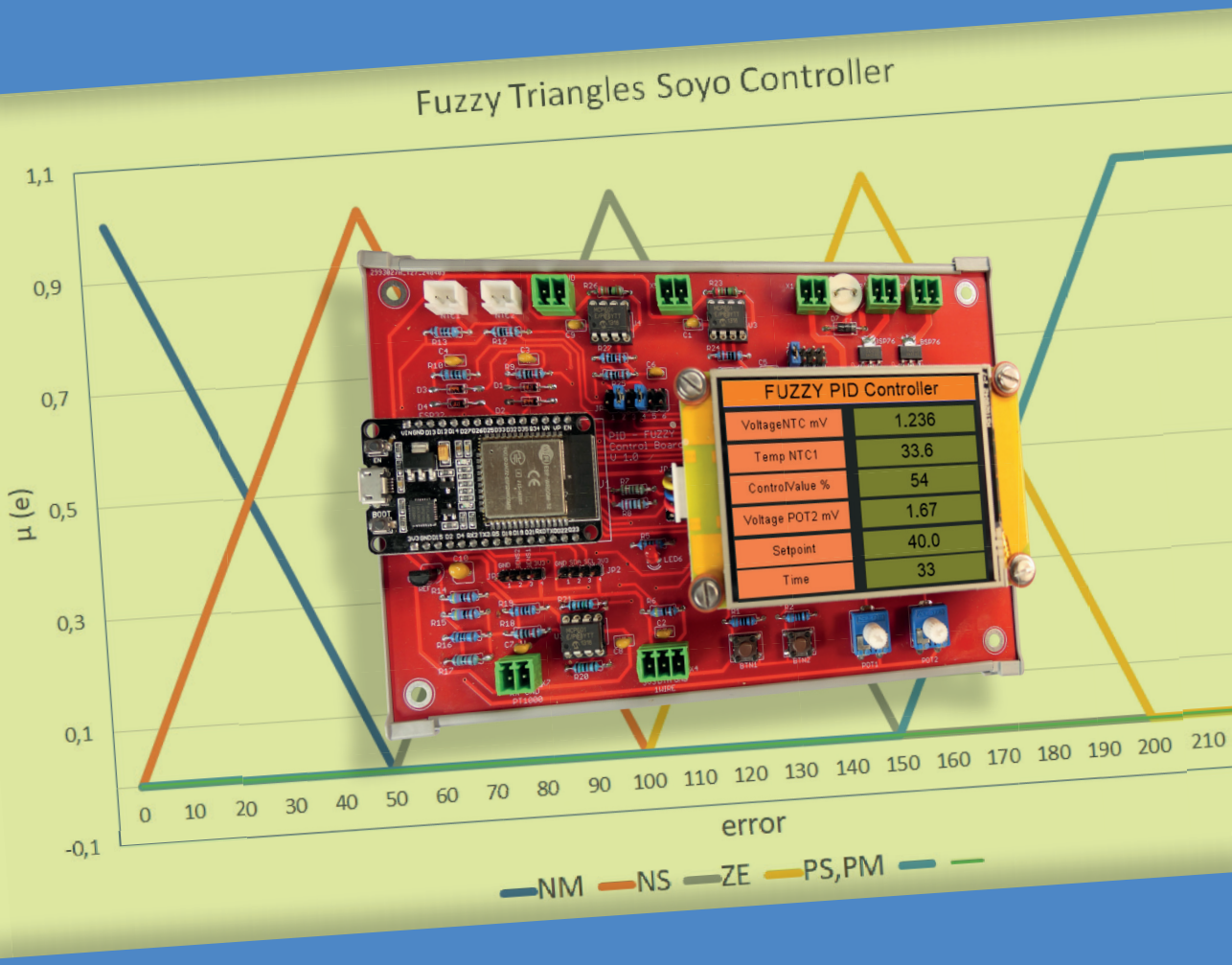
elektor books

Control Engineering with Fuzzy Logic

Practical Applications and Projects with Arduino, ESP32, and RP2040



Josef Bernhardt

elektor
design > share > earn

Control Engineering with Fuzzy Logic

Practical Applications and Projects with Arduino, ESP32, and RP2040



Josef Bernhardt

elektor
design › share › earn

This is an Elektor Publication. Elektor is the media brand of
Elektor International Media B.V.
PO Box 11, NL-6114-ZG Susteren, The Netherlands
Phone: +31 46 4389444

All rights reserved. No part of this book may be reproduced in any material form, including photocopying, or storing in any medium by electronic means and whether or not transiently or incidentally to some other use of this publication, without the written permission of the copyright holder except in accordance with the provisions of the Copyright Designs and Patents Act 1988 or under the terms of a licence issued by the Copyright Licencing Agency Ltd., 90 Tottenham Court Road, London, England W1P 9HE. Applications for the copyright holder's permission to reproduce any part of the publication should be addressed to the publishers.

Declaration

The author and publisher have made every effort to ensure the accuracy of the information contained in this book. They do not assume, or hereby disclaim, any liability to any party for any loss or damage caused by errors or omissions in this book, whether such errors or omissions result from negligence, accident, or any other cause.

ISBN 978-3-89576-660-2 Print
ISBN 978-3-89576-661-9 eBook

© Copyright 2025 Elektor International Media
www.elektor.com
Editor: Ferdinand te Walvaart
Prepress Production: D-Vision, Julian van den Berg
Printers: Ipskamp, Enschede, The Netherlands

Elektor is the world's leading source of essential technical information and electronics products for pro engineers, electronics designers, and the companies seeking to engage them. Each day, our international team develops and delivers high-quality content - via a variety of media channels (including magazines, video, digital media, and social media) in several languages - relating to electronics design and DIY electronics. **www.elektormagazine.com**

Contents

Preface 9

Chapter 1 ● Arduino Hardware for Control Engineering 10

1.1 Arduino UNO 10

1.2 Arduino NANO 11

1.3 Espressif ESP32 13

1.4 Raspberry RP2040 controller 14

1.5 ESP32 Fuzzy PID controller board 15

1.6 ESP32 Modbus RTU controller board 19

1.7 Programming the Arduino boards 21

1.8 The Nextion graphic display 23

1.9 Test software for the ESP32 board 24

1.10 Program example with the ESP32 28

1.11 Arduino ESP32 controller 12 bit A/D converter test 34

1.12 CodeBlocks development environment for Ansi C 40

1.13 Serial Data Reader 42

1.14 Serial Port Data Reader 42

1.15 The Espressif ESP32 as a WebSocket server 43

Chapter 2 ● Sensors for Control Engineering 54

2.1 NTC temperature sensors 54

2.2 RTD temperature sensors 57

2.3 Linear Interpolation 60

2.4 Thermocouples 61

2.5 Applications of thermocouples 62

2.6 Sensors with integrated electronics 65

2.7 pH sensors 67

2.8 Rotary encoder 68

2.9 Force measurement 70

2.10 Linear regression 73

2.11 Spline interpolation 77

2.12 Polynomial regression 81

Chapter 3 • Actuators - MOSFET and SSR 87

3.1 MOSFET BSP76 87

3.2 MOSFET circuit with optocoupler for 10 A 88

3.3 H-bridge L298 89

3.4 H-bridge with BTS7960 from Infineon up to a maximum of 43 A 89

3.5 Thyristor actuator for AC voltage 91

Chapter 4 • Control, regulation and control systems. 92

4.1 Formula symbols 92

4.2 Difference between control and control with feedback according to IEC 60050-351 . . 92

4.3 Examples control technology 93

4.4 Control of a real heating system 97

4.5 Control of a DC motor with PWM 108

Chapter 5 • On-Off and PID controllers 119

5.1 Two-point controller 119

5.2 PID controller 124

5.3 Parameter determination using Ziegler-Nichols 126

5.4 PID parameter determination with the relay method 127

5.5 Arduino PID controller with PT2 simulation 129

5.6 Arduino PID controller with real controlled system (ALU block) 135

5.7 Arduino PID controller for standing pendulum 146

Chapter 6 • Fuzzy Logic basics. 155

6.1 Fuzzy Logic history 155

6.2 Fuzzy controller structure 156

6.3 Fuzzification 156

6.4 Inference 156

6.5 Defuzzification. 156

6.6 Fuzzy controller example for a fan 157

6.7 Control of an inverted pendulum 163

6.8 Fuzzy sets 168

6.9 Simulation of a function with fuzzy logic 177

6.10 Application of a fuzzy function for a temperature profile 183

Chapter 7 • Fuzzy Logic temperature controller 186

7.1 Fuzzy controller with PT2 simulation 186

7.2 Fuzzy controller with the ESP32 and real heating. 193

7.3 Fuzzy examples with Gauss, Bell and Sigmoid 206

Chapter 8 • Fuzzy logic temperature controller with two inputs. 207

8.1 Temperature controller with two inputs 207

8.2 Fuzzy control example for a heating control system 211

Chapter 9 • Fuzzy controller examples with a DC motor 220

9.1 DC motor basics 220

9.2 Control of the motor with PWM (speed control) 222

9.3 Motor control with PWM and speed measurement 223

9.4 Motor speed control with fuzzy logic 227

9.5 Motor position control 240

Chapter 10 • Sliding mode controller. 250

10.1 General description of the sliding mode controller 250

10.3 Sliding mode controller example with real system 256

10.4 Optimization of the parameters $c1$ and $c2$ with fuzzy logic 268

Chapter 11 • Neural networks 272

11.1 History of neural networks 272

11.2 Biological neuron 273

11.3 Basics. 273

11.4 Activation functions 274

11.5 The perceptron 276

11.6 Feedforward networks 276

11.7 Training the neural network 279

11.8 ANN example XOR 282

11.9 Fuzzy Gauss controller with neural network 287

11.10 Color sensor evaluation with neural network 294

11.11 Training a neural network with two hidden layers. 307

Chapter 12 • Fuzzy controller for feeding electricity into the domestic grid. 320

12.1 Introduction 320

12.2 Shelly 3EM Energy Meter 320

12.3 Measurements with the Shelly Plus 1PM 323
12.4 Charge controller - Victron Smart Solar 100 I 20 326
12.5 Windows software SharpDevelop_Shelly3emWinform 329
12.6 Soyo Micro Inverter for feeding into the grid 329
12.7 ESP32 PID Fuzzy Control Board for closed-loop control 331
12.8 Fuzzy controller function 336
Appendix A • Circuits layouts and parts list 338
Appendix B • Bibliography 353
Appenix C • Weblinks 354
Index 356

Preface

This book is written for students and electronics enthusiasts interested in practical applications of control engineering. The aim of this book is to teach the fundamentals of control engineering, with a special focus on Fuzzy Logic control systems, and to illustrate these concepts through various practical examples.

In particular, the basics of sensors and linearization are presented in an easily understandable manner, providing a solid foundation for acquiring sensor data. Furthermore, we offer detailed information and examples on controlling different actuators, demonstrating how you can build your own projects.

The book is supplemented with numerous examples and describes real-world applications of control engineering, all implemented on the Arduino platform. This approach allows readers to gain hands-on experience and apply what they have learned directly in practice.

In addition to covering the basics of the Arduino Uno, we also delve into the use of the powerful ESP32 to give you a comprehensive insight into the world of microcontrollers. You will be guided step-by-step through applications for temperature control using Fuzzy Logic, and learn how to control motors, both in terms of speed and position.

The book concludes with chapters on sliding mode controllers, the basics of neural networks, and controllers for grid feed-in, all complemented with practical examples.

We hope this book not only provides enjoyment while reading but also assists you in successfully implementing your own projects. We wish you much fun and success with this book and your future projects!

Finally, we would like to thank everyone who supported us, repeatedly checked details, read the manuscript, provided feedback, allowed us to quote their comments, and helped with editing, proofreading, and design.

Josef Bernhardt

Chapter 1 • Arduino Hardware for Control Engineering

Arduino is an open-source platform for electronics projects that combines hardware and software. It is user-friendly and suitable for both beginners and experienced developers. There is a wide variety of Arduino boards, sensors, and components to choose from.

The name Arduino originates from a bar in the town of Ivrea, Italy, where in 2005, students Massimo Banzi, David Cuartielles, Tom Igoe, Gianluca Martino, and David Mellis conceived the idea of developing a programmable board for prototyping. Arduino was quickly released as an open-source platform, which led to the founding of the Arduino company in 2009. The company supports the production and distribution of the boards and continually develops new products.

With Arduino boards, projects can be realized in various fields, such as robotics and industrial control systems. For programming the hardware platform, the Arduino IDE is used, which allows users to write, test, and upload programs.

The IDE is based on C/C++ and makes it easier for beginners to get started by offering numerous libraries and example codes. Arduino provides endless possibilities for creativity and innovation.

In this book on control engineering, the focus is mainly on the Arduino UNO, Arduino Nano, and the ESP32 from Espressif. The Raspberry RP2040 is also of great interest for more complex projects.

1.1 Arduino UNO

The Arduino UNO is one of the most popular Arduino boards and is based on the ATmega328P microcontroller. It offers numerous digital and analog pins, as well as features like analog/digital converters, PWM outputs, and serial communication. The UNO is easy to use and has a large developer community, making it ideal for control engineering. Programming is done via the Arduino IDE using C code. With the UNO, we can control sensors, heaters, motors, LEDs, and other components, and build high-quality controllers. Its functionality can be extended with expansion boards, such as displays, network connections, or wireless communication. The UNO is widely used in educational and hobby projects, as well as in professional applications for prototyping and rapid development.

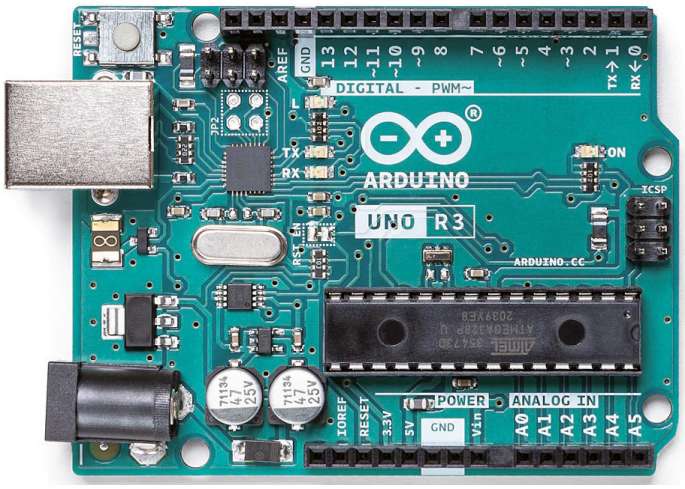


Figure 1.1: Arduino UNO Board

1.2 Arduino NANO

The Arduino Nano is highly suitable for electronics projects due to its compact form factor and is also based on the ATmega328P microcontroller. It offers similar features to the Arduino UNO but in a smaller size, making it ideal for space-constrained projects. The Nano has 14 digital I/O pins, 8 analog inputs, and supports serial communication as well as PWM outputs. Due to its small size and flexibility, the Arduino Nano is often used in portable and space-critical applications.

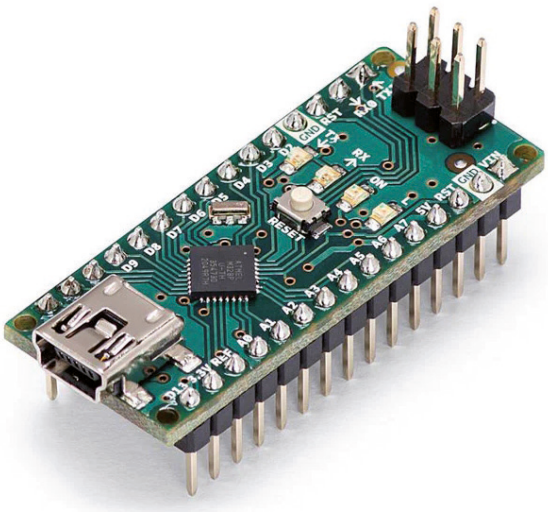


Figure 1.2: Arduino Nano board

Features and Specifications of the CPU

- Microcontroller: Atmel / Microchip ATmega328P
- Operating Voltage: 5 Volts
- Input Voltage (recommended): 7-12 Volts
- Digital I/O Pins: 14 (6 of which support PWM)
- Analog Input Pins: 6 with 10-bit resolution
- Flash Memory: 32 KB (0.5 KB reserved for the bootloader)
- SRAM: 2 KB
- EEPROM: 1 KB
- Clock Frequency: 16 MHz

Arduino Nano Pin	Board	Description
D2	Button S1	Digital Input active High
D3	Button S2	Digital Input active High
D4	Button S3	Digital Input active High
D5	Button S4	Digital Input active High
D6	Button S5	Digital Input active High
D7	LED 1	Digital Output
D8	LED 2	Digital Output
D9	LED 6	PWM Output
D10	LED 7	PWM Output
D11	LED 8	PWM Output
D12	LED 3	Digital Output
D 13	LED 4	Digital Output
ADC0	Poti R13	Analog Input
ADC1	Poti R14	Analog Input
ADC2	Poti R15	Analog Input
ADC3	Poti R16	Analog Input
ADC4	Poti R17	Analog Input

Table 1.1: Pin Assignment Arduino Nano test board.

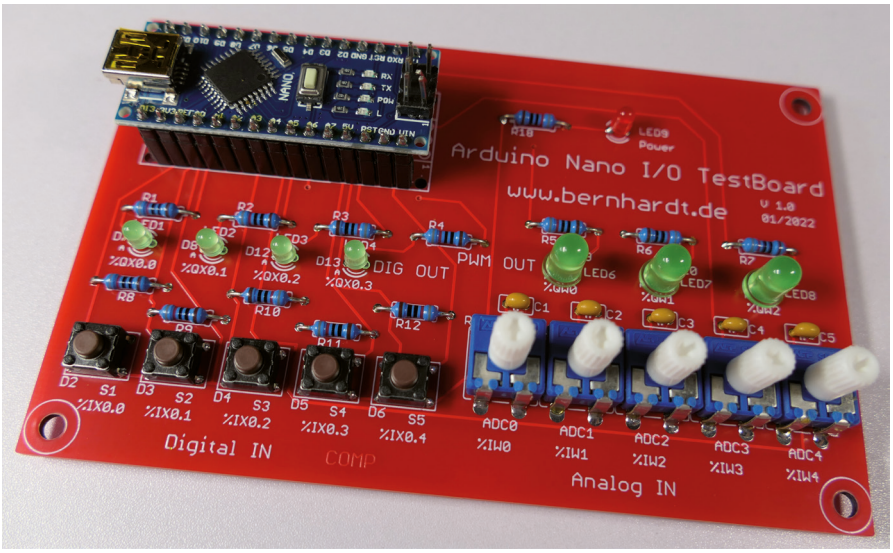


Figure 1.3: Arduino Nano Test board

1.3 Espressif ESP32

The ESP32 from Espressif is a powerful development board based on the ESP32 microcontroller. It features integrated Wi-Fi and Bluetooth, making it ideal for IoT applications. Programming is also done through the Arduino IDE, which can be downloaded from the official Arduino website [LINK1]. After installing the ESP32 board support, we can program the ESP32 using C++ code and access its hardware. The ESP32 has numerous GPIO pins for connecting sensors and actuators, and it can function as both a server and a client in Wi-Fi networks. Additionally, it allows communication with other devices via Bluetooth.

Due to its versatility, the ESP32 is suitable for projects in smart home, robotics, and control engineering. It is also beginner-friendly and benefits from a large developer community and numerous online resources that help with getting started. With the ESP32, we can collect sensor data and display it over Wi-Fi or on a TFT display, as well as control actuators.

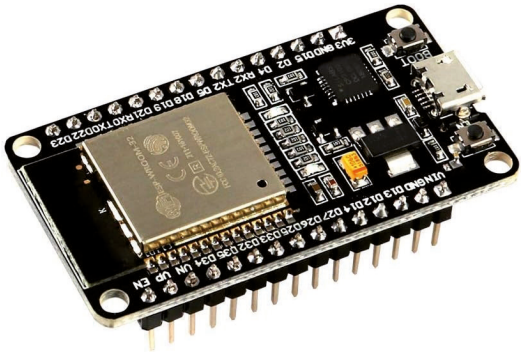


Figure 1.4: ESP32 board

Technical Specifications of the ESP32 Microcontroller:

- **Processor**: Dual-Core Tensilica LX6 with up to 240 Mhz
- **Memory**:
  - **RAM**: 520 KB SRAM
  - **ROM**: 448 KB
  - **Flash Memory**: Varies by model, typically 4 MB

- **Features**:
  - **Wi-Fi**: 802.11 b/g/n
  - **Bluetooth**: v4.2 BR/EDR and BLE

- **GPIO Pins**: 34 pins, versatile for various peripherals

- **Communication Interfaces**:
  - **UART**: 3 interfaces
  - **SPI**: 4 interfaces
  - **I^2C**: 2 interfaces
  - **I^2S**: 2 interfaces
  - **PWM**: Up to 16 channels
  - **DAC**: 2 channels
  - **ADC**: Up to 18 channels, 12-bit resolution

- **Additional Features**:
  - **Touch Sensors**: Up to 10 capacitive touch inputs
  - **RTC (Real-Time Clock)**
  - **Hardware Acceleration for Encryption**: AES, SHA-2, RSA, ECC

- **Power Management**:
  - Various power-saving modes (Light Sleep, Deep Sleep)
  - **Operating Voltage**: 2.2 V – 3.6 V

- **Operating Temperature**: -40°C to +85°C

- **Peripherals**:
  - Hall Sensor
  - Temperature Sensor

- **Connectivity**: USB, external power supply via battery or rechargeable battery

1.4 Raspberry RP2040 controller

The RP2040 is a powerful microcontroller developed by the Raspberry Pi Foundation. It is an affordable microcontroller based on a dual-core architecture, specifically designed for embedded applications. It stands out for its high performance, flexibility, and energy efficiency.

The CPU features a range of functions, including GPIO pins, UART, SPI, and I²C interfaces, as well as a variety of peripheral modules. It also supports programming via USB and includes an integrated USB bootloader, enabling programming and operation without the need for external programming devices.

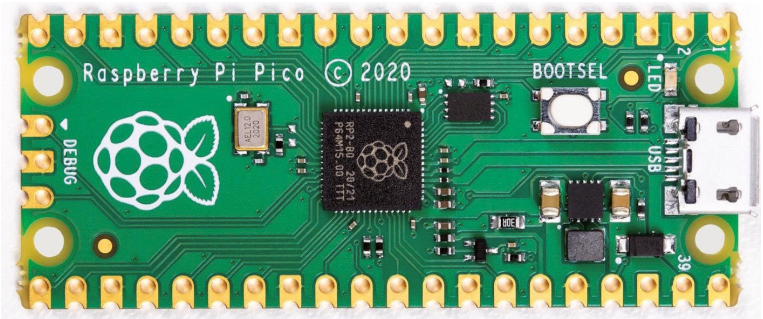


Figure 1.5: Raspberry RP2040 Board

Features and Specifications:

- **Microcontroller**: Dual-Core ARM Cortex-M0+ (up to 133 MHz)
- **Operating Voltage**: 3.3 Volts
- **Digital I/O Pins**: 30 (26 of which are PWM-capable)
- **Analog Input Pins**: 4
- **Flash Memory**: 2 MB (QSPI)
- **SRAM**: 264 KB
- **EEPROM**: Not available
- **Clock Speed**: Up to 133 MHz

The RP2040 is supported by a dedicated developer community that provides a variety of resources, including documentation, tutorials, forums, and example projects. These resources make project development easier and encourage the exchange of knowledge and experiences within the community.

1.5 ESP32 Fuzzy PID controller board

For the program examples in this book, we use a test board specifically designed for control engineering. The "ESP32 Fuzzy PID Controller" board is a versatile platform for precise control and regulation applications. With a wide range of interfaces, it allows seamless integration into numerous applications.

Additional practical applications, such as using it as a PLC or Modbus TCP I/O module, as well as protocols like MQTT or web applications with WebSockets, are always possible.

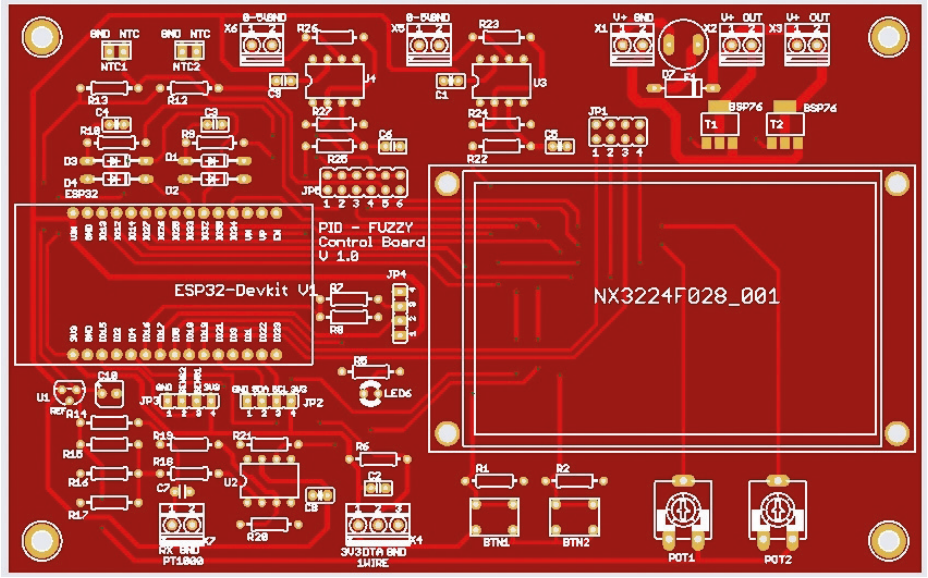


Figure 1.6: ESP32 Fuzzy PID Test board

Input Interfaces:

- **2 x NTC 10K**: These interfaces enable precise temperature change measurements using NTC temperature sensors.
- **1 x PT1000**: The PT1000 interface allows for accurate temperature readings using PT1000 temperature sensors.
- **1 x OneWire for DS18B20 or DHT11 / DHT22**: This flexible interface supports the use of DS18B20 temperature sensors or DHT11/DHT22 humidity sensors for reliable environmental monitoring.
- **2 x Buttons**: Two buttons provide an easy user interface for controlling and configuring the system.
- **2x Potentiometers**: Two potentiometers allow manual adjustment of parameters or thresholds for fine-tuned control.
- **2 x Sense Lines for Motor Speed and Position Monitoring**: These interfaces allow for precise measurement of motor speed and position for accurate control.

Output Interfaces:

- **2x 24V 1A for Motor or Relays, optionally also for PWM**: These outputs offer flexibility for controlling motors or relays, optionally via PWM signals for precise power control.

- **2 x Analog Outputs 0 to 5 Volts**: Two analog outputs enable the generation of voltage signals in the range of 0 to 5 volts for driving external devices.

- **1 x 4-pin I2C Connection for Expansion**: The I2C interface allows easy integration of expansion modules for additional functions or interfaces.

- **1 x UART2 TTL for Nextion Display NX3224F028 (2.8 inches)**: The Nextion display provides a clear, user-friendly interface for displaying data and, with touch functionality, allows interaction with the system.

With its powerful ESP32 chip and a wide range of interfaces, the ESP32 Fuzzy PID Controller Board offers a robust and flexible solution for a variety of applications, from temperature control to position monitoring. A complete layout and schematic diagrams can be found in the appendix.

Pin Assignment of the ESP32 from the Test Board:

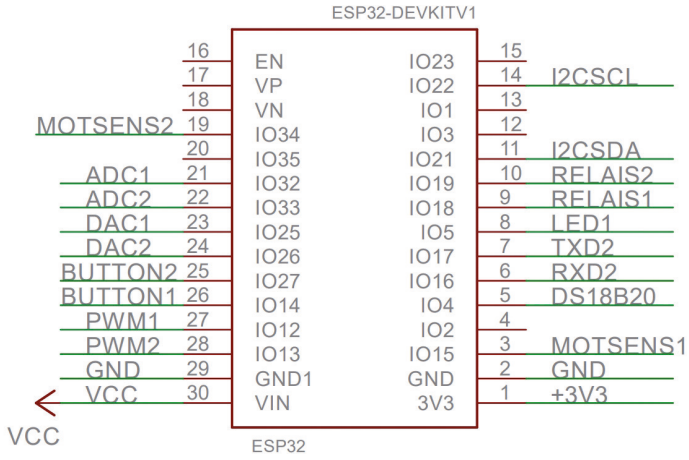


Figure 1.7: ESP32 Pinout

Pin Description	ESP32 Modul Pin Nr.	GPIO pin number	Description
VCC	30		VCC Pin 5V from USB
3V3	1		3.3 Volt pin from voltage regulator
GND	2 / 29		GND Pin
ADC1	21	IO32	Analogue digital converter channel 1
ADC2	22	IO33	Analogue digital converter channel 2
BUTTON1	26	IO14	Button 1 Input

BUTTON2	25	IO27	Button 2 Input
MOTSENS1	3	IO15	Motor Sensor 1 Input
MOTSENS2	19	IO34	Motor Sensor 2 Input
I2CSDA	11	IO21	I^2C Data
I2CSCL	14	IO22	I^2C Clock
RXD2	6	IO16	UART RX Input
TXD2	7	IO17	UART TX Output
DS18B20	5	IO4	OneWire Interface
DAC1	23	IO25	Analog Output 1 0 to 5.0 V
DAC2	24	IO26	Analog Output 2 0 to 5.0 V
PWM1	27	IO12	PWM Output 1
PWM2	28	IO13	PWM Output 2
RELAIS1	9	IO18	Activation for relays 1
RELAIS2	10	IO19	Activation for relays 2
LED1	8	IO5	LED Board

Table 1.2: Pin assignment for the GPIO ESP32 module

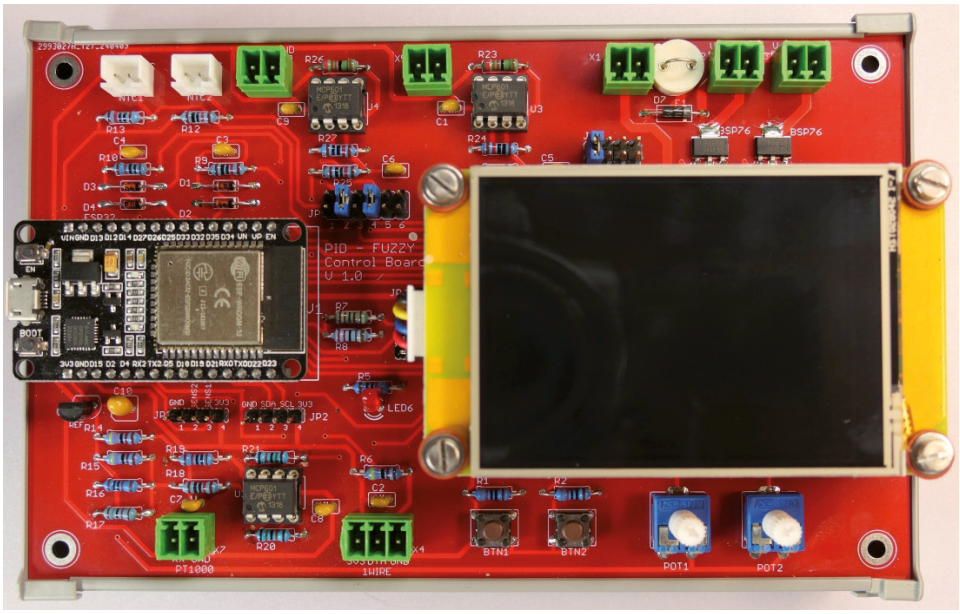


Figure 1.8: ESP32 Fuzzy PID Board

1.6 ESP32 Modbus RTU controller board

This board is also suitable for capturing measurement values such as current, voltage, and power, and is known as the **ESP32BoardModbusRTUControl** board. Schematic, diagrams and layouts are included in the appendix.

