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

Date of bachelor’s thesis assignment: **05.02.2024** Deadline for bachelor thesis submission: **24.05.2024**

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 bachelor’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 bachelor’s thesis, the author must state the names of consultants and include a list of references.

Date of assignment receipt	Student’s signature

Acknowledgements

I deeply thank my family for their support during my studies. Regarding the bachelor thesis, I would like to thank Ing. Pavel Píša, PhD. for introducing me to new topics in the motor and control theory and deepening the knowledge in embedded programming. I would also like to thank Ing. Petr Porazil for the help during the PCB design, like the components selection, Bc. Michal Lenc for the cooperation and advice regarding NuttX and Ing. Květoslav Belda, PhD. who paved the way for the creation of this controller alongside the supervisor.

Declaration

I declare that this work is all my own work and I have cited all sources I have used in the bibliography.

In Nýrsko and Prague, May 2024.

.....

Abstract

There are many motion controllers on the market, some of them open like ODrive or VESC controllers. However, their primary task is to only control a motor. Many controllers lack the feature of running model-based control applications, created, for example, by the known MATLAB/Simulink suite.

This thesis proposes a general prototyping platform, useful in research or early stages of development, on which high-level model-based applications could be run. As a tool for model-based prototyping, the open-source pysimCoder suite is used, alongside NuttX RTOS.

We want this platform to be extensible and modular, meaning making this open source and hardware is the best choice. The controller features a powerful Microchip's ATSAMV71Q21B microcontroller with rich connectivity and a powerful power stage board for motors.

Keywords: ATSAMV71Q21B, pysimCoder, NuttX RTOS, open hardware and software, KiCad, Ethernet AD converter, pulse width modulation, permanent magnet synchronous motor

Supervisor: Ing. Pavel Píša, Ph.D.

Abstrakt

Na trhu je mnoho řídicích jednotek pro motory, přičemž některé z nich jsou otevřené, jako ODrive či kontroléry z projektu VESC. Sice jejich primární účel je řídit motory, ale tyto platformy nejsou stavěny pro modelový návrh pomocí aplikací vytvořené např. nástrojem MATLAB/Simulink.

Tato bakalářská práce přichází s řídicí jednotkou vhodnou pro výzkum či prvotní fáze vývoje, schopna spouštět aplikace vytvořené pomocí modelového návrhu. Nástroj pro tvorbu těchto aplikací je otevřený balíček pysimCoder běžící nad systémem reálného času NuttX.

Tento kontrolér by měl být otevřený jak z hlediska hardwaru, tak softwaru, protože je to nejlepší z hlediska udržovatelnosti a modulárnosti. Tento kontrolér obsahuje výkonný mikrokontrolér ATSAMV71Q21B od firmy Microchip s mnoha komunikačními periférierniemi a výkonnovou deskou pro řízení motorů.

Klíčová slova: ATSAMV71Q21B, pysimCoder, systém reálného času NuttX, otevřený hardware a software, KiCad, Ethernet, AD převodník, pulzně šířková modulace, PMS motor

Překlad názvu: Otevřená jednotka pro řízení motorů využitelná pro modelový návrh pod systémem NuttX

Contents

Figures

Tables

Chapter 1

Introduction

This bachelor thesis proposes a modern motion controller capable of driving various kinds of actuators, mostly commutated DC, BLDC (brushless DC), PMS (permanent magnet synchronous), stepper motors and other types of actuators. There are many available motion controllers on the market, both open and closed hardware. Also, a lot of configurable integrated circuits designed for precise motor control are available.

However, we intend to come up with a general embedded prototyping platform, capable of running model-based control applications, useful in research and early stages of development. This implies we need to use a microcontroller with a lot of communication interfaces at our disposal (like Ethernet, UART, SPI, I^2C, CAN) and peripherals useful for motion control, like PWM and ADC. The general model-based design however comes with some overhead, thus a powerful core must be used too. The hardware analysis is done in chapter 5 and the implementation is described in chapter 6. The final product is meant to be open hardware and open source for the purpose of better maintainability.

In case of a platform upgrade (or the worst scenario, an another chip shortage), we intend not to write bare-metal applications, as this would make the switch between platforms harder. The solution is to use the POSIX-compatible NuttX RTOS with unified hardware abstraction across a wide range of different platforms. The introduction to NuttX's main features is mentioned in Chapter 2 and the adaptation to our platform is mentioned in Chapter 7.

In the field of model-based design, MATLAB/Simulink is the first suite that comes to mind. In this thesis, we plan to use the pysimCoder suite instead, an open tool capable of generating target-compatible generated C-code. PysimCoder is introduced in Chapter 3 and its applications are described in Chapter 8. The important aspect of pysimCoder is it supports code generation using the API of NuttX.

This thesis builds upon the *Open Rapid Control Prototyping and Real-Time Systems* bachelor's thesis by Michal Lenc [9], also supervised by Pavel Píša, PhD. Michal Lenc has done a lot of work in the field of open rapid control prototyping. Namely, the first contribution was the implementation of low-end peripheral drivers of the used microcontroller in this thesis. He also added a feature to make the pysimCoder's internal parameters configurable using the Silicon Heaven protocol created by Elektroline, a.s. (a Czech company specializing in tram transportation). The project-related repositories can be found here [6].

To make pysimCoder as convenient as Simulink to use, a lot of work will have to be still done. During the work on this controller, project teamwork as part of B3MPVTY1 (Práce v týmu) subject on master's Cybernetics and Robotics program was running to enhance the capabilities of pysimCoder.

Unfortunately, during the testing of pysimCoder generated code on NuttX, the RTOS showed serious problems with sampling capabilities above 1 kHz, even with the tickless mode (described in 7.2.4) activated. Despite this issue, an example of simple PMS motor control is shown, alongside with current control in the dq axes (d and q denote the direct and quadrature axes in the rotor's reference frame). Also, the measurement of a piezoactuator's deformation angle is presented, as the applied voltage is tuned with the help of pysimCoder and Silicon Heaven. Several commits have been accepted by the NuttX mainline to support features used in our applications.

The purchase of components and PCBs was funded by the PiKRON company and the Institute of Theory of Information and Automation of the Czech Academy of Sciences (ÚTIA, AV ČR). The feature of fast prototyping may prove useful when experimenting with different actuators even from the field of experimental mechatronics.

Chapter 2

Realtime NuttX OS

This chapter presents the main characteristics of the NuttX real-time operating system, the supported platforms, and the process of building.

Using higher levels of software abstraction usually comes with a cost in terms of performance overhead. However, in terms of maintenance and portability, RTOS is a good choice, especially for our controller which targets to be extensible and adaptive. NuttX also provides a unified API, can work with filesystems, and features a full IP stack on which network applications can be built.

2.1 Introduction

NuttX is a real-time operating system (RTOS) with an emphasis on standards compliance and small footprint. Scalable from 8-bit to 64-bit microcontroller environments, the primary governing standards in NuttX are POSIX and ANSI standards [27]. NuttX is an open-source project written in C and released under the Apache 2.0 license first introduced by Gregory Nutt in 2007. Currently, the project is maintained at GitHub [1].

As previously mentioned, NuttX obeys the POSIX standard which means all the interaction with the microcontroller's peripherals is done by accessing the peripheral's registered `/dev` files (for example, an AD converter may be registered as `/dev/adc0`, the CAN peripheral as `/dev/can0`, etc...). The files are accessed using system calls like `open`, `close`, `read` or `write`. As many peripherals do not share many similarities between each other, `ioctl` (the `ioctl()` system call manipulates the underlying device parameters of special files) is commonly used to interface the peripheral, which can sometimes lead to harder maintainability. In our case, these system calls do not behave the same way the system calls behave on personal computers (our microcontroller does not switch between the privileged and the user mode). However, this makes NuttX POSIX compatible.

NuttX offers a full standard C library, including sockets for network programming over UDP/IP or TCP/IP. It is also possible to work with filesystems and register hardware memories (like EEPROMs, SPI Flash memories, etc...) as filesystems which can be mounted into the filesystem later on.

[1] `https://github.com/apache/nuttx`

2.2 Source Code Structure

The project's structure is similar to the Linux kernel's one. This section mentions a few directories used in the NuttX project.

2.2.1 `arch`

This directory contains all architecture-dependant drivers which in the context of NuttX driver implementation is called the lower half part of the driver. It also contains low-end functions needed for context saving, stack manipulation, processor initialization, and so on.

The supported platforms span from 8bit AVR microcontrollers, to Mips, RISC-V used in ESP32 microcontrollers from Espressif, to last but not least ARM. For the context of this thesis, the ARM architecture is the most important because of the used microcontroller.

There are many supported ARM platforms in NuttX, like STMicroelectronic STM32 series, NXP lpc or imx series or Microchip SAM series. Taking a look at `arch/arm/src/stm33h7` for example, nearly all peripherals have a driver implemented. This proves NuttX's good platform support.

2.2.2 `boards`

This directory is architecture dependant too, but it provides BSP (board support package) directories for all kinds of architectures and all sorts of evaluation kits and modules. It contains functions from which the NuttX shell is started, booting procedures for each microcontroller, initialization of peripherals and its registration as `/dev` files. Each BSP contains various `defconfig` files needed for NuttX build configuration.

2.2.3 `drivers`

In this directory the so-called upper-half implementation of drivers can be found. NuttX supports a lot of peripheral drivers, like PWM, AD converter I^2C, SPI, serial line or USB. MTD (memory technology device) drivers are also present, allowing the registration of block devices based on EEPROMs, SPI flash memories and so on.

2.2.4 Other Directories

Amongst other directories, `include` and `net` is worth mentioning. The `include` directory contains all header files related to drivers, standard library, etc. The `net` directory contains the implementations of network stacks.

2.3 Configuration and Compilation

The main principle of NuttX is the right to build the project on any platform (Windows, Linux, BSD, MacOS) and on any architecture that has the right build tools. NuttX project features a lot of custom scripts and makefiles for the project building. NuttX can also be built using `CMake`.

The figure 2.1 shows the commands needed for the project configuration. The options for `board:config` can be obtained by running `./tools/configure.h -L` and they correspond to what has been mentioned previously as BSP `defconfig`s. NuttX can also be compiled with various examples and helper apps, which are located in the `nuttx-apps` GitHub repository.

```
$ git clone https://github.com/apache/nuttx.git nuttx
$ git clone https://github.com/apache/nuttx-apps.git apps
$ cd nuttx
$ ./tools/configure.sh board:config
```

Figure 2.1: NuttX build configuration using `make`

Afterwards, the command

```
$ make menuconfig
```

can be run to turn on or turn off various features by selecting `CONFIG_*` options. Running `make` compiles and links into `.elf` and raw binary executables. The `make export` command creates an archive which contains all the compiled parts of NuttX with exported header files. This may be useful when linking with other projects, like pysimCoder generated code.

Chapter 3

Control Theory and Related Software

Control theory is important for studying the behaviour of various physical, electrical, economical and biological systems. The first step of describing these systems is to develop a mathematical description, called the model, of the process to be controlled [7]. With the help of the model we are able to design controllers capable of driving these systems to desired states. This chapter describes simple theory behind control and introduces reader to software associated with control and regulation.

3.1 Basic Description of Systems

The term model, as it used and understood by control engineers, means a set of differential equations that describe the dynamic behaviour of the process. There are three domains within which to study dynamic response: the Laplace transform (s-plane), the frequency response, and the state space (analysis using the state-variable description) [7].

With the help of Laplace transform, the differential equations in the time domain can be transformed into a complex function in the s-plane. Let's denote $U(s)$ as the input and $Y(s)$ as the output functions in the complex Laplace domain. Then an impulse response (or transfer function) is defined as

$$H(s) = \frac{Y(s)}{U(s)}. \tag{3.1}$$

$H(s)$ is a complex function used to determine the behaviour or stability of a given system.

Generally, when working with the system described by the system of nonlinear equations, the first step involves linearizing the differential equations and obtaining system of *linear* differential equations. This system then describes a *linear time-invariant* (LTI) system whose system of linear differential equations can be compactly written in the form

$$\dot{\mathbf{x}} = A\mathbf{x} + B\mathbf{u}, \tag{3.2}$$

$$\mathbf{y} = C\mathbf{x} + D\mathbf{u}, \tag{3.3}$$

where $\mathbf{x}$ is the vector of state variables, $\mathbf{u}$ is the input vector and $\mathbf{y}$ is the vector of system's output. When given matrices A, B, C and D, the impulse response is calculated as follows:

$$H(s) = C(sI - A)^{-1}B + D = \frac{C \cdot \mathrm{adj}(sI - A) \cdot B}{\det(sI - A)} + D, \tag{3.4}$$

where I is an eye matrix of the same dimensions as A, adj denotes the adjugate matrix and det denotes the determinant.

3.2 Controllers

Given impulse response $H(s)$, system's behaviour can be modified by a controller. Controller is a system designed to maintain system's output fixed regardless of outer disturbances. The system is controlled by the difference between the given reference $U(s)$ and the output $Y(s)$ (feedback), also called the error $E(s)$. If we denote the controller's step response $G(s)$, then the impulse response of the simplest feedback controlled system can be calculated as follows:

$$\frac{Y(s)}{U(s)} = \frac{G(s)H(s)}{1 + G(s)H(s)}. \tag{3.5}$$

The controlled system schematic is shown in the figure 3.1.

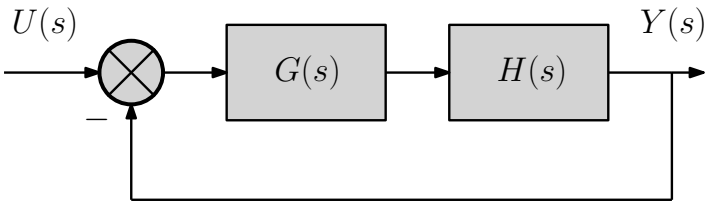


Figure 3.1: The simplest block diagram of feedback controlled system $H(s)$

3.2.1 PID Controller

Many types of controllers exist, however let's mention the simplest controller that is being frequently used in control applications. PID controller is a controller comprising of three parts: P being the proportional part, I being the integral part and D being the derivative part. The overall transfer function in the Laplace form of this controller can be written as

$$H(s) = k_P + \frac{k_I}{s} + k_D s. \tag{3.6}$$

Let's discuss each term's behaviour and impact on the control of the system. Let's denote $e(t)$ as the input signal of the regulator and $u(t)$ as the output of the regulator in the time domain.

Proportional Control

This result of the proportional feedback is

$$u(t) = k_P e(t), \tag{3.7}$$

where k_P is the proportional constant (gain). It is the the simplest controller, however with most systems this controller is not capable of achieving zero steady-state error. The error can be minimalised by setting higher k_P which can however result in big overshoots or instability.

Integral Control

Integral control can be expressed in the form

$$u(t) = k_I \int_{t_0}^{t} e(\tau)\,\mathrm{d}\tau, \tag{3.8}$$

where k_I is called the integral gain. The goal of integral control is to minimize the steady-state tracking error and the steady-state output to disturbances. It can be seen the output control signal at each time is a summation of all past values of the input signal. Setting k_I too big can lead to instability of the whole system.

When implementing integral control, care must be taken when working with systems whose output can be saturated. It can happen the integral control will continue growing (windup) and then signal overshooting or poor transient response can happen. The solution to this problem is implementing anti-windup integral control which turns off the integral controller when saturation is reached.

It must noted that in discrete controllers the integral control is not used, but rather the summated control is used.

Derivative Control

The derivative part of the PID controller can be expressed in the form

$$u(t) = k_D \frac{\mathrm{d}e(t)}{\mathrm{d}t}, \tag{3.9}$$

where k_D is the derivate gain. This action is used to improve closed-loop stability and speeding up the system's response. Derivative control is almost never used by itself, it is mostly used in conjuction with P or PI controller. The derivative controller gives sharp response to quick responses, hence it can amplify parasitic noise signals, worsening the overall control.

3.3 Model-based Design Approach

Software related to control engineering offers tools to model and simulate complex systems. Modeling and simulation help with testing, especially during early phases of development where hardware setup is unavailable. Early identification of the errors reduces the project time and cost and drastically improves software quality [18] (Section II: Simulink, chapter 5). With the help of these tools, engineers can devise control algorithms or design controllers capable of controlling complex dynamic systems.

Most model-based software for control engineering is based on blocks with different behaviour. Connection of these blocks creates a model and running a simulation means simulating a signal propagating through these blocks. The solver of this software can then calculate dynamic system's states at consecutive time steps during a specified period [18] (Section II: Simulink, chapter 8).

Due to the block design with signal flows, the whole block model can be regenerated into a C, C++ or any HDL (hardware description language) code for specific hardware which realises the designed control algorithms. This makes the shift from the computer aided design of a controller to a specific hardware much easier.

3.3.1 MATLAB/Simulink

One of the most known tools for computer systems modelling and simulation is the Simulink graphical environment based on MATLAB scripting language. It also allows automatic generation of C code if hardware dependant peripheral drivers are implemented.

The Simulink suite offers many modelling blocks for various branches of engineering, such as robotics or power electronics. The Simulink's solver, which can be either fixed-step or variable-step implements various numerical methods which calculate the system's next states using differential equation or numerical integration methods. The examples of solvers are ([18] Section II: Simulink, chapter 8):

- Fixed-step solvers:
  - ode1 (Euler method),
  - ode2 (Heun method),
  - ode4 (Runge-Kutta),
  - ode8 (Dormand-Prince RK8).
- Variable-step solvers:
  - ode45 (Dormand-Prince) - Runge-Kutta (4,5),
  - ode23 (Bogacki-Shampine).

The description of these solvers is beyond the scope of this thesis and are only mentioned for clarity.

However, Simulink is expensive and closed. For the purposes of open rapid prototyping, we will be using the pysimCoder open suite capable of generating C compatible code.

3.3.2 pysimCoder

This section introduces a suite of Python scripts and GUI for Simulink like model-based design. This piece of software is the suite we will be using throughout this thesis as an alternative to Simulink.

Introduction

PysimCoder is an open-source suite created by Professor Roberto Bucher from University of Applied Sciences and Arts of Southern Switzerland (SUPSI). The main task of this piece of software is to transform the drawn block diagram into a generated C-code. The GUI for the block editor is written in `Python` using the `PyQT` graphical library. The GitHub repository of this tool can be found at [4].

The pysimCoder GUI is shown on figure 3.2. The GUI is divided into two windows, the one being the library of all available blocks and the second window being the block editor where the control application can be drawn.

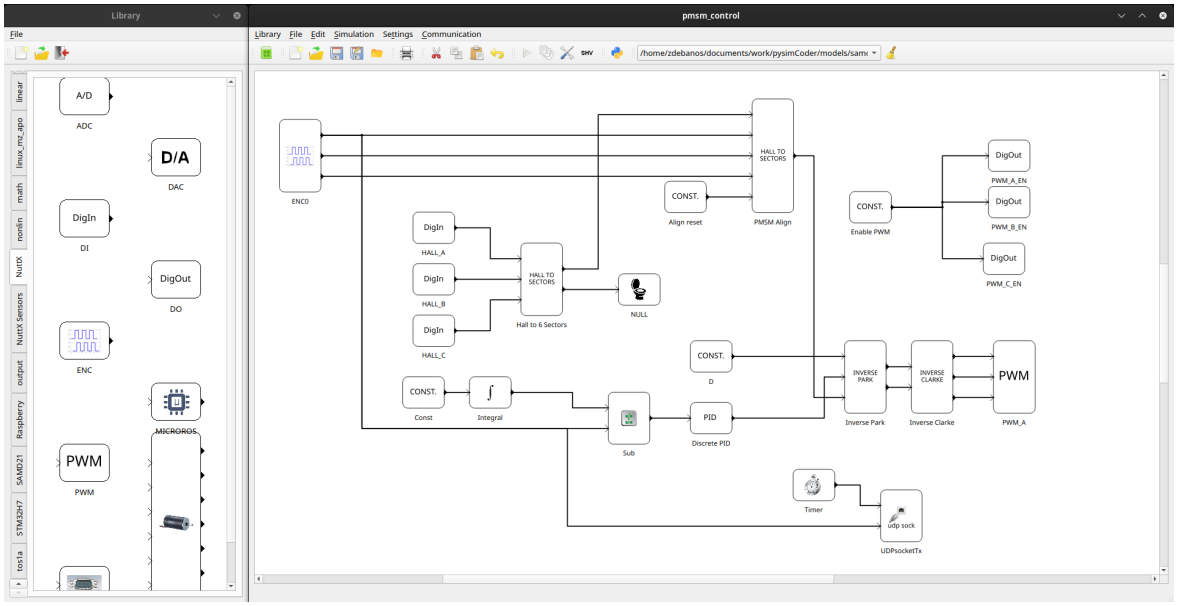


Figure 3.2: The GUI of pysimCoder - block menu and the block editor.

Code Generation

Each block is described by a JSON `.xblk` file, defining the overall behaviour of the block - the number of inputs, the number of outputs, the number of internal parameters, the library the block belongs to, etc. The `.xblk` files are located (relative to repository's root) in `resources/blocks/blocks/lib`, where `lib` is `linear`, `input`, `output`, etc. The directory `resources/blocks/rcpBlk/lib` contains the wrapping Python code which returns a new object for each block. Each returned block is an object of `RCPblk` class.

The `RCPblk` is a general wrapping class for each block and is located here [1]. The code snippet in 3.3 shows the constructor of this object, showing all parameters needed for the description of the block behaviour.

We can see the `*args` variable contains all the values needed for the general block, namely

- `fcn`: the C function name it should look for,
- `pin`: the array of input signals,
- `pout`: the array of output signals,
- `realPar`: block's internal parameters of the `double` type,
- `intPar`: block's internal parameters of the `int` type.

The Python scripts then iterate over all the blocks in the block diagram, drawn in GUI, and determine the data passing between the blocks. After this step, the C code must be generated and substitutions for C functions must be made.

The base struct for the C block functionality in pysimCoder is determined by the `python_block` datatype. The struct's definition is given by code shown in 3.4 and can be found here [2].

[1] `toolbox/supsisim/supsisim/RCPblk.py`
[2] `CodeGen/Common/include/pyblock.h`

```
class RCPblk:
def __init__(self, *args):
    if len(args) == 8:
        (fcn,pin,pout,nx,uy,realPar,intPar,str) = args
    elif len(args) == 7:
        (fcn,pin,pout,nx,uy,realPar,intPar) = args
        str=''
    else:
        raise ValueError("Needs 6 or 7 arguments;\
              received %i." % len(args))

    self.name = None
    self.fcn = fcn
    self.pin = array(pin)
    self.pout = array(pout)
    self.dimPin = ones(self.pin.shape)
    self.dimPout = ones(self.pout.shape)
    self.nx = array(nx)
    self.uy = array(uy)
    self.realPar = array(realPar)
    self.realParNames = []
    self.intPar = array(intPar)
    self.intParNames = []
    self.str = str
    self.sysPath = ''
    self.no_fcn_call = False
```

Figure 3.3: The `RCPblk's` constructor

We can observe the inputs and outputs variables (`u` and `y`) are of the general `void**` datatype. This means the block takes `nin` inputs, each input being an array which can, in general, be used as a method for passing serialized data. Namely, pysimCoder uses the `double` datatype for the outputs, inputs and non-integer internal parameters, which may become ineffective on architectures with no double-precision float support. However, the generated code is in the end more general and high precision calculations are guaranteed.