The ESP32 board is designed to read energy meters such as the Shelly 3EM via Wi-Fi, as well as the Eastron SDM series (compatible with Controlin) and KBR multimess 96 via RS485 (Modbus RTU). A functioning Wi-Fi connection is required for operation. The data from the meters is displayed on the built-in screen and sent to a server every second in JSON format via HTTP, where it is stored. XML format is also possible. The controller software was developed using the Arduino IDE. Possible protocols include HTTP POST and GET, MQTT, REST, Modbus TCP Server, and other TCP/IP protocols.

The board also allows for the control of inverters to achieve zero injection using the power data.

Technical Specifications:

- **CPU**: ESP32 by Tensilica
- **Power Supply**: 12-24 Volts DC
- **Interfaces**:
  - RS485
  - Wi-Fi 2.4 GHz and Bluetooth
  - UART RX TX for 5V Nextion Display 2.4-inch NX3224T024
- **Outputs**:
  - 2 x 24V DC outputs for relays or contactors
- **Inputs/Outputs**:
  - 1 x 10V analog input (expandable to 0-20V or 0-50V)
  - 1 x 10V analog output (or 0-5V)
  - 1 x OneWire for DS18B20 temperature sensor

Information on the Software of the ESP32 Module:

The module automatically connects to the Wi-Fi network after power-on, using the SSID and password that are stored in the program. If no Wi-Fi network is available, an access point must be installed.

Listing: ESP32_SDM_EnergyMeter.ino

```
// Wlan Parameter
char* ssid = "FRITZ!Box 7590 RI";
char* password = "52x6165x80538x77x967";
```

You then have the option of either reading the Shelly 3EM via WLAN or the Eastron SDM or Controlin meter via Modbus RTU.

Here is an example of querying the Shelly 3EM every second:

```
// Shelly 3 em
String serverName1 = "http://192.168.178.40/emeter/0";
/// Answer from Shelly
{"power":15.32,"pf":0.25,"current":0.27,"voltage":226.96,"is_
valid":true,"total":2.0,"total_returned":0.0}
String serverName2 = "http://192.168.178.40/emeter/1";
// Answer from Shelly
{"power":49.16,"pf":0.78,"current":0.27,"voltage":225.98,"is_
valid":true,"total":5.9,"total_returned":0.0}
String serverName3 = "http://192.168.178.40/emeter/2";
/// Answer from Shelly
{"power":11.56,"pf":0.40,"current":0.13,"voltage":227.10,"is_
valid":true,"total":2.0,"total_returned":0.0}
```

The data is converted from JSON format to float values, and the total performance is calculated.

The current timestamp is then queried from a time server and the data is sent to the Firebase or Postgre SQL Server.

An RS485 energy meter is connected via the UART1 and the following library:

https://github.com/reaper7/SDM_Energy_Meter

```
// Nextion Display
#define RXD2 16
#define TXD2 17

// RS485 Modbus RTU
#define RXD1 19
#define TXD1 18
#define REDE1 26

// Init RS485 Modbus RTU
SDM sdm(Serial1, SDM_UART_BAUD, REDE1, SERIAL_8N1, RXD1, TXD1);
```

The following Modbus registers are read out:

```
PowerArr[0] = sdm.readVal(SDM_PHASE_1_VOLTAGE);
PowerArr[1] = sdm.readVal(SDM_PHASE_1_CURRENT);
PowerArr[2] = sdm.readVal(SDM_PHASE_1_POWER);
PowerArr[3] = sdm.readVal(SDM_FREQUENCY);

PowerArr[4] = sdm.readVal(SDM_IMPORT_ACTIVE_ENERGY);
PowerArr[5] = sdm.readVal(SDM_EXPORT_ACTIVE_ENERGY);
```

For Modbus RTU meters from other manufacturers, the registers must be adapted and tests carried out.

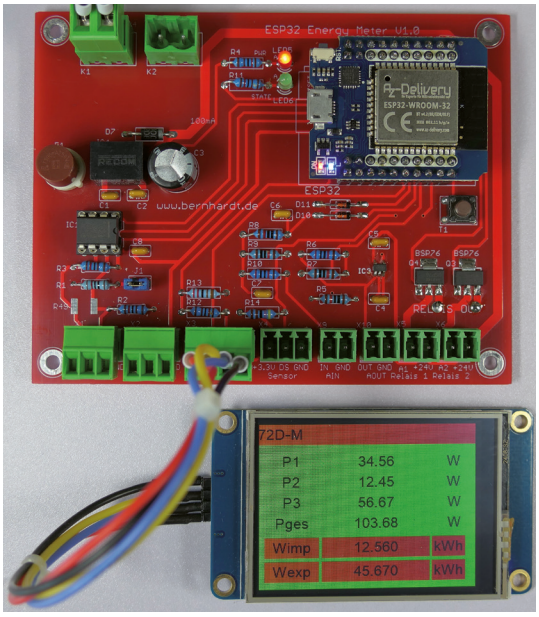


Figure 1.9: ESP32 Modbus RTU Board

The appendix also contains circuit diagrams and layouts for an Arduino Nano board and an Arduino Mega board with four digital and four analogue lines.

1.7 Programming the Arduino boards

A development environment is available for programming the boards, which can be downloaded for free from the Arduino website.

The software is available for Windows, Linux, and macOS.

The website address is: https://www.arduino.cc/en/software

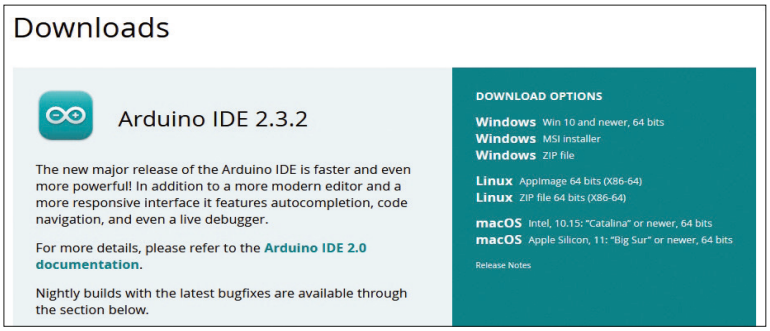


Figure 1.10: Download Arduino IDE

After installing the IDE, missing boards such as the ESP32 can be installed through the Board Manager settings. A list of installable Arduino boards can be found here:

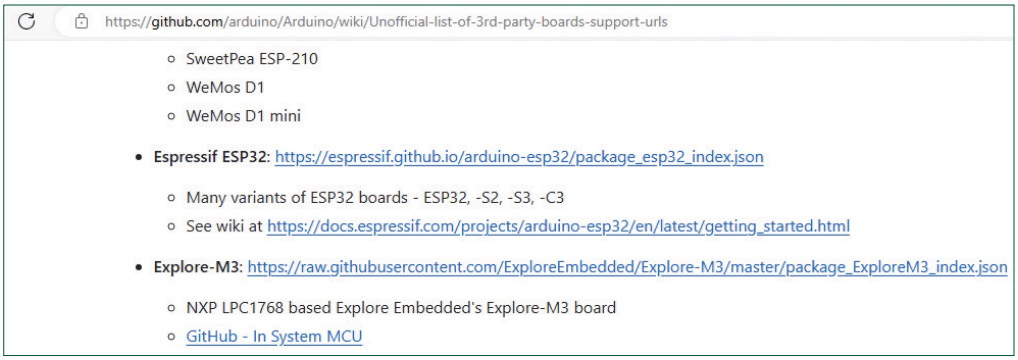


Figure 1.11: ESP32 Github website

Unofficial list of 3rd party boards support urls · arduino/Arduino Wiki · GitHub

1. Open the **Arduino IDE 2.3.2** on your computer.

2. Go to 'Tools' > 'Board' > 'Board Manager'.

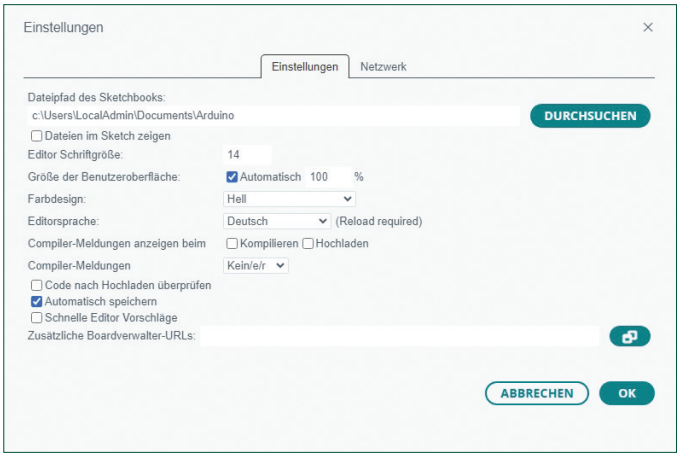


Figure 1.12: Settings in the Arduino IDE

3. Enter 'ESP32' in the search bar at the top right of the board manager and press Enter.

4. A list of available ESP32 boards should appear. Select the ESP32 board you want to use (e.g., 'ESP32 Dev Module').

5. Click on the Install button next to the ESP32 board to install the board support.

6. Wait until the installation is complete. This will take a few minutes.

7. Once the installation is complete, you can select the ESP32 board from the list of available boards under 'Tools' > 'Board'.

After the ESP32 board is successfully installed, you can use it like any other board in the Arduino IDE and write and upload programs for the ESP32.

1.8 The Nextion graphic display

To display the measurement data in the examples, we use a 2.8 inch graphic display NX3224F028_001 from Nextion. The big advantage over other displays, such as those with an IL9341 graphics chip, is the control with the UART via the RX/TX lines and, above all, the simple programming.

A free development environment for the PC is available from Nextion for programming the display. This can be used to design the user interface for the display on the PC. Many graphic elements such as buttons, text boxes, labels etc. are available.

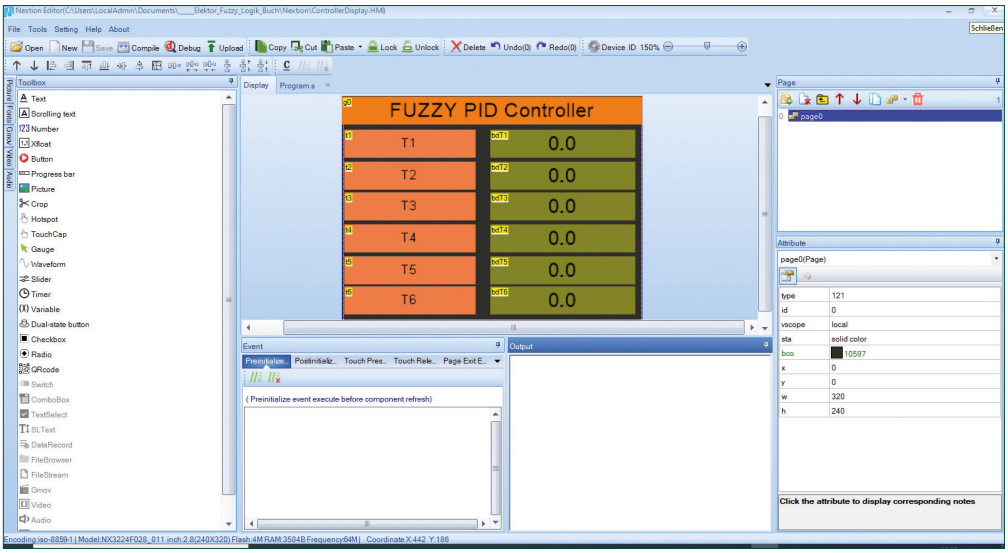


Figure 1.13: Nextion Editor IDE

Also included in the IDE is a debug function and an option to transfer the TFT file to the display. A simple USB-UART module can be used for this (upload).

A second option is to save the generated TFT file to a micro-SD card. This menu item can be found under 'File' 'TFT file output.' The SD card is then inserted into the SD slot of the display and voltage is applied. The program is now transferred from the display to the microcontroller. The card is then removed again. This saves you having to plug and unplug the display when it is mounted on the ESP32 board.

The ControllerDisplay.HMI and ControllerDisplay.tft files are stored in the 'Nextion' directory.

You can find lots more information and tutorials on the Nextion website.

1.9 Test software for the ESP32 board

This program is for initializing the hardware. The timer is initialized to 10 ms using the Ticker.h library. Then the two UART for the USB connection to the PC and UART2 for the Nextion display. In the timer the variable counter10ms is increased, if the value 100 is reached it sets a flag oneSecond which is queried in the loop(). This provides an exact time cycle for the output of the second counter. The program can be used as a 'basic framework' for further programs.

Listing 1: Arduino_UART2_NextionDisplay.ino

```
//-----------------------------------------------------------------------
//
//            Arduino Testsoftware
//            ====================
//
//      Testprogramm for Arduino ESP32 PID Fuzzy Control Board
//      10ms Timer Interrupt Function
//      Init UART2 for Nextion Display
//      Display seconds on Display NX3224F028_001
//
//  Date:     28/04/2024
//  Author:   Josef Bernhardt
//  Hardware: Arduino (ESPRESSIF) ESP32
//
//  File:     Arduino_UART2_NextionDisplay.ino
//-----------------------------------------------------------------------
#include <Arduino.h>
#include <Ticker.h>

int LED_BUILTIN = 2;

// ESP32 UART 2 Pins
#define RXD2 16
#define TXD2 17

// I2C Pins
#define I2CSCL 22
#define I2cSDA 21

// ESP32 ADC Pins
#define ADC1 32
#define ADC2 33
// Motor Sensor Pins
#define MOTSENS1 15
#define MOTSENS2 34
// Buttons
#define BTN1 14
#define BTN2 27

// Relay and Mosfet (BSP76) Pins
#define RELAIS1 18
#define RELAIS2 19
// LED1
#define LED1 5
```

```
// PWM1 and PWM2 Pins
#define PWM1 12
#define PWM2 13
// DAC Outputs 0..5V
#define DAC1 25
#define DAC2 26
// One Wire
#define DS18B20 4
```

In this code, a `ticker` object called `timer` is used to call a timer function every 10 milliseconds. A counter (`counter10ms`) is incremented by 1 with each call, and when this counter reaches the value 100, which corresponds to one second, it is reset. The variable `OneSecond` is set to `true` to indicate that one second has elapsed. The second counter (`CounterSeconds`) is used in loop() to call the output every second. OneSecond is set to false for the next run.

```
// Define 10ms Period
const unsigned long timerInterval = 10;

// Ticker-Objekt for the Timer-Funktion
Ticker timer;

// Counter
int counter10ms=0;

// When 1 second is elapsed
bool OneSecond = false;

// Counter for seconds
int CounterSeconds = 0;

// Timer-Funktion
// Called every 10 ms
void timerCallback()
{
  // Inc 10ms Counter
  counter10ms++;
  // If one Second
  if(counter10ms==100)
  {
     counter10ms = 0;
     // One Second
     OneSecond = true;
  }
}
```

This function sends data to a Nextion display to update the text of a textbox. It uses the serial interface `Serial2` to send a command that selects the textbox (`textbox`) and inserts the new text (`float`). The text is enclosed in quotes and the command is terminated with three terminating bytes (`0xFF`) to mark the end of the command. This enables the Nextion display to show the text correctly.

```
// Sends Data to Nextion Display in a textbox
void Nextion_Send_Text_to_Textbox(char *textbox,String sfloat)
{
        Serial2.write(textbox);
        Serial2.write(".txt=\"");
        Serial2.print(sfloat);
        Serial2.write("\"");
        Serial2.write(0xFF);
        Serial2.write(0xFF);
        Serial2.write(0xFF);
}

// Initializing the hardware
void setup()
{
  // Init USB UART
  Serial.begin(115200);
  // Nextion Display
  // TFTSerial.begin(9600);
  Serial2.begin(9600, SERIAL_8N1, RXD2, TXD2);
  // Init Hardware
  pinMode(LED_BUILTIN, OUTPUT);
  pinMode(LED1, OUTPUT);
  pinMode(RELAIS1, OUTPUT);
  pinMode(RELAIS2, OUTPUT);
  pinMode(BTN1, INPUT);
  pinMode(BTN2, INPUT);

  // Clear Textboxes
  Nextion_Send_Text_to_Textbox("t1","Counter");
  Nextion_Send_Text_to_Textbox("t2"," ");
  Nextion_Send_Text_to_Textbox("t3"," ");
  Nextion_Send_Text_to_Textbox("t4"," ");
  Nextion_Send_Text_to_Textbox("t5"," ");
  Nextion_Send_Text_to_Textbox("t6"," ");

  Nextion_Send_Text_to_Textbox("txtT2"," ");
  Nextion_Send_Text_to_Textbox("txtT3"," ");
  Nextion_Send_Text_to_Textbox("txtT4"," ");
  Nextion_Send_Text_to_Textbox("txtT5"," ");
```

```
  Nextion_Send_Text_to_Textbox("txtT6"," ");

  // Init Timer with 10ms
  timer.attach_ms(timerInterval, timerCallback);
}
```

The `loop()` function checks whether a second has elapsed based on the `OneSecond` variable. If this is the case, `OneSecond` is reset to `false` and the `CounterSeconds` counter is incremented by one. The new value of `CounterSeconds` is sent to the serial monitor via the serial interface and sent to a text box on the Nextion display. In addition, the status of two LEDs is switched: the internal LED and another LED (`LED1`). The code thus ensures regular updates and visual feedback via the LEDs.

```
void loop()
{

  // Check if one second has passed
  if (OneSecond) {
    // Reset the boolean variable OneSecond
    OneSecond = false;
    CounterSeconds++;

    // Sends variable to USB UART and Seriell Monitor
    Serial.print("Timer: ");
    Serial.println(CounterSeconds);

    // Sends variable to display
    Nextion_Send_Text_to_Textbox("txtT1",String(CounterSeconds));

    // Switching LED
    digitalWrite(LED_BUILTIN,!digitalRead(LED_BUILTIN));
    digitalWrite(LED1,!digitalRead(LED1));

  }
  // Additional code for the main loop
}
```

1.10 Program example with the ESP32

In the next program, the buttons are queried. The button on BTN1 is queried and debounced with the function read_btn1(). In the timer function, which is called every 10 ms, the push-button at BTN2 is read. If BTN2 is at low level three times in succession, i.e., 30 ms, the key_press variable is set to True. The key press is queried with the get_key() function. When the information is pressed, the values of the state variables state1 or state2 are changed to save the current state and then switch the RELAIS1 and RELAIS2 outputs. Press once to switch on, press again to switch off. These functions will be used later to set the setpoint of controllers.

Listing 2: Arduino_TimerTick_USB_Hardware.ino

```
//---------------------------------------------------------------------
//
//            Arduino Timer Test
//            ==================
//
//      Timer Testprogramm for Arduino ESP32 PID Fuzzy Control Board
//      10ms Timer Interrupt Function
//      Read 2 Buttons and switch Relais Pin with BTN1 and LED1 with BTN2
//
//  Date:     08/05/2024
//  Author:   Josef Bernhardt
//  Hardware: Arduino (ESPRESSIF) ESP32
//
//  File:     Arduino_TimerTick_USB_Hardware.ino
//---------------------------------------------------------------------

#include <Arduino.h>
#include <Ticker.h>

int LED_BUILTIN = 2;

// ESP32 UART 2 Pins
#define RXD2 16
#define TXD2 17

// I2C Pins
#define I2CSCL 22
#define I2cSDA 21

// ESP32 ADC Pins
#define ADC1 32
#define ADC2 33
// Motor Sensor Pins
#define MOTSENS1 15
#define MOTSENS2 34
// Buttons
#define BTN1 14
#define BTN2 27

// Relay and Mosfet (BSP76) Pins
#define RELAIS1 18
#define RELAIS2 19
// LED1
#define LED1 5
```

```
// PWM1 and PWM2 Pins
#define PWM1 12
#define PWM2 13
// DAC Outputs 0..5V
#define DAC1 25
#define DAC2 26
// One Wire
#define DS18B20 4

// Define 10ms Period
const unsigned long timerInterval = 10;

// Ticker-Objekt for the Timer-Funktion
Ticker timer;

// Counter 10 for Timer
int counter10ms=0;

// When 1 second is elapsed
bool OneSecond = false;

// State for Buttons
int state1 = 0;
int state2 = 0;

// Variable for BTN2 in Timer
int key_state = 0;
int key_counter = 0;
int key_press = 0;
```

In addition to the time measurement, the status of the BTN2 button is queried in the timer callback function. To ensure more stable button detection, the button must be pressed for a total of at least 30 milliseconds. When this counter is reached, the push-button status is updated, and key_press is set to 1 if the push-button is pressed for 30 ms. This method ensures that the button is actually pressed for a certain time before it is recognized as pressed.

```
// Timer-Funktion
// Called every 10 ms
void timerCallback()
{
    // Inc 10ms Counter
    counter10ms++;
    // If one Second
    if(counter10ms==100)
    {
```

```
        counter10ms = 0;
        // One Second
        OneSecond = true;
    }

    // Read Button 2 in Timer function
    int input = digitalRead(BTN2);
    if( input != key_state )
    {
      key_counter++;
      if( key_counter == 3 )
      {
        key_counter = 0;
        key_state = input;
        if( input ) key_press = 1;
      }
    }
    else key_counter = 0;
}

// Read Button 2
int get_key()
{
      int result;
      result = key_press;
      key_press = 0;
      return result;
}
```

The `read_btn1()` function checks the status of the BTN1 push-button and recognizes whether it has been pressed. If the button changes from a `LOW` state to `HIGH` (rising edge), a short debounce period of 20 milliseconds is observed and the status of the button is read again. If the button is still 'HIGH,' 'key' is set to 1 to indicate that the button has been pressed for 20 ms. Finally, the previous button status is saved for the next check and the value of `key` is returned.

```
unsigned char read_btn1()
{
    unsigned char key = 0;
    unsigned char btn;
    static unsigned char btn_old = 0; // Set initial value to 0 since the default
state of the button is not pressed (logically inverted)
    btn = digitalRead(BTN1);          // Using digitalRead to read the state of
the button
                                      // Start of button check Detect rising edge
    if (btn && (!btn_old))            // If the current state of the button is
```

```
HIGH and the previous state was LOW
    {
        // Debouncing
        delay(20);
        // Read again
        btn = digitalRead(BTN1);
        // If still HIGH, return 1
        if (btn)
        {
            key = 1;
        }
    }
    // Remember old value
    btn_old = btn;
    return key;
}


// Initializing the hardware
void setup()
{
  // Init USB UART
  Serial.begin(115200);
  // Init Hardware
  pinMode(LED_BUILTIN, OUTPUT);
  pinMode(LED1, OUTPUT);
  pinMode(RELAIS1, OUTPUT);
  pinMode(RELAIS2, OUTPUT);
  pinMode(BTN1, INPUT);
  pinMode(BTN2, INPUT);

  // Init Timer with 10ms
  timer.attach_ms(timerInterval, timerCallback);
}
```

The code in switch(state1) and switch (state2) implements two state machines to control two outputs RELAIS1 and LED1, each with two states.

1st state machine for state1:
State 0: In state 0, the automaton checks whether the BTN1 button has been pressed (function read_btn1(returns 1)). If this is the case, the machine switches to state 1 and outputs a message via the serial interface.

State 1: In state 1, the system checks again whether the BTN1 button has been pressed. If this is the case, the machine switches back to state 0 and outputs a corresponding message.

2nd state machine for state2:
State 0: Here the machine checks whether the BTN2 button has been pressed (function get_key() returns 1). If this is the case, the machine switches to state 1 and outputs a message via the serial interface.

State 1: In state 1, the system checks whether the BTN2 button has been pressed again. If this is confirmed, the machine switches back to state 0 and outputs a message.

Each state machine monitors the state of the respective button and switches between two states when the button is pressed.

```
void loop()
{

    switch (state1) {
        case 0:
            if (read_btn1()) {
                state1 = 1;
                Serial.println("Button 1 Pressed, state1 = 1");
            }
            break;
        case 1:
            if (read_btn1()) {
                state1 = 0;
                Serial.println("Button 1 Pressed, state1 = 0");
            }
            break;
    }

    switch (state2) {
        case 0:
            if (get_key()) {
                state2 = 1;
                Serial.println("Button 2 Pressed, state1 = 1");
            }
            break;
        case 1:
            if (get_key()) {
                state2 = 0;
                Serial.println("Button 2 Pressed, state1 = 0");
            }
            break;
    }


    // Switching relays with Buttons
```

```
    // digitalWrite(RELAIS1,digitalRead(BTN1));
    if (state1) digitalWrite(RELAIS1,1); else digitalWrite(RELAIS1,0);

    if (state2) digitalWrite(LED1,1); else digitalWrite(LED1,0);


    // if BTN2 is pressed then output 1
    digitalWrite(RELAIS2,digitalRead(BTN2));

    // Check if one second has passed
    if (OneSecond) {
      // Reset the boolean variable OneSecond
      OneSecond = false;
      Serial.println("Timer Test... ");
      // Switching LED
      digitalWrite(LED_BUILTIN,!digitalRead(LED_BUILTIN));
  }
  // Additional code for the main loop
}
```

1.11 Arduino ESP32 controller 12 bit A/D converter test

When testing the ADC of the ESP32 controller, it was noticed that the measured voltages were not correct. After testing with a calibrator, a table was created in Excel to obtain an overview of the deviation. Using these values, an array could be generated for an Arduino program to write a function for the exact calculation of the output values. This now enables very precise use of the ADC.

	A	B	C	D	E
1	Voltage calibrator	ADC1 Measurement	Caluclated Value		Error
2	0,00	0	0		0
3	0,10	0	124		124
4	0,20	55	248		193
5	0,30	174	372		198
6	0,50	406	621		215
7	0,75	704	931		227
8	1,00	1003	1241		238
9	1,25	1301	1552		251
10	1,50	1606	1862		256
11	2,00	2202	2482		280
12	2,50	2823	3103		280
13	3,00	3745	3724		-21
14	3,10	3984	3848		-136
15	3,15	4095	3910		-185
16	3,30	4095	4096		1

Figure 1.14: Calibrator Table

The right-hand column clearly shows that there are quite large deviations that need to be corrected.

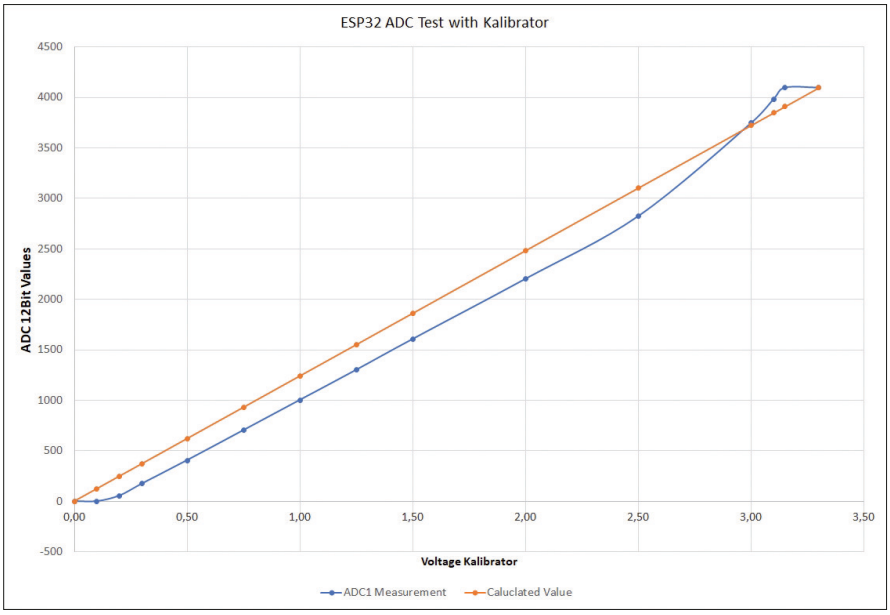


Figure 1.15 ADC characteristic curve

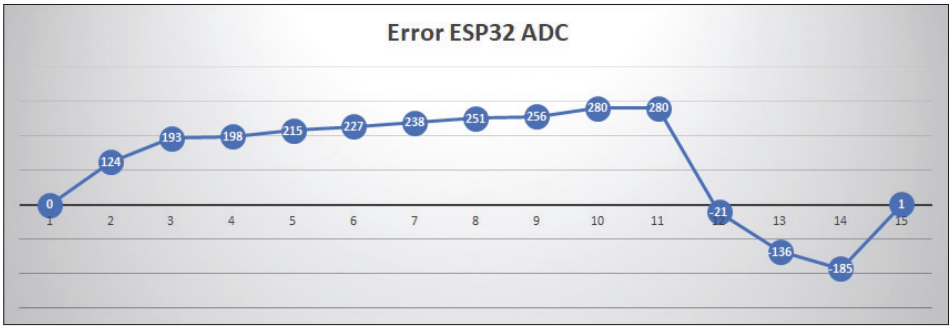


Figure 1.16 ADC Error

The values from the calibrator are entered in the array float Ua[]. The measured values, which are shown in the Nextion display or in the serial monitor of the Arduino, in the array ADC[]. The ADC is first read out in the main program. The read_value_linear() function is used for linear interpolation. In the line

```
while (j < n-1 && xn > x[j+1]) j++;
```

the interval in which the ADC voltage lies is checked and then linearly interpolated with the corresponding values. The calculated voltage value in the range from 0 to 3.3 V is output.

Listing 3: Arduino_ADC_Interpolate.ino

```
//---------------------------------------------------------------------
//
//            Arduino Timer and ADC Test and interpolation
//            ============================================
//
//      Testprogramm for Arduino ESP32 PID Fuzzy Control Board
//      10ms Timer Interrupt Function
//      Init UART2 for Nextion Display
//      Display ADC1 and ADC2 Values on Display NX3224F028_001
//
//      With a calibrator, exact voltages were applied and the values
//      were logged from the ADC. The conversion of ADC digits to
//      voltage is done with a stepwise interpolation
//
//
//  Date:     28/04/2024
//  Author:   Josef Bernhardt
//  Hardware: Arduino (ESPRESSIF) ESP32
//
//  File:     Arduino_ADC_Interpolate.ino
//---------------------------------------------------------------------

#include <Arduino.h>
#include <Ticker.h>

int LED_BUILTIN = 2;

// ESP32 UART 2 Pins
#define RXD2 16
#define TXD2 17

// I2C Pins
#define I2CSCL 22
#define I2cSDA 21

// ESP32 ADC Pins
#define ADC1 32
#define ADC2 33
// Motor Sensor Pins
#define MOTSENS1 15
#define MOTSENS2 34
// Buttons
#define BTN1 14
#define BTN2 27
```

```
// Relay and Mosfet (BSP76) Pins
#define RELAIS1 18
#define RELAIS2 19
// LED1
#define LED1 5
// PWM1 and PWM2 Pins
#define PWM1 12
#define PWM2 13
// DAC Outputs 0..5V
#define DAC1 25
#define DAC2 26
// One Wire
#define DS18B20 4

// Define 10ms Period
const unsigned long timerInterval = 10;

// Ticker-Objekt for the Timer-Funktion
Ticker timer;

// Counter
int counter10ms=0;

// When 1 second is elapsed
bool OneSecond = false;

// Counter for seconds
int CounterSeconds = 0;

// 12 Bit ADC-Values from 0 to 4095
float ADC[] = {0, 0, 55, 174, 406, 704, 1003, 1301, 1606, 2202, 2823, 3745, 3984,
4095, 4095};

// (Ua) in volts values applied to ADC by calibrator
float Ua[] = {0.00, 0.10, 0.20, 0.30, 0.50, 0.75, 1.00, 1.25, 1.50, 2.00, 2.50,
3.00, 3.10, 3.15, 3.30};

int n_adc = 15;
```

The function `readvalue_linear_voltage()` calculates an interpolated value for a given input value (`xn`) based on two arrays of `x` and `y` values. First, the interval in which the value `xn` lies is determined by iterating through the `x` values. Once the appropriate interval has been found, the value for `xn` within this interval is calculated. The interpolation is performed by calculating a weighted average between the neighbouring `y` values. The calculated value is then returned as the result.

```
// Linear interpolation
// Input Parameters:
// n = Number of values -1
// x = x Array
// y = y Array
// xn = Value to be calculated
float readvalue_linear_voltage(int n,float *x,float *y, float xn)
{
    int j = 0;
    // Determine interval of xn
    while (j < n-1 && xn > x[j+1]) j++;
    // interpolate xn
    return ((y[j+1]-y[j]) / (x[j+1]-x[j])) * (xn-x[j]) + y[j];
}


// Timer-Funktion
// Called every 10 ms
void timerCallback()
{
  // Inc 10ms Counter
  counter10ms++;
  // If one Second
  if(counter10ms==100)
  {
     counter10ms = 0;
     // One Second
     OneSecond = true;
  }
}

// Sends Data to Nextion Display in a textbox
void Nextion_Send_Text_to_Textbox(char *textbox,String sfloat)
{
        Serial2.write(textbox);
        Serial2.write(".txt=\"");
        Serial2.print(sfloat);
        Serial2.write("\"");
        Serial2.write(0xFF);
        Serial2.write(0xFF);
        Serial2.write(0xFF);
}

// Initializing the hardware
void setup()
{
```

```
  // Init USB UART
  Serial.begin(115200);
  // Nextion Display
  // TFTSerial.begin(9600);
  Serial2.begin(9600, SERIAL_8N1, RXD2, TXD2);
  // Init Hardware
  pinMode(LED_BUILTIN, OUTPUT);
  pinMode(LED1, OUTPUT);
  pinMode(RELAIS1, OUTPUT);
  pinMode(RELAIS2, OUTPUT);
  pinMode(BTN1, INPUT);
  pinMode(BTN2, INPUT);

  pinMode(ADC1, INPUT);
  pinMode(ADC2, INPUT);

  Nextion_Send_Text_to_Textbox("t1","Voltage 1");
  Nextion_Send_Text_to_Textbox("t2"," Float V1 ");
  Nextion_Send_Text_to_Textbox("t3"," ");
  Nextion_Send_Text_to_Textbox("t4","Voltage 2");
  Nextion_Send_Text_to_Textbox("t5"," Float V2 ");
  Nextion_Send_Text_to_Textbox("t6"," ");

  // Init Timer with 10ms
  timer.attach_ms(timerInterval, timerCallback);
}
```

In the loop() function, the values of two analogue inputs, ADC1 and ADC2, are read and stored in the variables voltage1 and voltage2. The value of voltage1 is then interpolated by the readvalue_linear_voltage() function in order to obtain a more precise value, which is saved in u1. The same is done for voltage2, whose interpolated value is stored in u2. The interpolation uses the given data arrays ADC[] and Ua[], as well as the current voltage value, to calculate more precise measured values.

```
void loop()
{
  // Read ADC1 value
  int voltage1 = analogRead(ADC1);
  // Read ADC2 value
  int voltage2 = analogRead(ADC2);
  // Interpolate voltage 1
  float u1 = readvalue_linear_voltage(n_adc-1,ADC,Ua,(float)voltage1);
  // Interpolate voltage 2
  float u2 = readvalue_linear_voltage(n_adc-1,ADC,Ua,(float)voltage2);

  // Check if one second has passed
```

```
  if (OneSecond) {
    // Reset the boolean variable OneSecond
    OneSecond = false;
    CounterSeconds++;

    // Sends variable to USB UART and Seriell Monitor

    Serial.print("ADC1: ");
    Serial.print(voltage1);
    Serial.print(" U1: ");
    Serial.print(String(u1,3));

    Serial.print(" ADC2: ");
    Serial.print(voltage1);
    Serial.print(" U2: ");
    Serial.println(String(u2,3));

    // Sends variable to display
    Nextion_Send_Text_to_Textbox("txtT1",String(voltage1));
    Nextion_Send_Text_to_Textbox("txtT2",String(u1,3));
    Nextion_Send_Text_to_Textbox("txtT3"," ");
    // Sends variable to display
    Nextion_Send_Text_to_Textbox("txtT4",String(voltage2));
    Nextion_Send_Text_to_Textbox("txtT5",String(u2,3));
    Nextion_Send_Text_to_Textbox("txtT6"," ");
    // Switching LED
    digitalWrite(LED_BUILTIN,!digitalRead(LED_BUILTIN));
    digitalWrite(LED1,!digitalRead(LED1));

  }
  // Additional code for the main loop
}
```

1.12 CodeBlocks development environment for Ansi C

To save time, it is a good idea to use a development environment such as CodeBlocks to test programs in Ansi C.

CodeBlocks is a popular, free IDE for ANSI C, which is particularly known for its user-friendliness and cross-platform support. Programs created on the PC can also be used on the Raspberry PI. It offers an intuitive user interface that simplifies the creation, editing and debugging of C projects. With its powerful code completion and syntax highlighting, it improves code quality and increases productivity. The integrated support for various compilers such as GCC enables flexible development. CodeBlocks is also extensible, so additional plugins and tools can be integrated to extend its functionality.

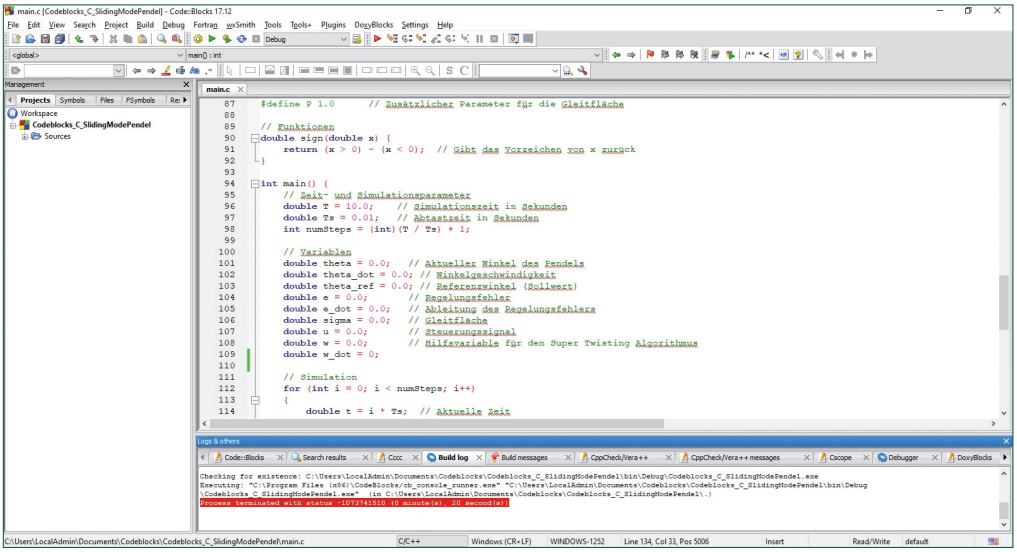


Figure 1.17: Codeblocks IDE

Many of the examples in this book for the Arduino were first tested on the PC and then adapted for the Arduino.

On the following page you will find different versions for different operating systems (Windows, macOS, Linux). Select the appropriate version for your system and follow the instructions for download and installation.

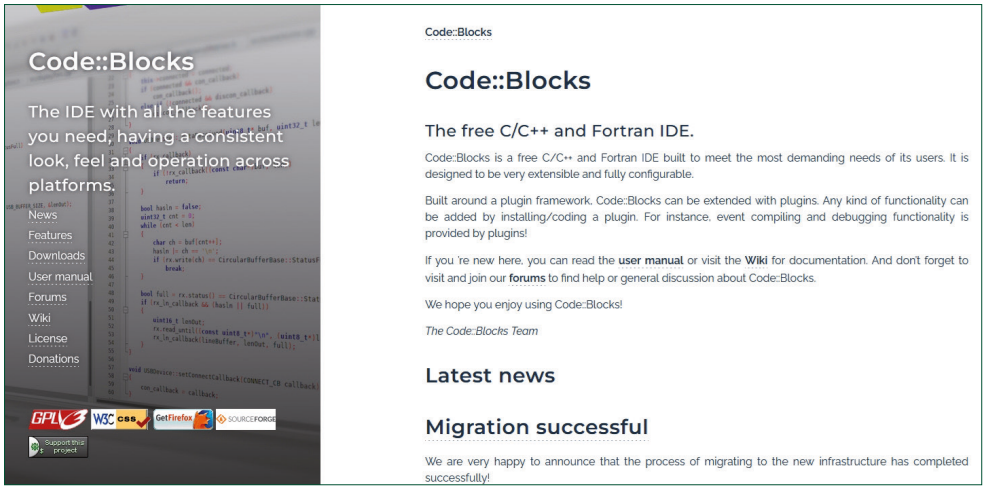


Figure 1.18: Codeblocks download site

An overview of the code blocks and program examples can be found in the appendix.

Download Link of CodeBlocks IDE: https://www.codeblocks.org/

1.13 Serial Data Reader

The Serial Data Reader program was created for the graphical display of data, in particular for the various two-point, PID and fuzzy controllers.

The software was created with the development of the environment Sharp Develop. It receives data from the COM port (USB). It can display a maximum of six different measured values in one data set. The values are displayed both as text and as graphics in a chart.

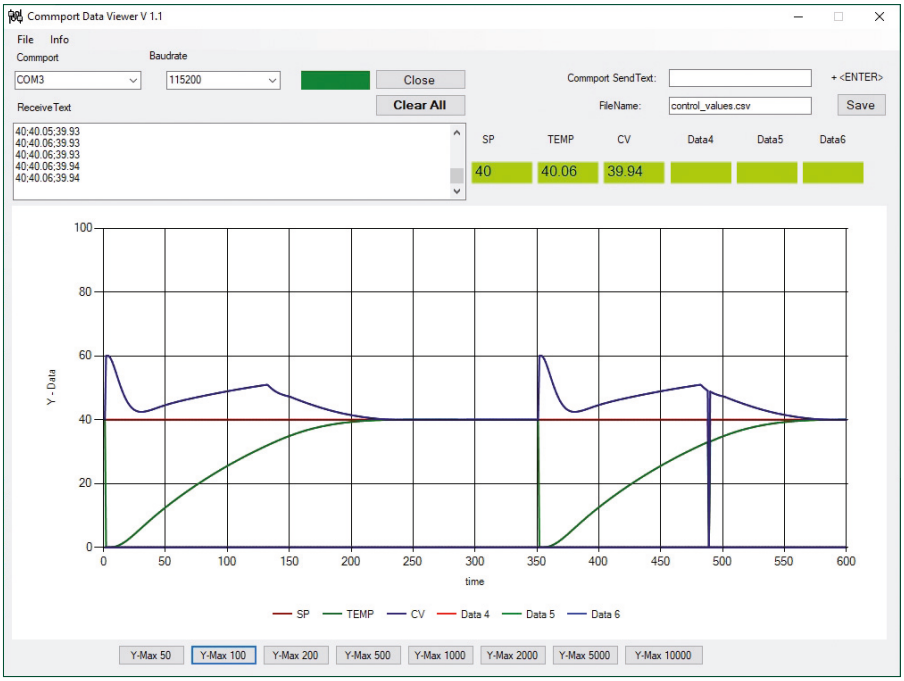


Figure 1.19: COM port Data Viewer

1.14 Serial Port Data Reader

It can be found in the download archive of the book, as well as the program SerialPortReader. This is very suitable for transferring data via the clipboard and preparing it for Excel.

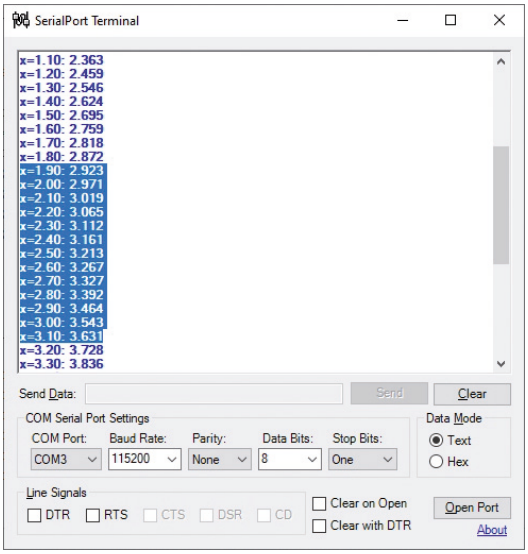


Figure 1.20: Serial Data Reader

1.15 The Espressif ESP32 as a WebSocket server

The WebSocket protocol enables bidirectional, permanent communication between a client and a server via a single TCP connection. In contrast to HTTP, which opens and closes a new connection for each request, the WebSocket connection remains open, which enables more efficient and faster data exchange. Communication begins with an HTTP handshake, after which the connection is switched to the WebSocket protocol. The data is then transferred into small frames, which enables low latency and high data throughput.

WebSockets are particularly useful for real-time applications installed on microcontrollers such as the ESP32. This also makes it possible to program a web interface for fuzzy and PID controllers.

The web interface of the WebSocket test program for the ESP32:

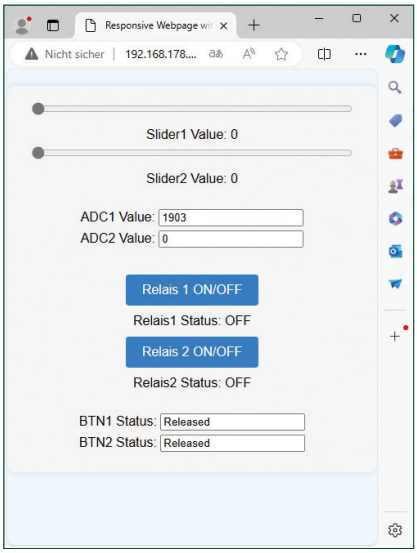


Figure 1.21: Example HTML page with ESP32

Listing 4: Arduino_ESP32_WebsocketSlider.ino

```
//----------------------------------------------------------------------
//
//            Arduino ESP32 Websocket Example NTC Relais 1
//            ============================================
//
//      WEB Software for Arduino ESP32 PID Fuzzy Control Board
//
//      Example with Websockets
//
//      https://de.wikipedia.org/wiki/WebSocket
//
//      Output Slider1 and Slider2 values to PWM1 and PWM2
//
//      Read ADC1 and ADC2
//
//      switch Relais 1 and Relais 2
//
//      Reading BTN1 and BTN2
//
//  Date:     18/7/2024 13.17
//  Author:   Josef Bernhardt
//  Hardware: Arduino (ESPRESSIF) ESP32  // ESP32 DEV Module
//  IDE:      Arduino IDE 2.3.2
//  File:     Arduino_ESP32_WebsocketSlider.ino
//----------------------------------------------------------------------
```

```
#include <WiFi.h>
#include <AsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include <AsyncWebSocket.h>
#include <ArduinoJson.h>


// WLAN Parameter
const char* ssid = "FRITZ!Box 7590 RI";
const char* password = «5226x1x80x385776xx7»;

// ESP32 ADC
#define ADC1 32
#define ADC2 33

// Relais
#define RELAIS1 18
#define RELAIS2 19

// DAC Outputs 0..5V
#define DAC1 25
#define DAC2 26

#define PWM1 12
#define PWM2 13

// Buttons
#define BTN1 14
#define BTN2 27

// LED1
#define LED1 5
// ESP32 Board blue LED
#define LED_BUILTIN 2

// use 10 or 12 bit precision for LEDC timer
#define PWM_RESOLUTION 10
// use 1000 Hz as a LEDC base frequency
#define PWM_FREQUENZ 1000

AsyncWebServer server(80);
AsyncWebSocket ws("/ws");

int slider1Value = 0;
int slider2Value = 0;
int adc1Value = 0;
```

```
int adc2Value = 0;
bool relais1Status = false;
bool relais2Status = false;
bool btn1Status = false;
bool btn2Status = false;
```

The CSS code in <style> defines a responsive design for a website with a modern, soft color scheme. The main container has a maximum width of 600 pixels and is centered with a subtle shadow effect and rounded corners. The slider elements are given a user-friendly width and a soft blue background color to make them easier to use. Buttons are designed in a dark shade of blue and change color slightly on hover to signal interactivity.

```
const char index_html[] PROGMEM = R"rawliteral(
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Responsive Webpage with Components</title>
  <style>
    /* Stildefinitionen für die responsive Gestaltung */
    body {
      font-family: Arial, sans-serif;
      margin: 0;
      padding: 0;
      background-color: #f0f7fc; /* Sanfter Blauton für den Hintergrund */
    }
    .container {
      max-width: 600px;
      margin: 20px auto;
      padding: 20px;
      background-color: #f5f5f5; /* Sanfter Grauton für den Hintergrund des
Containers */
      border-radius: 10px;
      box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
    }

    input[type="range"] {
      width: calc(100% - 20px);
      margin: 0 10px;
      margin-bottom: 10px;
      border-radius: 5px;
      background-color: #eaf2f8; /* Sanfter Blauton für den Hintergrund der
Slider */
    }
```

```
    button {
      display: block;
      margin: 10px auto;
      padding: 10px 20px;
      font-size: 16px;
      background-color: #357ebd; /* Dunkler Blauton für die Buttons */
      color: white;
      border: none;
      border-radius: 4px;
      cursor: pointer;
    }

    button:hover {
      background-color: #2d6ca3; /* Dunkler Blauton für die Buttons 3EMHover-
Zustand */
    }

    p {
      margin: 5px auto;
      text-align: center;
    }
  </style>
</head>
<body>
```

This HTML code represents a user interface for controlling an ESP32 WebSocket server. The title 'ESP32 WebSocket Server' is displayed in a container, followed by two sliders, which can take values between 0 and 1023. The current value is displayed in real time under each slider. There are also two text fields that display the values of two ADCs (analogue-to-digital converters) and two buttons that can be used to switch relays on and off. The status of the relays and two buttons is also displayed in text fields.

```
  <div class="container"><h1><p>ESP32 WebSocket Server</p></h1></div>

  <div class="container">
  <input type="range" min="0" max="1023" value="0" id="slider1">
  <p>Slider1 Value: <span id="slider1Value">0</span></p>

  <input type="range" min="0" max="1023" value="0" id="slider2">
  <p>Slider2 Value: <span id="slider2Value">0</span></p>
  <br>
  <p>ADC1 Value: <input type="text" id="adc1Value" readonly></p>
  <p>ADC2 Value: <input type="text" id="adc2Value" readonly></p>
  <br>
  <button id="button1">Relais 1 ON/OFF</button>
  <p>Relais1 Status: <span id="relais1Status">OFF</span></p>
```

```
<button id="button2">Relais 2 ON/OFF</button>
<p>Relais2 Status: <span id="relais2Status">OFF</span></p>
<br>
<p>BTN1 Status: <input type="text" id="btn1Status" readonly></p>
<p>BTN2 Status: <input type="text" id="btn2Status" readonly></p>
</div>
```

The following script initializes a WebSocket connection between a client (e.g., a web browser) and a server. It opens a connection to the server, receives messages and updates the user interface based on the data received. The sliders and buttons on the website are programmed to send their values to the server when the user changes them. The data received controls various elements of the interface such as slider values, relay status and button states. If the connection is lost, an attempt is made to automatically re-establish the WebSocket connection after a short pause.

```
<script>
  var gateway = `ws://${window.location.hostname}/ws`;
  var websocket;

  window.addEventListener('load', onLoad);

  function initWebSocket() {
    console.log('Trying to open a WebSocket connection...');
    websocket = new WebSocket(gateway);
    websocket.onopen = onOpen;
    websocket.onclose = onClose;
    websocket.onmessage = onMessage;
  }

  function onOpen(event) {
    console.log('Connection opened');
  }

  function onClose(event) {
    console.log('Connection closed');
    setTimeout(initWebSocket, 2000);
  }

  function onMessage(event) {
    var data = JSON.parse(event.data);
    console.log('Received data: ' + event.data);

    document.getElementById('slider1').value = data.slider1;
    document.getElementById('slider1Value').innerHTML = data.slider1;
    document.getElementById('slider2').value = data.slider2;
```

```
      document.getElementById('slider2Value').innerHTML = data.slider2;
      document.getElementById('adc1Value').value = data.adc1;
      document.getElementById('adc2Value').value = data.adc2;

      document.getElementById('relais1Status').innerHTML = data.relais1 ? "ON" :
"OFF";
      document.getElementById('relais2Status').innerHTML = data.relais2 ? "ON" :
"OFF";

      document.getElementById('btn1Status').value = data.btn1 ? "Pressed" :
"Released";
      document.getElementById('btn2Status').value = data.btn2 ? "Pressed" :
"Released";
    }

    function onLoad(event) {
      initWebSocket();
      initSliders();
      initButtons();
    }

    function initSliders() {
      var slider1 = document.getElementById('slider1');
      var slider1Output = document.getElementById('slider1Value');
      slider1.oninput = function() {
        slider1Output.innerHTML = this.value;
        sendSliderValues();
      }

      var slider2 = document.getElementById('slider2');
      var slider2Output = document.getElementById('slider2Value');
      slider2.oninput = function() {
        slider2Output.innerHTML = this.value;
        sendSliderValues();
      }
    }

    function sendSliderValues() {
      var slider1 = document.getElementById('slider1').value;
      var slider2 = document.getElementById('slider2').value;

      websocket.send(JSON.stringify({
        slider1: slider1,
        slider2: slider2
      }));
    }
```

```
    function initButtons() {
      document.getElementById('button1').onclick = function() {
        websocket.send(JSON.stringify({ pressed1: true }));
      }
      document.getElementById('button2').onclick = function() {
        websocket.send(JSON.stringify({ pressed2: true }));
      }
    }
  </script>
</body>
</html>
)rawliteral";
```

The `handleWebSocketMessage()` function processes incoming WebSocket messages that have been sent as JSON data. It checks whether the data is text data and deserialises it in order to analyze the values it contains. Depending on the values received, hardware components such as DACs, relays and LEDs are controlled. An updated message with the current values of the sliders, ADCs, relays, and buttons is then sent to all WebSocket clients. The function thus enables real-time control and feedback of hardware via WebSocket connections.

```
void handleWebSocketMessage(void *arg, uint8_t *data, size_t len) {
  AwsFrameInfo *info = (AwsFrameInfo*)arg;
  if (info->opcode == WS_TEXT) {
    data[len] = 0;
    Serial.printf("Received data: %s\n", (char*)data);

    // Parse the received JSON data
    DynamicJsonDocument doc(1024);
    DeserializationError error = deserializeJson(doc, (char*)data);

    if (!error) {
      if (doc.containsKey("slider1")) {
        slider1Value = doc["slider1"];
        dacWrite(DAC1, slider1Value);
      }
      if (doc.containsKey("slider2")) {
        slider2Value = doc["slider2"];
        dacWrite(DAC2, slider2Value);
      }
      if (doc.containsKey("pressed1")) {
        relais1Status = !relais1Status;
        digitalWrite(RELAIS1, relais1Status ? HIGH : LOW);
        digitalWrite(LED1, relais1Status ? HIGH : LOW);
      }
```

```
      if (doc.containsKey("pressed2")) {
        relais2Status = !relais2Status;
        digitalWrite(RELAIS2, relais2Status ? HIGH : LOW);
        digitalWrite(LED_BUILTIN, relais2Status ? HIGH : LOW);
      }

      // Send the updated slider, ADC, relais, and button values back to the
WebSocket clients
      String message = "{\"slider1\":" + String(slider1Value) + ",\"slider2\":" +
String(slider2Value) +
                       ",\"adc1\":" + String(adc1Value) + ",\"adc2\":" +
String(adc2Value) +
                       «,\»relais1\»:» + String(relais1Status) + «,\»relais2\»:»
+ String(relais2Status) +
                       ",\"btn1\":" + String(btn1Status) + ",\"btn2\":" +
String(btn2Status) + "}";
      ws.textAll(message);
    }
  }
}
```

The `setup()` function initializes the basic hardware and network connection of the ESP32. First, the serial connection is started, and a WLAN connection is established, whereby the connection status and network details (such as IP address and signal strength) are output. The pins for relays, DACs, LEDs, and buttons are defined as inputs or outputs and set to their initial states. The PWM configuration for the DACs is also set up. As soon as the WLAN connection is established, a WebSocket server is set up to process the WebSocket events and an HTTP server instance is started that responds to requests with an HTML page.

```
void setup() {
  Serial.begin(115200);
  WiFi.begin(ssid, password);
  pinMode(RELAIS1, OUTPUT);
  pinMode(RELAIS2, OUTPUT);
  pinMode(BTN1, INPUT);
  pinMode(BTN2, INPUT);
  // DAC-Pins as output
  pinMode(DAC1, OUTPUT);
  pinMode(DAC2, OUTPUT);
  pinMode(LED1, OUTPUT);
  pinMode(LED_BUILTIN, OUTPUT);
  digitalWrite(LED1, LOW);
  digitalWrite(LED_BUILTIN, LOW);
  digitalWrite(RELAIS1, LOW);
  digitalWrite(RELAIS2, LOW);
  digitalWrite(LED1, LOW);
```

```
  ledcWrite(PWM1, slider1Value);
  ledcWrite(PWM2, slider2Value);
  // Setup timer and attach timer to a led pin
  ledcAttach(PWM1, PWM_FREQUENZ, PWM_RESOLUTION);
  ledcAttach(PWM2, PWM_FREQUENZ, PWM_RESOLUTION);

  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }
  // Print IP-Adress
  Serial.print(„IP-Adresse: „);
  Serial.println(WiFi.localIP());

  // Output of the signal strength (RSSI)
  Serial.print(„Signalstärke (RSSI): „);
  Serial.println(WiFi.RSSI());

  // Print BSSID (MAC-Adress)
  Serial.print(„BSSID: „);
  Serial.println(WiFi.BSSIDstr());

  // Print SSID of WLAN Network
  Serial.print("SSID: ");
  Serial.println(WiFi.SSID());
  Serial.println("Connected to WiFi");

  server.addHandler(&ws);

  ws.onEvent([](AsyncWebSocket *server, AsyncWebSocketClient *client, AwsEventType
type, void *arg, uint8_t *data, size_t len) {
    handleWebSocketMessage(arg, data, len);
  });

  server.on("/", HTTP_GET, [](AsyncWebServerRequest *request) {
    request->send_P(200, "text/html", index_html);
  });

  server.begin();
}
```

The loop() function continuously reads the values of two analogue inputs and the status of two digital buttons in a loop. Based on the values of the sliders, it controls two PWM channels. The collected data, including the ADC values, relay states and button states, is summarized in a JSON-formatted message. This message is then sent to all connected

clients via a WebSocket connection. A short delay of 250 milliseconds ensures that the data is updated regularly.

```
void loop() {
  // Read ADC values
  adc1Value = analogRead(ADC1);
  adc2Value = analogRead(ADC2);

  // Read button states
  btn1Status = digitalRead(BTN1) == HIGH;
  btn2Status = digitalRead(BTN2) == HIGH;

  ledcWrite(PWM1, slider1Value);
  ledcWrite(PWM2, slider2Value);

  // Prepare the JSON message with ADC, relais, and button values
  String message = "{\"slider1\":" + String(slider1Value) + ",\"slider2\":" +
String(slider2Value) +
                   ",\"adc1\":" + String(adc1Value) + ",\"adc2\":" +
String(adc2Value) +
                   «,\»relais1\»:» + String(relais1Status) + «,\»relais2\»:» +
String(relais2Status) +
                   ",\"btn1\":" + String(btn1Status) + ",\"btn2\":" +
String(btn2Status) + "}";

  // Send ADC, relais, and button values to WebSocket clients
  ws.textAll(message);

  delay(250); // Update every half-second
}
```

Chapter 2 • Sensors for Control Engineering

2.1 NTC temperature sensors

An NTC thermistor is a resistor that changes its value depending on the temperature.