From the control theory perspective, each system's behaviour in discrete time can be described by the set of two equations:

$$\mathbf{x}_{k+1} = \mathbf{f}(\mathbf{x}_k, \mathbf{u}_k, k), \tag{3.10}$$

$$\mathbf{y}_k = \mathbf{g}(\mathbf{x}_k, \mathbf{u}_k, k), \tag{3.11}$$

where k is the current time, $\mathbf{x}_k$ represents the vector of the system's current state and $\mathbf{u}_k$ is the input vector. The pysimCoder block method must essentially perform these two calculations. It is better to compute 3.11 first and 3.10 afterwards because the function would have to save $\mathbf{x}_k$ first in order to compute $\mathbf{y}_k$ in the opposite way.

Each block must have a method which performs the computation described by the equations 3.11, 3.10. In the figure 3.5 an example of the integrator's implementation is presented.

```
typedef struct {
  int nin;              /* Number of inputs */
  int nout;             /* Number of outputs */
  int * dimIn;          /* Port signal dimension */
  int * dimOut;         /* Port signal dimension */
  int *nx;              /* Cont. and Discr states */
  void **u;             /* inputs */
  void **y;             /* outputs */
  double *realPar;      /* Real parameters */
  int realParNum;       /* Number of real parameters */
  int *intPar;          /* Int parameters */
  int intParNum;        /* Number of int parameters */
  char * str;           /* String */
  void * ptrPar;        /* Generic pointer */
  char **realParNames; /* Names of real parameters */
  char **intParNames;  /* Names of integer parameter */
} python_block;
```

Figure 3.4: The definition of `python_block` datatype

We can observe an `int Flag` parameter is also passed which determines the block's method functionality. The flag can be either

- `CG_INIT`: initialize the block's internal states,
- `CG_STUPD`: perform the state update,
- `CG_OUT`: compute the output of the block,
- `CG_END`: perform the terminating tasks.

Using with NuttX

To interact with the outer world, pysimCoder must implement blocks which interact various sensors, peripherals, etc. In motor control applications, AD converter, encoder counter and PWM peripherals are important.

PysimCoder supports platform dependant I/O and communication blocks for various platforms, such as STM32H7 series, SAMD21 series or Arduino. However, implementing a peripheral driver for each platform can become time consuming. Hence, pysimCoder can generate portable C code for NuttX RTOS which should make the switch between different platforms smoother, supposing the required `/dev` devices are registered on each platform.

The control application is then realized by periodic execution of blocks' output and update functions (3.11, 3.10). However, the system must be sampled at equidistant intervals periodically. This can be done by using POSIX time handling functions, like `clock_nanosleep` in a high priority task.

The configuration of the target binary can be done by clicking the icon shown in figure 3.6. A configuration window then pops up, allowing us to choose the default `Makefile` for

```
void integral(int Flag, python_block *block)
{
  double * realPar = block->realPar;
  double *y;

  double h = realPar[0];
  double *U = block->u[0];

  switch(Flag){
  case CG_OUT:
    y = (double *) block->y[0];
    y[0] = realPar[1];
    break;

  case CG_STUPD:
    /* Runga Kutta */
    realPar[1] = realPar[1] + U[0]*h;
    break;
  case CG_INIT:
  case CG_END:
    break;
  default:
    break;
  }
}
```

Figure 3.5: Integrator C implementation in pysimCoder.

our target platform (in our case, `nuttx.tmf` is used). Also the time for periodic sampling must be configured in the *Sampling Time* field.

Remote Introspection of the Model's State

Thanks to Michal Lenc, the pysimCoder suite supports the introspection of blocks during runtime using the Silicon Heaven protocol developed by Elektroline, a Czech company. Silicon Heaven's features and its integration into pysimCoder are discussed in Michal Lenc's bachelor's thesis [9].

Runtime monitoring of model's parameters makes the usage of pysimCoder much more convenient and allows the user to experiment with constants (for example a PID controller's ones) during runtime.

The infrastructure requires running a broker as a server. User applications as pysimCoder control application or GUI designed to interact with the broker are then registered to the broker as clients. Each client can have different rights and settings based on a broker's configuration [9].

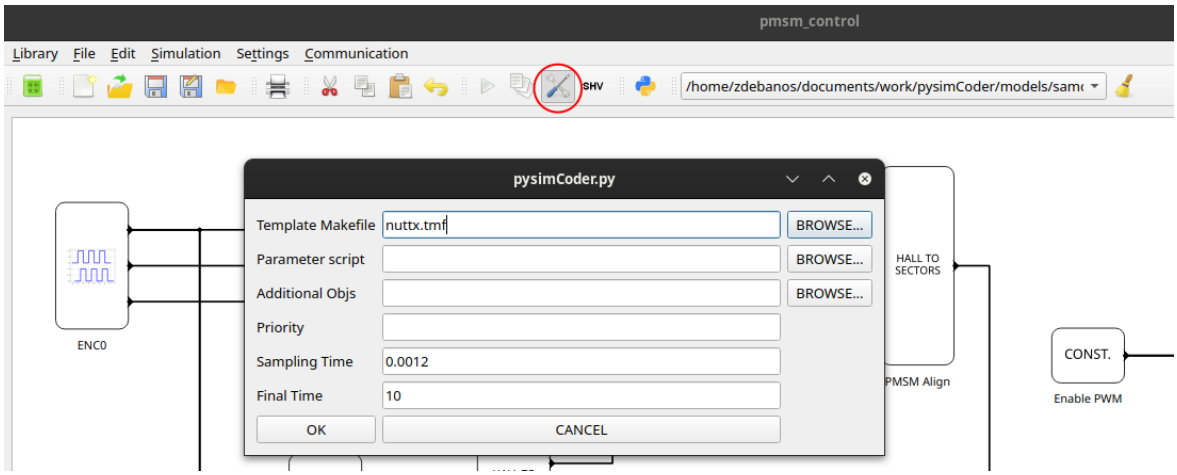


Figure 3.6: The build configuration of the target binary

Chapter 4

Motors and Actuators

This chapter introduces actuators which can be controlled by the proposed controller. AC motors will not be discussed as the controller was not designed to control these types of actuators.

4.1 DC Brushed Motor

The DC brushed motor is the simplest motor to control. The motor consists of a stator, rotor and a commutator. The stator is either a magnet or an electromagnet, responsible for magnetic field. The rotor is a series of wound coils in grooves, the ends of the coils are connected to the commutator pads.

The commutator is used to control which coils are conneted to the power source. This way, an alternating magnetic field can be created, perpendicular to the field of the stator and thus causing the rotor to rotate. Even though the magnetic field changes, the current flowing to the rotor has a DC character.

The current to the commutator flows through brushes, which tend to wear out. When the brushes wear our, sparking can happen, resulting in unwanted electromagnetic interference. The brushes contribute to friction, lowering the motor's efficiency and reliability.

Despite the cons, it is by far the easiest motor to control, requiring only a voltage source to rotate. To control the direction of rotation and the speed itself, H-bridge can be used, shown in figure 4.1. If the transistors A and D are switched, current flows in one direction, if transistors B and C are switched, current flows in the opposite direction. Switching the high transistors with a PWM signal controls the motor's speed, as the PWM duty corresponds to the average current in the coils. The H-bridge can also be made out of two half-bridge switches.

4.2 BLDC and PMS Motor

BLDC stands for *brushless direct current* and PMSM stands for *permanent magnet synchronous motor*. This section presents similarities and differences between these two machines, the mathematical description of the machine reference frames and the control methods. Construction wise, these motors do not differ too much, but they differ in the way of how these machines are controlled.

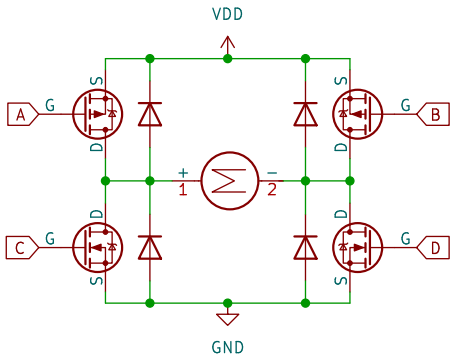
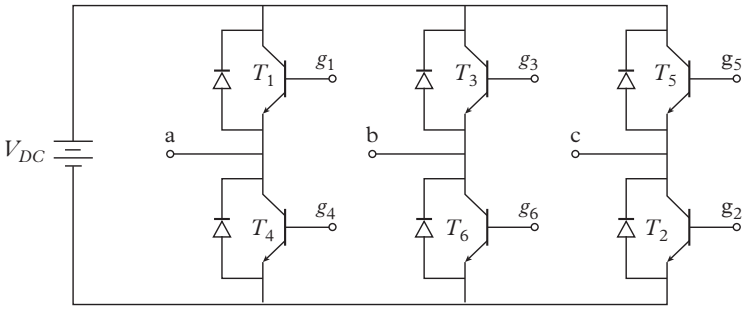


Figure 4.1: H-bridge and a DC motor

4.2.1 Construction of a Motor

The stators of both machines are made out of coils, providing a magnetic field, and the rotor contains permanent magnets. There are four types of permanent magnet materials, i.e., neodymium iron boron (NdFeB), SmCo, ferrite, and alnico. NdFeB magnets were discovered in 1983 and have linear demagnetization characteristics with very high remanence and coercivity. As a result, the energy product of the material is highest among all hard magnetic materials. ([31], 1.3.2). NdFeB permanent magnets are the most used ones. The commutation of the current flowing through the stator coils is done by the inverter. Also, both types of motor contain three *phases* used for the commutation. Compared to brushed DC motors, higher efficiency and reliability is achieved with the absence of the commutator.

Circuit wise, three half-bridge switches are used for the current commutation, connecting the winding either to VCC or GND. The circuit with a BLDC or a PMS motor and halfbridge switches is shown in figure 4.2. The way the half-bridge switches are switched for each motor type will be discussed later.



.

Figure 4.2: The driving of a BLDC or a PMS motor ([31], *Figure 1.2*)

4.2.2 Three Phase DC Motor Model

In this section, we refer to the modelled machine simply as PMS because of the cited sections from [31]. The differences between a BLDC and PMS motor are presented too. Before we start

the description of these machines, we need to distinguish between a mechanical and an electrical angle. In general, the motor can have multiple windings, which we denote by $P \in \mathbb{N}$ as *pole pairs*. A mechanical angle relates to the physical rotation angle of a rotor. In contrast, an electrical angle relates to the position of stator's magnetic field within one pole pair. That means a driver must rotate the magnetic field P times to turn around the rotor once.

Motor's Reference Frames

A schematic view of two popular PMS machines with four surface-mounted and interior permanent magnet poles is depicted in 4.3(a) and (b), respectively, to show the machine stator, rotor, and air gap. The stator windings of PMS machines are distributed sinusoidally in the stator slots around the periphery of the stator core with 120° displacements, producing a smooth rotating magnetic field in steady-state operation. Winding axes are, therefore, shown in the physical model as three stationary axes fixed to the winding centers and are 120° apart as noted by a, b, and c in 4.3. Also, the axis of a permanent magnetic pole, as the direct axis denoted by d, is shown in the figure for surface-mounted PM and interior PM machines. This axis is placed on the point of maximum flux density of the pole through its length, which is at the center of the pole. A quadrature axis, which is 90° (electrical) apart from the d-axis and denoted by q is also shown in the figure (note: the figure assumes two pole pairs so the d-q axes are 45° degrees apart from the mechanical perspective). The axes are fixed to the rotor body and rotate with the rotor. The two sets of a-b-c and d-q axes are referred to as phase variable RF and rotor RF, respectively ([31], *2.2.1 Machine schematic view*).

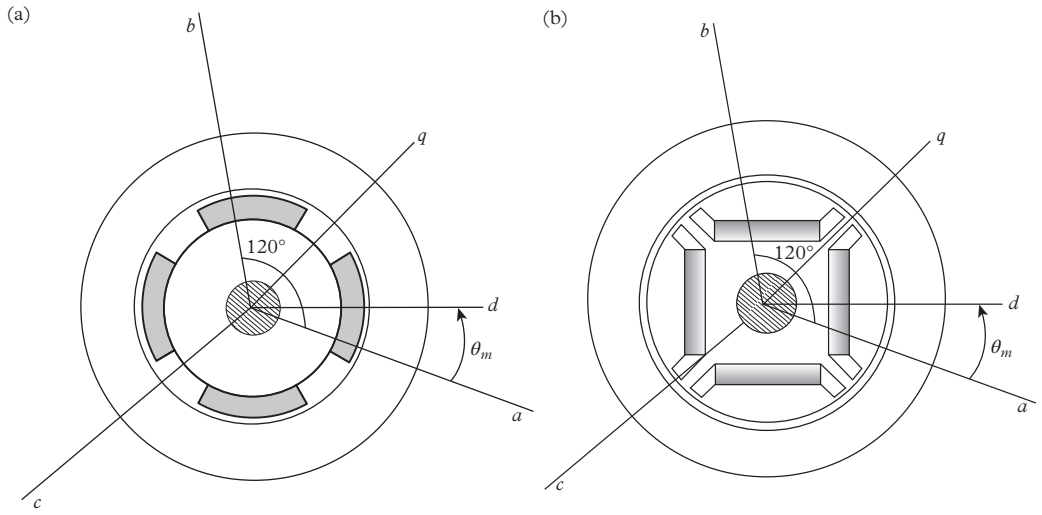


Figure 4.3: Schematic view of PMS machines with (a) surface-mounted poles and (b) interior poles ([31], *Figure 2.1*)

PMSM and BLDC Motor Differences

During modelling of a PMSM, several assumptions must be made:

- The stator windings of PMS machines are distributed sinusoidally in the stator slots around the stator core's periphery to produce a smooth rotating magnetic field in steady-

state operation. However, stator slot harmonics distort the magnetic flux produced by the machine windings. ([31], *2.2.2 Modeling assumptions*, point 1.).

- A sinusoidal flux density distribution around the periphery of the machine air gap produced by permanent magnet poles mounted on the rotor surface or buried inside the rotor core is assumed in the physical model despite an almost trapezoidal pattern of flux distribution in most actual machines ([31], *2.2.2 Modeling assumptions*, point 3). Shown in figure 4.4.

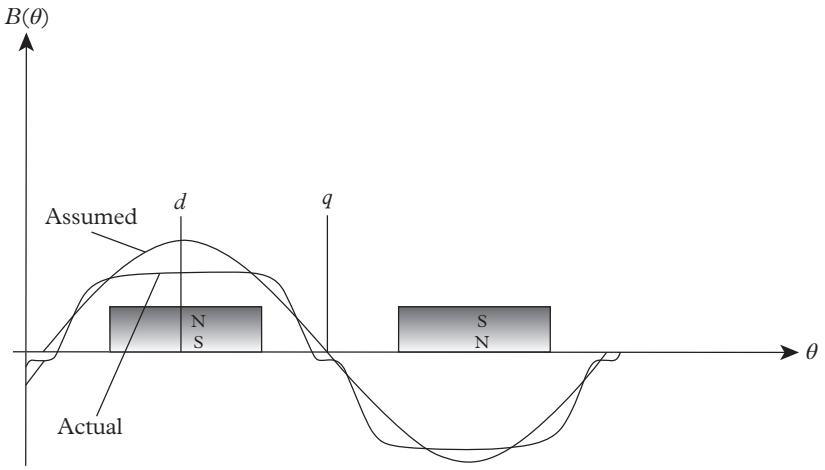


Figure 4.4: Actual and assumed patterns of air gap flux density distribution produced by a pair of PM poles ([31], *Figure 2.2*)

If a motor does not have sinusoidally distributed windings or the flux density of the magnets is not sinusoidal, then the motor's movement is not smooth when driven by sinusoidal voltage. This is what differentiates BLDC machines from the PMS ones.

Two-axis Reference Frame

To simplify the work with the stator's reference frame, the transformation from the three-phase *abc* frame to the orthogonal $\alpha\beta$ frame is called the *Clarke transformation*. It essentially projects *abc* quantities into an $\alpha\beta$ frame with a being in the same direction as α. We can express this by a matrix multiplication of a general vector $\mathbf{f}$:

$$\begin{bmatrix} f_\alpha \\ f_\beta \\ f_0 \end{bmatrix} = \frac{2}{3} \begin{bmatrix} 1 & -\frac{1}{2} & -\frac{1}{2} \\ 0 & \frac{\sqrt{3}}{2} & -\frac{\sqrt{3}}{2} \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \end{bmatrix} \begin{bmatrix} f_a \\ f_b \\ f_c \end{bmatrix}. \tag{4.1}$$

In the PMSM models, it is assumed that

$$f_a + f_b + f_c = 0, \tag{4.2}$$

as is the case for the variables of a star-connected three-phase winding with no neutral connection or a delta-connected three-phase winding ([31], *Stationary two-axis model*).

From that $f_0 = 0$ (hence the 0 index) and

$$f_\alpha = f_a, \tag{4.3}$$

$$f_\beta = \frac{f_a}{\sqrt{3}} + \frac{2f_b}{\sqrt{3}}. \tag{4.4}$$

The f_0 quantity is mentioned too because this way the matrix is invertible. The *Inverse Clarke Transformation* can be expressed as

$$\begin{bmatrix} f_a \\ f_b \\ f_c \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ -\frac{1}{2} & \frac{\sqrt{3}}{2} & 1 \\ -\frac{1}{2} & -\frac{\sqrt{3}}{2} & 1 \end{bmatrix} \begin{bmatrix} f_\alpha \\ f_\beta \\ f_0 \end{bmatrix}. \tag{4.5}$$

Again, if we suppose $f_0 = 0$, then

$$f_a = f_\alpha, \tag{4.6}$$

$$f_b = -\frac{f_\alpha}{2} + \frac{\sqrt{3}f_\beta}{2}, \tag{4.7}$$

$$f_c = -\frac{f_\alpha}{2} - \frac{\sqrt{3}f_\beta}{2}. \tag{4.8}$$

Rotation of a Two-axis Frame

A vector represented by the $\alpha\beta$ two-axis system coordinates can be represented in the orthogonal dq system, whose base vectors are rotated by an angle φ against $\alpha\beta$. This transformation is called the *Park Transformation* and can be expressed using matrix multiplication:

$$\begin{bmatrix} f_d \\ f_q \end{bmatrix} = \begin{bmatrix} \cos\varphi & \sin\varphi \\ -\sin\varphi & \cos\varphi \end{bmatrix} \begin{bmatrix} f_\alpha \\ f_\beta \end{bmatrix}. \tag{4.9}$$

The *Inverse Park Transformation* can be expressed as

$$\begin{bmatrix} f_\alpha \\ f_\beta \end{bmatrix} = \begin{bmatrix} \cos\varphi & -\sin\varphi \\ \sin\varphi & \cos\varphi \end{bmatrix} \begin{bmatrix} f_d \\ f_q \end{bmatrix}. \tag{4.10}$$

Equations 4.3 - 4.4, 4.6 - 4.8, 4.9 and 4.10 are important because they are implemented in the pysimCoder source code [1], [2].

4.2.3 Control Methods

This section summarizes the BLDC and PMSM control methods. The commutation of the stator's magnetic field for these machines must be done by an external driver. This section discusses various commutation methods using the half-bridge trio, as shown in figure 4.2.

[1] `CodeGen/Common/common_dev/clarke_trans.c`
[2] `CodeGen/Common/common_dev/park_trans.c`

Trapezoidal Switching

In a motor employing a trapezoidal control scheme, the MOSFET bridge switching must occur in a precisely defined sequence for the BLDC motor to operate efficiently. The switching sequence is determined by the relative positions of the rotor's magnet pairs and the stator's windings. A three-phase BLDC motor requires a six step commutation sequence to complete one electrical cycle [5].

To determine the right switch step, we need to estimate the rotor's position. Two technologies offer a solution for positional feedback. The first and most common uses three Hall-effect sensors embedded in the stator and arranged at equal intervals, typically 60° or 120° (described in 4.4.2). A second, sensorless control technology comes into its own for BLDC motors that require minimal electrical connections [5].

During each step of the commutation, one winding is driven high, one is driven low and the other is left unconnected. If we know the position of the rotor, we know what windings to drive. This is usually described by a diagram or a switching table.

Being this the easiest type of motor control, torque ripple is present, causing an uneven movement of the rotor. Instead of driving the windings with a PWM modulated direct current, a SPWM (*sinusoidal pulse width modulation*) or field oriented control can be used.

Vector Control

In electrical machines, a torque is developed due to the interaction of two magnetic fields. In a mathematical sense, the torque is proportional to the outer product of the two magnetic flux linkage vectors. In separately excited DC machines, the two magnetic fields are provided by two separate flux sources, i.e., stator or field winding and rotor or armature winding. Furthermore, in these machines a commutation system always locates the two magnetic fields spatially perpendicular to each other, providing the best space position of the two fields for the purpose of torque production ([31], *1.5.1 Vector Control*).

The main goal of a vector control (VC) method is to make the two magnetic fields independent and perpendicular as much as possible. VC is a means by which the stator phase currents of AC machines are transformed into a current vector of two perpendicular components, analogous to the field current and the armature current of a separately excited DC motor, to control the torque and flux independently. The function of the commutation system in DC machines is carried out by the current transformation in vector-controlled AC machines to keep the current components in normal position with each other. Information on rotor position or flux linkage vector position is needed for the transformation ([31], *1.5.1 Vector Control*).

As was mentioned in 4.2.2, the stator's reference frame can be described by the *a*-*b*-*c* axes and the rotor's reference frame is described by *d*-*q* axes. The vector control is based on measuring the i_a, i_b and i_c currents from the three windings. By applying the *Clarke transformation*, these currents are projected into an orthogonal $\alpha\beta$ reference frame. When the angle of the rotor is estimated, by applying *Park transformation* we get i_d and i_q.

The vector control consists of two controllers, one being an inner pair of i_d and i_q current controllers and the outer one setting the d and q current references, which we denote by i_d* and i_q*. The inner controllers are responsible for keeping the currents in the d and q axes. By applying the *Inverse Park* and *Inverse Clarke transformations* we get the v_a*, v_b* and v_c* actions that we apply to the inverter.