NTC stands for 'Negative Temperature Coefficient'. This means that the electrical resistance of an NTC thermistor decreases as the temperature rises.

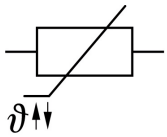


Figure 2.1: Symbol for an NTC sensor

NTC thermistors consist of transition metal oxides that are sintered to form a ceramic body. The materials used are usually metal oxides of manganese, copper, nickel, titanium, cobalt or iron. When heated, electrons are released from the lattice atoms and improve conductivity. The thermistors operate in the temperature range from -50 to +200°C. In contrast to metals, where the resistance increases with temperature, NTC thermistors have more free electrons at higher temperatures, which facilitate the flow of current. Electrical conduction takes place through the 'hopping' of electrons and holes. NTC thermistors have a higher temperature sensitivity coefficient than silicon temperature sensors and resistance temperature sensors.

In practical use, the resistance value for 25 °C is specified for the NTC temperature sensors. Here is the characteristic curve for an NTC10K. NTC 10K characteristic curve

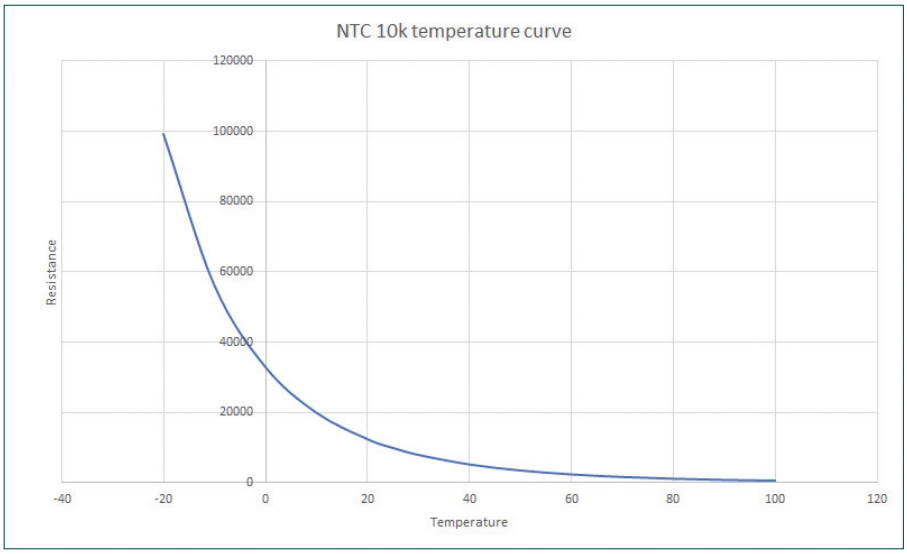


Figure 2.2: NTC 10K characteristic curve

To determine the resistance value of the sensor, a voltage divider is usually used, which should be connected to a precise reference as shown here with 3.3 volts to minimize the influence of voltage fluctuations on the measurement result.

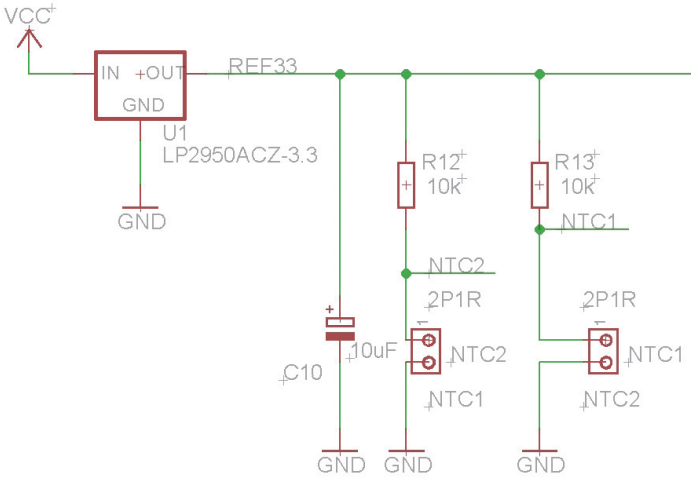


Figure 2.3: Circuit for the NTC sensors

The wires NTC1 and NTC2 can be connected to the ADC inputs of the ESP32 via the jumper field JP5.

As you can see, the characteristic curve of the sensor is not linear, so in the program example Arduino_ADC_NTC10K_Calculation.ino two arrays for resistance and voltage are initialized with 18 values each to calculate and save the values at intervals of 10 K. This enables step-by-step linear interpolation. This provides the option of step-by-step linear interpolation.

This calculation is performed using the following three functions.

Listing 1: Extract from Arduino_ADC_NTC10K_Calculation.ino

```
// Arrays for NTC10K calculation
float Voltages[18] = {0};
float Temperatures[18] = {0};
float VoltageESP32 = 3.3;

// Function for calculating the resistance of an NTC thermistor based on the
temperature
float calculateResistance(float T)
{
    // Constants
    const float R0 = 10000.0; // Nominal resistance of thermistor at 25°C
    const float B = 3950.0;   // B-value of the thermistor from the data sheet
```

```
    return R0 * exp(B * ((1.0 / (T + 273.15)) - (1.0 / (25 + 273.15))));
}

// Voltage from voltage divider 10k input NTC resistor
float calulateVoltage(float Rt)
{
    float R25 = 10000;
    float UE = VoltageESP32;
    float Ua=(UE*Rt)/(Rt+R25);
    return Ua;
}

void initArrayNTC()
{
    // Temperature range from -35°C to 135°C
    int min_temp = -35;
    int max_temp = 135;
    float voltage;
    int loop = sizeof(Voltages) / sizeof(Voltages[0]) - 1;
    // Loop over the temperature range
    for (float temp = min_temp; temp <= max_temp; temp+=10)
    {
        // Calculating the resistance for the current temperature
        float resistance = calculateResistance(temp);
        // Calculating the voltage for the current resistance
        float voltage = calulateVoltage(resistance);
        // Saving the values for temperature and voltage in arrays
        Temperatures[loop] = temp;
        Voltages[loop] = voltage;
        loop--;
    }
}
```

The calculation and initialization takes place once the program is started in the setup() function. Here, the two arrays' Voltages and Temperatures are initialized with the calculated values. In the main loop, the two analogue inputs ADC1 and ADC2 are read out and the temperature is calculated from the values in the arrays using the LinearInterpolate(u2); function.

Read out ADC and calculate temperature:

```
// Read ADC1 value
  int voltage1 = analogRead(ADC1);
  // Read ADC2 value
  int voltage2 = analogRead(ADC2);
  // Interpolate voltage 1
```

```
float u1 = readvalue_linear_voltage(n_adc-1,ADC,Ua,(float)voltage1);
// Interpolate voltage 2
float u2 = readvalue_linear_voltage(n_adc-1,ADC,Ua,(float)voltage2);

float temperature_NTC10K_1 = LinearInterpolate(u2);
```

NTC
NTC 10k temperature sensors are very inexpensive and available in various designs.

2.2 RTD temperature sensors

The term RTD for temperature sensors stands for 'Resistance Temperature Detector', known as PT100 and PT1000.

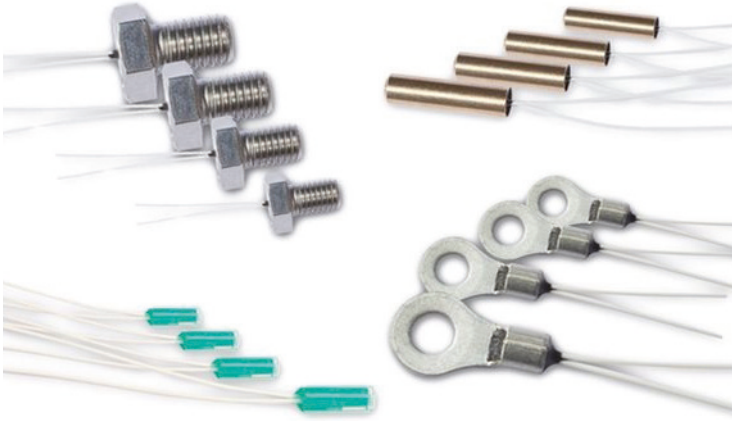


Figure 2.4: Mounting forms of PT1000 temperature sensors
Source: Variohm

The resistive element is usually made of platinum, as it is stable, linear and chemically inert, but can also be made of nickel or copper. RTDs work by measuring the change in electrical resistance, which varies with temperature. The sensors are mainly used for applications that require high accuracy, long-term stability and interchangeability. They are widely used in industry, mechanical engineering and medical technology.

RTDs offer advantages such as high accuracy, consistency, long-term stability, repeatability, suitability for extreme environments and a wide temperature range from -200° C to 850° C.

The designation PT100 or PT1000 indicates the resistance value at 0k. This means 100 Ohm for the PT100 and 1000 Ohm for the PT1000.

For the range from 0 to 850°, the resistance is calculated according to the Callendar van Dusen equation, an empirical equation for describing the temperature dependence of the resistance of a platinum RTD.

$$RT = R0(1 + AT + BT^2)$$

For negative temperatures from 0° to -200°:

$$RT = R0(1 + AT + BT^2 + CT^3(T - 100))$$

A = 3,9083e-3
B = -5,775e-7
C = -4,183e-12

The temperature for positive values from the resistance is calculated using the following equation:

$$T = \frac{-R0A + \sqrt{(R0A)^2 - 4R0B(R0 - RT)}}{2R0B}$$

After division with R0, the result is:

$$T = \frac{-A + \sqrt{(A)^2 - 4B(1 - \frac{RT}{R0})}}{2B}$$

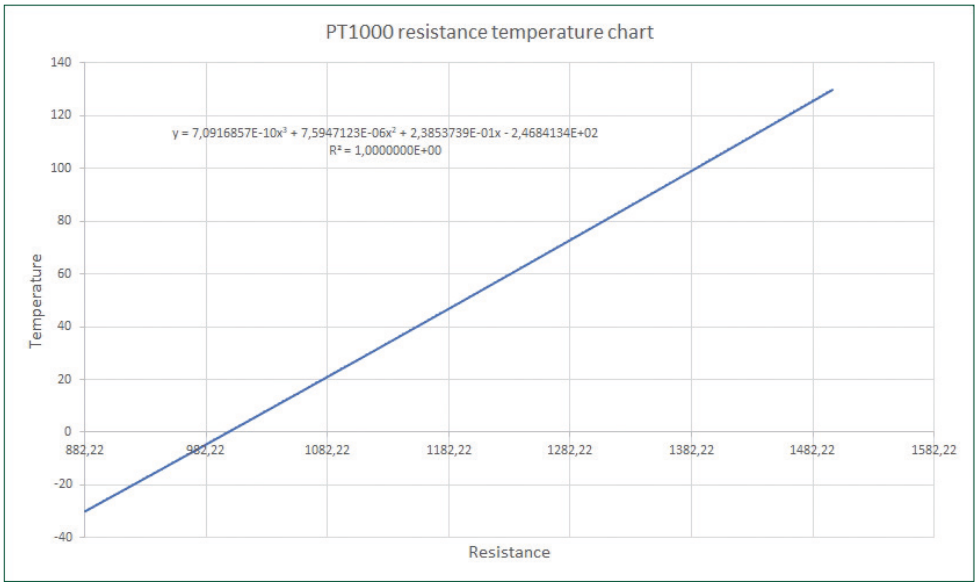


Figure 2.5: PT1000 temperature characteristic curve

From the chart you can see that the change in resistance over temperature is not very high. A measuring amplifier can be constructed with an operational amplifier as a subtractor. Here is a circuit with the PT1000 in a bridge circuit and the OP as a subtractor that amplifies the differential voltage. The output of the OP can be connected to ADC1 with a jumper via JP5.

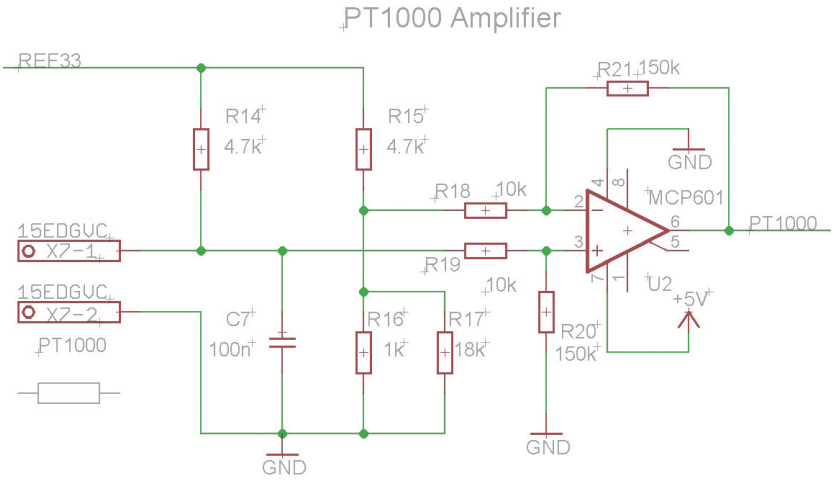


Figure 2.6: Measuring amplifier

2.3 Linear Interpolation

In practice, it has been shown that the best measurement results are obtained when using a PT1000 simulator. To do this, the temperature value is entered into 10 K increments and the voltage values shown on the display are noted.



Figure 2.7: PT1000 Kalibrator from Elementa
Image source: elementa gmbh

The displayed voltages and the corresponding temperatures are stored in two arrays.

Listing 2: Arduino_ADC_PT1000_Interpolation.ino

```
// PT1000 Voltage
float adc_pt1000[] =          { 0,     56,   385,  717,  1042, 1358, 1678, 1981,
2282, 2559, 2877, 3186, 3538,  3984,  4080,  4095};
// PT1000 temperature
float temperatures_pt1000[] = {-20.0, -10.0, 0.0, 10.0,  20.0, 30.0, 40.0, 50.0,
60.0, 70.0, 80.0, 90.0, 100.0, 110.0, 112.0, 113.0};
int n_pt1000 = 16;
```

Now the accuracy of the temperature measurement only depends on the accuracy of the calibrator.

The exact temperature is now calculated using the function for interpolating measured values.

```
// Linear interpolation
// Input Parameters:
// n = Number of values -1
// x = x Array
// y = y Array
// xn = Value to be calculated
float readvalue_linear_voltage(int n,float *x,float *y, float xn)
```

```
{
    int j = 0;
    // Determine interval of xn
    while (j < n-1 && xn > x[j+1]) j++;
    // interpolate xn
    return ((y[j+1]-y[j]) / (x[j+1]-x[j])) * (xn-x[j]) + y[j];
}
```

Here is the part of the program in loop() with the temperature calculation from the voltage from ADC1.

```
  // Read ADC1 value
  int voltage1 = analogRead(ADC1);
  // Interpolate PT1000 temperature
  float pt1000_temperature =   readvalue_linear_voltage(n_pt1000-1,adc_pt1000,
temperatures_pt1000,(float)voltage1);
```

2.4 Thermocouples

Thermocouples are temperature sensors based on the thermoelectric effect. The voltage at the junction of two different metals generates an electrical voltage. This very small voltage can be measured and used to determine the temperature. Here are some important points about thermocouples:

Structure and how they work

A thermocouple consists of two different metallic wires joined together at one end to form a sensing junction (or 'hot junction'). The other end of the wires, the reference junction (or 'cold junction'), remains open. A temperature difference between the measuring junction and the reference junction generates a thermoelectric voltage that is proportional to the temperature difference. This voltage is referred to as thermoelectric voltage.

Types of thermocouples

There are several different types of thermocouples, which consist of different combinations of metals. The most common include:

- Type K (Chromel-Alumel): Suitable for a wide temperature range from -200°C to 1260°C. Widely used and inexpensive.
- Type J (iron-constantan): For temperatures from -210°C to 760°C. Not suitable for oxidizing environments.
- Type T (copper constantan): Good for low temperature applications from -200°C to 350°C. Very accurate in the low temperature range.
- Type E (Chromel constantan): For temperatures from -200°C to 900°C. Higher sensitivity than type K.

- Type N (Nicrosil-Nisil): For temperatures from -200°C to 1300°C. Better stability and oxidation resistance than type K.

- Type S (platinum-rhodium): For high temperatures from -50°C to 1480°C. Very stable and accurate, but expensive.

Advantages of thermocouples
Thermocouples can measure extremely high and low temperatures.

They are mechanically robust and can be used in harsh environments.

Due to their simple design, thermocouples have a fast response time to temperature changes.

Disadvantages of thermocouples
Accuracy: Thermocouples are less accurate than some other temperature sensors, such as RTDs.

Non-linearity: The relationship between temperature and thermoelectric voltage is non-linear, often requiring calibration and complex conversion tables.

Sensitivity to environmental influences: Thermocouples can be affected by chemical reactions, mechanical stress or electromagnetic interference.

2.5 Applications of thermocouples

Industry: In process monitoring and control, such as in metal processing, plastics production and the chemical industry.

Science: In laboratories for measuring high and low temperatures.

Automotive industry: For monitoring exhaust gas temperature and other critical engine components.

Household appliances: In ovens, grills and heating systems.

Thermocouples are versatile and widely used temperature sensors that are used in many applications due to their robustness and wide temperature measurement range.

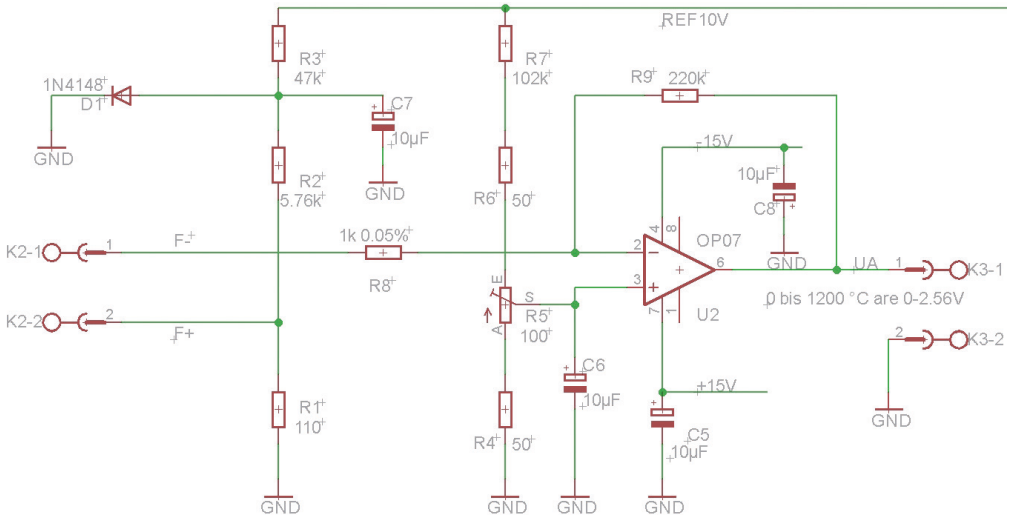


Figure 2.8: Thermocouple amplifier with OP07

As with the PT1000, simulators are available for calibrating the measuring amplifiers.



Figure 2.9: Thermocouple calibrator from Elementa
Image source: elementa gmbh

The biggest problem with the use of thermocouples is that the thermoelectric voltage depends not only on the temperature of the measuring point, but also on the temperature of the cold junction. To obtain an accurate temperature measurement, the temperature of the cold junction must be known and included in the calculation. This is the process of cold junction compensation.

Types of cold junction compensation:

1. Manual cold junction compensation

   Traditionally, the cold junction was kept at a known and stable temperature, often at 0°C by using ice water. The thermoelectric voltage was then corrected using tables or formulae.

   Disadvantage: Impractical and not suitable for mobile or industrial applications.

2. Electronic cold junction compensation

   Modern systems measure the temperature of the cold junction using an additional sensor (such as an RTD, thermistor or silicon IC sensor). This temperature is then used to electronically correct the measured thermoelectric voltage.

A common approach is to use a thermistor or an IC temperature sensor such as the TC1047A. The output of the sensor is then included in the calculations to determine the actual temperature.

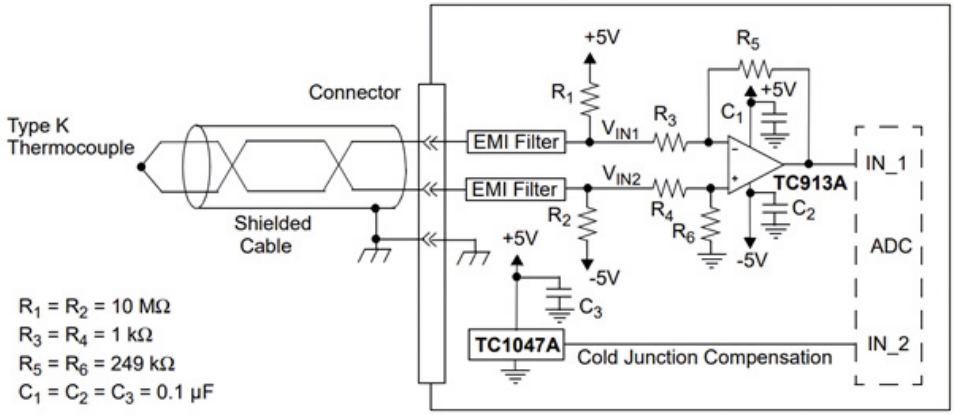


Figure 2.10: Cold junction compensation with TC1047A

Example of electronic cold junction compensation:

1. Measurement of the cold junction temperature

   An IC temperature sensor located near the cold junction measures the local temperature.

   For example, the TC1047A provides an output voltage of 10 mV/°C with an offset of 500 mV. If the cold junction is 25°C, the output voltage is 750 mV.

2. Calculation of the actual temperature

   The measured voltage of the thermocouple is converted into a temperature as if the cold junction was at 0°C.

   The temperature of the cold junction (measured by the IC sensor) is added to this calculated temperature to determine the actual temperature.

Advantages of electronic cold junction compensation:

- Accuracy: Improved accuracy of temperature measurement.
- Practicality: Suitable for use in harsh or changing environments.
- Automation: Enables automated and continuous temperature measurements.

Example circuit for cold junction compensation:

- An example of a circuit that can be used for a remote thermocouple measurement application includes:
- Operational amplifier (e.g. TC913A): For amplifying the thermocouple voltage.
- Resistors: High precision resistors (e.g. 0.1% tolerance) to improve common mode rejection ratio (CMRR).
- Temperature sensor (e.g. TC1047A): For cold junction temperature measurement.

Summary

Cold junction compensation is an essential part of accurate temperature measurement with thermocouples. Modern electronic methods provide a precise and practical solution for cold junction compensation by measuring the temperature of the cold junction and using this information to correct the measurements and calculate the actual temperature of the measuring point.

2.6 Sensors with integrated electronics

Dallas DS18B20 One Wire Temperature Sensor.

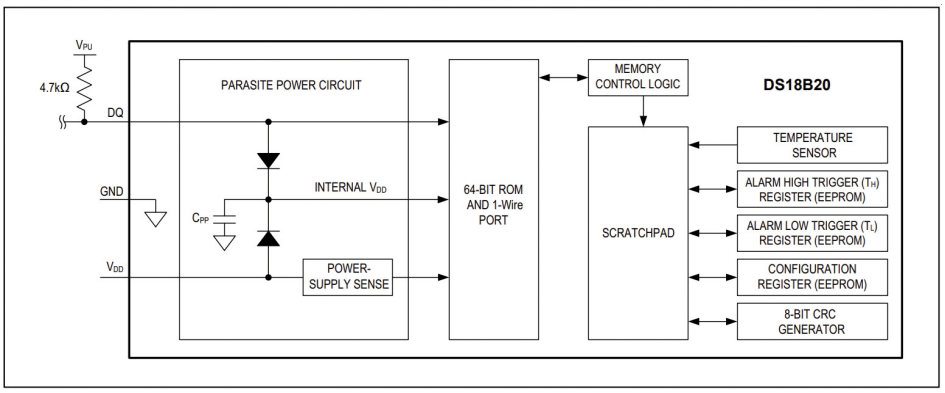


Figure 2.11: Temperature sensor Dallas 18B20

The Dallas DS18B20 temperature sensor is very popular in the Arduino scene.

There are various designs on the market, such as TO92 and versions in a waterproof stainless steel tube. SMD designs are also available.

Communication with the DS18B20 sensor takes place via the one-wire bus. This makes it possible to connect and query several sensors simultaneously via a single data line. Each sensor has a unique 64-bit address that enables specific queries to be made via the one-wire bus.

For the connection to the ESP32 you only need a data line and the 3.3V supply voltage. A pull-up resistor of 4.7k is recommended.

Advantages and features

- Unique 1-Wire® interface requires only one port pin for communication.
- Temperature measurement range from -55°C to +125°C (-67°F to +257°F).
- Accuracy of ±0.5°C in the range from -10°C to +85°C.
- Programmable resolution from 9 bit to 12 bit.
- Parasitic power supply mode requires only 2 pins for operation (DQ and GND).
- Simplified distributed temperature sensing applications with multi-drop capability.
- Each device has unique 64-bit serial code stored in on-board ROM.
- Flexible, user-definable non-volatile (NV) alarm settings with alarm search command to identify devices with temperatures outside programmed limits.

Listing 3: Arduino_ADC_PT1000_OneWire_DS18B20.ino
The following libraries should be installed.

```
#include <OneWire.h>
#include <DallasTemperature.h>

Define pin:

// One Wire
#define DS18B20 4

Create an instance:

DallasTemperature sensors(&oneWire);
```

Initialize the sensor:

```
// Init DS18B20 Sensor
OneWire oneWire(DS18B20);
```

The following function is suitable for readout:

```
float readDS18B20(int index)
{
  // Read Dallas Sensor DS18B20
  sensors.requestTemperatures();
  float temperatur = sensors.getTempCByIndex(index);
  if(temperatur == -127.00) {
    return -1;
  } else
  {
    return temperatur;
  }
}
```

Here is the function call in loop()

```
float ds18b20_temperature = readDS18B20(0);
```

Other sensors such as the DHT11, DHT22 and the TMP117 (for I2C) are also available.

2.7 pH sensors

pH sensors are used in chemistry to precisely measure the pH value of a solution. They are based on electrochemical principles in which a pH-sensitive electrode, often coated with a special glass or polymer membrane, is used. This membrane reacts to hydrogen ions (H^+) in the solution, creating a potential difference between the electrode and the surrounding solution. The measured potential is then recorded electronically and converted into the corresponding pH value, which is displayed on a scale from 0 (very acidic) to 14 (very alkaline).



Figure 2.12: pH Elektroden

Calibration of pH sensors is essential to ensure accurate measurements. This is done by immersing them in buffer solutions with known pH values. pH sensors are widely used in various fields, including environmental monitoring, food industry, pharmaceuticals,

chemicals and water treatment. They play an important role in the monitoring and control of chemical processes as well as quality assurance in industrial plants.

Modern pH sensors can also compensate for temperature fluctuations to improve measurement accuracy. Regular maintenance and care are required to maintain their performance and longevity. Continuous advances in materials science and electronics are making pH sensors more precise, robust and versatile, further increasing their usability and efficiency in numerous industrial and scientific applications.

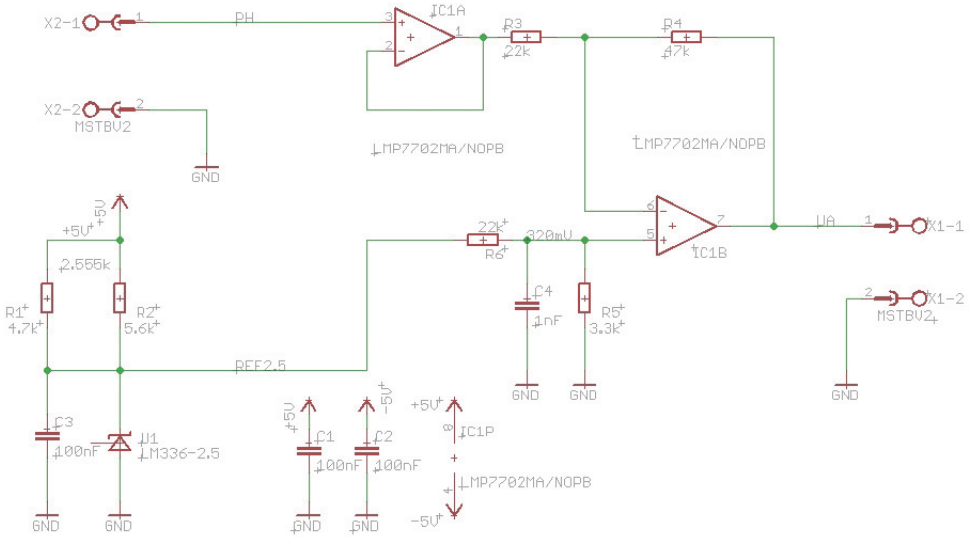


Figure 2.13: pH measuring amplifier with OP LMP7702

The pH amplifier requires a high-impedance input so as not to distort or weaken the weak output signal of the pH electrode. High-impedance inputs prevent the loss of signal strength and ensure the accuracy of the pH measurement.

2.8 Rotary encoder

A rotary encoder (also known as an encoder) is a sensor used to detect the position or speed of the rotary motion of an object, such as a direct current motor (DC motor). Encoders are essential in many industrial applications and automated systems as they provide precise feedback on the movements of a motor. There are two main types of encoders: Incremental encoders and absolute encoders.

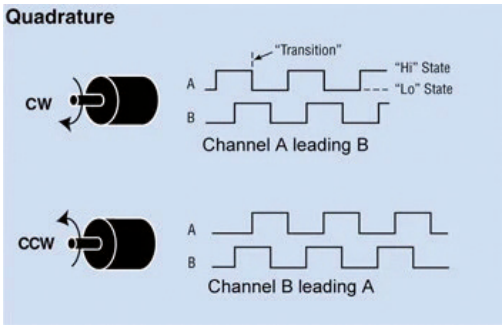


Figure 2.14: Encoder output of the rotary encoder
Source: dynapar company

Incremental encoder
An incremental encoder detects the rotary movement by emitting pulses that represent a relative movement. It typically consists of a rotating disc with evenly spaced slots or patterns and an optical or magnetic sensor.

How it works
A light source (e.g. an LED) shines through the slots of the rotating disc. A sensor (e.g. a photodiode) recognizes the light signals and generates corresponding electrical pulses. These pulses are counted to determine the rotational speed and relative position of the motor.

Advantages
- Simple design and low cost.
- High resolution possible.
- Suitable for applications where only relative movements need to be detected.

Disadvantages
- Loses the absolute position in the event of a power failure or if the movement is stopped.
- Additional measures are required to determine the initial position, e.g. limit switches.

Absolute encoder
An absolute encoder provides a unique position value for each position of the shaft. This is achieved by a more complex coding on the disc.

How it works
The disc has several tracks with different patterns that represent different binary codes. Sensors read the patterns and provide a binary code that indicates the absolute position of the disc.

There are optical, magnetic and capacitive absolute encoders, depending on the type of pattern recognition.

Advantages

- Provides absolute position at all times, regardless of power failures or motion stops.
- Ideal for applications where precise positioning is required.

Disadvantages

- More complex and expensive than incremental encoders.
- Requires a higher processing capacity to decode the position.

Application with DC motors

Rotary encoders are often used in combination with DC motors to enable precise motion control. A typical application example is motor speed control or positioning in servo systems.

Speed control

The encoder continuously supplies pulses that are analyzed by a controller.
The controller adjusts the motor power to ensure a constant speed.

Position control

The encoder indicates the exact position of the motor shaft.
A control system can thus control precise movement sequences and positioning.

A rotary encoder is an essential component of DC motor control systems, providing precise feedback on position and speed. While incremental encoders are characterized by their simplicity and cost-effectiveness, absolute encoders offer the advantage of precise positioning, making them essential for more demanding applications.

2.9 Force measurement

Force sensors are used in control engineering for various important reasons. Here are some of the most common applications and benefits of force sensors in this field:

- In many industrial processes, accurate control of applied forces is critical. Force sensors measure these forces and enable precise control to ensure that the desired force parameters are maintained.
- In the manufacturing industry, force sensors help to monitor the quality of products by ensuring that the correct forces are applied during assembly, pressing or cutting.
- In robotics, force sensors are used to measure the interaction between the robot and its environment. This is particularly important for tasks that require precise handling and adjustment of forces, such as gripping delicate objects or assembling components.
- In automated systems, force sensors help to ensure accurate and repeatable movements. They enable machines to react to changes in load or resistance and adjust their movements accordingly.

- Force sensors can help to make machines safer by monitoring forces and triggering warnings in the event of overloads or anomalies or stopping the machine to prevent damage.
- In material testing, force sensors are used to measure the strength and resilience of materials. This helps with the development and quality control of new materials and products.
- In the automotive industry, force sensors are used to monitor and control forces in various systems, such as braking and steering systems, to improve vehicle safety and performance.
- In research and development, force sensors enable the study of forces and their effects on different systems and materials, leading to new insights and innovations.
- By accurately measuring and controlling forces, force sensors contribute significantly to efficiency, quality and safety in numerous control engineering applications.

Strain gauges are sensory devices that are used to measure deformations or stresses in a material. They typically consist of a thin wire or foil attached to a backing material. When the material to which the strain gauge is attached deforms, the wire or foil also stretches. This stretching leads to a change in the electrical resistance of the wire or film. The change in resistance can be measured and converted into a strain or voltage. Strain gauges are used in a wide range of applications, from material testing to the monitoring of structures. They are extremely sensitive and can detect very small deformations with high accuracy. There are different types of strain gauges, including metallic and semiconductor-based variants. The metallic strain gauges are widely used due to their robustness and accuracy. Semiconductor strain gauges offer higher sensitivity but are also more sensitive to environmental influences.

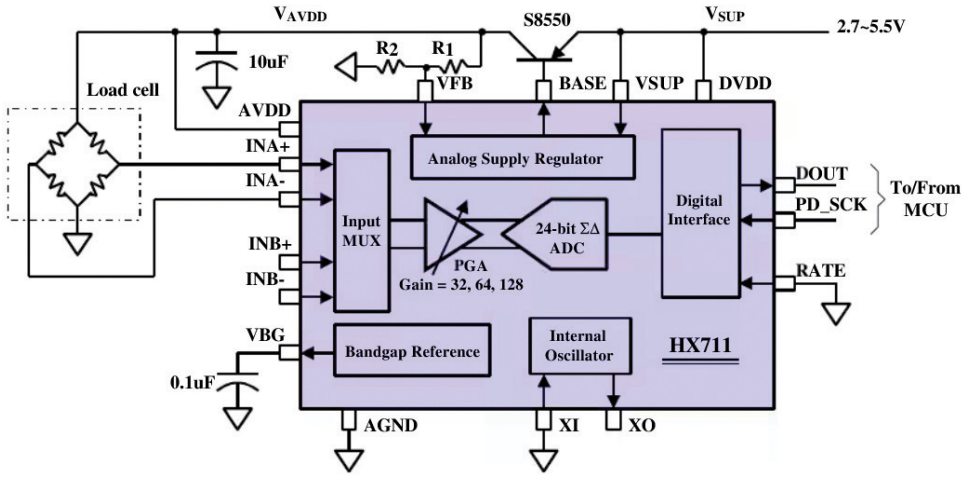


Figure 2.15: Measuring amplifier with HX711



Figure 2.16: LC201 force gauge

Software example for reading in data with the HX711 measuring amplifier and A/D converter.

```
// Read the HX711 load cell on channel A with gain of 128
// Return the result as a 32-bit integer
#include "Arduino.h"

const int CLK_PIN = 2;  // Pin for CLK
const int DTA_PIN = 3;  // Pin for DTA

void setup() {
  pinMode(CLK_PIN, OUTPUT);
  pinMode(DTA_PIN, INPUT);
  digitalWrite(CLK_PIN, LOW);
  Serial.begin(9600);
}

int32_t ReadCount1() {
  int32_t count = 0;
  char i;

  // Wait until DTA_PIN goes LOW
  while (digitalRead(DTA_PIN) == HIGH);

  for (i = 0; i < 24; i++) {
    digitalWrite(CLK_PIN, HIGH);
    count = count << 1;
    digitalWrite(CLK_PIN, LOW);
    if (digitalRead(DTA_PIN) == HIGH) {
      count++;
    }
  }

  digitalWrite(CLK_PIN, HIGH);
```

```
  count = count ^ 0x800000;  // Invert the MSB
  digitalWrite(CLK_PIN, LOW);

  return -count;  // Negate the result and return
}

void loop() {
  int32_t value = ReadCount1();
  Serial.println(value);
  delay(500);  // Delay to make the readings readable
}
```

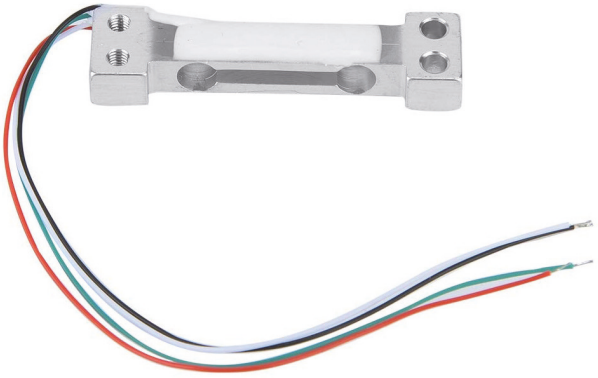


Figure 2.17: Force sensor for a scale

2.10 Linear regression

Linear regression is a statistical method used to model the relationship between a dependent variable (also known as the target variable) and one or more independent variables (predictors). The aim is to set up a linear equation that provides the best possible estimate of the dependent variable based on the independent variables. This equation has the form y = a * x + b, where y is the dependent variable, x is the independent variable, b is the intercept and a is the slope (coefficient).

The parameters a and b are determined in such a way that the sum of the squared deviations (residuals) between the actual values and the predicted values is minimized. This is often achieved using the method of least squares. Linear regression is used in many fields, including economics, biology and engineering, to analyze trends and make predictions. An important measure of the quality of the model is the R-squared value, which indicates how well the independent variables explain the variance of the dependent variables.

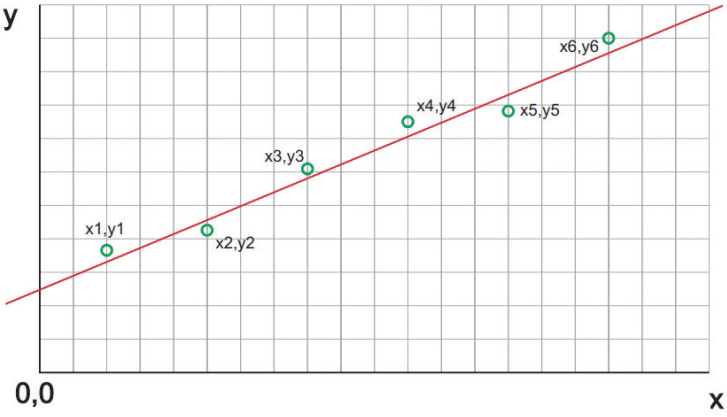


Figure 2.18: Example of linear regression

$$a = \frac{N\sum(x_i y_i) - \sum x_i \sum y_i}{N\sum(x_i^2) - (\sum x_i)^2}$$

Equation Linear regression for the gradient

$$b = \frac{\sum y_i - a\sum x_i}{N}$$

Equation Linear regression for the distance

This example of linear regression includes test values and an application with a pH sensor.

Listing 4: ArduinoLinearRegression.ino

```
//--------------------------------------------------------------------
//
//            Arduino Linear Regression
//            =========================
//
//      Testprogramm for Arduino ESP32 PID Fuzzy Control Board
//
//      calculate Parameters fo Y = ax + b from x and y array values
//
//  Date:     16/07/2024
//  Author:   Josef Bernhardt
//  Hardware: Arduino (ESPRESSIF) ESP32 and Arduino UNO
//
//  File:     Arduino_LinearRegression.ino
//--------------------------------------------------------------------
#include <Arduino.h>
```

```
// Example Values x axis
float x[] = {1, 2, 3, 4, 5};
// Example Values y axis
float y[] = {5, 7, 9, 11, 13};
int QtyTest = sizeof(x) / sizeof(float);

// pH Example
// Voltage from ADC
float ax[] = {0.158556,0.586422,1.014377,1.442273,1.870113};
//Voltage from pH Amplifier
float ay[] = {0.4,0.2,0.0,-0.2,-0.4};
int QtypH = sizeof(ax) / sizeof(float);

// Variables for the y-intercept (a) and the slope (b)
float a, b;
```

LinearRegression() calculates the linear regression for a set of data points. First, the function sums the x-values, the y-values, the products of x and y and the square values of the x-values. It then uses these sums to determine the slope and the intersection of the best-fit lines. The slope indicates how steep the line is, while the intercept indicates where the line intersects the y-axis. To summarize, the function finds the straight line that best fits the data points.

```
// Function for the linear regression
void LinearRegression(float* x, float* y, int AnzX, float* a, float* b)
{
    float sumX = 0, sumY = 0, sumXY = 0, sumX2 = 0;

    // Calculation of the sums
    for (int i = 0; i < AnzX; ++i) {
        sumX += x[i];
        sumY += y[i];
        sumXY += x[i] * y[i];
        sumX2 += x[i] * x[i];
    }

    // Calculation of the gradient (a) and the y-axis intercept (b)
    *a = (AnzX * sumXY - sumX * sumY) / (AnzX * sumX2 - sumX * sumX);
    *b = (sumY - (*a) * sumX) / AnzX;
}

void setup()
{
    // Initialisation of the serial communication
    Serial.begin(115200);
}
```

In this code example, the LinearRegression() function is called twice to calculate the linear regression for two different data sets.

1st example: The LinearRegression() function is called with the data `x` and `y`. The calculated slope and the intersection point of the regression line are displayed. The value of the regression line is then calculated and displayed.

2nd example: The LinearRegression() function is called again, but this time with the data `ax` and `ay`. The slope and intercept are also displayed here. The value of the regression line is also calculated and output.

The program waits one second between the two calculations before repeating the process.

```
void loop()
{
    // Example 1
    LinearRegression(x, y, QtyTest, &a, &b);
     // Display Result
    Serial.print("Gradient (a): ");
    Serial.println(a, 2);
    Serial.print("Intercept (b): ");
    Serial.println(b, 2);

    float result_example = a * 3.52 + b;
    Serial.print("Result fuer x = 3.52 :");
    Serial.println(result_example,3);


    // Example 2
    LinearRegression(ax, ay, QtypH, &a, &b);
     // Display Result
    Serial.print("Gradient (a): ");
    Serial.println(a, 2);
    Serial.print("Intercept (b): ");
    Serial.println(b, 2);

    float result_ph = a * 1.634 + b;
    Serial.print("Result fuer x = 1.634 :");
    Serial.println(result_ph,3);

    delay(1000);
}
```

2.11 Spline interpolation

The term 'spline' originally comes from construction and shipbuilding. A spline is a flexible strip of wood or metal used to draw smooth curves through a series of points. This was particularly useful in shipbuilding to create the curved shapes of ship hulls.

In mathematics and computer graphics, the term has been adopted to describe a method of interpolating smooth curves through a series of data points. A mathematical spline is a function consisting of several polynomial segments that are connected in such a way that they exhibit a certain degree of smoothness at the connection points.

This smoothness is achieved by the requirement that the first and often higher derivatives of the function coincide at the joints. This results in smooth, continuous curves that pass through the given data points exactly or approximately.

The following Arduino example is intended to show how spline interpolation can be used to calculate exact intermediate values from calculated support values for voltages and associated temperatures.

Ua=(3.3*Rt)/(Rt+R25) (Rt = NTC Sensor1 / R25 = 10k)

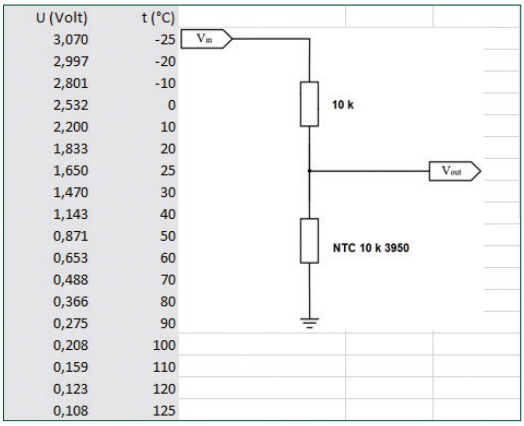


Figure 2.19: NTC voltage divider

Here is the chart with the values for voltage and temperature:

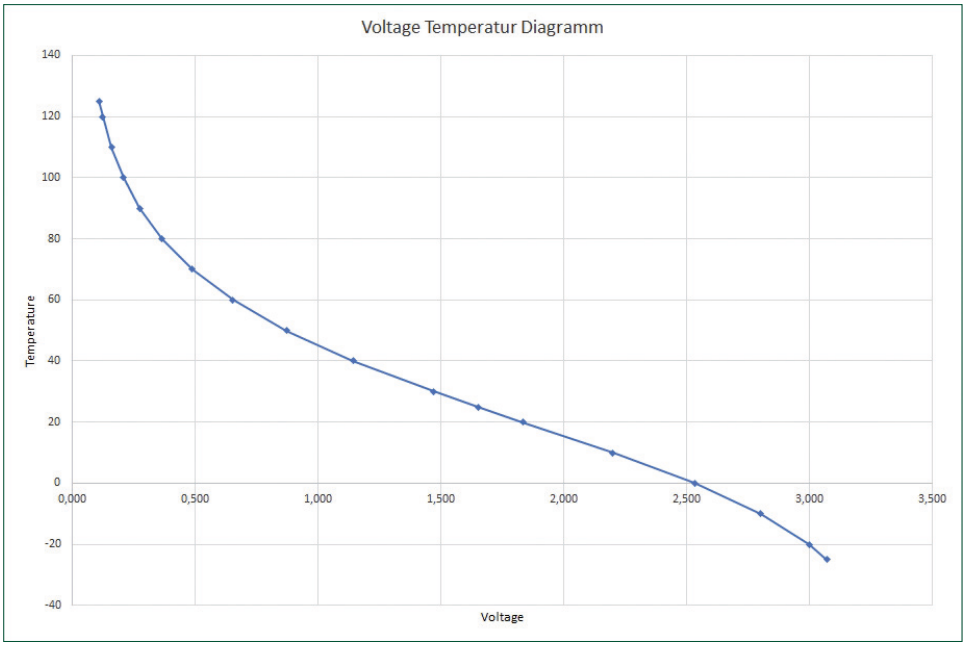


Figure 2.20: Spline diagram Voltage Temperature

The program calculates the parameters for calculating the intermediate values from the values in the xs[] and ys[] arrays using the spline_natural_init() function, which is called once when the program is started in the setup() function, and stores these in the hx[] , hy[] and M[] arrays.

Listing 5: Arduino_ADC_NTC_CubicSplineInterpolation.ino

```
// 3.3 Volt Values for ESP32 or RP2040
float xs[] = {0.108, 0.123, 0.159, 0.208, 0.275, 0.366, 0.488, 0.653, 0.871,
1.143, 1.470, 1.650, 1.833, 2.200, 2.532, 2.801, 2.997, 3.070};
// 5 V Values for Arduino Nano
// float xs[] = {0.164, 0.186, 0.241, 0.316, 0.417, 0.554, 0.740, 0.989, 1.320,
1.731, 2.227, 2.5, 2.778, 3.333, 3.836, 4.244, 4.542, 4.652};
float ys[] = {125.000, 120.000, 110.000, 100.000, 90.000, 80.000, 70.000,
60.000, 50.000, 40.000, 30.000, 25.000, 20.000, 10.000, 0.000, -10.000, -20.000,
-25.000};

/*
xs and ys are the arrays of the support points.
hx and hy store the differences between the support points.
M stores the calculated moments of the splines.
n is the number of support points minus 1 (since the index starts at 0).
```

```
*/
int spline_natural_init(int n, float *x, float *y, float *hx, float *hy, float
*M) {
    int i;
    // Allocate memory for temporary arrays
    float *mue, *z, *lambda, *d;
    mue = (float *)malloc((n + 1) * sizeof(float));
    z = (float *)malloc((n + 1) * sizeof(float));
    lambda = (float *)malloc((n + 1) * sizeof(float));
    d = (float *)malloc((n + 1) * sizeof(float));
    // Check if memory allocation was successful
    if (mue == nil || z == nil || lambda == nil || d == nil)
        return 0;  // Error during memory allocation

    // Step 1: Calculate differences hx and hy
    for (i = 1; i <= n; i++) {
        hx[i] = x[i] - x[i - 1];
        hy[i] = y[i] - y[i - 1];
    }

    // Initialize the arrays z, lambda, and d for the tridiagonal equation system
    z[0] = 2;
    lambda[0] = 0;
    d[0] = 0;

    // Step 2: Calculate the coefficients mue, z, lambda, and d
    for (i = 1; i < n; i++) {
        mue[i] = hx[i] / (hx[i] + hx[i + 1]);  // Calculate μ (mu)
        z[i] = 2;
        lambda[i] = hx[i + 1] / (hx[i] + hx[i + 1]);  // Calculate λ (lambda)
        // Calculate d
        d[i] = 6 / (hx[i] + hx[i + 1]) * (hy[i + 1] / hx[i + 1] - hy[i] / hx[i]);
    }

    // Boundary conditions for a natural spline (second derivative at the ends is
zero)
    mue[n] = 0;
    z[n] = 2;
    d[n] = 0;

    // Step 3: Forward elimination to solve the tridiagonal equation system
    for (i = 1; i <= n; i++) {
        z[i] = z[i] - mue[i] / z[i - 1] * lambda[i - 1];
        d[i] = d[i] - mue[i] / z[i - 1] * d[i - 1];
    }
```

```
    // Step 4: Back substitution to calculate the second derivatives M
    M[n] = d[n] / z[n];
    for (i = n - 1; i >= 0; i--) {
        M[i] = (d[i] - lambda[i] * M[i + 1]) / z[i];
    }

    // Free the temporarily allocated memory
    free(mue);
    free(z);
    free(lambda);
    free(d);

    return 1;  // Successful calculation
}
```

The spline_value() function calculates the interpolated value of a natural cubic spline at the position x0 based on the given points (x, y) and the calculated second derivatives M.

```
float spline_value(int n, float *x, float *y, float *hx, float *hy, float *M,
float x0) {
    int j = 0;
    float A1, A2, A3, y0;

    // Find the interval where x0 lies
    while (j < n - 1 && x0 > x[j + 1]) j++;

    // Calculate the coefficients A1, A2, and A3
    A1 = hy[j + 1] / hx[j + 1] - hx[j + 1] / 6 * (2 * M[j] + M[j + 1]);
    A2 = M[j] / 2;
    A3 = (M[j + 1] - M[j]) / 6 / hx[j + 1];

    // Use Horner's scheme to compute the interpolated value y0
    y0 = A3;  // Initial value for Horner's scheme
    y0 = A2 + (x0 - x[j]) * y0;  // Compute the next level
    y0 = A1 + (x0 - x[j]) * y0;  // Compute the next level
    y0 = y[j] + (x0 - x[j]) * y0;  // Add the final term

    return y0;  // Return the interpolated value
}
```

In the main program loop(), the spline_value() function can be used to calculate the exact temperature from the measured voltages.

```
void loop()
{
    float Temperature = 0;
    // Read ADC1 value
    int voltage1 = analogRead(ADC1);
    float u1 = readvalue_linear(n_adc-1,ADC,Ua,(float)voltage1);
    Temperature = spline_value(n_ntc - 1, xs, ys, hx, hy, M, u1);
    Serial.print("Voltage:");
    Serial.print(u1,3);
    Serial.print(" Temperature:");
    Serial.println(Temperature);
    // Switching LED
    digitalWrite(LED_BUILTIN,!digitalRead(LED_BUILTIN));
    delay(200);
}
```

Conclusion

Cubic splines are a powerful and versatile tool for creating smooth, continuous curves through a given set of points. Due to their mathematical properties, they offer high accuracy and smooth transitions, making them indispensable in many technical and scientific applications. The given code implements a cubic spline to interpolate a function that uses values from an array of sample points xs and ys.

2.12 Polynomial regression

Polynomial regression is an extremely versatile tool in data analysis.

One of its main applications is curve fitting, where it is used to fit mathematical functions to experimental data. This helps to identify patterns and trends in the data, which is particularly useful in technical applications.

Polynomial regression is also used to smooth data. Experimental data is often irregular or noisy. Here, polynomial regression can be used to create a smoother function that better represents the underlying trends, allowing for more precise analysis.

Polynomial regression is a flexible and powerful method used in various disciplines to model data, identify patterns and make predictions. Its versatility makes it an important tool for metrology.

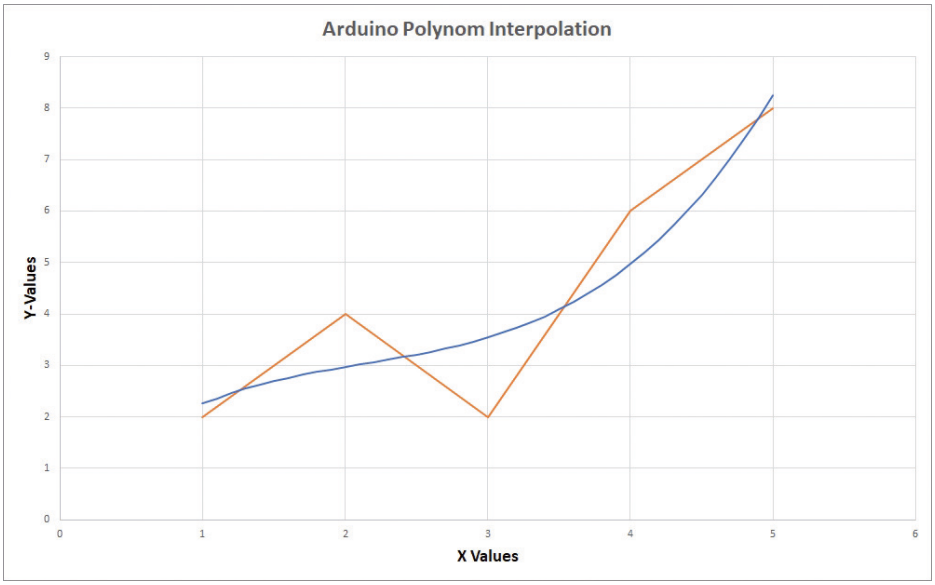


Figure 2.21 Chart Polynomial Interpolation

The PolynomReg function calculates the coefficients for a polynomial regression model with a degree of polynomiality based on a given set of data points (x, y). It takes the arrays x and y as input, which contain the x and y values of the data points. The function creates a matrix G and a vector c to solve a linear system of equations G * a = c, where a are the coefficients of the polynomial. This is done by matrix inversion and backward substitution. At the end, the function returns an array a containing the calculated coefficients of the polynomial. All dynamically allocated memory areas are released to avoid memory leaks.

Listing 6: Arduino_Polynom_Interpolation.ino

```
//----------------------------------------------------------------------
//
//            Arduino Polynomial Interpolation
//            ================================
//
//      Testprogramm for Arduino ESP32 PID Fuzzy Control Board
//
//      calculate Parameters for Polynom Interpolation
//
//  Date:     05/08/2024
//  Author:   Josef Bernhardt
//  Hardware: Arduino (ESPRESSIF) ESP32 and Arduino UNO
//
//  File:     Arduino_Polynom_Interpolation.ino
//----------------------------------------------------------------------
#include <Arduino.h>
```

The PolynomReg() function calculates the coefficients for a polynomial regression model with a specified degree for a given set of data points (x, y). It receives as parameters an array of x-values, an array of corresponding y-values, the degree of the desired polynomial and the number of data points. The return value is an array containing the coefficients of the polynomial, starting with the constant term up to the highest degree. These coefficients define the best fit function that describes the data points by a polynomial. These coefficients can be used to calculate the regression line or curve for any x-values.

```
float* PolynomReg(float* x, float* y, int Polynomgrad, int QtyX) {
    float* Sxk = (float*)malloc((2 * Polynomgrad + 1) * sizeof(float));
    float* Sxkyk = (float*)malloc((Polynomgrad + 1) * sizeof(float));
    // float G = (float)malloc((Polynomgrad + 1) * sizeof(float*));
    float** G = (float**)malloc((Polynomgrad + 1) * sizeof(float*));

    for (int i = 0; i <= Polynomgrad; ++i) {
        G[i] = (float*)malloc((Polynomgrad + 1) * sizeof(float));
    }
    float* g1 = (float*)malloc((Polynomgrad + 1) * sizeof(float));
    float* c = (float*)malloc((Polynomgrad + 1) * sizeof(float));
    float* a = (float*)malloc((Polynomgrad + 1) * sizeof(float));

    // Initialization of Sxk and Sxkyk
    for (int i = 0; i <= 2 * Polynomgrad; ++i) {
        Sxk[i] = 0;
        for (int k = 0; k < QtyX; ++k) {
            Sxk[i] += pow(x[k], i);
        }
    }
    for (int i = 0; i <= Polynomgrad; ++i) {
        Sxkyk[i] = 0;
        for (int k = 0; k < QtyX; ++k) {
            Sxkyk[i] += pow(x[k], i) * y[k];
        }
    }

    // Creation of the coefficient matrix G and the constant vector c
    for (int i = 0; i <= Polynomgrad; ++i) {
        for (int j = 0; j <= i; ++j) {
            G[i][j] = Sxk[i + j];
            G[j][i] = Sxk[i + j];
        }
        c[i] = Sxkyk[i];
    }

    // Solving the equation system G * a = c with matrix inversion
    // and matrix multiplication a = G1 * c
```

```
    float det = 1.0;
    for (int i = 0; i <= Polynomgrad; ++i) {
        for (int j = i + 1; j <= Polynomgrad; ++j) {
            float ratio = G[j][i] / G[i][i];
            for (int k = i; k <= Polynomgrad; ++k) {
                G[j][k] -= ratio * G[i][k];
            }
            c[j] -= ratio * c[i];
        }
        det *= G[i][i];
    }

    // Back substitution for the polynomial coefficients
    for (int i = Polynomgrad; i >= 0; --i) {
        a[i] = c[i];
        for (int j = i + 1; j <= Polynomgrad; ++j) {
            a[i] -= G[i][j] * a[j];
        }
        a[i] /= G[i][i];
    }

    free(Sxk);
    free(Sxkyk);
    for (int i = 0; i <= Polynomgrad; ++i) {
        free(G[i]);
    }
    free(G);
    free(g1);
    free(c);

    return a;
}
```

The PolynomEval() function evaluates a polynomial at a specific point x using given coefficients. It receives an array of coefficients, the degree of the polynomial and the value x at which the polynomial is to be evaluated as parameters. The return value is the value of the polynomial at position x. This function makes it possible to calculate the function value of the polynomial for any x-values. This allows you to check how the polynomial behaves at different points.

```
float PolynomEval(float* a, int Polynomgrad, float x) {
    float px = 0.0;
    for (int i = Polynomgrad; i >= 0; --i) {
        px = px * x + a[i];
    }
    return px;
```

```
}

void setup()
{
    // Initialisation of the serial communication
    Serial.begin(115200);
}
```

In the loop() function, polynomial regression coefficients are calculated for given test values and then analyzed. Firstly, two example data arrays x[] and y[] are defined, which can represent either test values or ADC values. The PolynomReg() function is used to calculate the coefficients for a third-degree polynomial. The calculated coefficients are then output to the serial console. The polynomial is then evaluated for a range of x-values within the range of the test data using PolynomEval() and the results are also displayed on the serial console. Finally, the resources used for the calculation are released and the loop ends.

```
void loop()
{

// Example Testvalues
float x[] = {1.0, 2.0, 3.0, 4.0, 5.0};
float y[] = {2.0, 4.0, 2.0, 6.0, 8.0};

// Example ADC Values
// float x[] = {0,   0,    123,   249, 496,  817,  1141, 1456, 1778, 2416, 3067,
3922, 4095, 4095, 4095};
// (Ua) in volts values applied to ADC by calibrator
// float y[] = {0.00, 0.10, 0.20, 0.30, 0.50, 0.75, 1.00, 1.25, 1.50, 2.00, 2.50,
3.00, 3.07, 3.15, 3.30};

       int Polynomgrad = 3;
       int QtyX = sizeof(x) / sizeof(x[0]);

       float* coefficients = PolynomReg(x, y, Polynomgrad, QtyX);
       Serial.print("Polynomial coefficients: ");
       for (int i = 0; i <= Polynomgrad; ++i) {
               Serial.print(coefficients[i], 2);
               Serial.print(" ");
       }
       Serial.println();
  // Polynomial coefficients: 7.99 -12.49 8.32 -2.00 0.17

       float start = x[0];
       float end = x[QtyX - 1];
       Serial.print("Polynomial values from ");
       Serial.print(start, 1);
```

```
        Serial.print(" to ");
        Serial.println(end, 1);

        // for (float value = start; value <= end; value += 10.0)  // ADC Values
    for (float value = start; value <= end; value += 0.1)      // Test Values
    {
                float result = PolynomEval(coefficients, Polynomgrad, value);
                Serial.print("x=");
                Serial.print(value, 2);
                Serial.print(": ");
                Serial.println(result, 3);
        }
        free(coefficients);
    while (true){}
}
```

Here is the serial output with the Windows C# program SerialPortTerminal.exe to transfer the data to Excel.