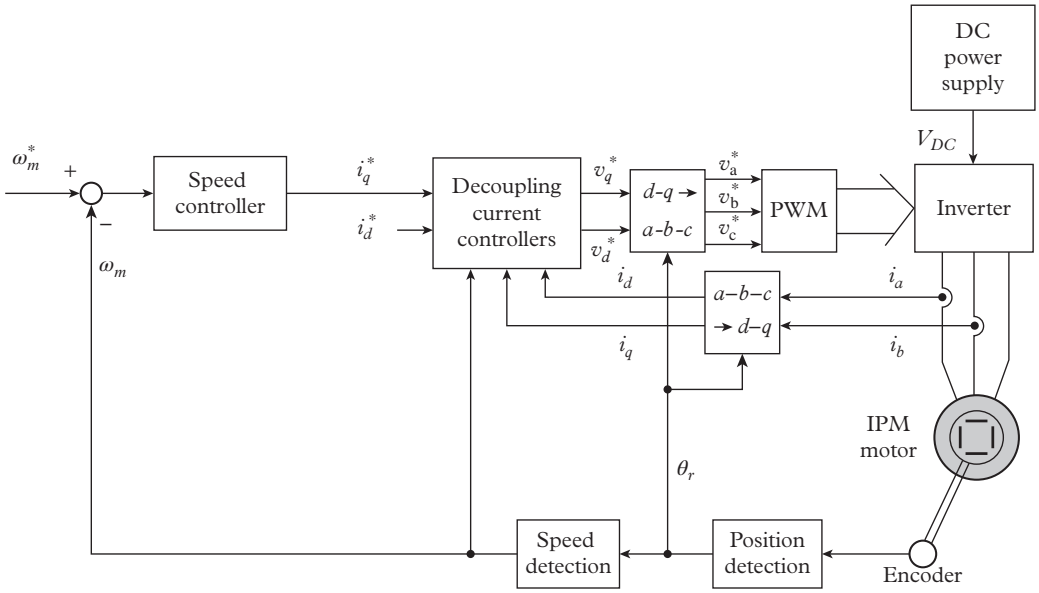


Figure 4.5: A schematic of vector control based speed controller ([31], *Figure 3.6*)

The outer controller sets i_d* and i_q* references, according to the mathematical model of the whole controlled system, for example a speed or position controller. The schematic of the vector control is depicted in figure 4.5. Several strategies for i_d* and i_q* can be deployed, but the simplest one is to control only i_q*, while $i_d* = 0$. This way, the current controller is trying to keep the stator current only in the rotor's q axis, perpendicular to the d axis, producing the highest torque.

4.3 Other Types of Actuators

While the motion controller currently only works with the DC and BLDC/PMS motors, other actuators are briefly explained as well, because of the modulatity of our controller.

4.3.1 Stepper Motors

As all with electric motors, stepper motors have a stationary part (the stator) and a moving part (the rotor). On the stator, there are teeth on which coils are wired, while the rotor is either a permanent magnet or a variable reluctance iron core.

The stator is the part of the motor responsible for creating the magnetic field with which the rotor is going to align. The number of phases is the number of independent coils, while the number of pole pairs indicates how main pairs of teeth are occupied by each phase. Two-phase stepper motors are the most commonly used.

There are two types of stepper motors: unipolar and bipolar. In unipolar motors, each coil has a central tap, connected to the power supply. Each end of the coil is switched to the ground. In bipolar stepper motors, each coil has only two leads available, and to control the direction it is necessary to use an H-bridge [16].

There are many types of stepper driving techniques. The advantage of a stepper motor is

the fact that it is operated as a synchronous machine by performing steps. Thus open-loop control can be used for precise positioning if the required torque is not too high.

4.3.2 Piezoactuators

The piezoelectric ceramic material used in the piezoelectric actuator generates electrical energy when it is subjected to mechanical energy (piezoelectric effect) and generates mechanical energy when it is subjected to electrical energy (inverse piezoelectric effect). The piezoelectric actuator is a device that makes use of the inverse piezoelectric effect [32].

At operating frequencies well below the resonant frequency, a piezo actuator behaves like a capacitor. The actuator displacement is proportional to the stored electrical charge, as a first order estimate. The capacitance of the actuator depends on the area and thickness of the ceramic as well as the material properties. In the case of actuators that are constructed of several ceramic layers electrically connected in parallel, the capacitance also depends on the number of layers [20].

The piezoactuators need to be driven by a linear amplifier. However, we can put a lowpass filter after the inverter, turning the PWM signal into a DC signal which depends on the duty.

4.4 Position estimation

4.4.1 Optical encoder

Encoders can be either linear or rotational, having encoder strip or disk, respectfully. This device operates on the principle of converting mechanical displacement into an electrical signal through the modulation of light. The key components of an optical encoder include the encoder disk or strip, a light source (typically an LED), and photodetectors. The encoder disk or strip is patterned with transparent gaps.

In operation, the light source illuminates the encoder disk or strip, and as it moves, the alternating gaps modulate the light reaching the photodetectors. The photodetectors then convert this modulated light into an electrical pulses which can be counted by a counter on FPGA or MCU. Position can be then calculated from the number of counts and the density of gaps per milimeter or degree.

An absolute optical encoder provides a unique digital code for each position on its measurement range, enabling it to instantly determine the exact position of a moving object without requiring a reference point. In contrast, a relative optical encoder measures changes in position relative to a starting point.

Incremental Rotary Encoder (IRC)

In this sensor, there are two signals shifted by quarter of a period. From the phase shift of these two signals, the orientation of movement can be deducted (left $\times$ right movement, clockwise $\times$ counterclockwise rotation). The disadvantage of relative measurement can be solved by adding an index (or Z) signal which generates signal at a precisely defined location. In rotary optical encoders, it is usually once per revolution.

The figure 4.6 shows an industrial optical encoder from Omron. The simplest rotary encoder has 5 wires in total: VCC, GND, 2 phase shifted signals for counting and an index. However,

more expensive encoders feature differential signalling to reduce the effect of electromagnetic interference. Such controller must then have differential line receiver.

Many microntrollers have a built-in quadrature encoder logic cooperating with a timer peripheral where a counter is incremented or decremented depending on the two incoming signals. From the value saved in counter, the positon of the motor can be calculated.

Figure 4.6: Example of a rotary encoder from Omron taken from [17]

4.4.2 Hall Sensors

Hall sensor is a sensor working on the principle of Hall effect. Hall sensor produces voltage proportional to the perpendicular magnetic field. This sensoring method is used as a position feedback method to measure the postiion of shaft made out of permanent magnet. The outputs of Hall sensors are of logical value and are typically open drain, so microcontroller GPIO pins are used to read the values.

Three Hall sensors are mostly used in BLDC and PMS motors to roughly estimate the absolute position of the shaft. The allowed boolean sequence of codewords is 001, 101, 100, 110, 010, 011, corresponding to readings from $H_3H_2H_1$, codewords 000 and 111 are not allowed. Totally, this gives 6 combinations mapping to 2π radians of the electrical angle which means each reading estimates one third of π radians.

These sensors are not intented to be used as precise estimation of shaft's angle, but may used as the initial estimation before an index mark of optical encoder is hit. In applications where precise position is not a concern, they can be used standalone.

Chapter 5

Introduction to Used Hardware and Interfaces

This chapter introduces the tools, electronic parts used in our controller. Also, the analysis of the problem is described in this chapter, alongside with the proposal of the controller's main features.

5.1 PCB Design

This section mentions several factors which need to be taken in account when designing custom PCBs for this controller.

5.1.1 PCB Copper Layers

Advantages of More Layers

Using more layers where traces can be routed simplifies the PCB design. It is convenient to use inner layers as a ground plane and a power plane, allowing us to get power and grounding for electrical parts on outer layers with the help of vias, while not routing grounds and power on outer layers.

By providing an uniform conductive surface, the power and the ground plane also help in reducing the electromagnetic interference, because these planes act as shields and form a nonnegligable decoupling capacitor. Due to the high area of copper, these planes also help in heat dissipation of the whole PCB.

More layers can be helpful if there is limited space for traces. If the circuit contains a lot of signal traces, more layers can be used, however it makes the fabrication of PCB more expensive. A handful of signal traces can be routed in power or ground planes but one must be careful not to divide the power or ground plane into smaller parts with these traces, because it breaks the uniformity of these layers.

Copper Thickness

Thicker copper can take more current and helps in heat dissipation. However, wider tracks must be used on layers with thicker copper because of underetching. The upper part of the trace is exposed to etching substance for a longer time than the lower part, which means more of the upper part is etched away. It is important to obey manufacturer's minimum track width for given copper thickness.

5.1.2 Track and Via Ampacity

When designing high current PCBs, the track widths must be taken in account. Ampacity of the trace can be estimated by the formula

$$I = K(\Delta T)^{0.44}(WH)^{0.725}, \tag{5.1}$$

where I is the estimated current, W is the thickness of the trace, H is the thickness of the copper layer, K is either 0.024 for inner layers or 0.048 for external layers and ΔT is the temperature rise of the track. This is an empiric formula devised by the IPC organization in their IPC 2221 document.

There was no need to design tracks of precise width, but it helped me to estimate widths of high current tracks with small temperature rise.

Ampacity of the vias must be considered too, when routing current through different layers is desired.

5.1.3 Electronic Design Automation Software

Such software helps user draw schematics and design custom PCBs. Since our controller is open hardware, using tool like KiCad is the best choice since it is an open source software and is fairly known among electrical engineers. The figure 5.1 shows the KiCad's PCB editor. There are many tools available, like Altium Designer, Cadence's OrCad, Autodesk's Eagle or the open source gEDA suite.

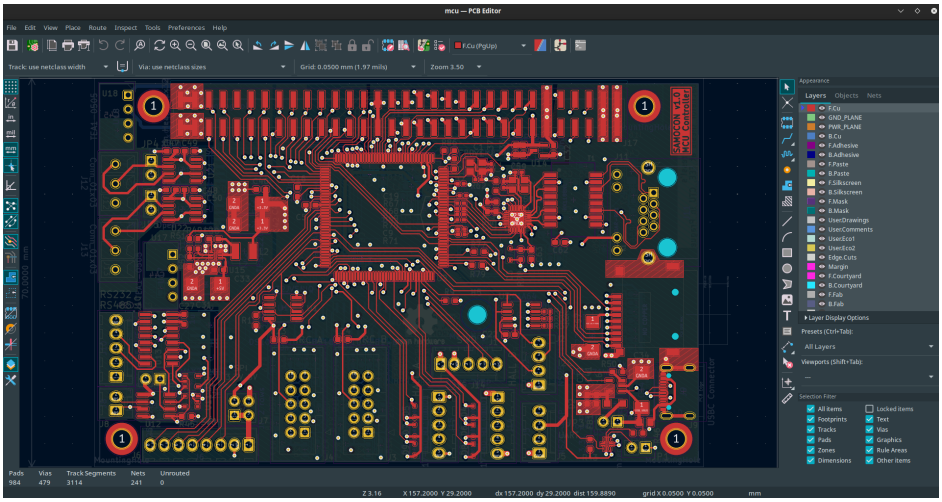


Figure 5.1: KiCad's PCB editor

5.2 Communication Interfaces

In this section, a few communication interfaces are presented because of their needed hardware adaptation in our circuit. All the communication interfaces like SPI, I^2C and USB won't be discussed and the familiarization with these interfaces is left to the reader.

5.2.1 Serial Communication

The simplest protocol for serial communication is UART (*Universal Asynchronnous Receiver and Transmitter*). UART uses RX for data reception and TX for data transmission. The speed of the communication is given by the baud rate. This thesis supposes the UART communication is known to the reader, so only the physical layer of serial communication will be discussed.

The standard UART is compatible with TTL (5 V) or LVTTL (3.3 V) logic. However, UART voltage levels are not suitable for long transmission lines and UART itself is only point to point.

RS232

This standard defines the used voltage levels, as well as other signals like RTS (*ready to send*) or CTS (*clear to send*), suitable for control flow.

The standard also defines the pinout on a typical DE-8 connector. The typical logical high level is -3 to -15 V and the logical low level is 3 to 15 V. MAX232 is a very known IC used for converting standard levels to RS232 levels. By using external capacitors, RS2321compatible voltage levels are produced while the IC powering voltage and UART voltage levels are 3.3 or 5 V.

RS422

RS-422 is a unidirectional, multidrop network, allowing for the use of a single driver with an output signal that can be dropped at multiple (up to 10) receiver nodes [23]. The physical medium is usually a twisted pair cable with differential signalling. Because data traffic is unidirectional, only one termination resistor is required to eliminate signal reflections on the bus. Its value should match the characteristic cable impedance, typically in the range of 100 Ω to 130 Ω ([23], Figure 1). If point-to-point full duplex communication is desired, two RS422 interfaces are used.

RS485

RS-485 is a bidirectional multipoint interface, which allows for multiple drivers to be connected to the same bus. To avoid bus contention, bus access is controlled with driver enable pins. Only one driver can access the bus at a time ([23], Figure 4). The transceiver example is shown in figure 5.2. The data can be received on RO (RX) and transmitted on DI (TX) pin, data reception is activated by the $\overline{\text{RE}}$ pin and transmission of data is activated by the DE pin when the bus is idle.

5.2.2 Ethernet

Introduction

Ethernet is a family of network technologies first standartized as IEEE 802.3. From the ISO/OSI layer model, Ethernet is responsible for the physical and link layer. Nowadays, Ethernet supports a lot of physical mediums, like the optical fibre or the most widely used UTP (*unshielded twisted pair*) or STP (*shielded twisted pair*) with the characteristic impedance

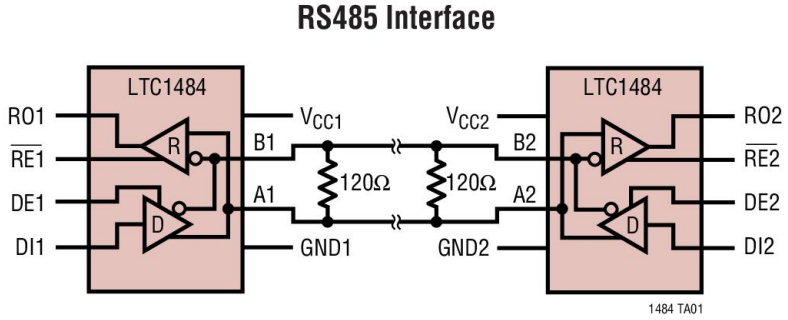


Figure 5.2: Example usage of two LTC1484 transceivers (taken from [2])

around 100 Ω. Even though Ethernet was mainly used as a network technology on personal computers, it has become favourite in the industrial fields, like automotive.

For the purpose of this thesis, the 10/100 Base-T/TX standards will be discussed, as these are the only standards supported by our microcontroller. Having Ethernet on our controller is benefitial, because it is the most popular technology that allows us to access our controller remotely from the network.

The standard defines two interconnected interfaces, the first one being MAC (*Media Access Control*) and the second one being PHY (*Physical Layer*). MAC is responsible for the link layer, while PHY is responsible for the physical layer. Microncontrollers often include a MAC peripheral interfaced from software while an external PHY chip must be connected.

Access Control

Originally, Ethernet was designed as a protocol to run over a shared medium. Simultaneous transmission from multiple nodes would result in garbled data on the medium, and subsequent loss of data. From this simple example, we can derive some basic requirements for a network protocol:

- multiple nodes must be able to transmit on a shared medium,
- each node must be able to detect when another node is transmitting,
- a transmitting node must be able to determine when simultaneous transmission occurs in the case where multiple nodes see the medium as idle and start transmitting at the same time,
- when a collision is detected, each node must have a method to determine when to retransmit without each node continually trying to retransmit at the same time.

These requirements are met in Ethernet using a scheme known as *Carrier Sense Multiple Access with Collision Detect* (CSMA/CD) [10] and the algorithm used to determine the retransmision is called the exponential backoff algorithm. This method plays a role in half-duplex shared medium operation. In full-duplex operation collisions cannot happen.

PHY

Before the frame is transmitted, it needs to be encoded to be effectively transmitted over a long transmission line. In the 10 Mb/s Ethernet, Manchester encoding is used. Because Unshielded Twisted Pair (UTP) wires are low-pass in nature, the same encoding scheme that was used for 10Base-T will not work when we increase the speed by 10x ([10], *100 Mb/s STREAM CONTENTS*). For the 100 Mb/s operation MLT-3 and 4B/5B encodings are used.

The PHY is responsible for the encoding/decoding and the transmission line driving. Firstly, a PHY receives a frame from the MAC which must be encoded before it is transmitted onto the physical medium. The MAC communicates with the PHY using MII (*Media Independent Interface*) or RMII (*Reduced* MII). PHY is also responsible for the *auto negotiation* (a mechanism used by Ethernet to negotiate common parameters, like the duplex mode, speed, ...) and it also includes R/W registers accessed by the MDIO (*Management Data Input/Output*), from which the PHY can be configured (e.g. configure the maximum negotiated bitrate) or the auto negotiation state with an another device (like a switch) can be read.

The way MAC is connected to PHY is shown on figure 5.3. As the PHY must drive the physical medium, these components can be power hungry.

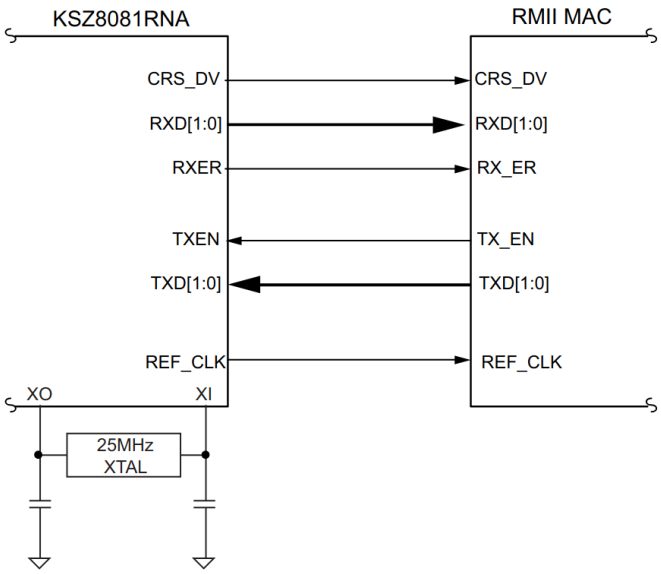


Figure 5.3: Connecting KSZ8081 PHY using RMII to a MAC device (e.g. a MCU) (taken from [12])

5.3 Motion Controller Analysis

Most of the communication interfaces mentioned in the previous section should be present on our controller. The following table 5.1 describes all the peripherals our controller should contain and should serve as the baseline for our design. We also require that our motion controller should be capable of driving at least 2 stepper motors. Since a stepper motor requires 4 half bridge switches, we need total of 8 half-bridge switches. A BLDC/PMS motor requires 3 halfbridge switches to be controlled and a DC brushed motor requires 2 half-bridge

Peripheral	*Use Cases*
TTL UART	Simple console user interface
RS232, RS422 or RS485	Communication with other devices, sensor data acquistion
(Q)SPI	Ext. SPI memory (e.g. W25), sensor data acquisiton
I^2C	Ext. I^2C memory (e.g. 24xxxx), sensor data acquistion
Counter	Position measurement from encoders
JTAG or SWD	Program flashing and debugging
CAN	Communication with other devices
Ethernet	Data logging, remote access, pysimCoder parameter tuning
SD slot	Data logging
PWMs	Control of power switches
ADCs	Current measurement for motor FOC (*field oriented control*)
GPIOs	Reading from Hall sensors, general stuff

Table 5.1: Table of peripherals and corresponding use cases

switches (forming a H-bridge). Choosing 8 half-bridge switches is enough for controlling 2 BLDC/PMS motors and 4 DC brushed motors.

With that, we need at least 8 PWM outputs for power switches (or 16 if each gate driver requires complementary signals), at least 8 ADC channels for current measurement, two counter peripherals for encoder pulse counting and in the best case all the communication interfaces mentioned above.

The last design rule is to divide the motion controller into a *Control MCU board* and a *Power stage board* as two PCBs. The complete motion controller then comprises of these two PCBs connected together. The MCU board should contain the microcontroller and all the connectors for communication interfaces. The power board should contain the half-bridge switches and measurement circuits, like operational amplifiers. The MCU board is a more complex board, so this board should be kept the same, while new revisions of a power board should adapt to the MCU board. Overall, the design is inspired by the PiKRON's Lx-RoCoN motion controller [21]. We also require all used components must be rated as industry parts, so the controller can be used in harsher environments (at least -40 to 85°C).

5.3.1 The Used Microcontroller

The chosen microcontroller is ATSAMV71Q21B from Microchip. The reason for choosing such microcontroller is because of supervisor's and Michal Lenc's good experience with this microncontroller.

The intention is to route as many SAMV71's peripherals as possible because of high modularity and extensibility. The biggest LQFP144 package was chosen. However, Microchip also manufactures this microcontroller in smaller LQFP100, LQFP64 and BGA packages so in case of smaller designs, a whole new motion controller can be designed while keeping software the same.

This microcontroller features an ARM Cortex M7 core running at 300 MHz, it has 2 MB of flash memory and 384 kB of SRAM. This microcontroller features a lot of needed peripherals, ranging from Ethernet, to two PWM peripherals each featuring 4 channels with complementary outputs with configurable deadzones, two AFEC (*Analog Front End Controller*) peripherals

each having up to 11 AD channels at disposal and four TC (*Timer/Counter*) peripherals with encoder counter logic. It also features 2x CAN, a HSMCI peripheral for SD interfacing and also many UARTs, SPIs, TWIHSs (Microchip's I^2C) and so on. Another advantage is vast driver support in NuttX RTOS and also support for double precision float instructions, suitable for pysimCoder general generated code with the `double` data type.

5.3.2 The Used Power Switch

IFX007T is a half-bridge switch developed by Infineon Technologies. It is an all-in-one switch featuring two N power switching MOSFETs and its gate drivers, It also features a built-in high and low side current limitations alongside with overtemperature protection. This makes the switch very easy to use, as it only requires one PWM driving signal (IN) and an inhibit (INH) input controlled by a GPIO. The figure 5.4 shows the internal structure of the IFX007 chip. We can see the switch also features an IS output whose current is proportional to

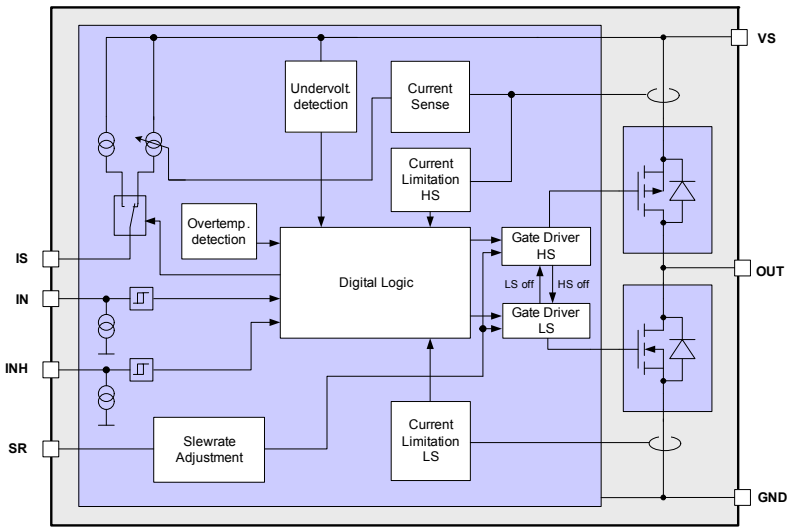


Figure 5.4: Block diagram of IFX007 internal structure [8], 2.1

the current flowing through the high side transistor to the load. The high side current can be then measured by measuring the voltage drop across the connected resistor. The IS pin also serves the purpose of a fault indication in case of overcurrent or high temperature. The characteristics of IS output current is shown in the figure 5.5.

Current Measurement

A decision was made to include additional low side transistor measurement using a shunt connected to ground. This way, a high-side and low-side currents can be measured. However, we use the IS high-side value as an indicator of average current by connecting a parallel capacitor and the indicator of a fault. As the figure 5.5 suggests, the fault output current is bigger than expected current during normal operation. This way, a fault voltage drop can be compared with a reference on a comparator. One fault output per motor can be generated by chaining 4 open drain/collector comparators.