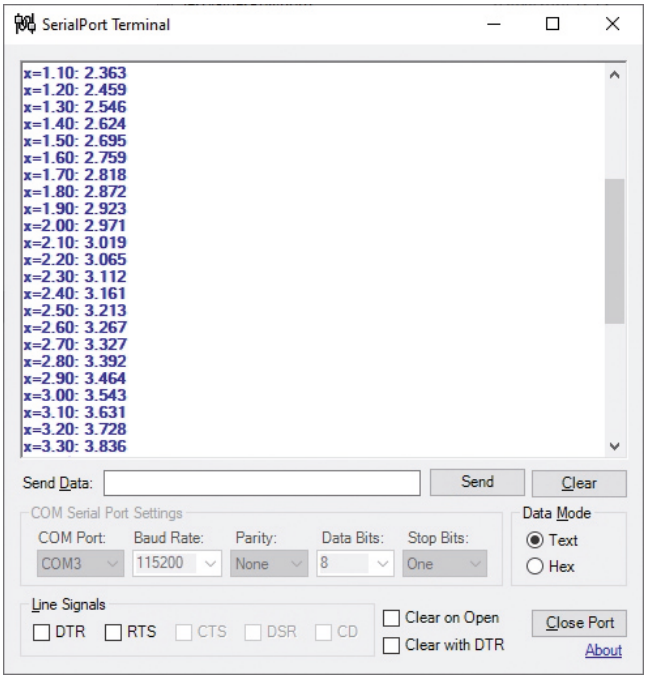


Figure 2.22: C# Program Serial Port Terminal

Chapter 3 • Actuators - MOSFET and SSR

3.1 MOSFET BSP76

Two Infineon BSP76s are installed on the ESP32 board to control larger loads up to 70 watts, such as heaters and motors. The component is an Infineon HITFET™ Smart Low-Side Switch, which are versatile power transistors that are ideal for automotive and industrial applications. Their built-in intelligence and protection features offer higher performance and reliability than conventional discrete solutions. The HITFET Smart Low-Side Switches offer a higher level of protection against over-temperature, short circuit, over-current, over-voltage and ESD. Infineon HITFET combines all these protection functions in a single device.

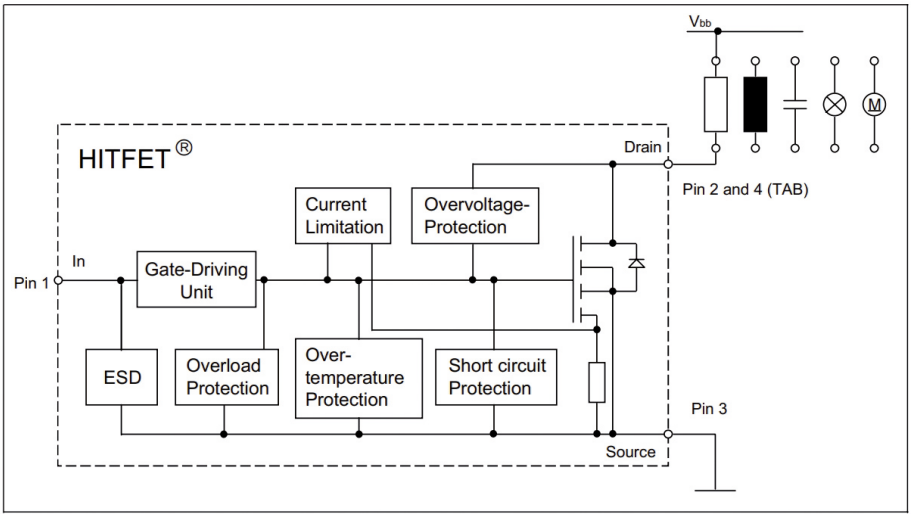


Figure 3.1: Internal circuitry of the BSP76

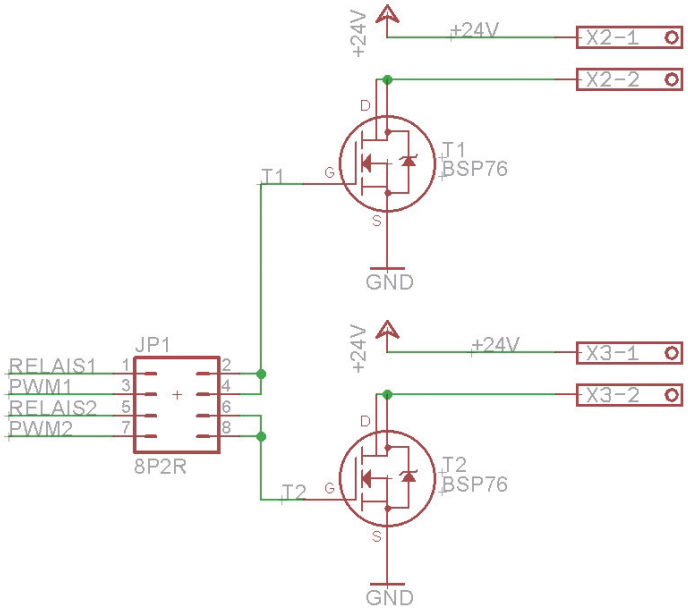


Figure 3.2: Schematic of the BSP76 from the ESP32 board

Here are the most important technical data for the BSP76

- Nominal load current per channel 1.4 A
- Operating temperature min max -40°C 150°C
- Power consumption max 3.8 W
- Protection strategy Auto restart
- RDS (on) (@ T_j = 25°C) 190 mΩ
- RDS (on) (@ T_j = 150°C) max 480 mΩ
- Recommended operating voltage max 42 V

3.2 MOSFET circuit with optocoupler for 10 A

This circuit is designed for higher outputs up to approx. 250 watts. The supply voltage should not be less than 24 V so that the IRFB4410 mosfet is adequately driven.

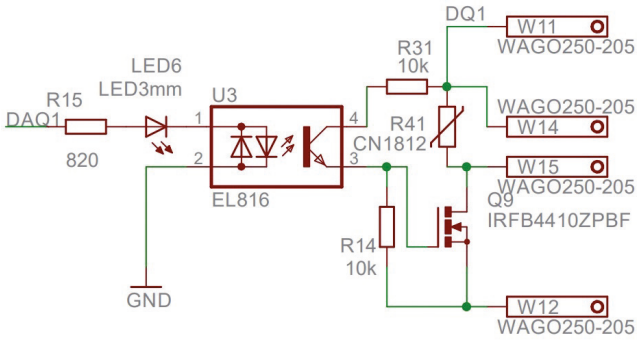


Figure 3.3: MOSFET circuit with optocoupler and IRFB4410

H-bridge circuits are electrical circuits that are widely used in electrical engineering. They consist of four transistors arranged in the shape of an 'H.' These circuits are often used in power electronics, especially in applications such as motor control and for Peltier elements. By controlling the switches, H-bridge circuits can generate bidirectional current paths, making them ideal for applications where reversible power transfer is required. A typical application is the control of DC motors, where the direction of rotation is controlled by reversing the direction of the current. H-bridge circuits also enable efficient control of power and offer high reliability in various applications. However, they are also challenging to control and require appropriate circuitry to avoid unwanted switching states and control the switches effectively. Despite these challenges, H-bridge circuits are an important part of many electronic systems due to their versatility and efficiency.

3.3 H-bridge L298

The L298 is an integrated monolithic circuit in a 15-pin multiwatt and PowerSO-20 package. It is a high voltage, high current dual full bridge driver that accepts standard TTL logic levels and drives inductive loads such as relays, solenoids, DC, and stepper motors. Two enable inputs allow the device to be activated or deactivated independently of the input signals. The emitters of the lower transistors of each bridge are connected together and the corresponding external connection can be used to connect an external measuring resistor. An additional supply input is provided so that the logic operates at a lower voltage.

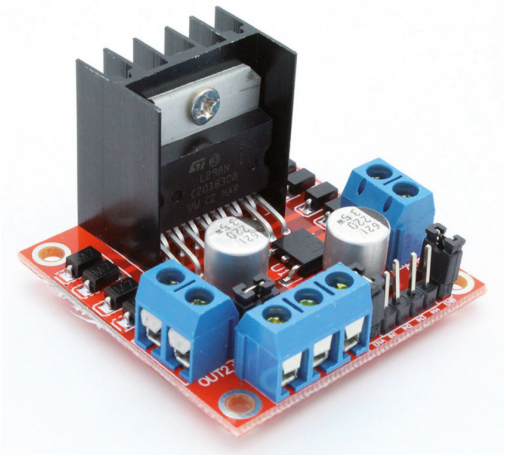


Figure 3.4: H-bridge with L298

3.4 H-bridge with BTS7960 from Infineon up to a maximum of 43 A

The BTS 7960 is a fully integrated high current half-bridge for motor drive applications. It is part of the NovalithICTM family and contains a p-channel high-side MOSFET and an n-channel low-side MOSFET with an integrated driver IC in one package. The p-channel highside switch eliminates the need for a charge pump, minimizing EMI. Interfacing to a microcontroller is facilitated by the integrated driver IC, which provides logic level inputs, diagnostics with current sensing, slew rate adjustment, dead time generation and protection against over temperature, over voltage, under voltage, over current and short circuit. The

BTS 7960 offers a cost-optimized solution for protected high-current PWM motor drives with a very small footprint on the circuit board.

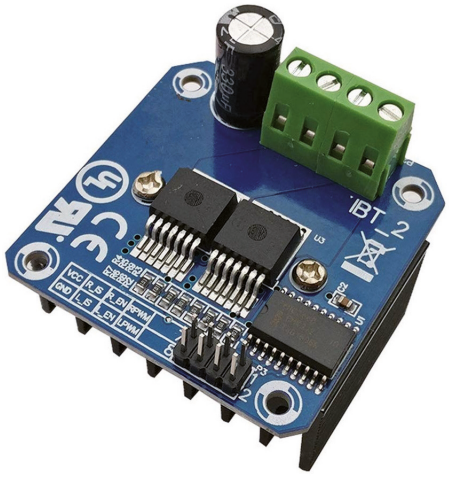


Figure 3.5 H-bridge with BTS7960 for high currents

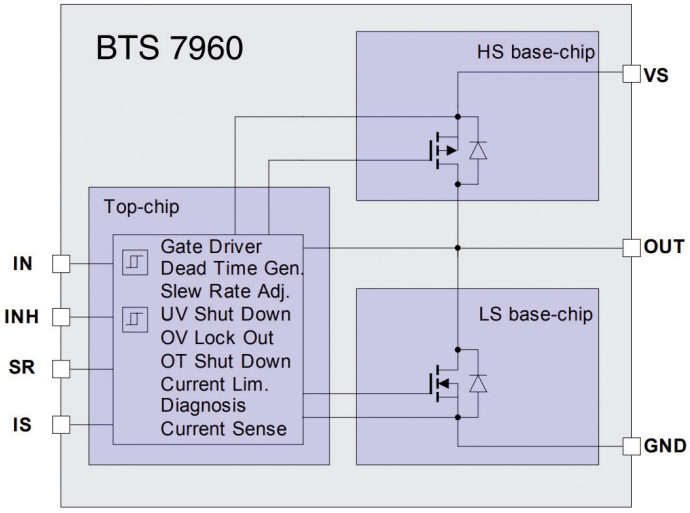


Figure 3.6: Internal circuit BTS7960

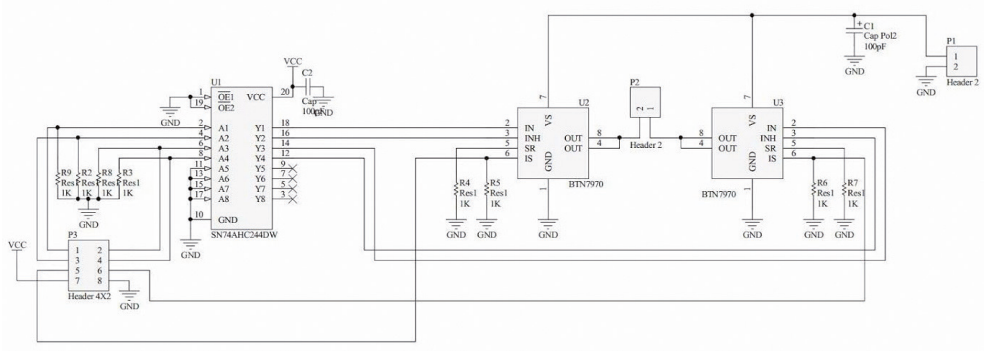


Figure 3.7: Example circuit with the BTS7960

3.5 Thyristor actuator for AC voltage

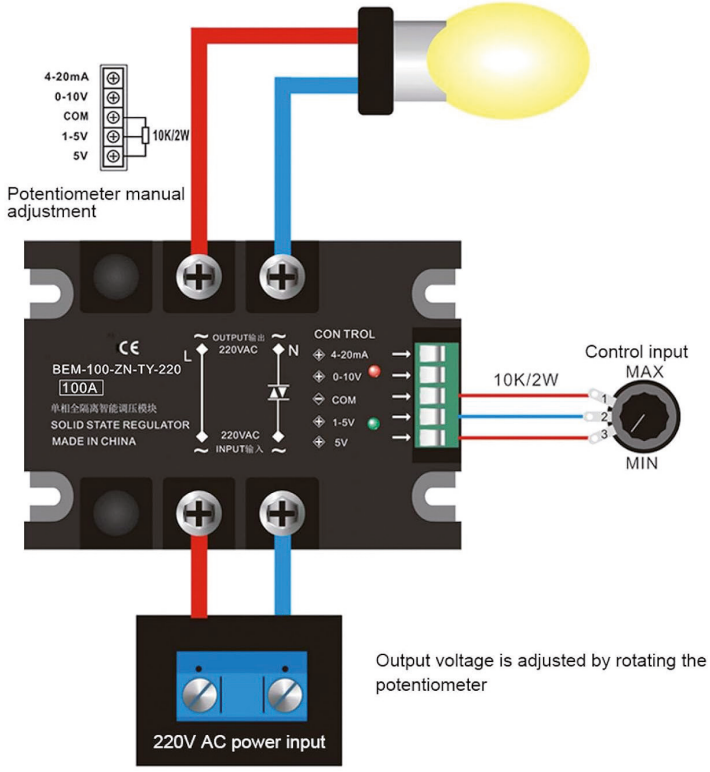


Figure 3.8: Power regulator for 230V

Chapter 4 • Control, regulation and control systems

This chapter provides an introduction to controllers and controlled systems. Firstly, here are the most important formula symbols that will be required later on.

4.1 Formula symbols

The formula symbols largely correspond to the standard control engineering conventions specified in DIN EN 60617-12:1999-08. This standard specifies uniform labelling of control and regulation systems and defines the meaning of the symbols.

Here are the corresponding formula symbols according to DIN EN 60617-12:

Formula symbol	Description
e or e(t)	error
u or u(t)	control variable
y or y(t)	controlled variable
r or r(t)	setpoint
Kp, Ki, Kd	Proportional, integral and derivative amplification (for PID controllers)
Td, Ti	Time constants for the derivative, integral components of a PID controller

It is important to note that these conventions are internationally recognized, but in some cases specific industries or organizations may use slightly different conventions.

4.2 Difference between control and control with feedback according to IEC 60050-351

Control is the process in a system in which one or more variables as input variables influence other variables as output variables based on the system's inherent laws. Control is characterized by the open path of action, in which the output variables influenced by the input variables do not act on themselves continuously via the same input variables.

The control chain can be represented by the following block diagram.

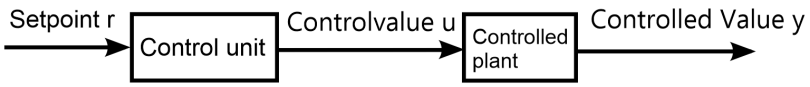


Figure 4.1: Control

Control is a process in which a variable, the controlled variable variable (variable to be controlled), is continuously recorded, compared with another variable variable, the reference variable, and influenced in the sense of an adjustment to the reference variable. Control is characterized by a closed action sequence in which the controlled variable continuously influences itself in the action path of the control loop.

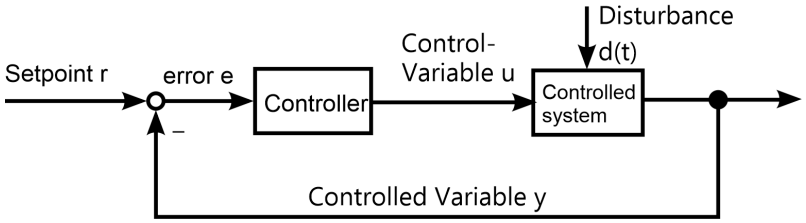


Figure 4.2: Control System

The controls will be discussed in more detail in the next chapter. Firstly, two examples of control technology for the Arduino are discussed.

4.3 Examples control technology

First an example of a PT2 controlled system, a function simulates a PT2 controlled system such as a heating system. A manipulated variable u with the value 100 is applied to the system for a certain time, then a value 0. The output value y could be a temperature t. After 300 seconds you can see that y rises slightly and then slowly falls. This PT2 function is ideal for testing the various controllers. A PT1 function is also available in the program.

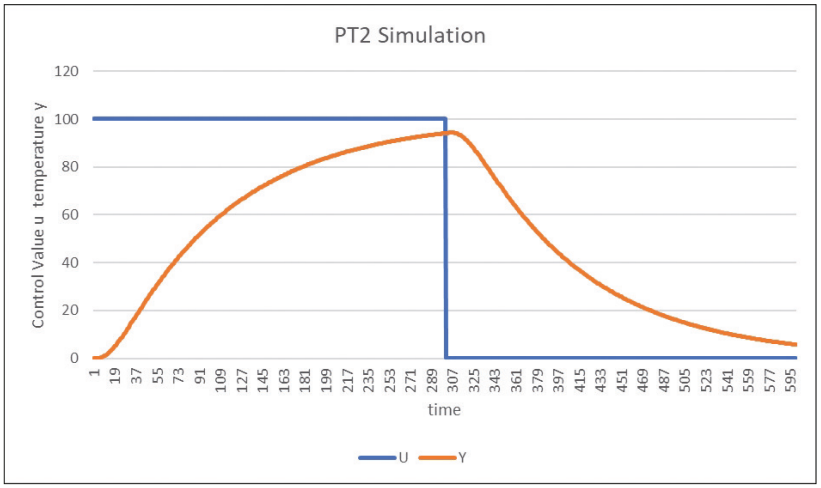


Figure 4.3: PT2 Transfer Function Simulation

The output can also be done with the serial plotter of the Arduino IDE, here is a screenshot that was created with the program Arduino_PT2_SystemSimulation.ino. This allows you to adjust the values for u and observe the results of the route directly.

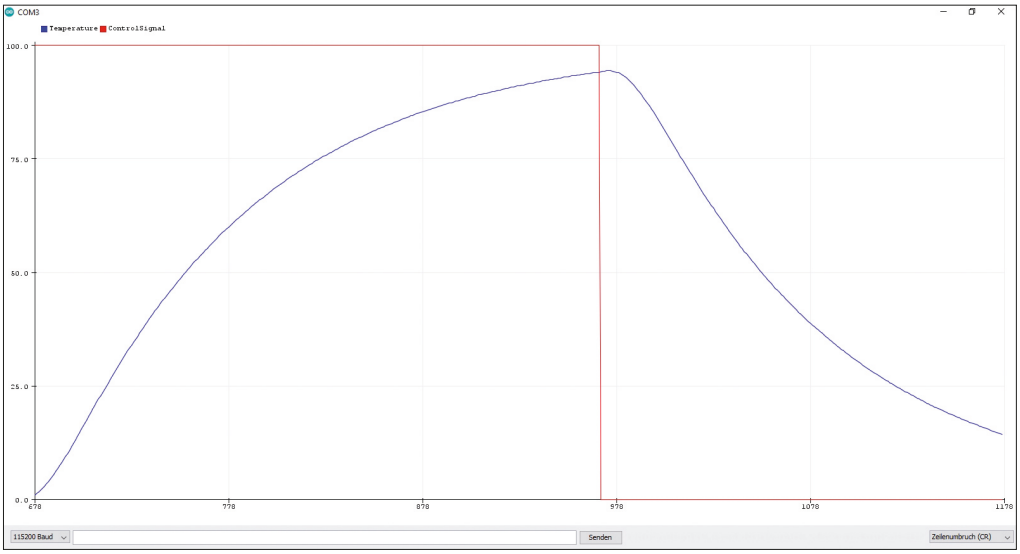


Figure 4.4: Arduino PT2 Simulation

Listing 4.1 Arduino_PT2_SystemSimulation.ino

```
//---------------------------------------------------------------------
//
//            Arduino PT2 System Simulation Test
//            ==================================
//
//      Testprogramm for Arduino ESP32 PID Fuzzy Control Board
//      PT1 and PT2 System Simulation Test Output to USB UART
//
//  Date:     29/04/2024
//  Author:   Josef Bernhardt
//  Hardware: Arduino (ESPRESSIF) ESP32
//
//  File:     Arduino_PT2_SystemSimulation.ino
//---------------------------------------------------------------------

// Include-Bibliotheken
#include <Arduino.h>

// Pin-Definitions
#define LED_BUILTIN 2 // GPIO 2 of ESP32 )
// LED1
#define LED1 5

// Simulating a PT1 control system
float PT1_Simulation(float controlInput, int reset)
```

```
{
    // Parameter
    float deltaT = 1.0;
    float ProportionalGain = 2.0;
    float IntegralGain = 5.0;
    float TimeConstant = 400.0;  // Time constant for the PT1 distance
    // Status variable
    static float output = 0;
    // Reset the static variable
    if (reset == 1)
    {
        output = 0;
    }
    // Model equation, Euler integration
    float d_output = (ProportionalGain * controlInput - output) / TimeConstant;
    output += d_output * deltaT;
    return output;
}
```

The function simulates a PT2 system, which is a simple model for describing second-order dynamic systems. In this case, the system is described by three state variables (output1, output2 and output3), which are calculated iteratively using the Euler method.

1. The PT2 system is described by three links (or time constants) that model the behavior of the system. Each element represents a certain delay or inertia of the system.

2. In the beginning, the system can be reset by setting the reset variable, which resets the state variables to 0.

3. The calculations d_output1, d_output2 and d_output3 model the derivatives of the state variables, whereby the ratio of input variable and state variable is determined by a time constant.

4. The proportional, integral and differential gains modify the response of the system to the control input and the various outputs.

5. The Euler method is used to update the state variables by numerical integration over time, where deltaT represents the time interval of the simulation.

6. Each state variable is updated based on the previous one, where output1 is influenced by the control input, output2 depends on output1 and finally output3 represents the final output of the system.

7. The function returns the last state variable, output3, as the result, which reflects the simulated system response to the controller input.

```
// Simulating a PT2 control system
float PT2_Simulation(float controlInput, int reset)
{
    // Parameter
    float deltaT = 1.0;
    float ProportionalGain = 1.0;
    float IntegralGain = 1.0;
    float DerivativeGain = 1.0;
    float TimeConstant1 = 100.0;
    float TimeConstant2 = 8.0;
    float TimeConstant3 = 10.0;

    // State variables
    static float output1 = 0;
    static float output2 = 0;
    static float output3 = 0;

    // Reset static Variabel
    if (reset == 1)
    {
        output1 = 0;
        output2 = 0;
        output3 = 0;
    }
    // Model equations, Euler integration
    float d_output1 = (ProportionalGain * controlInput - output1) / TimeConstant1;
    float d_output2 = (IntegralGain * output1 - output2) / TimeConstant2;
    float d_output3 = (DerivativeGain * output2 - output3) / TimeConstant3;

    output1 += d_output1 * deltaT;
    output2 += d_output2 * deltaT;
    output3 += d_output3 * deltaT;

    return output3;
}

// Init Hardware
void setup() {
  pinMode(LED_BUILTIN, OUTPUT); // Konfiguration des Pins als Ausgang
  // Init UART USB
  Serial.begin(115200);
}
```

```
// Main Loop
void loop()
{
  // main code to execute repeatedly:
  float temperature = 0;
  float control_signal = 100.0;

  // Reset PT2 System
  temperature = PT2_Simulation(0,1);
  // toggle LED
  digitalWrite(LED_BUILTIN, !digitalRead(LED_BUILTIN));
  // wait 200 ms
  delay(200);
  for (int time = 0; time < 600; time++)
  {
      // Simulating the temperature control system
      temperature = PT2_Simulation(control_signal,0);

      if (time > 300)  control_signal = 0.0;

      Serial.print(" Temperature:");
      Serial.print(temperature, 1);     // Temperature with one decimal place
      Serial.print(" ControlSignal:");
      Serial.print(control_signal, 1);  // Control signal with one decimal place
      Serial.println();                 // New line for the next output
      delay(20);
  }
}
```

4.4 Control of a real heating system

The example 'Arduino_PhysicalHeaterPulse.ino' is about controlling a heater and measuring the temperature.

An aluminium block with a heating resistor is used to test the controller.

Calculation of the heating of an aluminium block

1. Calculation of the heat capacity of the aluminium block:

The density of aluminium is approximately 2.7 g/cm^3.

Volume of the block:

$$V = 80\ mm \times 30\ mm \times 15\ mm = 36000\ mm^3 = 36\ cm^3$$

Mass of the block:

$$m = V \times rho = 36\ cm^3 \times 2.7\ {}^{g}/cm^3 = 97.2\ g = 0.0972\ kg$$

The specific heat capacity of aluminium is about 900 J/(kg-K).

Heat capacity:

$$C = m \times c = 0.0972\ kg \times 900\ J/(kg\text{-}K) = 87.48\ /K$$

2. Calculation of the power supplied by the heating resistor:

The power P supplied by an electrical resistor can be calculated using the equation

$$P = U^2 / R$$

For U = 24 V and R = 30 Ohm:

$$P = 24^2 / 30 = 576 / 30 = 19.2\ W$$

3. Determination of the energy required to heat the aluminium block:

The energy Q required to heat the block from T1 = 22 K to T2 = 60 K is

Q = C x Delta T
Delta T = T2 - T1 = 60 K – 22 K = 38 K
Q = 87.48 J/K x 38 K = 3324.24 J

4. Calculation of the required time:

The time t required to supply this energy is

$$t = Q / P\ t = 3324.24\ J / 19.2\ W = 173.14\ s \approx 2.89\ minutes$$

Summary
It takes about 2.89 minutes or 173.14 seconds to heat the aluminium block from 22 K to 60 K when using a heating resistor with a resistance of 30 Ohm at a voltage of 24 V.

This is controlled by the BSP76 MOSFET output which is controlled by the RELAIS1 output of the ESP32. For this purpose, a potentiometer on the ESP32 board is read in with the ADC1 and a value from 0 to 100 is calculated for the clocking of the heating. This corresponds to a duration of 0 to 1000 ms in 10 ms steps. In the main loop, a function is called every second to switch an output. The parameter that is passed is the duration. In the timer callback function, the value is counted down and the output is switched off at 0. A value of 75, for example, means that the output is on for 750 ms and off for 250 ms. This is an initial test, basically a control to set the temperature and observe the behavior of the track.