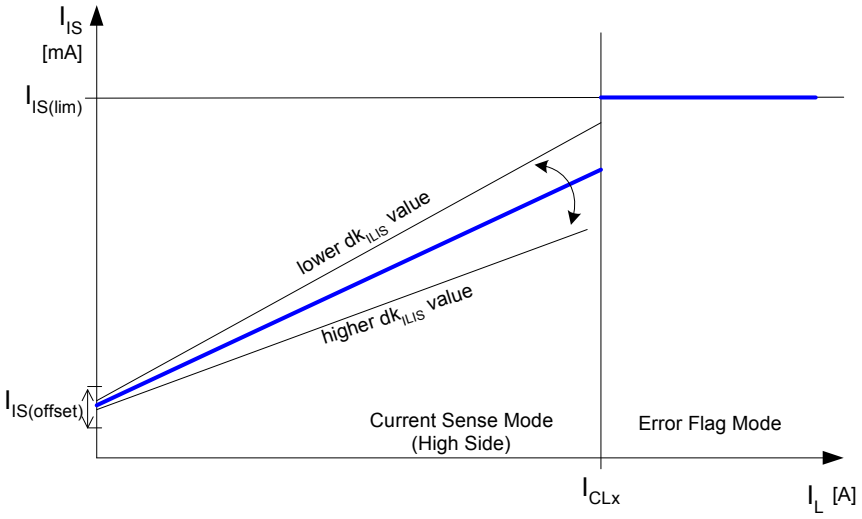


Figure 5.5: IS output current [8], Figure 13

The low-side shunt is used for more precise measurement of the motor winding current. The current measurement and fault generation scheme is depicted on figure 5.6. If the voltage drop on the R_{is} resistor is bigger than V_{ref}, the comparator output goes low. Since the operational amplifiers must be connected to the analog ground, the measurement of the shunt R_{s} is done using a differential amplifier because R_{s} is connected to the power ground. Since the current passing through the transistors can be negative, a bias voltage is applied to the differential amplifier, shifting the output voltage of the amplifier. Fast voltage spikes on the R_{IS} resistor can be filtered by connecting a parallel C_{IS} capacitor.

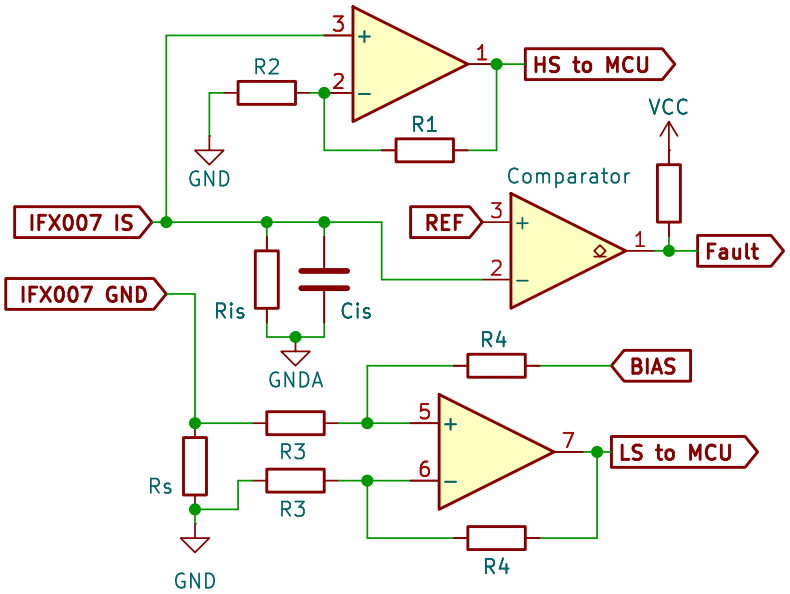


Figure 5.6: HS and LS current sensing alongside with fault generation

Chapter 6

Hardware Implementation

This chapter describes the hardware implementation details of the MCU control and the power stage board, together forming an interconnected stack. Since the MCU control board contains a lot of connectors, it is placed above the power board. The hardware is available publicly online under the OHL-CERN-W v2 license on the OTREES GitLab [1]. Since the used microcontroller is from the Microchip's SAM family, the motion controller model name was chosen to be *SaMoCon* as an abbreviation of ATSAMV71Q21B Motion Controller. The design is inspired by the PiKRON Lx-RoCon motion controller [21].

6.1 MCU Board

6.1.1 Interconnecting with Expanbility

An interconnection pinout between the boards needed to be defined. We need 16 inputs from the operational amplifiers for current measuring. We also need 8 PWM signals and 8 GPIOs for IFX007 control.

The connection of two boards is placed on the back of the PCBs. For the purposes of expansibility, it is good to have extra GPIOs, SPI or I^2C routed. There is an another connector on top of the MCU board with the same mirrored pinout as the lower interconnection which can be used for expanding boards placed above the MCU board. The upper MCU connector can be used as an oscilloscope probing place too. In upcoming sections, the peripherals routed to the separate peripheral connectors on the MCU board will be reffered to as *main peripherals* whereas the peripherals routed to the interconnection/expanding connector will be reffered to as the *extra peripherals*.

6.1.2 Microcontroller's Pinout

Before any routing took place, we needed to determine the SAMV7's pinout. Because of the microcontroller's alternative pin functions, there were a lot of combinations of how routing could be done. However, due to high count of desired peripherals, conflicts started to happen - two peripheral outputs or inputs were on the same pin. For example, we weren't able to include a QSPI memory because of these problems. To see the SAMV71's pinout, see [14], section 6.1.

[1] `https://gitlab.fel.cvut.cz/otrees/motion/samocon`

See table 6.1 for routed peripherals and the corresponding SAMV71's peripheral. The detailed description of pin assingments can be found in the spreadsheet here [2].

SaMoCon Peripheral	*SAMV7 peripheral*
2x CAN	MCAN0, MCAN1
SPI main & extra	SPI0, SPI1
TTL Console	UART3
RS232/RS485	USART2
Ethernet	GMAC in RMII mode
USB-C	HSUSB
2× IRC	TC0, TC2 (Timer/Counter)
SD Card	HSMCI
H PWM outputs for Motor A	PWM0 CH0, CH1, CH2, CH3
L PWM outputs for Motor A	PWM0 CH0, CH1, CH2, CH3
H PWM outputs for Motor B	PWM1 CH0, CH1, CH3
L PWM outputs for Motor B	PWM1 CH0, CH1, CH2, CH3
8 ADC channels for Motor A	AFEC0
8 ADC channels for Motor B	AFEC1
3 extra ADC channels	AFEC1
2x 3 Hall inputs	GPIOs on the same PIO
I^2C main & extra	TWIHS0, TWIHS1

Table 6.1: Routed peripherals

As previously mentioned in 5.3.2, each switch must be controlled by one PWM H signal and one GPIO pin. Even though we need 8 PWM H outputs, we also route all remaining PWM L outputs, because some gate drivers require complementary signals. In case of IFX007T, the L output can be configured as a simple output pin.

Unfortunately, the only one H output of PWM1 CH2 could not be routed due to collisions with the HSMCI peripheral. A decision was made to route the L channel output as the H channel, because, as it turned out, the PWM polarity can be configured in software. This leaves us with total of 7 complementary and 1 non-complementary outputs.

6.1.3 Power Components

All the components on this PCB are powered from the 5 or 3.3 V rail. The MCU board is designed to be powered from the power board through the interconnection. The 5 V rail can also be supplied from the USB-C connector.

To get 3.3 V from 5V, the TPS562207 buck regulator from Texas Instruments [24] has been used. The datasheet recommended circuit has been used and the implementation is shown in the figure 6.1.

Two CAN and RS232/RS485 peripherals are galvanically isolated, meaning small isolated switching power supplies are needed. For that purpose, small black 5V to 5V power supplies capable of outputting 1 W have been chosen.

[2] `https://gitlab.fel.cvut.cz/otrees/motion/samocon/-/blob/main/hw/pinouts.ods`

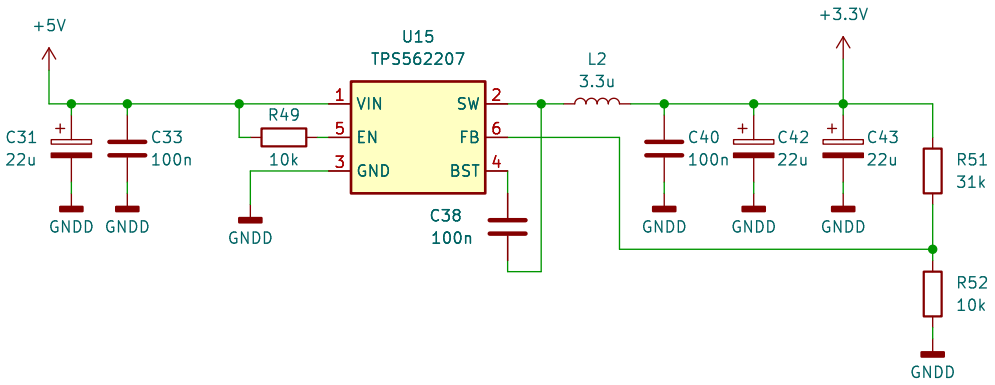


Figure 6.1: The TPS562207 buck regulator

We also need a stable voltage reference for analog measurements. The MCP1525-TT [11] reference from Microchip has been used. The advantage over the widely used TL431 is no cathode resistor is needed.

6.1.4 CAN

The Microchip's MCP2562FD [15] IC was used as a CAN physical layer transceiver. CAN was meant to be galvanically isolated, so optical ADuM1201 isolators from Analog Devices have been chosen. A 120 Ω resistor has been added parallel to each output together with a jumper if termination of the transmission line is required.

6.1.5 RS232/RS485

This peripheral is galvanically isolated too, using ADuM1201 as optical isolators. RTS and CTS pins needed to be connected too, needing two bidirectional optical isolators in total. To convert UART TTL levels to corresponding voltage levels, MAX232 and ST485CDR transceivers are used.

RS232 and RS485 are two distinct interfaces with different voltage levels and different flow control. However the solution on our board allows to have these interfaces on the PCB while using only one. The circuit of this whole serial communication interface is shown in the figure 6.2.

Both interfaces can be used without any changes to the circuit, except the jumper which disables/enables data reception from RS485. To prove our point, we will discuss the circuit when using RS232 only and RS485 only.

Using RS232 only. If something on RTS or TX is sent, it propagates to MAX232 IC. If data CTS or RX must be received, it can be seen that these signals are switched by AND gates. However, if RS485 differential line is not connected, then RO pin of the RS485 if high, leaving the CTS and RX signals on the ADuM1201 isolator being dependant only on signals from MAX232.

Using RS485 only. In order to send data, RTS pin must be set high to activate the transceiver. Since nothing is received from MAX232, the outputs of this IC are high, making the CTS and RX signals on the ADuM1201 isolator being dependant only on signals from

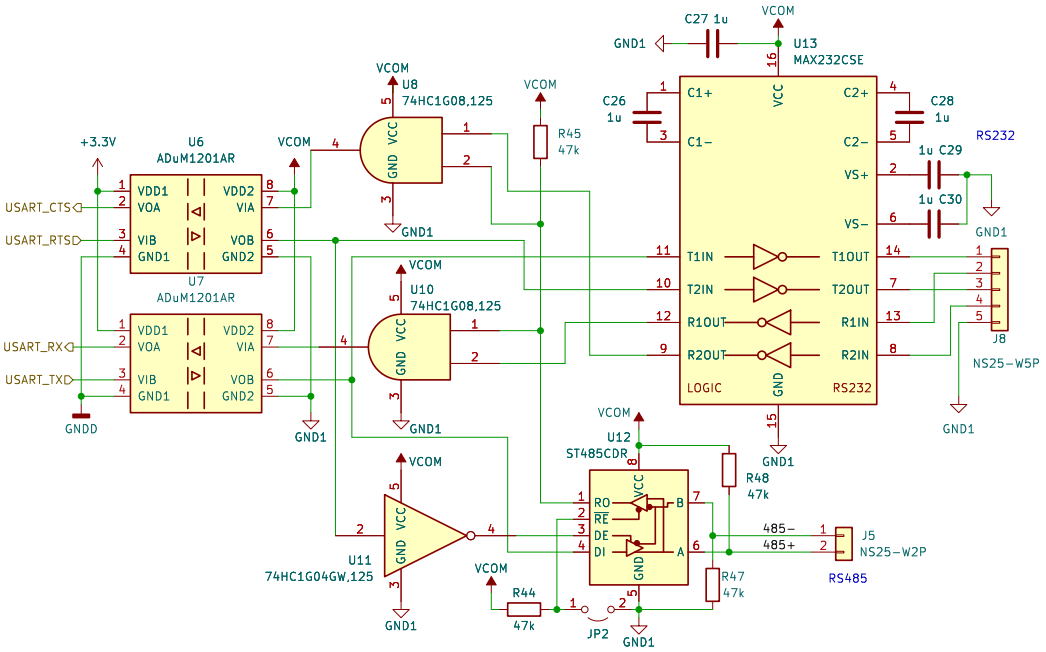


Figure 6.2: RS232 and RS485 communication interface

the RS485 transceiver. CTS used for signalling data activity is connected to RO pin as the RX pin. If TX data is to be sent, it gets immediatetly propagated to DI pin.

6.1.6 Ethernet

A so called PHY (*Physical Layer*) component must be connected to the MCU's MAC (*Media Access Controller*). For that, the Microchip KSZ8081 IC was used [12], working in the RMII mode. For galvanic isolation, the magnetics (de facto a transformer) needs to be connected between the PHY and the RJ45 connector. HX1188NL magnetics, supported by the PHY, was used. The routing of Ethernet components is shown in figure 6.3.

KSZ8081 PHY

The design is based on the recommendations given by the datasheet [12]. That includes the powering part (ferrites, capacitors) and all the pull up and pull down resistors connected to open drain outputs and a 25 MHz crystal. Some resistors also define the PHY's address, which was chosen to by PHYAD[4:0] = 0 (bits 4:3 are always zero). The PHY also includes an Ethernet activity pin intended for the RJ45 activity LED. A 560 Ω current limiting resistor needed to be added.

6.1.7 USB

The USB-C's CC1 and CC2 pins are connected to the TUSB321 IC from Texas Instruments which negotiates host or device operation, given voltage levels at certain pins. The board was designed in such way that it could function alone from either the connected power board or the power from USB. The data pins are connected to the HSMCI SAMV7 peripheral.

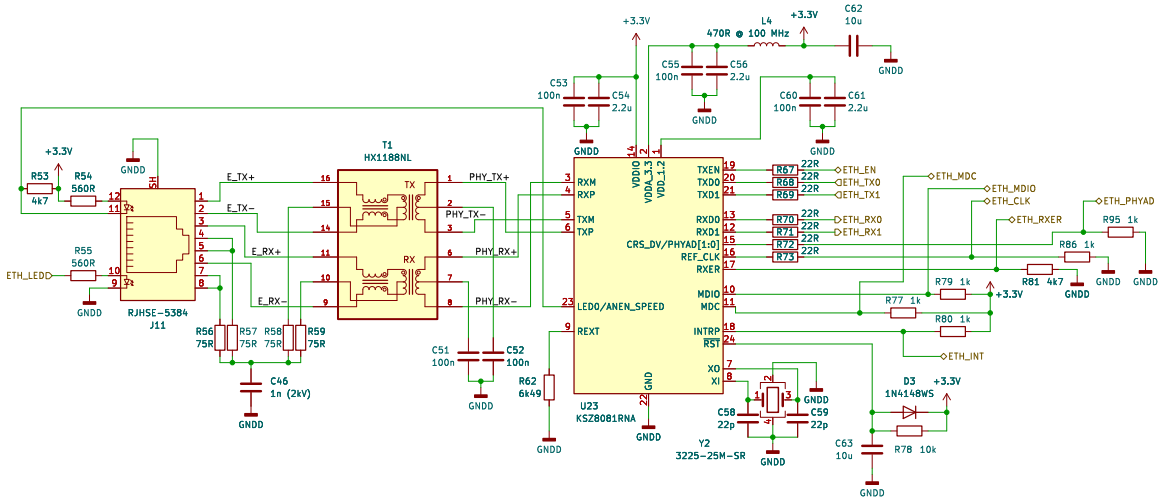


Figure 6.3: Connection of Ethernet components

An USB power switch AP2171W has been chosen. If the controller acts as a host, the EN pin is set high, enabling the power switch. If overcurrent is detected, an open collector FLG output is used to indicate error. If we want to power the board using an external USB supply, the current passes through the PMEG3010ER Schottky diode with a low voltage drop and bypasses the USB power switch. Some decoupling capacitors are connected for voltage filtering. The schematics of such solution is in figure 6.4.

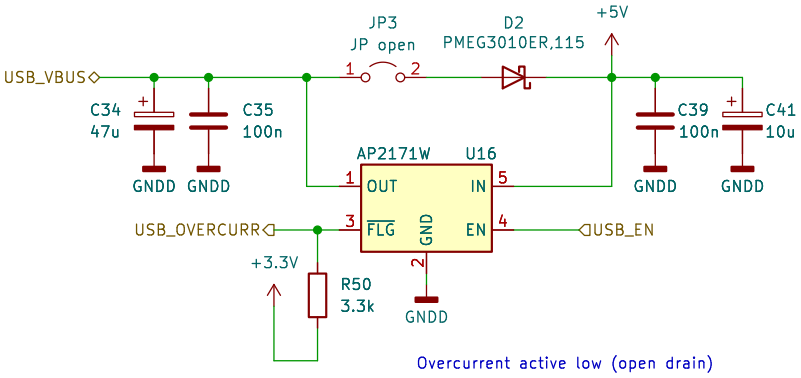


Figure 6.4: USB power components.

6.1.8 I²C

When the requirements were set, one I²C was needed. During routing, pins for a second I²C has been found which has been connected to the external connector. This way I²C can be routed to a sensor mounted on the lower board, for example a temperature sensor. The main peripheral has a Microchip 24AA64 EEPROM connected whose address is defined by bridged solder pads. The main I²C is routed on a 4 pin connector, alongside with 3.3 V and GND.

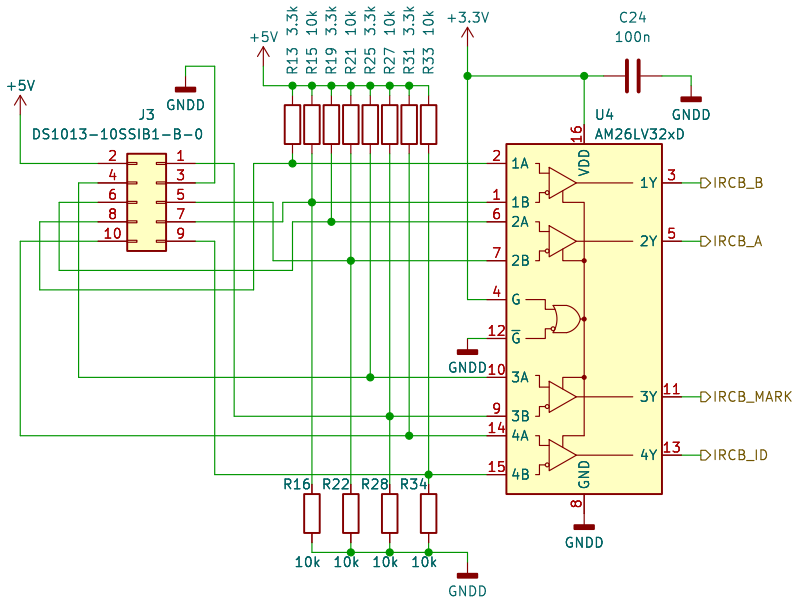


Figure 6.5: Routed optical encoder with HEDL series pinout

6.1.9 SPI

A main SPI is connected to a standard Molex 8 pin connector, alongside with 3 GPIO chip selects, 3.3 or 5 V and GND. Another SPI is routed onto the external connector, but some signals are in conflict with 3 extra ADC channels. The user must choose between one of these peripherals.

The main SPI incorporates a Winbond's W25Q32JV 32 Mbit NOR flash with a separate chip select, useful for data offloading. NOR flash memory typically exhibits higher reliability than NAND flash due to its simpler structure and lower susceptibility to errors, while NAND flash offers higher storage capacity. Since very high capacity is not needed NOR flash has been selected for the other better properties.

6.1.10 Feedback from the Motors

The board has 2 connectors for incremental encoders and 2 Hall input trios at disposal. The optical encoder connection is compatible with the HEDL encoder series, the connector being an IDC (insulation-piercing contact) 2 × 5 socket. HEDL series connector incorporates differential two optical phases, a differential zero cross index, grounding and 5 V power input too. These differential signals are decoded by the AM26LV32 receiver from Texas Instruments [25]. So called mark is used too, acting as an another user-defined helpful synchronizing signal. The schematic of the decoder circuit is shown in figure 6.5.

Since the voltage level for the inverted signals in the disconnected state is defined by voltage dividers, then an encoder with single ended outputs can be connected to the positive inputs. All the inputs have a pull up in case of open drain/collector outputs. The decoded signals from the receiver are connected to the SAMV7's TC0 and TC2 (*Timer/Counter*) peripheral.

Each Hall trio was connected to the same PIO (*Pararell Input/Output Controller*), to make readings faster in case of bare metal applications, because only one register reading is needed.

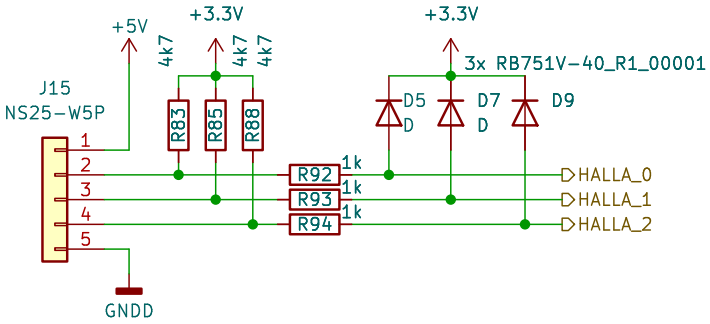


Figure 6.6: Open drain/collector output: the output is either 3.3 V or GND due to the pull up to 3.3 V. Push-pull: the voltage limitation is done by shorting the output to 3.3 V through diodes. The shorting current is limited by series resistors.

A standard 5 pin Molex connector (5 V power, GND and 3 Hall outputs) is used. In general, the outputs can be either push-pull or open/drain. A 5 V push-pull output could however potentially destroy the MCU, meaning the sensor's output voltage must be limited. The way Hall inputs are routed is shown in figure 6.6.

6.1.11 Analog Signal Routing and Grounding

Analog signals with sensing information need to be routed with special care. It is generally recommended to route analog signals far away from any fast digital signals. The figure 6.7 shows how some inputs to the AD converter are realised in the KiCad PCB editor. Due to a

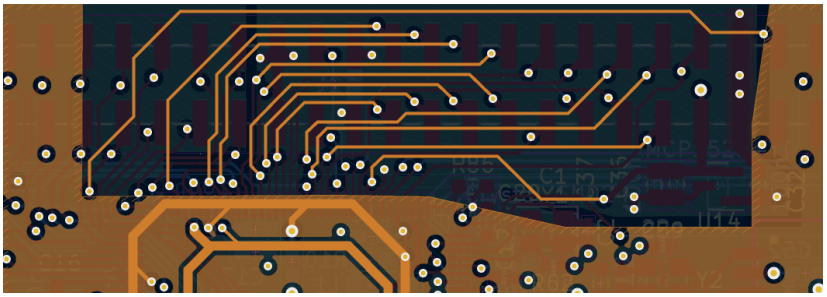


Figure 6.7: Analog signals in an empty space of power plane.

lot of space in the power plane a piece of the plane was decided to be cut away and instead be used to accomodate space for analog signals. There wasn't any space on the top or bottom plane due to the 2 × 32 header connectors.

The analog related electrical components must be connected to a stable analog grounding potential (an *analog ground*). Since all the measurement is done at the MCU board, it makes sense to realise the connection of the analog and digital ground next to the microcontroller on the MCU board.

6.1.12 Interconnection Pinout

The interconnection between the two boards is realised by a standard 2 × 32 header pin connector with a 2.54 mm pitch. The MCU board contains the pin connector heading down

towards the power board and a socket heading up. The pinouts of these connectors are the same, mirrored only, and are of SMD type, allowing us to route tracks in the inner layers.

Mentioned in 6.1.3, some pins must provide the 5 V source and the power digital ground to power the MCU board. In general the power board may contain 3.3 V powered circuits, so we need to route 3.3 V too, as well the 2.5 V reference. The lower board may also contain analog circuitry so we route the analog ground too.

All 15 PWM outputs must be routed alongside with 16 ADC channels for Motor A and Motor B. An extra I^2C, GPIOs and SPI or 3 extra ADC channels are routed too. The figure 6.8 shows the pinout of the interconnection. The analog signals are surrounded by a "ground shield" just for case.

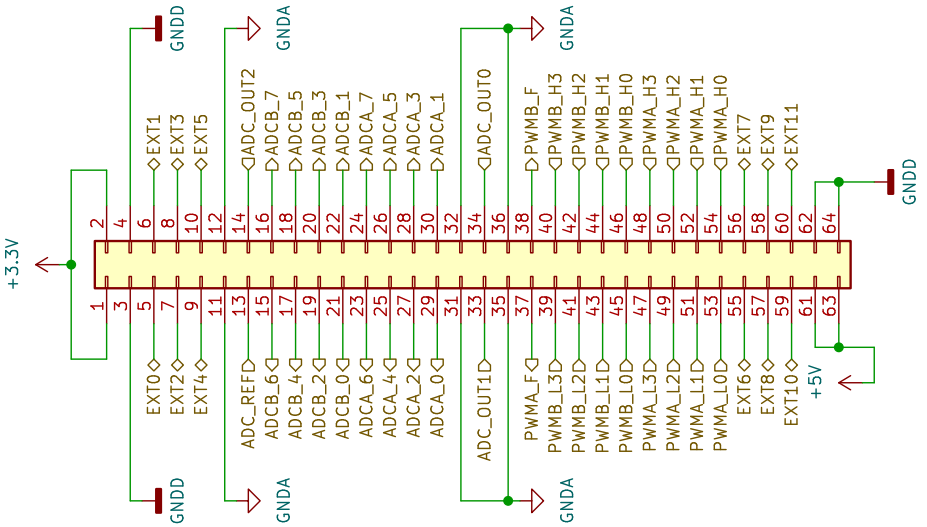


Figure 6.8: The interconnection pinout between the MCU and the power stage board.

6.1.13 PCB Realisation

Most of the traces are used for digital signals, so there is no explicit need for power considerations. The PCB was designed with 4 layers in mind, the outer layers are used for signals and the inner layers being used as a ground and power plane. The copper's thickness is not crucial in terms of power, but rather in terms of impedance matching of several differential signals like Ethernet and USB.

The positions of connected peripherals were chosen in such a way that it was as close as possible to the microcontroller's pins. Mounting M3 holes were put into each corner to secure a lower power board to this board. The board's shape is a rectangle with dimensions of 110 × 70 mm.

6.2 Power Stage Board

This section mentions the hardware implementation details of the first power board for the *SaMoCon* controller featuring IFX007 half bridge switches. In general, the power board should be interchangable, but should always contain a 5 V step down to power the higher

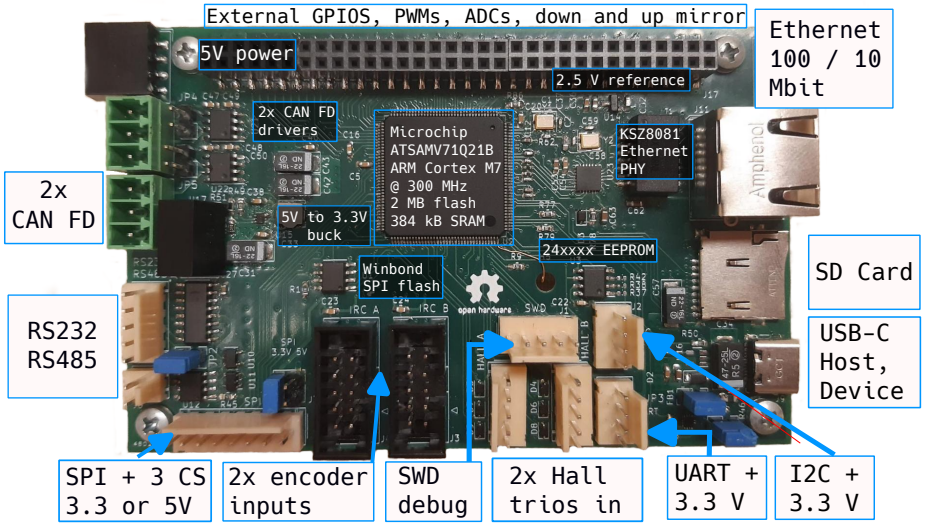


Figure 6.9: The depiction of the MCU board and its peripherals

control board. The voltage rail and the voltage rail for the control board are both 24 V. The continuous current should be 10 A at minimum, and peak 20 A should be possible too.

The board must perform the current measurement and amplification. These analog signals are then processed by the MCU board placed above.

6.2.1 Power Components

The power board features a 3 pin power connector, one pin being the common ground pin, the other ones are the positive voltage rails, one for the MCU board and the other for the motors. Each positive rail has an overvoltage protection diode and the rated voltage for both rails is 24 V.

The used buck regulator IC is AOZ1284 [1], again used in accordance with datasheet recommendations, shown in figure 6.10. We suppose the total power consumption of the power board should be around 0.5 A, the MCU board's USB power switch handles 1 A continous at max ([1] Recommended Operating Conditions) and we suppose the expanding board consumes another 1 A. The IC provides up to 4 A which is enough for our requirements.

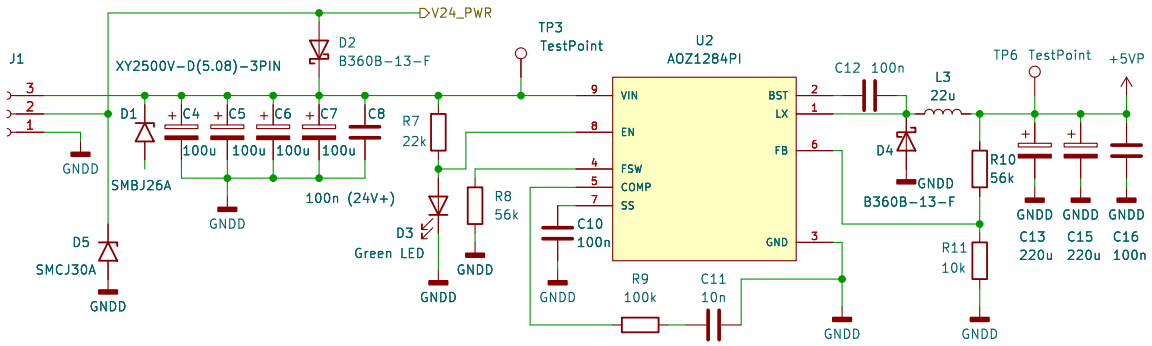


Figure 6.10: The 24 V to 5 V regulator circuit on the power board

The IFX007 switches and the buck regulator share the digital ground, while the integrated circuits used for analog sensing require a stable analog ground. The analog ground potential is provided by the upper MCU board (see 6.8) where it is also connected with the digital ground. All the analog circuitry powered from 5 V or 3.3 V (provided by the MCU board) is first filtered with a LC filter. The power rail for the IFX007 switches contains decoupling electrolytic capacitors.

6.2.2 Current Sensing and Fault Generation

Already mentioned in the section 5.3.2, we add a shunt resistor to ground from which the motor's winding current should be measured. The IFX007 IS output will be used for the measurement of average current. We must compute the values shown in the figure 5.6.

Shunt L Current

The reason we use a differential amplifier is due to different ground potentials, because the IS is connected to the analog ground, while the shunt is connected to the digital power ground. We set V_{Bias} in the middle of analog reference voltage, which is 1.25 V. This voltage is obtained by choosing 1:1 voltage divider with a buffer to lower the output impedance of the divider.

R_{s} was chosen to be 10 mΩ, because higher resistance values would lead to big power losses. The power loss of this at 10 A is

$$P = I^2 R = 1\,\text{W}, \tag{6.1}$$

a 2512 3 W resistor has been chosen. If $I = 10\,\text{A}$, then $V_{\text{RS}} = 0.1\,\text{V}$. To use the whole ADC bandwidth, we amplify the voltage by at least 10 ×. The output voltage formula for differential amplifier shown in figure 5.6 is (supposing there are only two values R_3 and R_4)

$$V_{\text{Out}} = \frac{R_4}{R_3} V_{\text{RS}} + V_{\text{Bias}}. \tag{6.2}$$

We select $R_3 = 1\,\text{k}\Omega$ and $R_4 = 10\,\text{k}\Omega$.

IS Current

Reffering to the figure 5.5, we need to set a threshold value indicating a fault. This threshold value should be equal to the point where $I = 20\,\text{A}$. Anything above this threshold is classified as a fault. First, we need to set the R_{IS} value. Referring to Table 11 in [8], the maximum analog sense current in fault condition is $I_{\text{IS(lim)}} = 6.1\,\text{mA}$. We need to make sure the voltage drop on R_{IS} won't be higher than 3.3 V to not damage the operational amplifier, powered from the 3.3 V rail. We choose a $R_{\text{IS}} = 510\,\Omega$ resistor from the E24 series. The maximum voltage drop across this resistor is

$$V_{\text{IS(lim)}} = 3.111\,\text{V} < 3.3\,\text{V}. \tag{6.3}$$

The differential current sense ratio in static on-condition is on average (Table 11 [8])

$$\frac{\text{d}I_{\text{L}}}{\text{d}I_{\text{S}}} = 19500. \tag{6.4}$$

From that we compute the threshold value for the comparator at $I = 20\,\mathrm{A}$:

$$V_{\mathrm{th}} = \frac{20}{19500} \cdot 510\,\mathrm{V} = 0.52\,\mathrm{V}. \tag{6.5}$$

Looking back in the datasheet, the differential ratio varies quite significantly and it also has quite a big current offset. This means we set V_{th} a bit higher, for example at $V_{\mathrm{th}} = 0.655\,\mathrm{V}$, provided by a voltage divider made out of $2.2\,\mathrm{k\Omega}$ and $6.2\,\mathrm{k\Omega}$ resistors. This compare value is again amplified by a buffer to lower the output impedance.

We need to set R_1 and R_2 resistors for the high side amplification too. Since all quantities in this current sensing method are quite uncertain, we choose $R_1 = 4.7\,\mathrm{k\Omega}$ and $R_2 = 1\,\mathrm{k\Omega}$, yielding a gain of 5.7 for a noninverting amplifier.

The last value that needs to be set is the value of C_{IS}. Initally, this value was supposed to be in the range of hundreds of pF, so only the high frequency noise would be filtered but with fast rising times, so current could be measured even during small PWM duty. Unfortunately, during testing I've spotted a high fault rate on the comparator's output due to fast voltage spikes on R_{IS}, which were not caused by high current but always after switching off the PWM signal. The reasons of this problem are unknown to me and will require further investigation. Currently $C_{\mathrm{IS}} = 22\,\mathrm{nF}$, giving

$$\frac{1}{\tau_{\mathrm{IS}}} = \frac{1}{R_{\mathrm{IS}} C_{\mathrm{IS}}} \doteq 89\,\mathrm{kHz}. \tag{6.6}$$

All these values mentioned in this section are supposed to be the first guess of such values. A place for capacitors in the operational amplifiers' feedbacks has been made on the PCB, so it can be tested whether these capacitors have a good impact on the noise.

6.2.3 Used Analog Components

The used operational amplifiers are MCP6022 from Microchip, a low offset and rail-to-rail operational amplifier [13]. The used comparator is LM2903 from Texas Instruments [26], a dual differential comparator with open-drain outputs. That means all comparators from 4 half bridges can be chained together. If at least one switch faults, the line goes low.

6.2.4 PCB realisation

The main design rule is to have the back side of the PCB flat with no components, because we want to route the outputs of IFX007s for the motor phases on this side of the board. The IFX007 output with an exposed solder pad is connected to a high area copper, on which a heatsink can be placed to cool down the switches. The output of IFX007 is routed on the back side with the help of high diameter vias.

The left part of the PCB is left for the buck regulator. The lower part is occupied by the connectors and power switches and the upper part features the operational amplifiers and comparators. Holes are needed for standoffs to hold the upper MCU board. The PCB is of a rectangular shape, the dimensions being 115×88 mm. Since the lower board contains the power connectors, one side must be a bit longer so the MCU board does not cover them.

The board is made out of 4 layers like the MCU board, but the current to the switches is routed on the upper side and the power plane. The ground is not routed on any outer layer, as it is only in the inner ground layer. The current sensing traces are routed in the inner

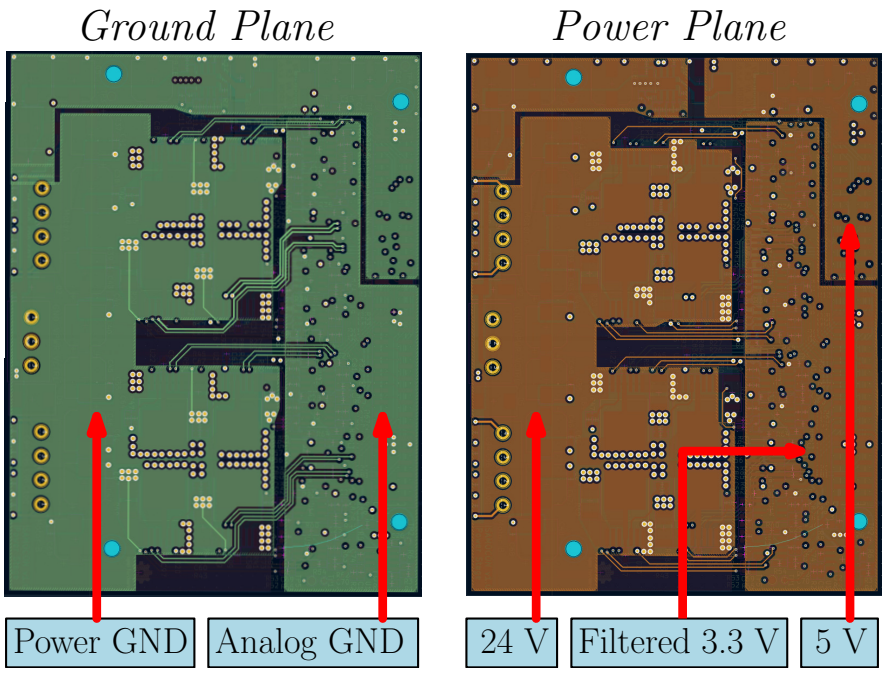


Figure 6.11: The inner layers of the power board (taken from KiCad)

layers, as it is not possible to route them on the outer layers. A cutout of copper is made near the current sensing traces to minimalize interference.

The upper part ground is disconnected from the power ground, as the analog ground connection to the whole ground is provided by the upper MCU board. The inner ground layer in the upper part is entirely the analog ground. Since the upper layer is mostly occupied by amplifiers and traces, the inner layers are mostly ground or filtered 3.3 or 5 V, some tracks need to be routed on the back side of the PCB. The figure 6.11 shows the filled copper areas of inner layers.

6.3 Design flaws and Needed Fixes

During the PCB design I made several mistakes which needed a quick fix during the electronics bringup. Here is the list of the most severe flaws which need to be fixed in the second design iteration:

- I had to draw the footprints of some KiCad footprints myself because of their absence in the KiCad library. However, I made the hole diameters of some parts small, so I needed to file the pins to reduce their diameters.
- I made a mistake when routing the KSZ8081 PHY. I routed the CRSDV pin to the SAMV7's GCRS pin because of the name similarity. However, I was supposed to route this pin to the GRXDV pin, when working in the RMII mode. The fix was to cut the PCB traces and reroute using a thin wire.
- The galvanically isolated communication interfaces on our board require a small switching power supply. For unknown reasons, I made a mistake when placing these supplies as

they have a mirrored pinout on the PCB. When soldering, I had to rotate them by 180 degrees.

- I did not galvanically isolate the Ethernet by connecting the ground before the magnetics to the common ground. This may have a bad impact on analog measurements.
- The left motor (Motor A) switches are driven by PWM1 and the currents are measured by the AFEC0 peripheral and vice versa for the right motor (Motor B) - PWM0 and AFEC1. We need the ADC measurements to be triggered by the PWM. However, it turned out the AFEC0 is synchronized only by PWM0 (analogous to AFEC1). The simplest fix was to reroute the traces on the power board using a thin wire, as rerouting the MCU board would be hard, because of MCU's close proximity to the connector.

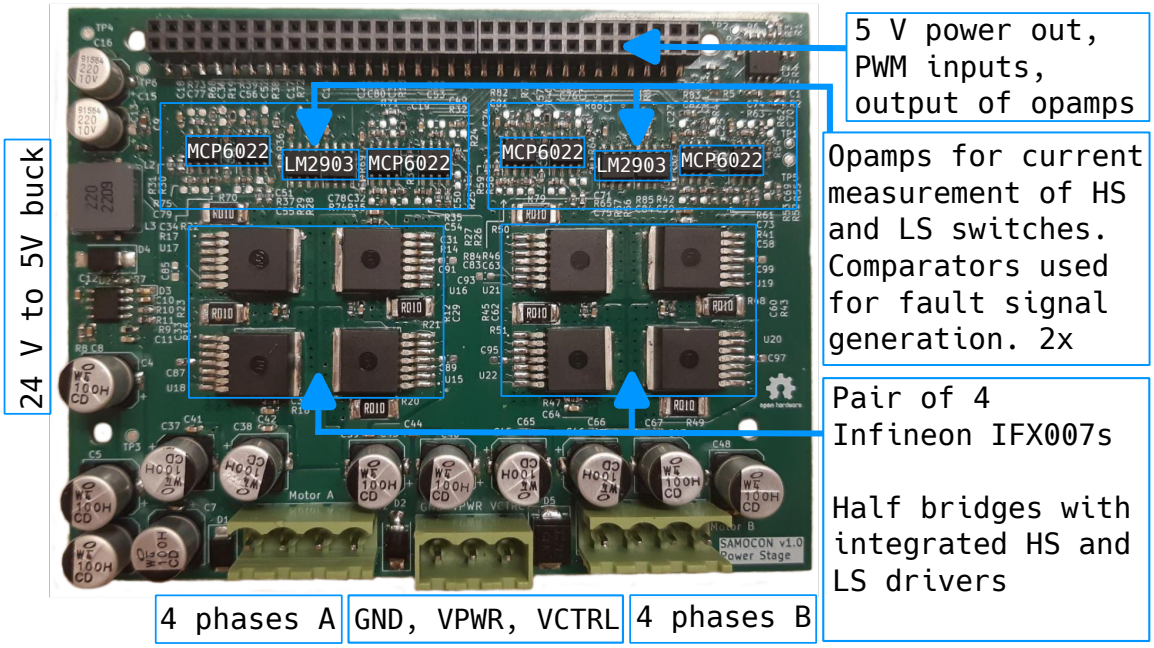


Figure 6.12: The power board showcase.

Figure 6.13: The MCU and power stage boards forming the motion controller.

Chapter 7

NuttX Adaptation

In this chapter, the configuration of the NuttX setup is described. During the tuning of pysimCoder applications, some features in the NuttX SAMV7 drivers were implemented to make everything work properly. The contributions to NuttX project hosted on GitHub are also documented here. The overall NuttX configuration can be viewed on my GitHub NuttX fork [1] on the `samocon-branch` branch.

7.1 NuttX Bringup

The first step was to configure NuttX to make the most needed peripherals work. Namely, for basic functionality and pysimCoder experiments, these peripherals needed to be configured:

- TTL console on UART3 peripheral for console interfacing,
- Ethernet on GMAC peripheral for UDP/IP and TCP/IP stacks,
- PWM0 and PWM1 peripherals for motor control,
- AD converters on AFEC0 and AFEC1 peripheral for current measurement,
- TC0 and TC2 peripherals (Timer/Counter) quadrature decoders for position feedback,
- Parallel Input/Output controller for pinmuxes and GPIOs (Hall inputs, interrupt requests, IFX007 INH enable outputs).

I managed to configure the peripherals listed below which aren't crucial for our experiments but are important in terms of connectivity for the future usage:

- I^2C with communicating with the 24xxxx EEPROM memory,
- MCAN0 and MCAN1 peripherals,
- USART2 peripheral connected to RS232 and RS485 converters.

Unfortunately, in the time of writing this thesis, I had not managed to configure the USBHS peripheral for the USB communication. Also I didn't have time to make the HSMCI peripheral and the SPI peripheral to work.

[1] `https://github.com/zdebanos/nuttx`

7.1.1 Custom BSP

To build NuttX, a custom BSP (*board support package*) was needed. NuttX has the support to have the BSP related sources in a different directory, but I created a new `samocon` directory in the `boards/arm/samv7` directory.

The `samv71-xult` BSP for the *SAMV71 Xplained Ultra Evaluation Kit* was used as the baseline, located in the `boards/arm/samv7/samv71-xult` directory. However, the Microchip's evaluation board contains many components which are not present on my board. Also, the pinouts of some peripherals are different. The first step was to edit the two files containing defines, including the pinouts. From now on, I will refer to files relative to the `boards/arm/samv7/samocon` directory if not specified otherwise.