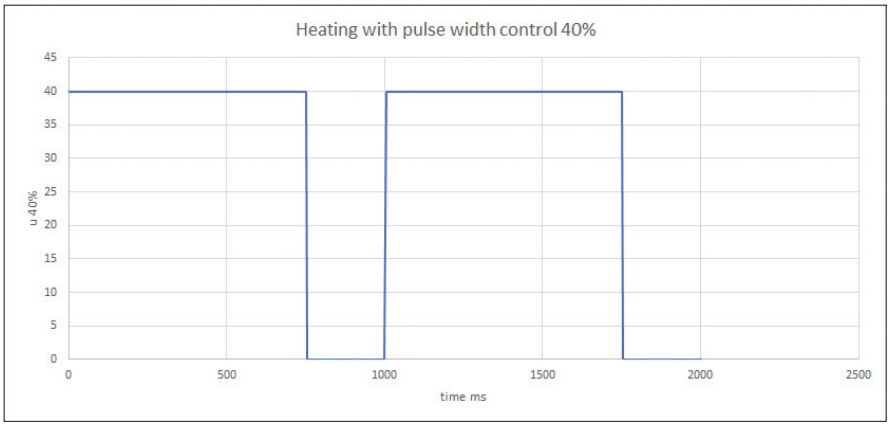


Figure 4.5: Pulse heating control

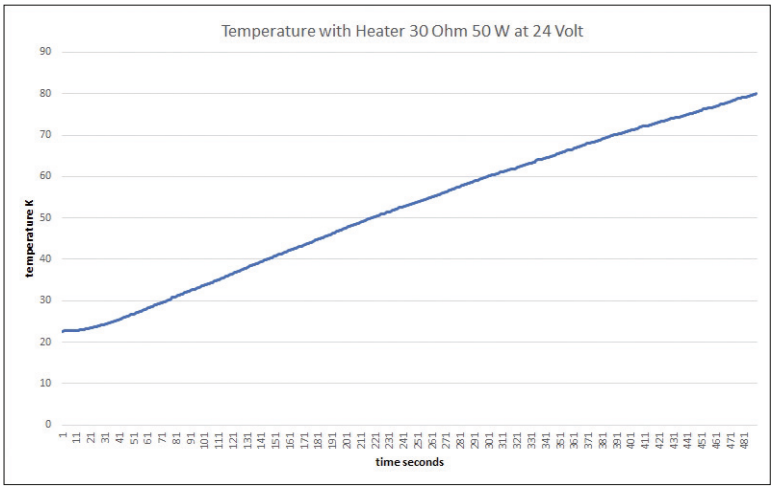


Figure 4.6: Step response of the heating

Here is another test with a heat output of 40 %, after approx. 100 seconds the heating is switched off. You can see very clearly how long the ALU block retains the heat. After 300 seconds, the temperature is still 28 °C.

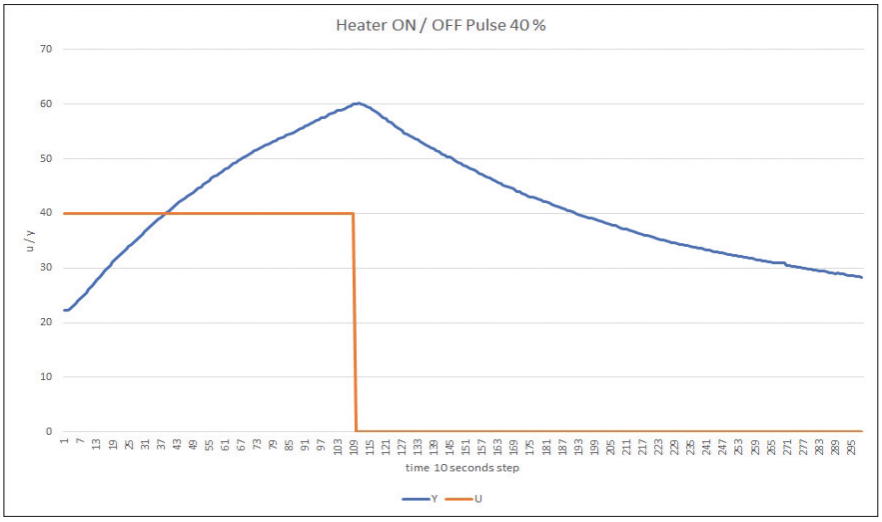


Figure 4.7: Heating test with the aluminium block

Listing 4.2: Arduino_PhysicalHeaterPulse.ino

```
//--------------------------------------------------------------------
//
//            Arduino Physical Heater Pulse Test
//            ==================================
//
//      Control Software for Arduino ESP32 PID Fuzzy Control Board
//      10ms Timer Interrupt Function
//      Init UART2 for Nextion Display
//      Display ADC1 and ADC2 Values on Display NX3224F028_001
//
//      Reading NTC1 Sensor / Display Temperature
//      POT 2 for Pulse Value 0 to 100
//      Outputting the pulses to RELAIS1 for the heater
//
//
//  Date:     30/04/2024
//  Author:   Josef Bernhardt
//  Hardware: Arduino (ESPRESSIF) ESP32
//
//  File:     Arduino_PhysicalHeaterPulse.ino
//--------------------------------------------------------------------

#include <Arduino.h>
#include <Ticker.h>

int LED_BUILTIN = 2;
```

```
// ESP32 UART 2 Pins
#define RXD2 16
#define TXD2 17

// I2C Pins
#define I2CSCL 22
#define I2CSDA 21

// ESP32 ADC Pins
#define ADC1 32
#define ADC2 33
// Motor Sensor Pins
#define MOTSENS1 15
#define MOTSENS2 34
// Buttons
#define BTN1 14
#define BTN2 27

// Relay and Mosfet (BSP76) Pins
#define RELAIS1 18
#define RELAIS2 19
// LED1
#define LED1 5
// PWM1 and PWM2 Pins
#define PWM1 12
#define PWM2 13
// DAC Outputs 0..5V
#define DAC1 25
#define DAC2 26
// One Wire
#define DS18B20 4

#define FILTER_SIZE 20

// Define 10ms Period
const unsigned long timerInterval = 10;

// Ticker-Objekt for the Timer-Funktion
Ticker timer;

// Counter
int counter10ms=0;

int counterRelais1 = 0;

// When 1 second is elapsed
```

```
bool OneSecond = false;

// Counter for seconds
int CounterSeconds = 0;

// Arrays for NTC10K calculation
float Voltages[18] = {0};
float Temperatures[18] = {0};
float VoltageESP32 = 3.3;

// 12 Bit ADC-Values from 0 to 4095
float ADC[] = {0, 0, 55, 174, 406, 704, 1003, 1301, 1606, 2202, 2823, 3745, 3984,
4095, 4095};

// (Ua) in volts values applied to ADC by calibrator
float Ua[] = {0.00, 0.10, 0.20, 0.30, 0.50, 0.75, 1.00, 1.25, 1.50, 2.00, 2.50,
3.00, 3.10, 3.15, 3.30};

int n_adc = 15;



void Heat1_ON(int value)
{
  //Serial.println("ON");
  if (value > 0)
  {
    counterRelais1 = value;
    digitalWrite(RELAIS1,1);
    digitalWrite(LED1,1);
  }
}

void Heat1_OFF()
{
  //Serial.println("OFF");
  digitalWrite(RELAIS1,0);
  digitalWrite(LED1,0);
}


int mapInteger(int value, int fromLow, int fromHigh, int toLow, int toHigh) {
    return (value - fromLow) * (toHigh - toLow) / (fromHigh - fromLow) + toLow;
}

float mapFloat(float x, float in_min, float in_max, float out_min, float out_max)
```

```
{
  return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
}

// Linear interpolation
// Input Parameters:
// n = Number of values -1
// x = x Array
// y = y Array
// xn = Value to be calculated
float readvalue_linear_voltage(int n,float *x,float *y, float xn)
{
    int j = 0;
    // Determine interval of xn
    while (j < n-1 && xn > x[j+1]) j++;
    // interpolate xn
    return ((y[j+1]-y[j]) / (x[j+1]-x[j])) * (xn-x[j]) + y[j];
}

// Function to calculate the resistance of an NTC thermistor based on temperature
float calculateResistance(float T)
{
    // Constants
    const float R0 = 10000.0; // Nominal resistance of the thermistor at 25°C
    const float B = 3950.0;   // B value of the thermistor see data sheet
    return R0 * exp(B * ((1.0 / (T + 273.15)) - (1.0 / (25 + 273.15))));
}

// Voltage from voltage divider 10k input NTC resistor
float calulateVoltage(float Rt)
{
    float R25 = 10000;
    float UE = VoltageESP32;
    float Ua=(UE*Rt)/(Rt+R25);
    return Ua;
}
```

The filterNTC_1() function calculates the moving average of the last values of an NTC sensor to smooth the measurements. It stores the latest measured values in an array and updates this array continuously by storing the values at the current index position. The average of the stored values is calculated and returned to provide a more stable and less noisy sensor reading.

```
// Filter of NTC1 Sensor
float filterNTC_1(float newValue) {
    // Array to save the last FILTER_SIZE values
    static float values[FILTER_SIZE] = {0};
    static int index = 0;
    float sum = 0;
    // Updating the value at the current index position
    values[index] = newValue;
    // Increment the index for the next value
    index = (index + 1) % FILTER_SIZE;
    // Calculation of the average of the values in the array
    for (int i = 0; i < FILTER_SIZE; ++i) {
        sum += values[i];
    }
    return sum / FILTER_SIZE;
}

// Interpolation of temperature from voltage from array
// for voltage and temperature
float LinearInterpolate(float voltage) {
    int size = sizeof(Voltages) / sizeof(Voltages[0]);
    // Find the indices for interpolation
    int i;
    for (i = 0; i < size - 1; i++) {
        if (voltage >= Voltages[i] && voltage <= Voltages[i + 1]) {
            break;
        }
    }
    // Linear interpolation
    float interpolated_temperature = Temperatures[i] + ((Temperatures[i + 1] -
Temperatures[i]) / (Voltages[i + 1] - Voltages[i])) * (voltage - Voltages[i]);
    return interpolated_temperature;
}
```

Calculation of temperature and voltage values for temperature measurement with the NTC resistor.

```
void initArrayNTC()
{
    // Temperature range from -35°C to 135°C
    int min_temp = -35;
    int max_temp = 135;
    float voltage;
    int loop = sizeof(Voltages) / sizeof(Voltages[0]) - 1;
    // Loop over the temperature range
    for (float temp = min_temp; temp <= max_temp; temp+=10)
```

```
    {
        // Calculating the resistance for the current temperature
        float resistance = calculateResistance(temp);
        // Calculating the voltage for the current resistance
        float voltage = calulateVoltage(resistance);
        // Saving the values for temperature and voltage in arrays
        Temperatures[loop] = temp;
        Voltages[loop] = voltage;
        loop--;
    }
}
```

The timerCallback() function is called every 10 milliseconds. It increments a counter for the 10 millisecond intervals (second counter) and decrements another counter for a relay. When the 10 millisecond counter reaches 100, which corresponds to one second, this counter is reset, and a flag is set to true to indicate that one second has passed. When the relay counter drops to zero, the function `Heat1_OFF` is called to switch off the output.

```
// Timer-Funktion
// Called every 10 ms
void timerCallback()
{
  // Inc 10ms Counter
  counter10ms++;
  counterRelais1--;
  // If one Second
  if(counter10ms==100)
  {
     counter10ms = 0;
     // One Second
     OneSecond = true;
  }
  if(counterRelais1==0)
  {
     Heat1_OFF();
  }
}

// Sends Data to Nextion Display in a textbox
void Nextion_Send_Text_to_Textbox(char *textbox,String sfloat)
{
        Serial2.write(textbox);
        Serial2.write(".txt=\"");
        Serial2.print(sfloat);
        Serial2.write("\"");
        Serial2.write(0xFF);
```

```
        Serial2.write(0xFF);
        Serial2.write(0xFF);
}

// Initializing the hardware
void setup()
{
  // Init USB UART
  Serial.begin(115200);
  // Nextion Display
  // TFTSerial.begin(9600);
  Serial2.begin(9600, SERIAL_8N1, RXD2, TXD2);
  // Init Hardware
  pinMode(LED_BUILTIN, OUTPUT);
  pinMode(LED1, OUTPUT);
  pinMode(RELAIS1, OUTPUT);
  pinMode(RELAIS2, OUTPUT);
  pinMode(BTN1, INPUT);
  pinMode(BTN2, INPUT);

  pinMode(ADC1, INPUT);
  pinMode(ADC2, INPUT);

  Nextion_Send_Text_to_Textbox("t1","VoltageNTC");
  Nextion_Send_Text_to_Textbox("t2","Temp NTC1");
  Nextion_Send_Text_to_Textbox("t3","");
  Nextion_Send_Text_to_Textbox("t4","Voltage POT2");
  Nextion_Send_Text_to_Textbox("t5","Pulse Value");
  Nextion_Send_Text_to_Textbox("t6","time");

  initArrayNTC();

  // Init Timer with 10ms
  timer.attach_ms(timerInterval, timerCallback);

}
```

The program is executed regularly in this loop() function. First, the value of two analogue inputs (ADC1 and ADC2) is read and voltage2 is scaled in a range from 0 to 100. The value of voltage1 is interpolated and the temperature is calculated based on this value. When one second has elapsed, the OneSecond flag is reset, an output is switched on for a certain time, and every 10 seconds the temperature, the scaled value and other variables are shown on the serial monitor and on the display. In addition, an LED flashes every second and the seconds counter is incremented.

```
void loop()
{
  // Place your main code here to execute repeatedly:
  float temperature = 0;

  // Read ADC1 value
  int voltage1 = analogRead(ADC1);
  // Read ADC2 value
  int voltage2 = analogRead(ADC2);

  int scaledValue = map(voltage2, 0, 4095, 0, 100);

  // Interpolate voltage 1
  float u1 = readvalue_linear_voltage(n_adc-1,ADC,Ua,(float)voltage1);
  // Interpolate voltage 2
  // float u2 = readvalue_linear_voltage(n_adc-1,ADC,Ua,(float)voltage2);

  temperature = LinearInterpolate(filterNTC_1(u1));

  // Check if one second has passed
  if (OneSecond)
  {
      // Reset the boolean variable OneSecond
      OneSecond = false;
      // Heat1 10-1000 ms on step 10ms
      Heat1_ON(scaledValue);

      // Check whether 10 seconds have elapsed
      if (CounterSeconds % 10 == 0)
      {
          // Sends variable to USB UART and Seriell Monitor
          Serial.print("t:");
          Serial.print(temperature, 2); // Temperature with 2 decimal places
          Serial.print(" u:");
          Serial.print(scaledValue);    // Control signal
          Serial.println();             // New line for the next output
      }
      // Sends variable to display
      Nextion_Send_Text_to_Textbox("txtT1",String(voltage1));
      Nextion_Send_Text_to_Textbox("txtT2",String(temperature,2));
      Nextion_Send_Text_to_Textbox("txtT3"," ");
      // Sends variable to display
      Nextion_Send_Text_to_Textbox("txtT4",String(voltage2));
      Nextion_Send_Text_to_Textbox("txtT5",String(scaledValue));
      Nextion_Send_Text_to_Textbox("txtT6",String(CounterSeconds));
      // Switching LED
```

```
      digitalWrite(LED_BUILTIN,!digitalRead(LED_BUILTIN));
      CounterSeconds++;
  }
}
```

4.5 Control of a DC motor with PWM

In the following example, a DC motor is to be controlled with PWM. To do this, two PWM channels are initialised in the ESP32 program.

```
// Pins
#define PWM1 12
#define PWM2 13

// use 10 bit precision for LEDC timer
#define PWM_RESOLUTION 10

// use 1000 Hz as a LEDC base frequency
#define PWM_FREQUENZ 1000
```

The initialisation of the two PWM channels:

```
// Setup timer and attach timer to a led pin
ledcAttach(PWM1, PWM_FREQUENZ, PWM_RESOLUTION);
ledcAttach(PWM2, PWM_FREQUENZ, PWM_RESOLUTION);
```

The PWM frequency is output via `ledcWrite(PWM2, controlvalue);`

In the PWM speed diagram, it is noticeable that the motor only starts to rotate from a PWM value of 300 (approx. 30 % power).

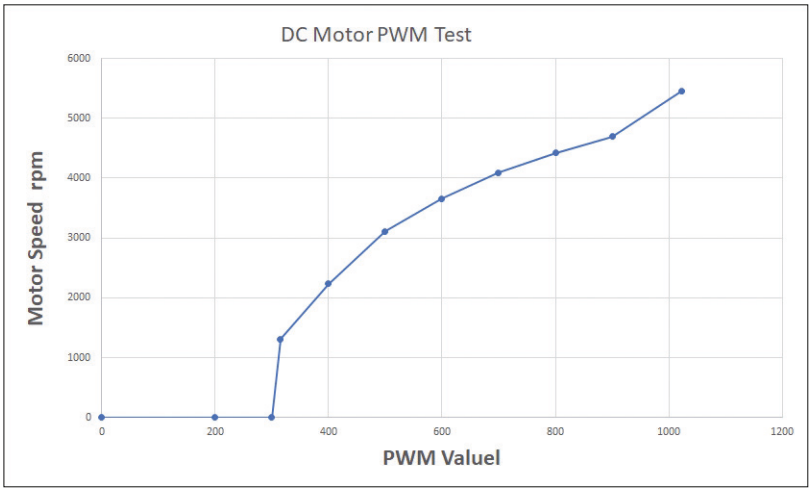


Figure 4.8: DC motor characteristic curve

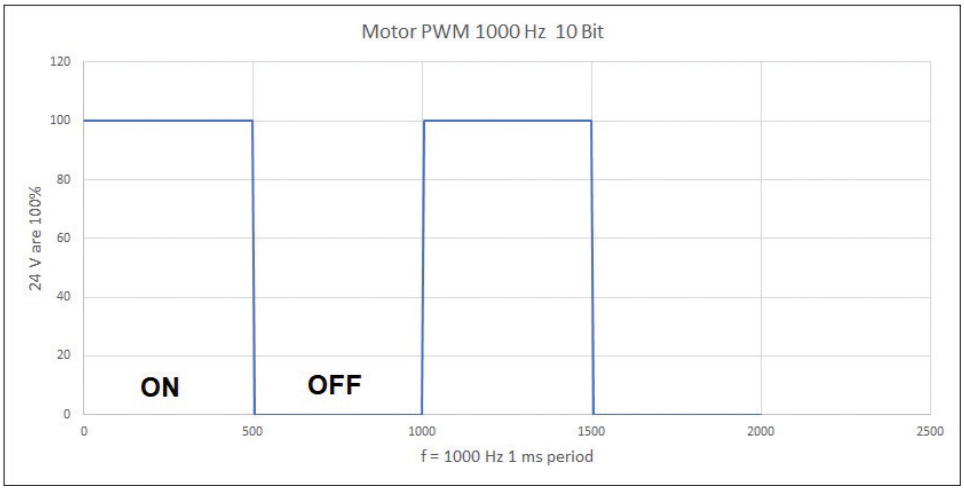


Figure 4.9: PWM output with 1000 Hz

Listung 4.3: Arduino_Motor_PWM_ADC.ino

```
//---------------------------------------------------------------------
//
//            Arduino Timer and PWM Test and interpolation
//            ============================================
//
//      Timer Testprogramm for Arduino ESP32 PID Fuzzy Control Board
//      10ms Timer Interrupt Function
//      Init UART2 for Nextion Display
//      Display ADC1 and ADC2 Values and PWM Values on Display NX3224F028_001
//
//      Converting the voltage values in the range of 0 to 4095
//      into a value from 0 to 1023 for PWM output.
//      Outputting the values to PWM1 and PWM2 channels for the
//      motor or to the heater
//
//
//  Date:     14/07/2024
//  Author:   Josef Bernhardt
//  Hardware: Arduino (ESPRESSIF) ESP32
//
//  File:     Arduino_Motor_PWM_ADC.ino
//---------------------------------------------------------------------

#include <Arduino.h>
#include <Ticker.h>

int LED_BUILTIN = 2;
```

```
// ESP32 UART 2 Pins
#define RXD2 16
#define TXD2 17

// I2C Pins
#define I2CSCL 22
#define I2CSDA 21

// ESP32 ADC Pins
#define ADC1 32
#define ADC2 33
// Motor Sensor Pins
#define MOTSENS1 15
#define MOTSENS2 34
// Buttons
#define BTN1 14
#define BTN2 27

// Relay and Mosfet (BSP76) Pins
#define RELAIS1 18
#define RELAIS2 19
// LED1
#define LED1 5
// PWM1 and PWM2 Pins
#define PWM1 12
#define PWM2 13
// DAC Outputs 0..5V
#define DAC1 25
#define DAC2 26
// One Wire
#define DS18B20 4

Setting the PWM resolution and frequency.

// use 10 or 12 bit precision for LEDC timer
#define PWM_RESOLUTION 10
// use 1000 Hz as a LEDC base frequency
#define PWM_FREQUENZ 1000

// Define 10ms Period
const unsigned long timerInterval = 10;

// Ticker-Objekt for the Timer-Funktion
Ticker timer;
```

```
// Counter
int counter10ms=0;

// When 1 second is elapsed
bool OneSecond = false;

// Counter for seconds
int CounterSeconds = 0;

// Timer-Funktion
// Called every 10 ms
void timerCallback()
{
  // Inc 10ms Counter
  counter10ms++;
  // If one Second
  if(counter10ms==100)
  {
     counter10ms = 0;
     // One Second
     OneSecond = true;
  }
}

// Sends Data to Nextion Display in a textbox
void Nextion_Send_Text_to_Textbox(char *textbox,String sfloat)
{
        Serial2.write(textbox);
        Serial2.write(".txt=\"");
        Serial2.print(sfloat);
        Serial2.write("\"");
        Serial2.write(0xFF);
        Serial2.write(0xFF);
        Serial2.write(0xFF);
}

// Initializing the hardware
void setup()
{
  // Init USB UART
  Serial.begin(115200);
  // Nextion Display
  // TFTSerial.begin(9600);
  Serial2.begin(9600, SERIAL_8N1, RXD2, TXD2);
  // Init Hardware
  pinMode(LED_BUILTIN, OUTPUT);
```

```
  pinMode(LED1, OUTPUT);
  pinMode(RELAIS1, OUTPUT);
  pinMode(RELAIS2, OUTPUT);
  pinMode(BTN1, INPUT);
  pinMode(BTN2, INPUT);

  pinMode(ADC1, INPUT);
  pinMode(ADC2, INPUT);

  Nextion_Send_Text_to_Textbox("t1","Voltage 1");
  Nextion_Send_Text_to_Textbox("t2"," PWM Value 1 ");
  Nextion_Send_Text_to_Textbox("t3"," ");
  Nextion_Send_Text_to_Textbox("t4","Voltage 2");
  Nextion_Send_Text_to_Textbox("t5"," PWM Value 2 ");
  Nextion_Send_Text_to_Textbox("t6"," Seconds");

  // Sets up a channel (0-15), a PWM duty cycle frequency, and a PWM resolution
(1 - 16 bits)
  ledcAttach(PWM1, PWM_FREQUENZ, PWM_RESOLUTION);
  ledcAttach(PWM2, PWM_FREQUENZ, PWM_RESOLUTION);

  // Init Timer with 10ms
  timer.attach_ms(timerInterval, timerCallback);
}

void loop()
{
  // Read ADC1 value
  int voltage1 = analogRead(ADC1);
  // Read ADC2 value
  int voltage2 = analogRead(ADC2);
  // Interpolate voltage 1
  int scaledValue1 = map(voltage1, 0, 4095, 0, 1023);
  int scaledValue2 = map(voltage2, 0, 4095, 0, 1023);

  ledcWrite(PWM1, scaledValue1);
  ledcWrite(PWM2, scaledValue2);

  // Check if one second has passed
  if (OneSecond) {
    // Reset the boolean variable OneSecond
    OneSecond = false;
    CounterSeconds++;

    // Sends variable to USB UART and Seriell Monitor
```

```
    Serial.print("ADC1:");
    Serial.print(voltage1);
    Serial.print(" PWM1:");
    Serial.print(String(scaledValue1));

    Serial.print(" ADC2:");
    Serial.print(voltage2);
    Serial.print(" PWM2:");
    Serial.println(String(scaledValue2));

    // Sends variable to display
    Nextion_Send_Text_to_Textbox("txtT1",String(voltage1));
    Nextion_Send_Text_to_Textbox("txtT2",String(scaledValue1));
    Nextion_Send_Text_to_Textbox("txtT3"," ");
    Nextion_Send_Text_to_Textbox("txtT4",String(voltage2));
    Nextion_Send_Text_to_Textbox("txtT5",String(scaledValue2));
    Nextion_Send_Text_to_Textbox("txtT6",String(CounterSeconds));
    // Switching LED
    digitalWrite(LED_BUILTIN,!digitalRead(LED_BUILTIN));
    digitalWrite(LED1,!digitalRead(LED1));

  }
  // Additional code for the main loop
}
```

The Arduino_ADC_PWM_Mosfet.ino programme can be adapted to your own requirements for controlling LEDs or other loads. It would be very well suited as a basis for LED controllers. The number of PWM channels can be extended at any time.

Listing 4.4 Arduino_ADC_PWM_Mosfet.ino

```
//-----------------------------------------------------------------------
//
//            Arduino Timer and PWM Test and interpolation
//            ============================================
//
//      Testprogramm for Arduino ESP32 PID Fuzzy Control Board
//      10ms Timer Interrupt Function
//      Init UART2 for Nextion Display
//      Display ADC1 and ADC2 Values and PWM Values on Display NX3224F028_001
//
//      Converting the voltage values in the range of 0 to 4095
//      into a value from 0 to 1023 for PWM output.
//      Outputting the values to PWM1 and PWM2 channels for the
//      motor or to the heater
//
```

```
//
//  Date:     04/07/2024
//  Author:   Josef Bernhardt
//  Hardware: Arduino (ESPRESSIF) ESP32
//
//  File:     Arduino_ADC_PWM_Mosfet.ino
//-----------------------------------------------------------------------

#include <Arduino.h>
#include <Ticker.h>

int LED_BUILTIN = 2;

// ESP32 UART 2 Pins
#define RXD2 16
#define TXD2 17

// I2C Pins
#define I2CSCL 22
#define I2CSDA 21

// ESP32 ADC Pins
#define ADC1 32
#define ADC2 33
// Motor Sensor Pins
#define MOTSENS1 15
#define MOTSENS2 34
// Buttons
#define BTN1 14
#define BTN2 27

// Relay and Mosfet (BSP76) Pins
#define RELAIS1 18
#define RELAIS2 19
// LED1
#define LED1 5
// PWM1 and PWM2 Pins
#define PWM1 12
#define PWM2 13
// DAC Outputs 0..5V
#define DAC1 25
#define DAC2 26
// One Wire
#define DS18B20 4

// use 10 or 12 bit precision for LEDC timer
```

```
#define PWM_RESOLUTION 10
// use 1000 Hz as a LEDC base frequency
#define PWM_FREQUENZ 1000

// Define 10ms Period
const unsigned long timerInterval = 10;

// Ticker-Objekt for the Timer-Funktion
Ticker timer;

// Counter
int counter10ms=0;

// When 1 second is elapsed
bool OneSecond = false;

// Counter for seconds
int CounterSeconds = 0;

/*
Set the voltage at ADC1 with the potentiometer to e.g. 1 volt.
Enter the value voltage1 e.g. 1141 in the display for the corresponding
value "float ADC[7]". Repeat this with the other voltages
from 100 mV to 3.3 volts.
12 Bit ADC-Values from 0 to 4095
*/
float ADC[] = {0,   0,    123,   249, 496,  817,  1141, 1456, 1778, 2416, 3067,
3922, 4095, 4095, 4095};
// (Ua) in volts values applied to ADC by calibrator
float Ua[] = {0.00, 0.10, 0.20, 0.30, 0.50, 0.75, 1.00, 1.25, 1.50, 2.00, 2.50,
3.00, 3.07, 3.15, 3.30};
int n_adc = 15;

// Linear interpolation
// Input Parameters:
// n = Number of values -1
// x = x Array
// y = y Array
// xn = Value to be calculated
float readvalue_linear(int n,float *x,float *y, float xn)
{
    int j = 0;
    // Determine interval of xn
    while (j < n-1 && xn > x[j+1]) j++;
    // interpolate xn
    return ((y[j+1]-y[j]) / (x[j+1]-x[j])) * (xn-x[j]) + y[j];
```

```
}


// Timer-Funktion
// Called every 10 ms
void timerCallback()
{
  // Inc 10ms Counter
  counter10ms++;
  // If one Second
  if(counter10ms==100)
  {
     counter10ms = 0;
     // One Second
     OneSecond = true;
  }
}

// Sends Data to Nextion Display in a textbox
void Nextion_Send_Text_to_Textbox(char *textbox,String sfloat)
{
        Serial2.write(textbox);
        Serial2.write(".txt=\"");
        Serial2.print(sfloat);
        Serial2.write("\"");
        Serial2.write(0xFF);
        Serial2.write(0xFF);
        Serial2.write(0xFF);
}

// Initializing the hardware
void setup()
{
  // Init USB UART
  Serial.begin(115200);
  // Nextion Display
  // TFTSerial.begin(9600);
  Serial2.begin(9600, SERIAL_8N1, RXD2, TXD2);
  // Init Hardware
  pinMode(LED_BUILTIN, OUTPUT);
  pinMode(LED1, OUTPUT);
  pinMode(RELAIS1, OUTPUT);
  pinMode(RELAIS2, OUTPUT);
  pinMode(BTN1, INPUT);
  pinMode(BTN2, INPUT);
```

```
  pinMode(ADC1, INPUT);
  pinMode(ADC2, INPUT);

  Nextion_Send_Text_to_Textbox("t1","Voltage 1");
  Nextion_Send_Text_to_Textbox("t2","PWM Value 1 ");
  Nextion_Send_Text_to_Textbox("t3","PWM_FREQ");
  Nextion_Send_Text_to_Textbox("t4","Voltage 2");
  Nextion_Send_Text_to_Textbox("t5","PWM Value 2 ");
  Nextion_Send_Text_to_Textbox("t6","");

  // Setup timer and attach timer to a led pin
  ledcAttach(PWM1, PWM_FREQUENZ, PWM_RESOLUTION);
  ledcAttach(PWM2, PWM_FREQUENZ, PWM_RESOLUTION);

  // Init Timer with 10ms
  timer.attach_ms(timerInterval, timerCallback);
}

void loop()
{
  // Read ADC1 value
  int voltage1 = analogRead(ADC1);
  // Read ADC2 value
  int voltage2 = analogRead(ADC2);
  // Interpolate voltage 1
  int scaledValue1 = map(voltage1, 0, 4095, 0, 1023);
  int scaledValue2 = map(voltage2, 0, 4095, 0, 1023);

  ledcWrite(PWM1, scaledValue1);
  ledcWrite(PWM2, scaledValue2);

  // Check if one second has passed
  if (OneSecond) {
    // Reset the boolean variable OneSecond
    OneSecond = false;
    CounterSeconds++;

    // Sends variable to USB UART and Seriell Monitor

    Serial.print("ADC1: ");
    Serial.print(voltage1);
    Serial.print(" PWM1: ");
    Serial.print(String(scaledValue1));

    Serial.print(" ADC2: ");
    Serial.print(voltage1);
```

```
    Serial.print(" PWM2: ");
    Serial.println(String(scaledValue2));

    // Sends variable to display
    Nextion_Send_Text_to_Textbox("txtT1",String(voltage1));
    Nextion_Send_Text_to_Textbox("txtT2",String(scaledValue1));
    Nextion_Send_Text_to_Textbox("txtT3",String(PWM_FREQUENZ));
    Nextion_Send_Text_to_Textbox("txtT4",String(voltage2));
    Nextion_Send_Text_to_Textbox("txtT5",String(scaledValue2));
    Nextion_Send_Text_to_Textbox("txtT6","");
    // Switching LED
    digitalWrite(LED_BUILTIN,!digitalRead(LED_BUILTIN));
    digitalWrite(LED1,!digitalRead(LED1));

  }
  // Additional code for the main loop
}
```

Chapter 5 • On-Off and PID controllers

This chapter focuses on two-point and PID controllers, which are widely used in electrical appliances.