Firstly, I renamed the `src/samv71-xult.h` to `src/samocon.h` to fit into my BSP and I also renamed the `#include` statements in all `src` files depending on this header file. This file contains all pin defines that are needed only by the board specific logic, for example Ethernet related GPIOs, shown on figure 7.1.

```
#define GPIO_EMAC0_INT (GPIO_INPUT | GPIO_CFG_PULLUP | GPIO_CFG_DEGLITCH |\
                        GPIO_INT_FALLING | GPIO_PORT_PIOC | GPIO_PIN25)
#define IRQ_EMAC0_INT   SAM_IRQ_PC25
#define GPIO_EMAC0_LINK_LED (GPIO_OUTPUT | GPIO_PORT_PIOB | GPIO_PIN13)
```

Figure 7.1: Ethernet related GPIOs and GPIO PHY's IRQ

The figure 7.1 also shows how the SAMV7's pin behaviour is configured with the help of bit masking. The defines of the GPIO flags can be found in this NuttX header file [2]. If a pin is not desired to be configured as a GPIO but rather as the input or an output of a different peripheral, `GPIO_{ALTERNATE, PERIPHA, PERIPHB, PERIPHC, PERIPHD}` must be used. Often the peripheral outputs needn't to be defined since the defines are already in this header file [3]. The KSZ8081 PHY has an interrupt pin indicating activity which is also configured in this file. These Ethernet related defines are used by the `src/sam_ethernet.c` functions, taken from the `samv71-xult` BSP.

The second file which needed to be edited is the `include/board.h` header. Files in the `include` directory are used by the files in the `arch` dependant drivers for SAMV7. Here the `GPIO_UART3_TXD` symbol was defined to `GPIO_UART3_TXD_2` to make the basic TTL console working. Other than that, MCAN1 peripheral pins are defined here and also all the used GPIOs pins for IRC marks and Hall sensor input GPIOs. I chose to define these GPIOs pin here because in the future the `include/board.h` file can be imported to a custom application. Last but not least, pins driving the IFX007 halfbridge switches are also configured. While the H pins are configured as pins used by the PWM peripherals (except for PWM1 CH2, as mentioned earlier in 6.1.2), the L pins are configured as GPIOs by default to drive the INH pin if `CONFIG_SAMV7_PWMx_CHy_COMP` is not selected.

[2] `arch/arm/src/samv7/sam_gpio.h`
[3] `arch/arm/src/samv7/hardware/samv71_pinmap.h`

7.2 Project Configuration

All the BSP directories include a `configs` directory, as well the `samv71-xult` BSP from which I started using most basic `configs/nsh/defconfig` config copied to my `samocon` BSP. I could have started with a more complex configuration but I have decided not to do so, because from configuring step by step I was able to catch all misconfigurations.

The project is configured by the command mentioned in 2.3. In the time of writing this thesis, a `configs/test` is the currently used development configuration. After configuring, running `make menuconfig` loads the TUI shown on figure 7.2.

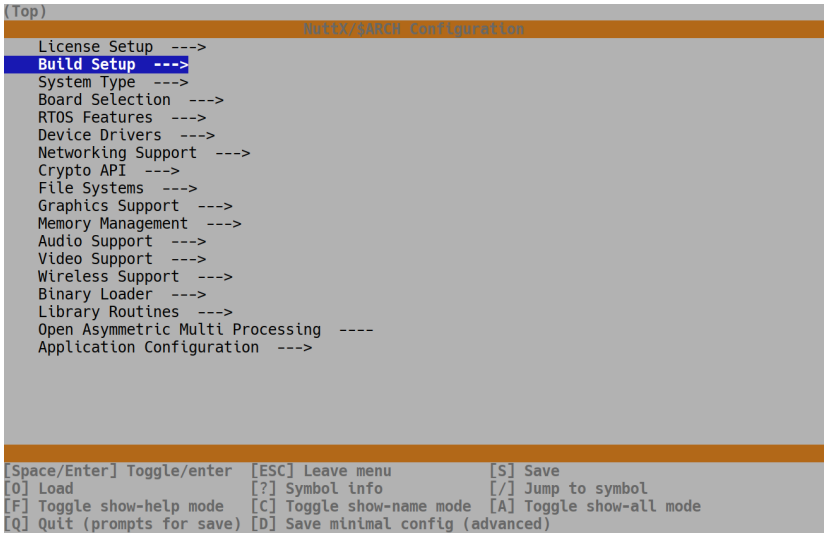


Figure 7.2: The start menu of configuration

TTL console was the first peripheral that needed to be configured. To set up the UART3 peripheral and the upperhalf console driver following options must be selected:

```
CONFIG_SAMV7_UART3=y,
CONFIG_SERIAL=y,
CONFIG_STANDARD_SERIAL=y,
CONFIG_UART3_SERIAL_CONSOLE=y.
```

By default, the ATSAMV71Q21B microcontroller boots from bootloader located in the internal ROM, however we want NuttX to boot from flash memory. To do so, a *General-purpose Non volatile Memory Bit 1* must be set to 1 ([14], table 11-4). The process of setting this bit is described in 7.3. NuttX memory ranges must be configured too:

```
CONFIG_BOOT_RUNFROMFLASH=y,
CONFIG_RAM_START=0x20400000,
CONFIG_RAM_SIZE=393216.
```

This controller features a lot of peripherals and only Ethernet, PWM and ADCs will be discussed, since these are the most crucial peripherals for our experiments.

7.2.1 Ethernet and IP Configuration

The NuttX EMAC drivers must be configured to communicate with the KSZ8081 PHY. It may be useful to enable the `phytool` application. This tool can be used to get the PHY's register readings from the MDIO interface Also it can be used to check if the PHY is alive:

`CONFIG_SYSTEM_MDIO=y.`

To configure EMAC lowerhalf driver, select these options:

```
CONFIG_SAMV7_EMAC=y, CONFIG_SAMV7_EMAC0=y,
CONFIG_SAMV7_EMAC0_PHYADDR=0,
CONFIG_SAMV7_EMAC0_RMII=y,
CONFIG_SAMV7_EMAC0_AUTONEG=y,
CONFIG_SAMV7_EMAC0_PHYSR=30,
CONFIG_SAMV7_EMAC0_PHYSR_ALTCONFIG=y,
CONFIG_SAMV7_EMAC0_PHYSR_ALTMODE=0x0007,
CONFIG_SAMV7_EMAC0_PHYSR_10HD=0x01,
CONFIG_SAMV7_EMAC0_PHYSR_100HD=0x02,
CONFIG_SAMV7_EMAC0_PHYSR_10FD=0x05,
CONFIG_SAMV7_EMAC0_PHYSR_100FD=0x06.
```

The PHY address is determined by PHYAD[2:0] pins ([12], figure 2-1 and table 2-1). In our case the resistor values are set that PHYAD[2:0] = 000. Referring to the table 4-2 in the datasheet [12], the autonegotiation status register address is $0x1E = 30_{10}$ (PHY Control 1). Since the mode indication is in the last three bits, the PHY Mode Mask is set to 0x7. The half and full duplex values are the values in the *Operation Mode Indication field*.

As the PHY has the interrupt pin connected to ATSAMV7's PC25 pin (defined in 7.1), the interrupts on GPIOC pins must be enabled:

```
CONFIG_SAMV7_GPIO_IRQ=y,
CONFIG_SAMV7_GPIOC_IRQ=y.
```

The Ethernet is configured, higher layers of the IP stack can be configured:

```
CONFIG_NET=y, CONFIG_NETDB_DNSCLIENT=y, CONFIG_NETDEV_PHY_IOCTL=y,
CONFIG_NETDEV_STATISTICS=y, CONFIG_NETINIT_DHCPC=y, CONFIG_NETINIT_DNS=y,
CONFIG_NETUTILS_DHCPC_HOST_NAME="samocon-nuttx", CONFIG_NETUTILS_DISCOVER=y,
CONFIG_NETUTILS_TELNETC=y, CONFIG_NETUTILS_TELNETD=y,
CONFIG_NET_ALLOC_DEVIF_CALLBACKS=1, CONFIG_NET_ARP_SEND=y,
CONFIG_NET_BROADCAST=y, CONFIG_NET_ETH_PKTSIZE=1518, CONFIG_NET_ICMP=y,
CONFIG_NET_ICMP_ALLOC_CONNS=3, CONFIG_NET_ICMP_NPOLLWAITERS=2,
CONFIG_NET_ICMP_SOCKET=y, CONFIG_NET_IPFRAG=y, CONFIG_NET_LOCAL=y,
CONFIG_NET_LOCAL_SCM=y, CONFIG_NET_ROUTE=y, CONFIG_NET_STATISTICS=y,
CONFIG_NET_TCP=y, CONFIG_NET_TCPBACKLOG=y,CONFIG_NET_TCPPROTO_OPTIONS=y,
CONFIG_NET_TCP_NPOLLWAITERS=3, CONFIG_NET_TCP_WRITE_BUFFERS=y,
CONFIG_NET_UDP=y, CONFIG_SYSTEM_PING=y.
```

These options turn on the DNS client, the DHCP client and UDP/IP and TCP/IP stacks. Also, the `ping` command is turned on, along with `telnet`. In that case, TTL console is not needed.

7.2.2 PWM and ADC Configuration

The PWM peripheral has undergone changes (described in 7.4) so this config works only with the newest mainline NuttX. For our controller, we need to enable all the H channels except for the CH2 where the only H channel pin is not available, as pointed out in 6.1.2. The only way to make this channel to work is to enable the L pin only and configure it later in the application to behave as an inverted L output. All channels must be configured as synchronous, i.e. all the channels share the same timebase and start at the same time.

The AFEC0 and AFEC1 peripherals must be configured too. In case of motor control, we want the AD converters to be triggered by the PWM signal. When measuring the current passing through the high side transistor, the measurement must be triggered in the beginning of the period. In case of the low side transistor, we want the measurement as late as possible before new PWM period start. AFEC0 is triggered by PWM0 and AFEC1 is triggered by PWM1.

Let's mention the configuration options for PWM1. We enable the PWM1 first, enable all the used 4 channels, make every channel synchronous (the CH0 provides timebase for every other channel) and enable only the L output for CH2. Turn on the EVENT0 output for AFEC1 trigger and set the TRIG0 to 92 %.

```
CONFIG_SAMV7_PWM1=y
CONFIG_SAMV7_PWM1_SYNC=y
CONFIG_SAMV7_PWM1_CH0=y
CONFIG_SAMV7_PWM1_CH1=y
CONFIG_SAMV7_PWM1_CH1_SYNC=y
CONFIG_SAMV7_PWM1_CH2=y
CONFIG_SAMV7_PWM1_CH2_LONLY=y
CONFIG_SAMV7_PWM1_CH2_SYNC=y
CONFIG_SAMV7_PWM1_CH3=y
CONFIG_SAMV7_PWM1_CH3_SYNC=y
CONFIG_SAMV7_PWM1_EVENT0=y
CONFIG_SAMV7_PWM1_SYNC=y
CONFIG_SAMV7_PWM1_TRIG0=92
```

The TRIG0 depends on the speed of the AD converter conversion. Reffering to section in 51.6.1 in [14], the conversion is given by three time values which depend on the clocking period t_{AFE}: the startup time (t_{start}), the conversion time (t_{conv}), which is always $23t_{\mathrm{AFE}}$, the tracking time (t_{track}) and the transport time (t_{trans}). In the AFEC_MR description (51.7.2) the t_{trans} is selected by the TRANSFER[1:0] bits and by default t_{trans} is equal to $9t_{\mathrm{AFE}}$. Also the tracking time is always $15t_{\mathrm{AFE}}$ (specified by TRACKTIM[3:0]). Since $t_{\mathrm{track}} < t_{\mathrm{conv}}$, we can refer to Figure 51-3 to calculate the time needed to convert N channels:

$$t_{\mathrm{total}} = t_{\mathrm{start}} + N(t_{\mathrm{conv}} + t_{\mathrm{trans}}). \tag{7.1}$$

Looking in the source code of NuttX AFEC drivers [4], the STARTUP[3:0] field in AFEC_MR is set that $t_{\mathrm{start}} = 64t_{\mathrm{AFE}}$ and the prescaler (PRESCAL[7:0]) is set to 2. A simple rearranging

[4] `static void afec_reset(struct adc_dev_s *dev)` in `arch/arm/srv/samv7/sam_afec.c`

gives ($f_{\text{periph}} = 150\,\text{MHz}$):

$$f_{\text{AFE}} = \frac{f_{\text{periph}}}{\text{PRESCAL} + 1} = 50\,\text{MHz}. \tag{7.2}$$

Assume measuring $N = 4$ L channels. We can now compute the total time of conversion:

$$t_{\text{total}} = 64t_{\text{AFE}} + 4 \cdot 31t_{\text{AFE}} = 188t_{\text{AFE}} = 3.76\,\mu\text{s}. \tag{7.3}$$

This means the conversion must be launched at least by t_{total} plus a little margin before the start of another PWM period ($t_{\text{PWM}} = 1/20\,\text{kHz}$):

$$\text{TRIG0} = \frac{t_{\text{PWM}} - t_{\text{total}}}{t_{\text{PWM}}} \doteq 0.92 \tag{7.4}$$

Even though we have 11 channels at disposal in total, we currently focus on measuring only the 4 L channels since we do not care about the measurements on the other channels. Measuring as late as possible allows us to use as much duty cycle bandwith as possible.

Following options must be turned on to configure AFEC1, the options are analogous to AFEC0. We must enable the trigger from the PWM1 peripheral and the DMA transmission of completed conversions too:

```
CONFIG_SAMV7_AFEC=y
CONFIG_SAMV7_AFEC_PWMTRIG=y
CONFIG_SAMV7_AFEC_DMA=y
CONFIG_SAMV7_AFEC_DMASAMPLES=2
CONFIG_SAMV7_AFEC1=y
CONFIG_SAMV7_AFEC1_RES=0
CONFIG_SAMV7_AFEC1_PWMTRIG=y
CONFIG_SAMV7_AFEC1_PWMEVENT=0.
```

The number of DMA samples could be bigger, but for the purpose of not breaking things up, I kept this value for now.

7.2.3 Initialization and Registration of Peripherals

The peripherals are interfaced using system calls on file descriptors. The BSP logic must first register the peripherals as `/dev` files. This is, most of the time, done using initialization functions that come from the lowerhalf part of the driver. The lowerhalf driver usually returns a pointer to an upperhalf related driver struct containing a pointer to a private lowerhalf structure of the related driver. The returned upperhalf pointer is then passed to a registration function, creating the desired file.

Examples of file registrations are in the `samocon` BSP directory. Most of the logic is taken from the `samv71-xult`.

7.2.4 Tickless Mode

By default, a NuttX configuration uses a periodic timer interrupt that drives all system timing. The timer is provided by architecture-specific code that calls into NuttX at a rate

controlled by CONFIG_USEC_PER_TICK. The default value of CONFIG_USEC_PER_TICK is 10000 microseconds which corresponds to a timer interrupt rate of 100 Hz ([28], Tickless OS).

While this is reliable way to clock the scheduler, the problem is the time resolution. This way, the smallest timeslice NuttX can measure is CONFIG_USEC_PER_TICK. While this configuration option may be lowered, it may lead to overhead if high frequency is selected, because each time the scheduler must be polled if anything needs to be done.

The better way is to configure the system in a tickless mode, where two timers are used. The first timer is a oneshot interval timer. It becomes event driven instead of polled: The default system timer is a polled design. On each interrupt, the NuttX logic checks if it needs to do anything and, if so, it does it. Using an interval timer, one can anticipate when the next interesting OS event will occur program the interval time and wait for it to fire. When the interval time fires, then the scheduled activity is performed ([28], Tickless OS).

The second timer is a freerunning timer where each counter increment corresponds to the smallest measurable timeslice. If the timeslice is small, high precision delays can be made, useful especially in fast periodic sampling. It is possible to combine the freerunning and oneshot timer into one timer by selecting CONFIG_SCHED_TICKLESS_ALARM.

7.3 Flashing

To flash the microcontroller, the OpenOCD application is used to flash and debug the processor. When flashing a new ATSAMV72Q21B chip, the GPNVM1 bit must be set to 1, so the microcontroller can boot from the flash memory instead of ROM, as shown in the figure 7.3.

```
openocd -f interface/stlink.cfg -f target/atsamv.cfg \
        -c 'set CHIPNAME atsamv71q21' \
        -c init -c targets \
        -c 'reset halt' \
        -c 'atsamv gpnvm set 1' \
        -c 'reset run' -c shutdown
```

Figure 7.3: Setting a GPNVM1 bit using STLink and OpenOCD.

Afterwards, the flashing of the microcontroller can be done by usin an ARM compatible SWD tool, like STLink. Command in figure 7.4 shows how to run OpenOCD with STLink to flash the microcontroller on the SaMoCon board.

7.4 Contributions to Mainline

Several things needed to be implemented in the quadrature decoder lowerhalf driver of the Timer/Counter peripheral in [5], introducing a better way to handle index counting and range extension to 32bit of the 16bit internal counter.

Also, changes needed to be made to the PWM lowerhalf driver in [6], the biggest change being the feature of synchronized channels.

[5] arch/arm/src/samv7/sam_qencoder.c
[6] arch/arm/src/samv7/sam_pwm.c

```
samocon-flash.sh file:
| #!/bin/sh
| if [ $# -ne 1 ]; then
|   echo "Usage: samocon-flash.sh [bin name]"
|   exit 1
| fi
|
| prog_name=$1
| openocd -f interface/stlink.cfg -f target/atsamv.cfg \
|         -c "set CHIPNAME atsamv71q21; reset_config none separate;
|             program $prog_name 0x400000; reset; exit"

$ samocon-flash.sh nuttx.bin
```

Figure 7.4: Flashing using STLink and OpenOCD.

7.4.1 Quadrature Decoder Driver

The internal counter of the Timer/Counter peripheral is decremented or incremented according to the incoming A and B encoder signals. The reading from the internal counter returns the number of optical signals. The block scheme of quadrature decoder logic is shown in the figure 7.5.

However, the default Timer/Counter channels are only 16bit. The first task is to extend the value read from 16bit to an internal 32bit software variable because the 16bit range is not enough for fast rotating motors. Another problem arises from the datasheet recommended quadrature decoder settings which causes the CH0 to be cleared by an incoming index signal, while incrementing CH1 by one. An interrupt can be triggered when the index pulse comes.

During the motor's homing procedure, we need to precisely catch the index signal. The best solution would be to perform the counting of optical signals and capture the channel's counter when the index pulse comes. However, the default settings make the implementation of this method not straightforward and getting this into NuttX would require some hacks. It makes sense to configure the TC peripheral such that the counter's value is captured when an index pulse comes.

Fortunately, the TC peripheral can be set to capture the value on a incoming index signal. The counter is zeroed due to the ABETRG bit and the ETRGEDG bits, which are recommended by the datasheet, in the TC_CMR register ([14], 49.7.2). This way, if an edge appears on the TC's input, the counter is zeroed. Our way is to not set the previously mentioned bits but use the capture into the TC_RA and TC_RB registers when an edge appears. Two capture registers must be used due to their exclusive access. The loading can be configured by setting the LDRA and LDRB bits in the TC_CMR register. The loading of the TC_RA or TC_RB register is indicated by the LDRAS and LDRBS bits respectively in the TC_SRx register.

Since the capture is done on hardware, the polling of TC_SRx is sufficient. If a capture flag is set, the reading of TC_RA or TC_RB is made, containing the counter's value when the index signal was caught. With each read, the counter value is read and an extension to 32bits is made.

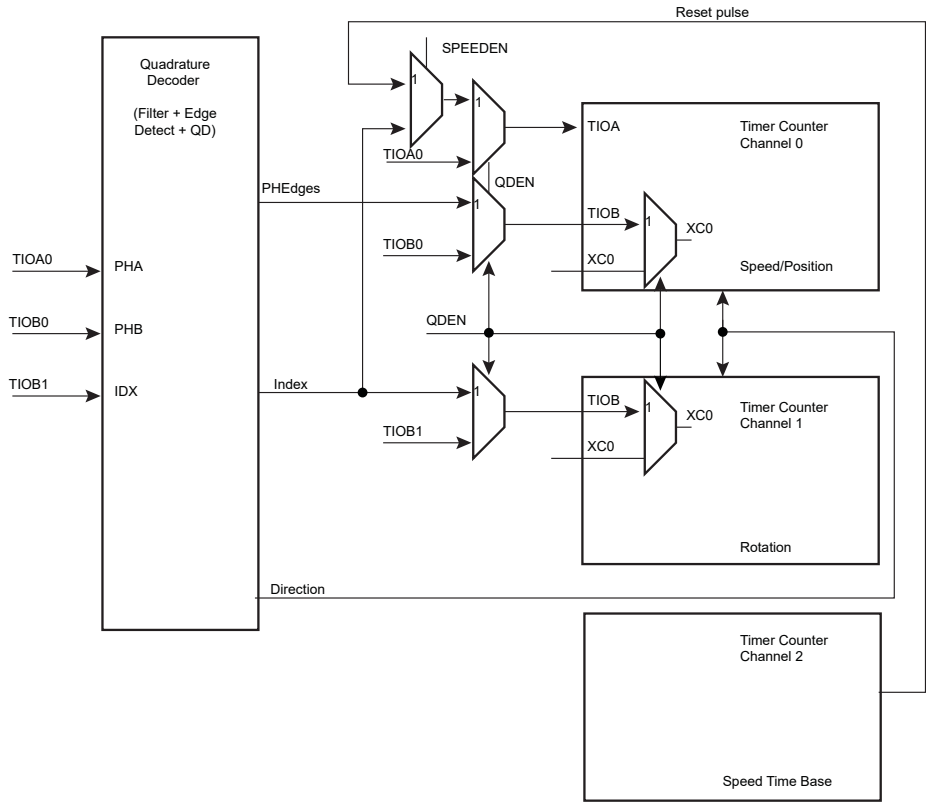


Figure 7.5: Timer/Counter quadrature decoder logic (Figure 49-17 in [14])

During the development of PiKRON motor controller based on the Teensy-4.1 board [22], a `QEIOC_GETINDEX ioctl` call has been introduced alongside with a `qe_index_s` struct to perform this index capturing task. The SAMV7's driver now supports this call, serving as a getter for the actual position, the last index value alongside of the number of caught indexes.

7.4.2 PWM Driver Changes

PWM Polarity

During the testing of pysimCoder applications, I've spotted an unwanted logical high default value when the PWM output was off. The reason was the SAMV7's DPOLI (disabled channel's polarity) bit in the PWM_CMRx register. With that, a decision was made to include `uint8_t dcpol` attribute in the `pwm_chan_s` and `pwm_info_s` structs (located in [7]). This attribute defines the default state of the PWM channel when it is off. With that, the DPOLI bit setting was implemented into the lowerhalf SAMV7 PWM driver.

Only L Output

This commit allows the activation of only the channel's L output. The selection is done in Kconfig and the selection then configures the SAMV7's pinmux such that the L output is

[7] `include/nuttx/timers/pwm.h`

turned on, while the H output is not.

Channel Synchronization

All the channels share the same timebase from the channel 0. This change allows the user to configure all the channels synchronized with the CH0 in Kconfig. The driver needed to be updated because the synchronized channels require additional register handling. If the xth bit in PWM_SCM register is defined ([14], 50.7.9), the channel x is synchronized with channel's 0 timebase. This activation is done in the `pwm_setup` function [8].

When new duty of a synchronous channel is desired, the UPDULOCK bit in the PWM_SCUC ([14], 50.7.11) register must be set. The update of the duty is applied after the new channel 0 PWM period. The driver implements the recommended *Method 1* defined in the section 50.6.2.9 [14].

List of All PWM Driver Commits

- Adding the `dcpol` attribute [9],
- Adjust SAMV7's PWM driver to the `dcpol` attribute [10],
- Enable only L outputs [11],
- Synchronous channels [12].

[8] located in `arch/arm/src/samv7/sam_pwm.c`
[9] `https://github.com/apache/nuttx/commit/21de46a4d12087fda42e982dd9745fe926376b31`
[10] `https://github.com/apache/nuttx/commit/bf3a5bb4cbbd760112216ca87f79a0577cd29262`
[11] `https://github.com/apache/nuttx/commit/88fa598ea2fb965b35168bff409cce78b950ad69`
[12] `https://github.com/apache/nuttx/commit/297b3b0209b1f6cc0f35ec0f95380d4747b18292`

Chapter 8

Applications

In this chapter applications using pysimCoder and the Silicon Heaven infrastructure for the parameter tuning are presented. Examples of PMSM control are presented, as well as the open loop control of a piezoactuator bending element. All the provided schematics should be compatible with the mainline pysimCoder. However, the current motion controller is only capable of achieving sampling rates below 1 kHz, probably due to the NuttX scheduler's limitations, making it unusable for the control of fast-moving systems or the *field oriented control*.

The concept of this controller is not to implement state of the art control methods (like predictive control) but rather an extensible prototyping platform. However, even for such applications, the sampling rate should be much higher.

Probably, a high priority thread running the control loop might be blocked by a long lasting interrupt or some kinds of synchronization locks during communication. Unfortunately, there was no time to find out the causes in NuttX responsible for the sampling issues. Luckily, there are a few ways to possibly overcome these issues in the future:

- Using ITM (*Instrumented Trace Macrocell*) as part of the ARM Cortex-M core. This allows us to sample the program counter, read it from STLink and then associate the program counter with compiled program's debug symbols. From that, a histogram of function durations can be drawn (an online guide can be found here [3]).
- Using NuttX internal profiling tools [30]. Events are collected in the NuttX kernel into a trace buffer which is then sent to a PC. Eclipse Trace Compass tool is then used to analyze the events.
- Placing the control loop in a high priority periodic interrupt, triggered by a timer for example. However, this method requires overriding NuttX interrupt handling [29].

8.1 PMSM Control with pysimCoder

Several schematics are presented in this section, demonstrating simple PMSM control with pysimCoder, as well as pysimCoder's great logging capabilities, despite the poor sampling rate. All the data presented in this section was logged to my computer by using UDP/IP. These examples have already been tried out by Michal Lenc in his thesis [9] but were recreated to show off the functionality of *SaMoCon*. All these schematics can be found here [1]. The

[1] `https://gitlab.fel.cvut.cz/otrees/motion/samocon/-/tree/main/control`

tested motor with the *SaMoCon* controller is shown in figure 8.1. The tested motor was not connected to any load and all shown examples are only for demonstrative purposes, as shown in figure 8.1.

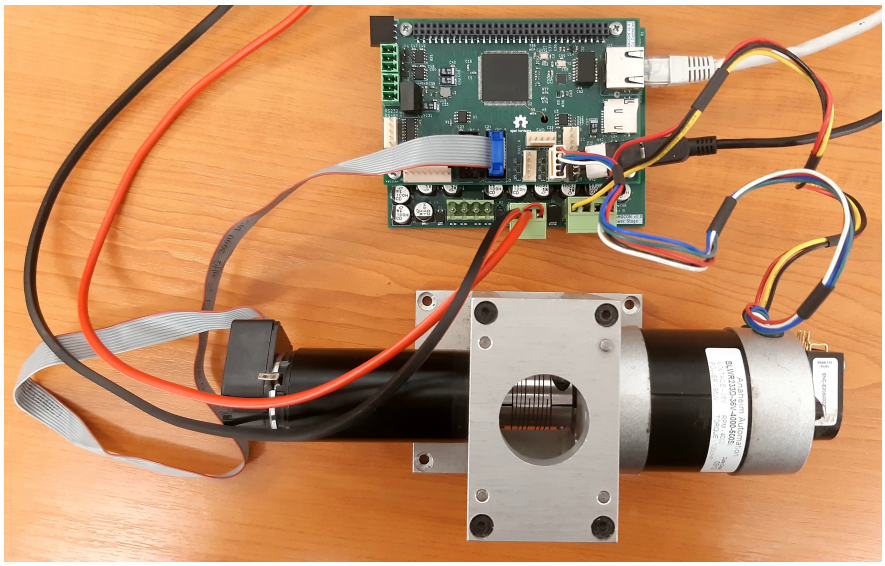


Figure 8.1: The testing setup with *SaMoCon* and a PMS motor on the right. A DC motor on the left is supposed to be used as a brake which was not used. DC motor's IRC was used because of a compatible connector.

Before any current measurements took place, I needed to find out the mapping from of an ADC word to current $I = I(\text{ADC})$. For that, I measured multiple values and then I used linear regression using my Python script. The measurement is done on the L shunts. Also the *NuttX/Encoder* block must be set to have 3 outputs. This way, a `QEIOC_GETINDEX` `ioctl` is performed, returning `qe_index_s` struct fields, as was discussed in 7.4.1.

Also, due to a high *Switch ON delay time* of IFX007, we weren't capable of operating below a certain threshold, effectively being limited by switches' deadzones. Due to this, we decided to operate near the "50% duty" point, eliminating the deadzones. However, this causes higher power consumption and also accounts for higher EMI.

8.1.1 Electrical Angle Calibration

To control PMSM effectively, we must estimate the electrical angle from the motor's encoder. Suppose C is the number of pulse counts, C_I is the counter value when the last index was hit and C_Turn is the number of pulses per one electical turn. The angle can be then estimated by this formula:

$$\varphi_\text{Est} = \frac{C - C_\text{I}}{C_\text{Turn}} \cdot 2\pi + \varphi_\text{Offset}. \tag{8.1}$$

Since we do not know φ_Offset (the angle between the start of the electrical angle and the index), the following method is used to heuristically determine the offset:

1. Set $d > 0$ and $q \leftarrow 0$, $\varphi_\text{Offset} \leftarrow \text{Randval} \in [0, 2\pi]$. That way the PMSM steadily follows the generated magnetic field. Generate $\varphi_\text{Ref} = \omega t$ (set ω to be small, for example $\omega = 1\,\text{rad}$). Generate a, b, c actions using the inverse Park and inverse Clarke transformations.

2. Visualize φ_{Est} and φ_{Ref}.

3. Adjust φ_{Offset} such that φ_{Est} and φ_{Ref} overlap as best as they can. Save the value φ_{Offset} for later experiments.

The equation 8.1 is used by the *Math/PMSM Align* block, created by Michal Lenc (showcased in [9], 7.3.2). The C source file of the block can be found here [2]. The figure 8.2 shows the UDP/IP output sent to my computer.

If no index pulse still has not been hit yet, the estimation is done using Hall sensors using the *Math/PMSM Align* block. For that, a *Math/Hall To 6 Sectors* block is used. Again, the Hall angle estimation must be done too, as there may be an offset between the start of the electrical angle and the first sector. This offset is again set in the *Math/PMSM Align* block.

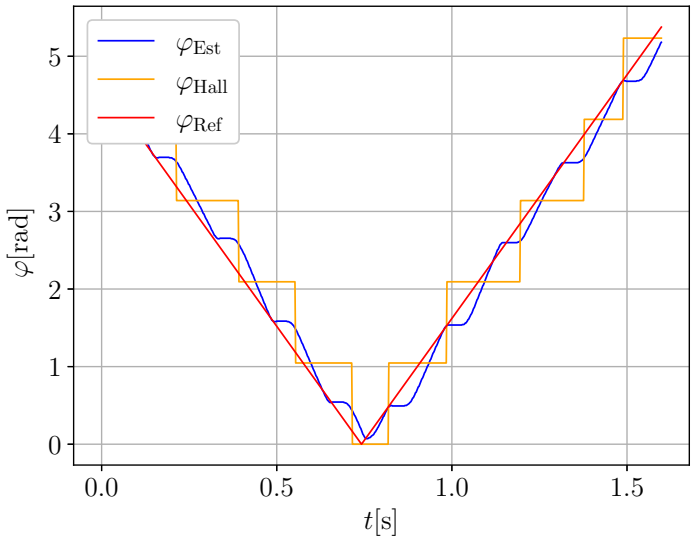


Figure 8.2: The electrical angle estimation with Halls and an IRC. The graph shows a well tuned φ_{Est}.

8.1.2 Open Loop Current Measurement

In this example we set $d > 0$, $q \leftarrow 0$ and $\omega \leftarrow \{5, 20\}$ rad. We then measure the currents by using the *Nuttx/ADC* block. The measurement is done on the L shunts. The pysimCoder diagram is shown in figure 8.3 and it demonstrates the trapezoidal shape of the winding current during open loop control, shown in figure 8.4 for different angular velocities. This diagram helped me prove I configured the SAMV7's AFEC peripherals correctly.

8.1.3 Simple Feedback Control

This example presents a simple control diagram with a PID controller. The controller's input is the difference between the target IRC count and the current IRC count. This is not an example of vector control, however electrical angle estimation is used to compute the a,

[2] `CodeGen/Common/common_dev/pmsm_align.c`

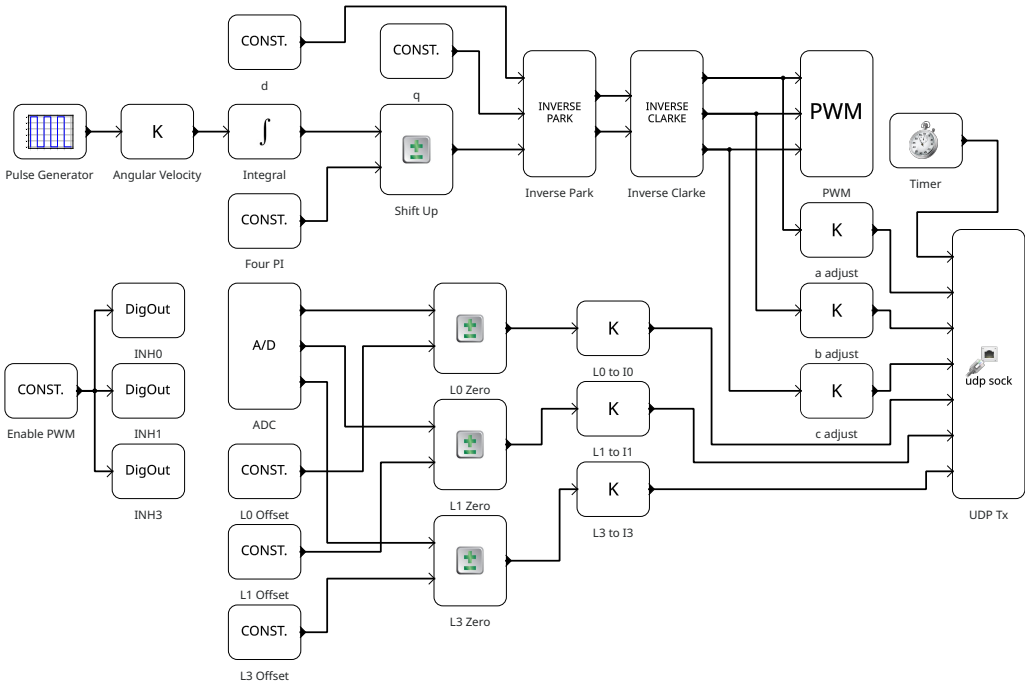


Figure 8.3: PysimCoder diagram for open loop PMSM control.

b and c actions from the inverse Park and inverse Clarke transformations. The controller's action output drives the q axis for the biggest torque while $d = 0$.

For showcase purposes, a reference rise limit has been added. Also, with the help of Silicon Heaven, a user can configure whether the motor should follow a ramp reference or a step by setting a constant in the *Pulse Or Int Control*, triggering certain switches in the diagram. The pysimCoder diagram is shown in figure 8.5, alongside with the motor's ramp and pulse responses, as shown in 8.6 and 8.7. I have configured the PID controller to provide the steepest responses, the constants being

$$k_P = 8 \cdot 10^{-4}, \quad k_I = 2 \cdot 10^{-5}, \quad k_D = 2 \cdot 10^{-4}.$$

8.1.4 Current Control

This example shows simple d and q axes PI controllers. The initial goal was to make whole vector control work, alongside position or speed control. I have tried tuning the PI controller multiple times but I was not able to tune it to have fast responses, as it led to instability due to the poor sampling rate.

The example, whose diagram is shown in the figure 8.9, shows a diagram for the dq current control with an open loop rotation control, with d being varied between two values and q set to zero. The angular velocity needed to be set small. Figures 8.8 show the i_d and i_q current control, the reference being steps. Some ADC noise can be seen, mostly in the left figure. Despite the problems, we can see pysimCoder and *SaMoCon* hardware is ready to quickly design control FOC applications and this diagram may be helpful to tune the PI current controllers in the future.

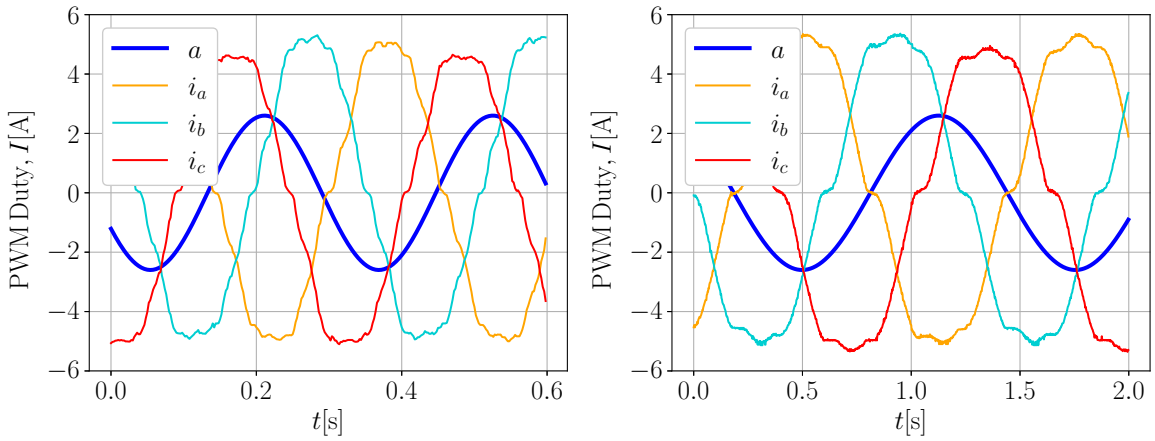


Figure 8.4: Open loop motor control. Left: $\omega = 20\,\text{rad}$, Right: $\omega = 5\,\text{rad}$. The output of the inverse transformations (a) is shown for a reference.

8.2 Piezoelectric Actuator Control with pysimCoder

The purpose of this experiment was to prove the controller is capable of controlling a piezoelectric actuator, whose datasheet can be found here [19] (P-871.140 is used). Even though this actuator features a full Wheatstone bridge made out of 4 tensometers to measure the bending angle, the experiment was initially planned for an actuator without tensometers. Unfortunately, due to lack of attention, I broke the fragile actuator, so this actuator was used as a quick replacement.

The tip of the piezoactuator is equipped with a mirror. The tip is illuminated by a laser beam at an angle. The beam is then reflected by a mirror and the beam can be viewed on for example a wall. If the actuator is bent, the incidence angle changes, and the position of the beam on the wall moves. The piezoactuator is mounted on a 3D-printed holder, as shown in 8.10.

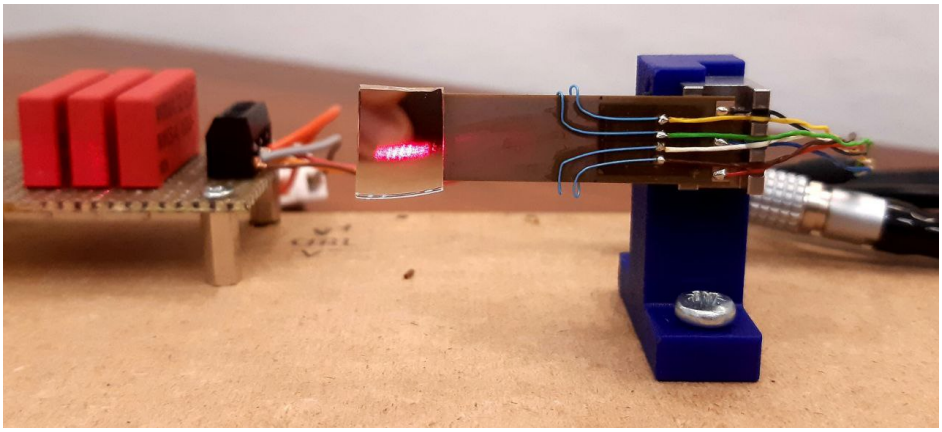


Figure 8.10: The optical setup with a piezoactuator from PI, GmbH.

Due to conditions in our lab, this experiment should serve as a proof of concept, since precise angle measurement would require a precisely mounted holder, precisely measured

incidence angle between the laser and the mirror and precisely measured distance between the wall and the mirror.

The way this actuator is controlled is shown in figure 8.11. Even though the rated voltage across *3-1* is up to $V_{\mathrm{Max}} = 60\,\mathrm{V}$, we can still suppose the position should change even when powered by 24 V, as that is the rated voltage of our power stage board. The input *2* is controlled by a voltage in the range of $[0, V_{\mathrm{Max}}]$, where $V_{\mathrm{Max}}/2$ corresponds to no tilt. To create DC voltage out of a PWM signal, a low pass filter is put between the power board output and the input *2*.

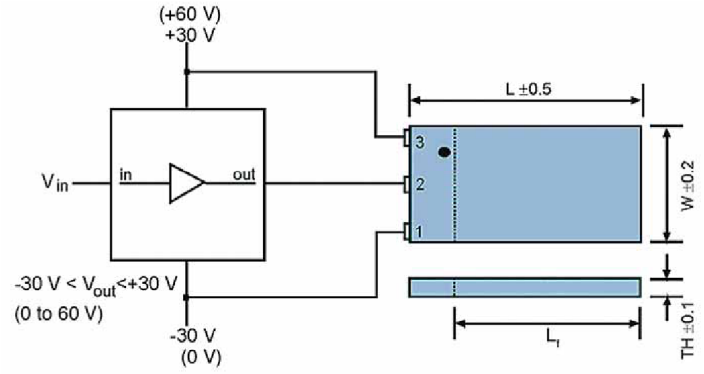


Figure 8.11: Piezoactuator tilt control [19].

As mentioned in 4.3.2, the element behaves as a capacitor. An inductor can be connected before the piezoactuator's input. Since we operate the PWM frequency at 20 kHz, we set the cutoff frequency around

$$f_{\mathrm{C}} = 5\,\mathrm{kHz}. \tag{8.2}$$

Since the capacitance of P-871.140 is $C = 2 \cdot 4\,\mu\mathrm{F}$ [19], we calculate the estimate value of L from this formula:

$$L = \frac{1}{4\pi^2 f_{\mathrm{C}}^2 C} = 126\,\mu\mathrm{H}. \tag{8.3}$$

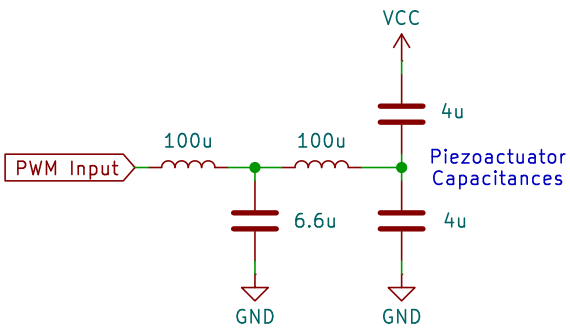


Figure 8.12: The used filter to control the piezoactuator.

We can connect another LC filter before the inductor to create a 4th-degree low pass filter with higher attenuations. The used inductors have inductances of 100 μH. The used filter is shown in figure 8.12.

Our experiment tried to prove the repeatability of the movement which means the same bending angle can be expected when the same voltage is applied after consecutive tries. Before the measurement took place, I turned on the laser beam and marked the point with a pencil on the wall, serving as a reference base point, when the actuator was in its base position with no voltage applied. A voltmeter was connected to the filter's output to monitor its output DC voltage. The experiment setup is shown in figure 8.14.

Afterwards, I turned on the voltage and applied approximately $6, 12, 18$, and 24 V to the piezoactuator's input and marked the beam's position referenced to the reference point. Anything on the left to the reference point has a positive value. This was tried out three times, and the values are shown in table 8.1. The values Δl_1, Δl_2, and Δl_3 denote the measured offsets referenced to the reference point during the first, second, and third try, respectively.

Voltage	Δl_1 [mm]	Δl_2 [mm]	Δl_3 [mm]
0 V	-22	-22	-22
6.10 V	-13	-12	-13
12.07 V	-3	-2	-3
18.03 V	9	9	9
23.82 V	21	21	21

Table 8.1: The measured values for the repeatability experiment with piezoactuator.

The voltage was controlled by a pysimCoder diagram shown in figure 8.13 while the applied voltage was tuned remotely using the Silicon Heaven protocol. The *Duty* value can be set anywhere between $[-1, 1]$ where -1 means 0 V is applied and 1 means 24 V is applied to the piezoactuator input. A low-pass filter has been added before the PWM block input to not stress actuator with fast voltage spikes. The transfer function of the low-pass filter is

$$G(s) = \frac{5}{s+5}. \tag{8.4}$$

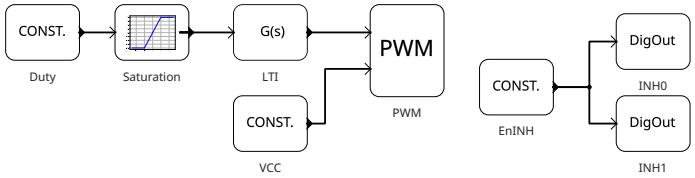


Figure 8.13: The piezoactuator voltage control pysimCoder diagram.

To prove the movement repeatability, more values should be measured in a better environment. However, the purpose of this experiment is to show that even with such a simple diagram shown in 8.13, the researcher is ready to control the piezoactuator bending angle.

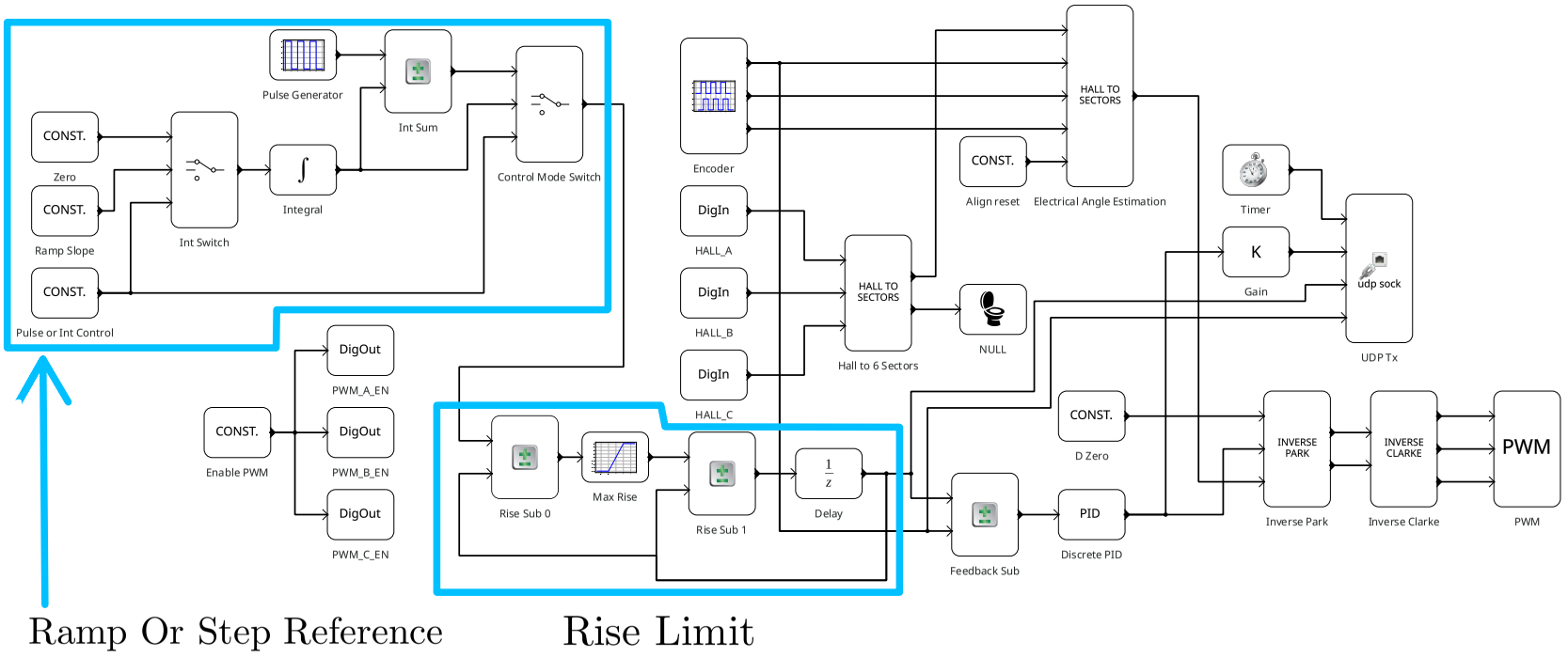


Figure 8.5: The diagram for a simple PMSM control with a PID controller, alongside with a rise limit and a user choosable reference (ramp or step)

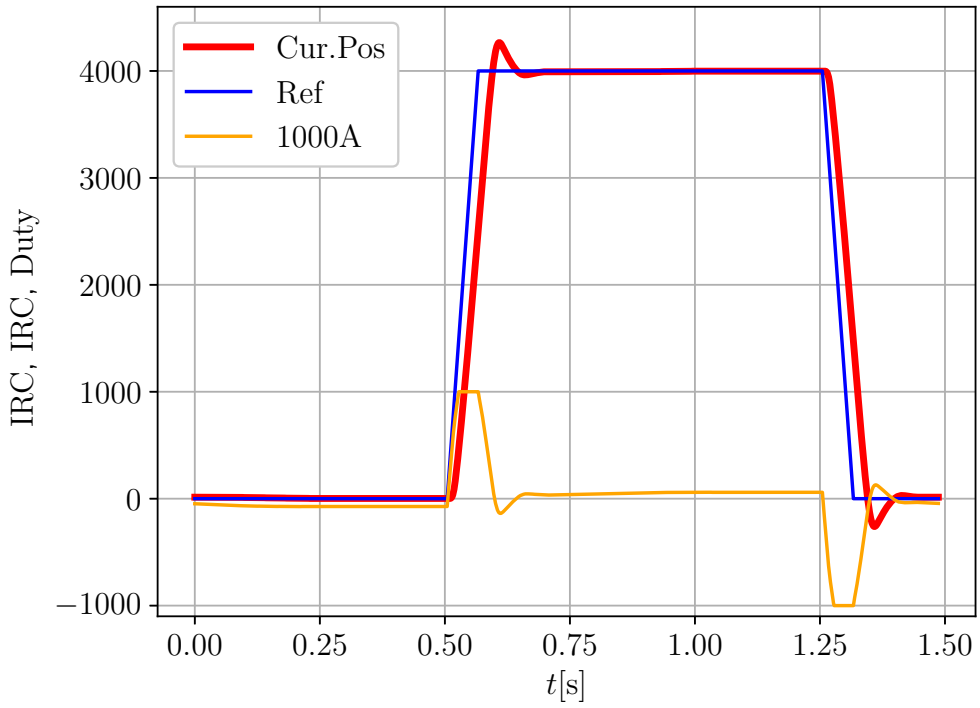


Figure 8.6: A periodic step reference between 0 and 4000 IRC pulses (corresponding to 2 mechanical turns) back and forth. The PID action is also shown (multiplied by 1000).

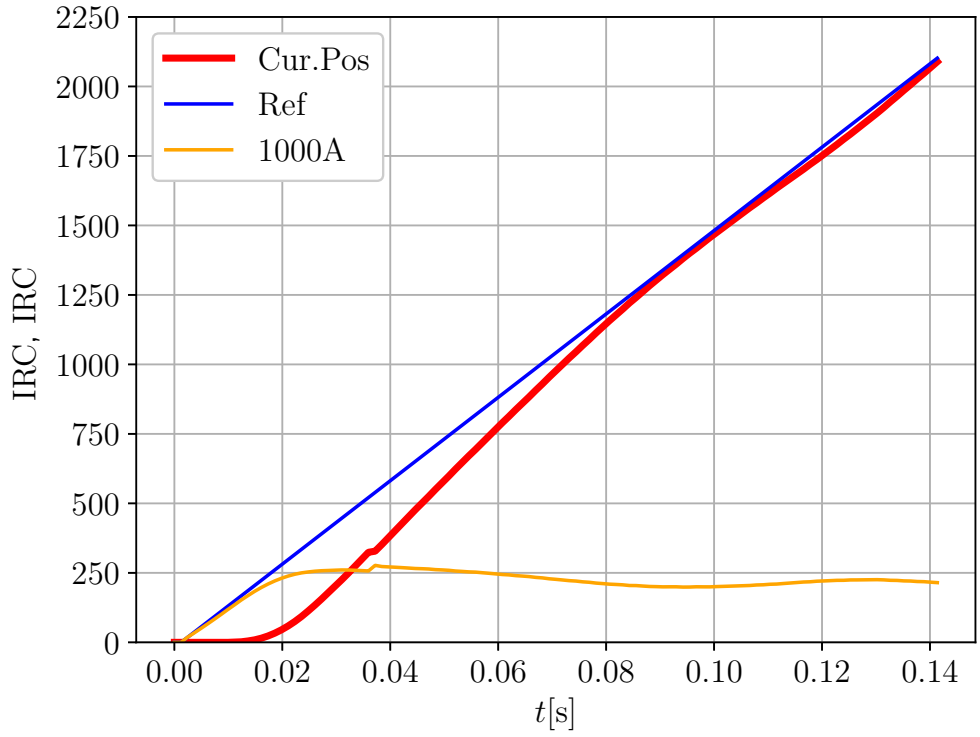


Figure 8.7: The motor following a ramp with a speed of 15000 IRC pulses/s. The PID action is also shown (multiplied by 1000).

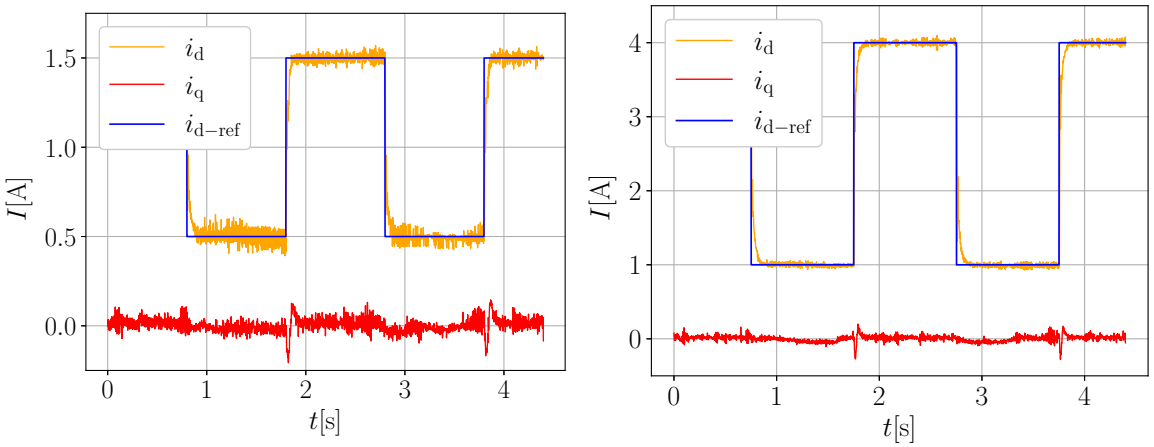


Figure 8.8: The current waveforms. i_q reference is set to zero, i_d varies between 0.5 and 1.5 A on the left and 1 and 4 A on the right.

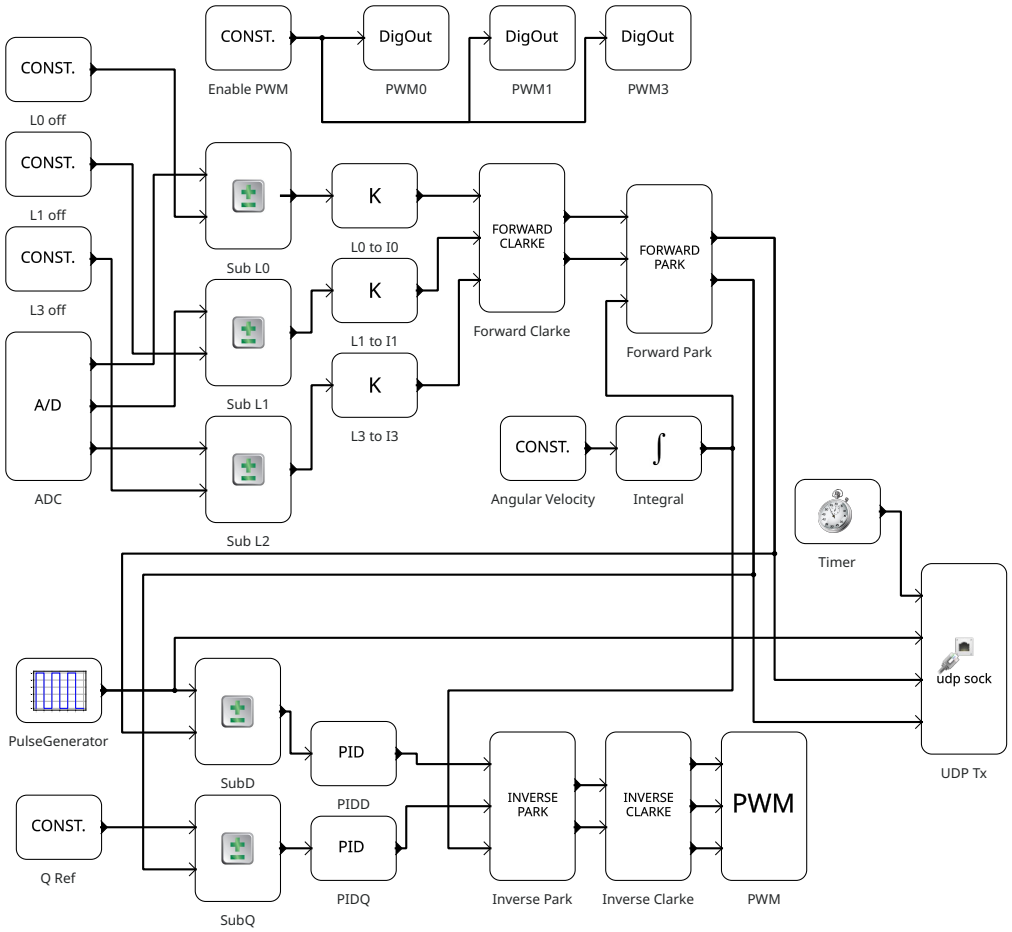


Figure 8.9: The dq current control diagram in pysimCoder.

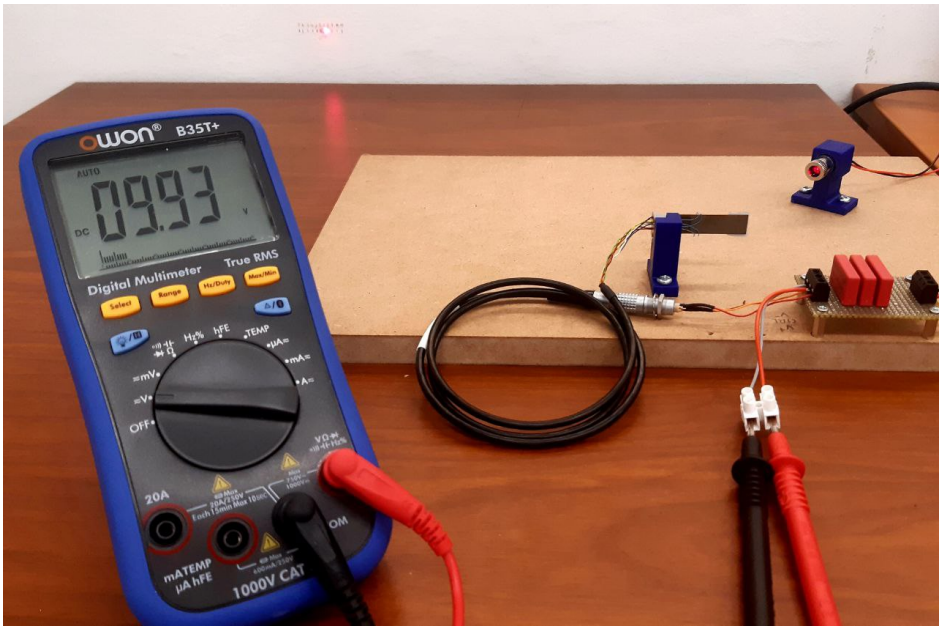


Figure 8.14: Measuring the reflected angle.

Chapter 9

Conclusion and Summary

The goal of this thesis was to design hardware for a platform used for rapid control prototyping. While it is possible to adapt the MCU board to a different power stage board and run bare metal applications on it to even control big induction motors, the main task was to create a generic platform which could be used for various control experiments. With that, the pysimCoder suite alongside the Silicon Heaven protocol was tested with a BLDC motor controlled in the PMSM way and proved useful in the optical experiment with a piezoactuator in cooperation with ÚTIA, AV ČR.

Despite some hardware flaws, I was able to bring up the two PCB boards. The MCU now runs NuttX with all the needed peripherals for motion control registered, alongside with configured network over the Ethernet and the power stage capable of providing enough power while measuring the winding currents. During the adaptation, I contributed to NuttX's lowend drivers to adapt the peripherals for our tasks.

Unfortunately, we have come across with serious NuttX issues. While the pysimCoder code adds some overhead, our microcontroller with the double precision float support should handle this problem with no hassle. Since there is no easy answer to these sampling issues, the only solution is to consult the NuttX developers and profile/debug NuttX itself to find out the source of problems. Placing the control loop in a high-priority interrupt may be possible but it can even turn out these sampling issues are unsolvable in NuttX so the configuration will have to be done on other RTOSs, like Zephyr or RTEMS while adapting pysimCoder to the API of these operating systems.

The hardware is currently available under the CERN-OWL-W v2 license on the FEL ČVUT GitLab [1], my NuttX fork can be found here [2] (the development is done in the `samocon-branch` branch). This project remains open and free to use as we believe this is the best way to keep the project fresh and updated. If we solve the issues with NuttX, a custom *SaMoCon* API should be created to allow for a nonoverhead motion control using protocols like TCP/IP and UDP/IP.

[1] `https://gitlab.fel.cvut.cz/otrees/motion/work-and-ideas/-/wikis/SaMoCon`
[2] `https://github.com/zdebanos/nuttx`

Bibliography

[1] Alpha Omega Semiconductor: *AOZ1284 EZBuck 4A Simple Buck Regulator*. https://www.aosmd.com/res/datasheets/AOZ1284PI.pdf, 2019. – Online, Accessed on May 15th, 2024

[2] Analog Devices, Inc.: *Low Power RS485 Transceiver with Receiver Fail-Safe*. https://www.analog.com/media/en/technical-documentation/data-sheets/1484f.pdf, . – Online, Accessed on January 25th, 2024

[3] Baldassari, François: *Profiling Firmware on Cortex-M*. https://interrupt.memfault.com/blog/profiling-firmware-on-cortex-m, 2020. – Online, Accessed on May 16th, 2024

[4] Bucher, Roberto: *The GitHub repository of pysimCoder*. https://github.com/robertobucher/pysimCoder, 2024. – Online, Accessed on April 17th, 2024

[5] Digikey North America: *How to Power and Control Brushless DC Motors*. https://www.digikey.com/en/articles/how-to-power-and-control-brushless-dc-motors, 2016. – Online, Accessed on May 18th, 2024

[6] Elektroline, a.s.: *The GitHub repositories of Silicon Heaven project*. https://github.com/silicon-heaven, 2024. – Online, Accessed on May 23rd, 2024

[7] Franklin, Gene F. ; Powell, J. D. ; Emami-Naeini, Abbas: *Feedback Control of Dynamic Systems, 8th edition*. 2020. – ISBN 1–292–27452–2

[8] Infineon Technologies: *High Current PN Half Bridge with Integrated Driver*. https://www.infineon.com/dgdl/Infineon-IFX007T-DS-v01_00-EN.pdf?fileId=5546d46265f064ff0166433484070b75, 2018. – Online, Accessed on May 12th, 2024

[9] Lenc, Michal: *Open Rapid Control Prototyping and Real-Time Systems*. https://dspace.cvut.cz/handle/10467/100938, 2022. – Online, Accesssed on January 28th, 2024

[10] Microchip Technology Inc.: *Ethernet Theory of Operation*. https://ww1.microchip.com/downloads/aemDocuments/documents/OTH/ApplicationNotes/ApplicationNotes/01120a.pdf, 2008. – Online, Accessed on May 12th, 2024

[11] Microchip Technology Inc.: *MCP1525, 2.5 Voltage Reference*. https://www.microchip.com/en-us/product/mcp1525, 2012. – Online, Accessed on May 15th, 2024

[12] Microchip Technology Inc.: *10BASE-T/100BASE-TX PHY with RMII Support*. https://ww1.microchip.com/downloads/aemDocuments/documents/UNG/ProductDocuments/DataSheets/KSZ8081RNA-RND-10BASE-T-100-BASE-TX-PHY-with-RMII-Support-DS00002199F.pdf, 2019. – Online, Accessed on January 21st, 2024

[13] Microchip Technology Inc.: *MCP6021/1R/2/3/4, Rail-to-Rail Input/Output, 10 MHz Op Amps*. https://ww1.microchip.com/downloads/aemDocuments/documents/MSLD/ProductDocuments/DataSheets/MCP6021-Data-Sheet-DS20001685.pdf, 2023. – Online, Accessed on May 15th, 2024

[14] Microchip Technology Inc.: *SAM E70/S70/V70/V71 Family Data Sheet*. https://ww1.microchip.com/downloads/aemDocuments/documents/MCU32/ProductDocuments/DataSheets/SAM-E70-S70-V70-V71-Family-Data-Sheet-DS60001527.pdf, 2023. – Online, Accessed on January 21st, 2024

[15] Microchip Technology Ltd.: *High-speed CAN FD Transceiver with Standby Mode and VIO Pin*. https://www.microchip.com/en-us/product/mcp2562fd, 2014. – Online, Accessed on May 15th, 2024

[16] Monolithic Power Systems: *Stepper Motors Basics: Types, Uses, and Working Principles*. https://www.monolithicpower.com/stepper-motors-basics-types-uses, 2024. – Online, Accessed on May 20th, 2024

[17] Omron: *Miniature size rotary encoder*. https://industrial.omron.eu/en/products/e6a2-c, . – Online, Accessed on January 21st, 2024

[18] Patankar, Priyanka ; Kulkarni, Swapnil: *MATLAB and Simulink In-Depth*. BPB Online, 2022. – ISBN 978–93–55511–997

[19] Physik Instrumente GmbH: *P-871 PICMA® Piezo Bender Actuators*. https://www.pi-usa.us/fileadmin/user_upload/pi_us/files/product_datasheets/P871_Piezo_Bimorph_Bender.pdf, 2008. – Online, Accessed on May 23rd, 2024

[20] Physik Instrumente GmbH: *Electrical Operation of Piezo Actuators*. https://www.piceramic.com/en/expertise/piezo-technology/properties-piezo-actuators/electrical-operation, 2024. – Online, Accessed on May 20th, 2024

[21] PiKRON s.r.o.: *LX_RoCoN, Motion And Robotic Controller*. https://pikron.com/pages/products/motion_control/lx_rocon.html, . – Online, Accessed on May 16th, 2024

[22] PiKRON s.r.o.: *imxRT Teensy-4.1 Base Board by PiKRON*. https://gitlab.com/pikron/projects/imxrt-devel/-/wikis/teensy_bb, 2021. – Online, Accessed on May 11th, 2024

[23] Renesas: *Application Note: RS-422 vs RS-485 Similarities and Key Differences*. https://www.renesas.com/us/en/document/apn/an1989-rs-422-vs-rs-485-similarities-and-key-differences, 2017. – Online, Accessed on May 12th, 2024

[24] Texas Instruments Incorporated: *TPS562207 4.3-V to 17-V Input, 2-A Synchronous Buck Converter in SOT563*. https://www.ti.com/lit/ds/symlink/tps562207.pdf, 2021. – Online, Accessed on May 15th, 2024

[25] Texas Instruments Incorporated: *AM26LV32 Low-Voltage, High-Speed Quadruple Differential Line Receiver*. https://www.ti.com/lit/ds/symlink/am26lv32.pdf, 2023. – Online, Accessed on May 15th, 2024

[26] Texas Instruments Incorporated: *LM393B, LM2903B, LM193, LM293, LM393 and LM2903 Dual Comparators*. https://www.ti.com/lit/ds/symlink/lm2903.pdf, 2023. – Online, Accessed on May 15th, 2024

[27] The Apache Software Foundation: *NuttX Documentation*. https://nuttx.apache.org/docs/latest/, . – Online, Accessed on May 2nd, 2024

[28] The Apache Software Foundation: *NuttX Documentation - System Time and Clock*. https://nuttx.apache.org/docs/latest/reference/os/time_clock.html, . – Online, Accessed on May 11th, 2024

[29] The Apache Software Foundation: *NuttX High Performance, Zero Latency Interrupts*. https://nuttx.apache.org/docs/latest/guides/zerolatencyinterrupts.html, 2023. – Online, Accessed on May 16th, 2024

[30] The Apache Software Foundation: *NuttX Task Trace*. https://nuttx.apache.org/docs/latest/guides/tasktrace.html, 2023. – Online, Accessed on May 16th, 2024

[31] Vaez-Zadech, Sadegh: *Control of Permanent Magnet Synchronous Motors*. Oxford University Press, 2018. – ISBN 978–0–19–874296–8

[32] Yoichi, Mamiya: *Applications of Piezoelectric Actuator*. https://www.nec.com/en/global/techrep/journal/g06/n05/pdf/t060519.pdf, 2006. – Online, Accessed on May 20th, 2024