5.1 Two-point controller

Two-position controllers, also known as on/off controllers or two-stage controllers, are used in particular where precise control is not required. A common example is the control of heating systems, where the controller switches the heating on when the temperature falls below a certain value and switches it off when the temperature reaches an upper threshold value.

Similar to heating systems, but for cooling systems, the controller switches the cooling system on when the temperature exceeds a certain upper value and switches it off when the temperature falls below a lower threshold.

Many household appliances, such as refrigerators, washing machines or dishwashers, use two-point controllers for various control tasks, such as temperature control or water level control.

The simplicity and reliability of two-point controllers make them a popular choice for applications where control does not need to be highly precise and simple on/off control is sufficient.

An example of the two-point controller is also available for the PT2 simulation. Here is the graphic. The example shows a temperature controller that switches off the heating when a set temperature is exceeded. If the temperature falls below the setpoint, the heating switches on again. A typical area of application is temperature control for refrigerators and hot water tanks in households as well as for solar hot water supply.

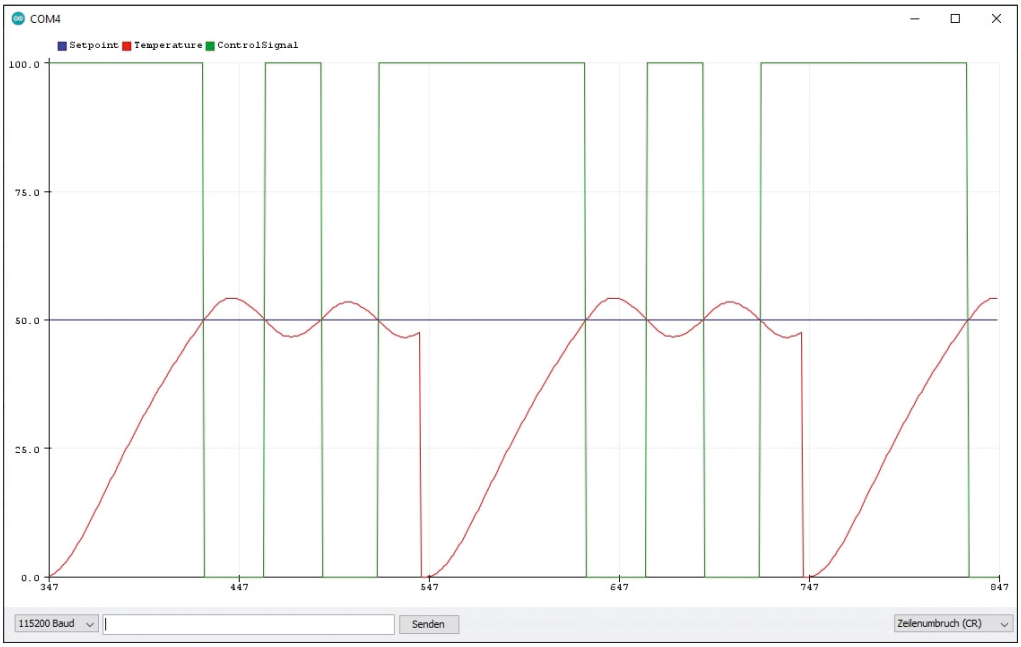


Figure 5.1 Test of a two-point controller

This graphic shows the progression of the setpoint, actual value and control value of the program Arduino_PT2_TwoPoint_Controller.ino

Listing 5.1: Arduino_PT2_TwoPoint_Controller.ino

```
//---------------------------------------------------------------------
//
//            Arduino PT2 System Two Point Controller
//            =======================================
//
//      Testprogramm for Arduino ESP32 PID Fuzzy Control Board
//      10ms Timer Interrupt Function
//      Init UART2 for Nextion Display
//      Display ADC1 and ADC2 Values and PWM Values on Display NX3224F028_001
//      Simulation of two point controller
//
//  Date:     29/04/2024
//  Author:   Josef Bernhardt
//  Hardware: Arduino (ESPRESSIF) ESP32
//
//  File:     Arduino_PT2_TwoPoint_Controller.ino
//---------------------------------------------------------------------

// Include-Bibliotheken
```

```
#include <Arduino.h>

// Pin-Definitionen
#define LED_BUILTIN 2 // LED Pin GPIO 2 from ESP32 )
// LED1
#define LED1 5

// PT1 Simulation of controlled system
float PT1_Simulation(float controlInput, int reset)
{
    // Parameter
    float deltaT = 1.0;
    float ProportionalGain = 5.0;
    float IntegralGain = 5.0;
    float TimeConstant = 400.0;  // Time constant PT1 controlled system
    // Zustandsvariable
    static float output = 0;
    // Reset der statischen Variable
    if (reset == 1)
    {
        output = 0;
    }
    // Model equation, Euler integration
    float d_output = (ProportionalGain * controlInput - output) / TimeConstant;
    output += d_output * deltaT;
    return output;
}
```

The function simulates a PT2 system, which is a simple model for describing second-order dynamic systems. In this case, the system is described by three state variables (output1, output2 and output3), which are calculated iteratively using the Euler method (see Chapter 4).

```
// Simulating a PT2 control system
float PT2_Simulation(float controlInput, int reset)
{
    // Parameter
    float deltaT = 1.0;
    float ProportionalGain = 1.0;
    float IntegralGain = 1.0;
    float DerivativeGain = 1.0;
    float TimeConstant1 = 100.0;
    float TimeConstant2 = 8.0;
    float TimeConstant3 = 10.0;

    // State variables
```

```
    static float output1 = 0;
    static float output2 = 0;
    static float output3 = 0;

    // Reset static Variabel
    if (reset == 1)
    {
        output1 = 0;
        output2 = 0;
        output3 = 0;
    }
    // Model equation, Euler integration
    float d_output1 = (ProportionalGain * controlInput - output1) / TimeConstant1;
    float d_output2 = (IntegralGain * output1 - output2) / TimeConstant2;
    float d_output3 = (DerivativeGain * output2 - output3) / TimeConstant3;

    output1 += d_output1 * deltaT;
    output2 += d_output2 * deltaT;
    output3 += d_output3 * deltaT;

    return output3;
}
```

The two-point controller works on the principle of binary control, in which the output signal is set to either 0% or 100%, depending on the current status of the system. In this program, the controller compares the current temperature with the setpoint and switches the output to 0 if the temperature reaches or exceeds the setpoint. If the temperature falls below the setpoint minus a hysteresis, the output signal is set to 100% to activate the system. The hysteresis prevents frequent switching operations by allowing a certain tolerance in the control behavior so that the system does not react immediately to small changes. Two-point controllers are easy to implement but can lead to oscillations as they switch the system back and forth between the two limit values.

```
// Two-point controller for the PT2 system
float twoPointController(float sollwert,float temperature,float hyst)
{
    static float stellwert;
    if (temperature >= sollwert)
    {
      stellwert = 0.0;
    }
    if (temperature < (sollwert-hyst))
    {
      stellwert = 100.0;
    }
    return stellwert;
```

```
}


// Init hardware
void setup() {
  pinMode(LED_BUILTIN, OUTPUT); //Configuration of the pin as an output
  // Init UART USB
  Serial.begin(115200);
}
```

The main loop simulates a temperature control in which the PT2 system calculates the temperature curve, and the two-point controller determines the control signal based on the difference between the setpoint and the temperature. At the beginning, the PT2 system is reset, the LED is switched and then a 200 ms waiting time is inserted. In the loop over 300 iterations, the temperature is simulated based on the current control signal, which is then regulated accordingly by the two-point controller. The setpoint, temperature and control signal values are output via the serial interface in each iteration, followed by a short delay of 20 ms.

```
// Main Loop
void loop()
{
  // Place your main code here to execute repeatedly:
  float setpoint = 50;
  float temperature = 0;
  float control_signal = 100.0;
  float hysteresis = 0.0;

  // Reset PT2 System
  temperature = PT2_Simulation(0,1);
  // toggle LED
  digitalWrite(LED_BUILTIN, !digitalRead(LED_BUILTIN));
  // wait 200 ms
  delay(200);
  for (int time = 0; time < 300; time++)
  {
      // Simulating the temperature control system
      temperature = PT2_Simulation(control_signal,0);
      // Control signal from the two-point controller
      control_signal = twoPointController(setpoint,temperature,hysteresis);

      Serial.print(" Setpoint:");
      Serial.print(setpoint, 1); // Setpoint with one decimal place
      Serial.print(" Temperature:");
      Serial.print(temperature, 1); // Temperature with one decimal place
      Serial.print(" ControlSignal:");
```

```
        Serial.print(control_signal, 1); // Control signal with one decimal place
        Serial.println(); // New line for the next output
        delay(20);
    }
}
```

5.2 PID controller

The history of PID controllers (proportional-integral-derivative controllers) began around 1920 with the work of Elmer Sperry and Nicholas Minorsky, who were working on the automatic control of ships. Minorsky developed the mathematical basis for the PID controller in 1922 and showed that the combination of proportional, integral and differential components enables stable and precise control. In the 1940s and 1950s, PID controllers became widely used in industry, particularly in process control and automation. With the introduction of digital computers and microprocessors in the 1970s, PID controllers were further developed and could now take on more complex and faster control tasks. Today, PID controllers are a central component of many industrial control systems and are used in numerous areas such as robotics, aerospace, and the automotive industry.

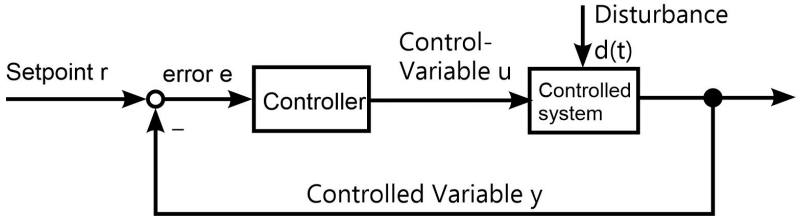


Figure 5.2: PID controller

A PID controller works by measuring the difference e (error) between a desired setpoint r and the current actual value y and using this error to calculate the manipulated variable u. The proportional component (P) responds directly to the current error. The proportional component (P) responds directly to the current error, the integral component (I) sums past errors to correct long-term deviations, and the differential component (D) responds to the rate of change of the error to make rapid corrections. By combining these three components, the PID controller can achieve precise and stable control by continuously making adjustments to minimize the error. This stabilizes the system and brings it to the desired setpoint.

The general controller equation for a PID controller using Kp, Ki and Kd:

$$u(t) = K_p \cdot e(t) + K_i \cdot \int_0^t e(\tau)\,d\tau + K_d \cdot \frac{de(t)}{dt}$$

The general controller equation for a PID controller using Tn and Tv:

$$u(t) = K_p \left(e(t) + \frac{1}{T_n} \int_0^t e(\tau)\, d\tau + T_v \frac{de(t)}{dt} \right)$$

Parameter:

Kp: Proportionality factor
Tn: Reset time (integration time)
Tv: Derivative time (differential time)
Ki: Integral factor
Kd: Differential factor
e(t): Control deviation (error) at time t
u(t): Output signal of the controller at time t

Here are the equations with Ki and Kd for the Arduino:

1. proportional part (P): u_P(t) = Kp * e(t)
2. integral component (I) (discrete): u_I(t) = Ki * sum(e(t) * Δt)
3. differential component (D) (discrete): u_D(t) = Kd * (e(t) - e(t-1)) / Δt

The complete PID controller equation then looks as follows:

u(t) = u_P(t) + u_I(t) + u_D(t) = Kp * e(t) + Ki * sum(e(t) * Δt) + Kd * (e(t) - e(t-1)) / Δt

In a more compact form: u(t) = Kp * (e(t)) + Ki * sum(e(t) * Δt) + Kd * (e(t) - e(t-1)) / Δt

Here are the equations with Tn and Tv

1. proportional part (P): u_P(t) = Kp * e(t)
2. integral component (I) (discrete): u_I(t) = Kp * (1 / Tn) * sum(e(t) * Δt)
3. differential component (D) (discrete): u_D(t) = Kp * Tv * (e(t) - e(t-1)) / Δt

The complete PID controller equation then looks as follows:

u(t) = u_P(t) + u_I(t) + u_D(t) = Kp * e(t) + Kp * (1 / Tn) * sum(e(t) * Δt) + Kp * Tv * (e(t) - e(t-1)) / Δt

In a more compact form: u(t) = Kp * (e(t) + (1 / Tn) * sum(e(t) * Δt) + Tv * (e(t) - e(t-1)) / Δt)

Unfortunately, in practice it is often difficult to find the correct parameters Kp, Ki and Kd or Tn and Tv for setting the controller. Various methods have proven themselves in practice.

5.3 Parameter determination using Ziegler-Nichols

The Ziegler-Nichols oscillation method, also known as the 'Ultimate Gain and Period' method, determines the PID parameters by experimentation. First, the proportional controller (without integral and differential components) is set so that the system just begins to produce stable, continuous oscillations. The proportional amplification factor Ku and the oscillation period Tu of these oscillations are measured. These values are then used to calculate the PID parameters:

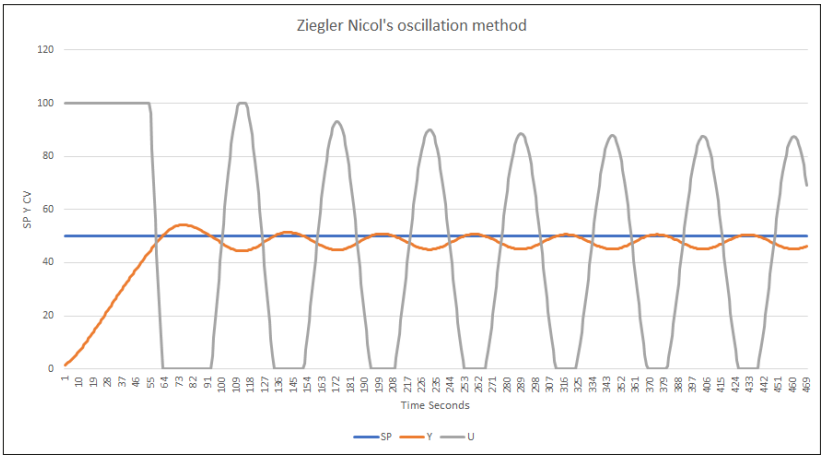


Figure 5.3: Chart with Kp 18 Ki 0 Kd 0

Ku is the value at which uniform oscillation occurs.

The equations for calculating the PID parameters according to Ziegler Nicols:

Parameter	Equation	Example Ku = 18 Tu = 57
Kp	Kp = 0,6 * Ku	10,8
Ti	Ti = 0,5 *Tu	28,5
Td	Td = 0,125 * Tu	7,125
Ki	Ki = Kp / Ti	0,379
Kd	Kd = Kp * Td	76,95

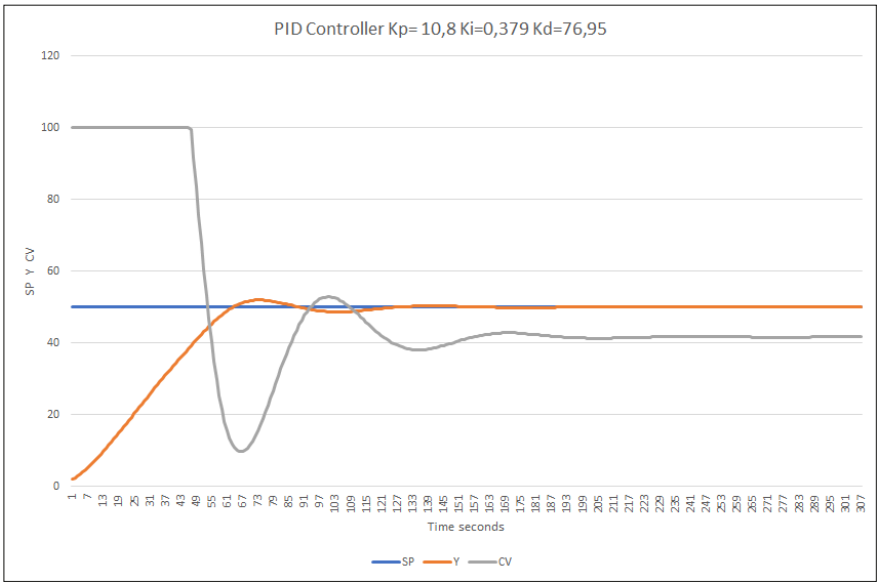


Figure 5.4: PID controller example

5.4 PID parameter determination with the relay method

The relay method for determining the parameters of PID controllers was developed by Åström and Hägglund. This method, which is also known as the Ziegler-Nichols method of the second kind, uses an on/off relay to set the system into a periodic oscillation. The PID parameters can be calculated from the measured oscillation period and amplitude. Åström and Hägglund published their work on this in the 1980s, making a significant contribution to the practical application and fine-tuning of PID controllers.

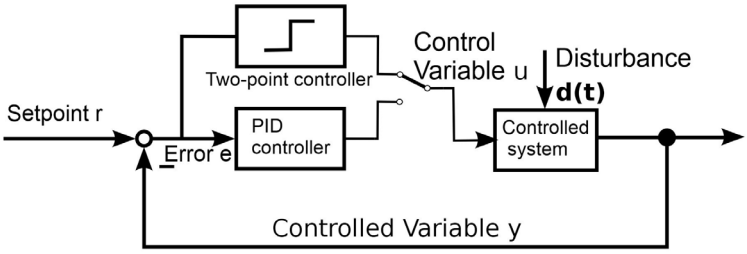


Figure 5.5: Relay method block diagram

To do this, the PID controller is replaced by a two-point controller in the program and the values are recorded. The program calculates the PID parameters Kp, Ki and Kd from these values a, Pu and h.

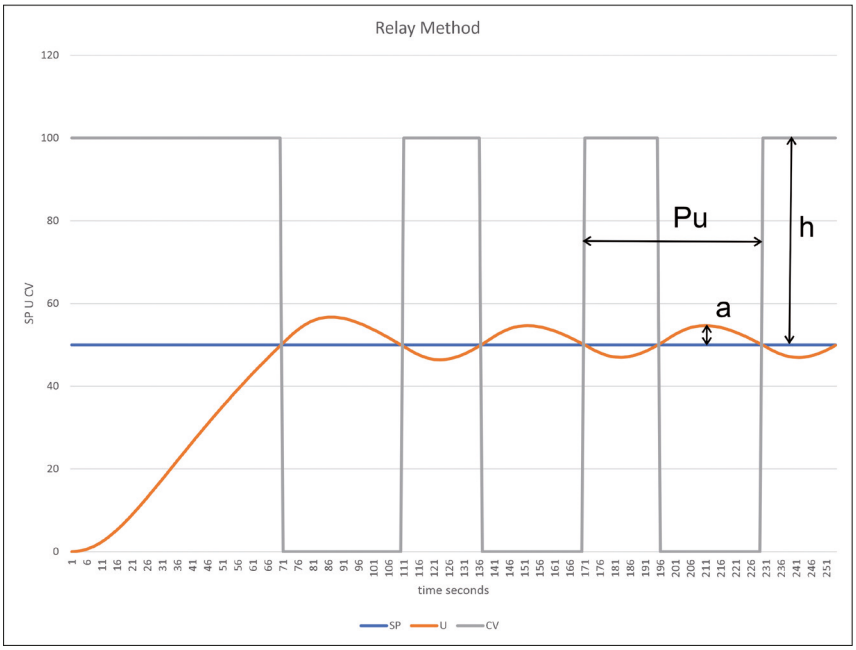


Figure 5.6: Relay method Parameter determination

Parameter	Example	Equation Pu=58 h=50 a=4,65
Ku	Ku = (4 * h) / (3.14159 * a)	Ku = 13.6908
Omega u	omega u = (2 * 3.14159) / Pu	Omega u = 0.1083
Kp	Kp = 0.45 * Ku	Kp = 6.1608
Ti	Ti = Pu / 1.2	Ti = 48.3333
Td	Td = Pu * 0.1	Td = 5.8000
Ki	Ki = Kp / Ti	Ki = 0.1275
Kd	Kd = Kp * Td	Kd = 35.7329

Website for calculating the PID controller parameters for the relay method:

https://www.bernhardt.de/relaymethod/

5.5 Arduino PID controller with PT2 simulation

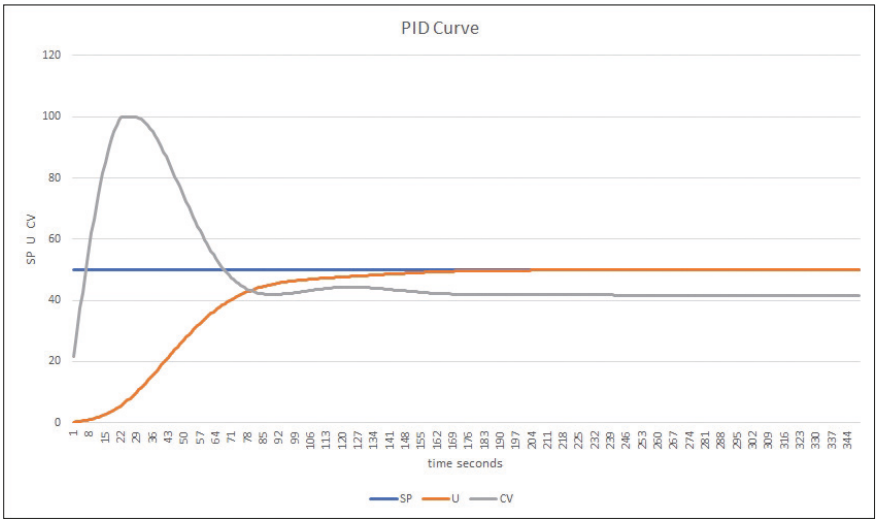


Figure 5.7: PID example controller position

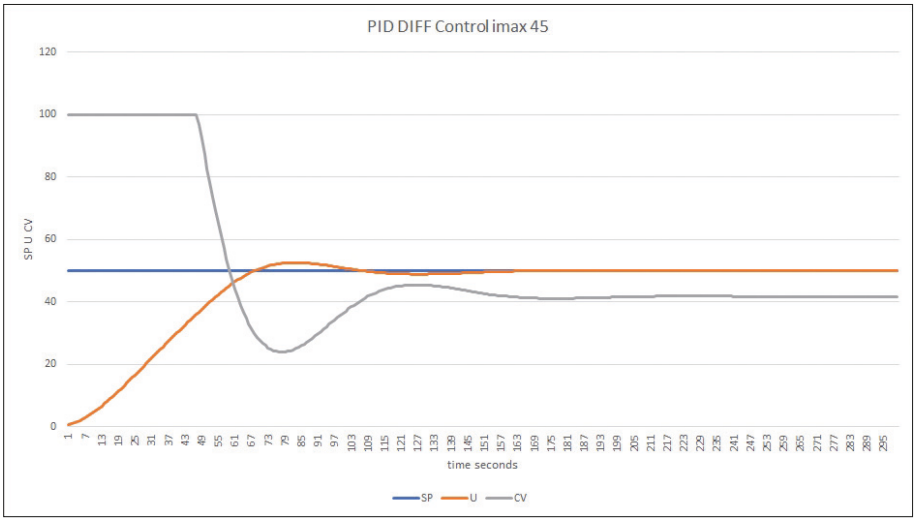


Figure 5.8: PID example differenz

As you can see from the charts, the two PID functions for the positional algorithm and the difference equations behave differently.

Listing 5.2: Arduino_PT2_PID_Controller.ino

```
//---------------------------------------------------------------------
//
//            Arduino PT2 PID Controller for PT2 Simulation
//            =============================================
//
//      Testprogramm for Arduino ESP32 PID Fuzzy Control Board
//      PID Controller Test
//
//
//  Date:     05/06/2024
//  Author:   Josef Bernhardt
//  Hardware: Arduino (ESPRESSIF) ESP32
//
//  File:     Arduino_PT2_PID_Controller.ino
//---------------------------------------------------------------------

// Include
#include <Arduino.h>

// Pin-Definitionen
#define LED_BUILTIN 2 // Pin,  GPIO 2 from ESP32 )
// LED1
#define LED1 5
#define Kp 6.1608
#define Ki 0.1275
#define Kd 35.7329
/*
 *
// float ProportionalGain = 1.25;
Kp = 5.4881
Ki = 0.1098
Kd = 32.9286
*/


// Simulating a PT2 control system
float PT2_Simulation(float controlInput, int reset)
{
    // Parameter
    float deltaT = 1.0;
    float ProportionalGain = 1.2;
    float IntegralGain = 1.0;
    float DerivativeGain = 1.0;
    float TimeConstant1 = 100.0;
```

```
    float TimeConstant2 = 8.0;
    float TimeConstant3 = 10.0;

    // state variable
    static float output1 = 0;
    static float output2 = 0;
    static float output3 = 0;

    // Reset static Variabel
    if (reset == 1)
    {
        output1 = 0;
        output2 = 0;
        output3 = 0;
    }
    // Model equation, Euler integration
    float d_output1 = (ProportionalGain * controlInput - output1) / TimeConstant1;
    float d_output2 = (IntegralGain * output1 - output2) / TimeConstant2;
    float d_output3 = (DerivativeGain * output2 - output3) / TimeConstant3;

    output1 += d_output1 * deltaT;
    output2 += d_output2 * deltaT;
    output3 += d_output3 * deltaT;

    return output3;
}
```

The PID controller calculates the control signal based on the difference between the setpoint (w) and actual value (x) using a difference equation. The proportional component (Kp) is based directly on the current deviation, the integral component (Ki) adds up all deviations, whereby a limitation (anti-windup) limits the integral compensation. The differential component (Kd) takes into account the change in deviation compared to the previous step to react to sudden changes. Finally, the calculated control signal is limited to a range between 0 and 100 to ensure that it remains within reasonable limits.

```
// PID controller with difference equation
float pid_regler_diff(float w, float x)
{
  float e;
  float imax = 45;
  static float y,e_alt,isum = 0;

  // Calculate deviation e
  e = w - x;
  isum +=e;
  // Anti Windup
```

```
  // Limit integral part

  if (Ki * isum > imax)
  {
    isum = imax / Ki;
  }
  if (Ki * isum < -imax)
  {
    isum = -imax / Ki;
  }

  // Calculate control signal here as a float
  y = Kp * e + Ki * isum + Kd * (e-e_alt);
  // Save deviation for the next iteration
  e_alt = e;
  // Limit control signal
  if (y > 100.0)
  {
    y = 100.0;
  }
  if (y < 0.0)
  {
    y = 0.0;
  }
  return y;
}
```

This function implements a PID controller with a position algorithm in which the control signal (y) is calculated directly from the current and past deviations. The coefficients q0, q1 and q2 represent the weightings of the proportional (Kp), integral (Ki) and differential (Kd) components, which are applied to the current deviation and the two previous deviations respectively. The new control signal is calculated by adding the weighted deviations to the previous control signal, resulting in the position control. As with the difference-based PID controller, the control signal is finally limited, and the old value is saved for the next iteration.

```
// PID controller (positioning algorithm)
float pid_regler_stell(float w, float x)
{
  float y,q0,q1,q2;
  static float ealt,ealt2,yalt = 0;
  float e = w - x;;

  q0 = Kp+Ki+Kd;
  q1 = -Kp-2.0*Kd;
  q2 = Kd;
```

```
  //Control
  y = yalt + q0*e + q1*ealt + q2*ealt2;
  ealt2 = ealt;
  ealt = e;
  // Limit actuating value
  if (y > 100.0)
  {
    y = 100.0;
  }
  if (y < 0.0)
  {
    y = 0.0;
  }
  yalt = y;
  return y;
}

// Two-point controller for the PT2 system
float twoPointController(float sollwert,float temperature,float hyst)
{
    static float stellwert;
    if (temperature >= sollwert)
    {
      stellwert = 0.0;
    }
    if (temperature < (sollwert-hyst))
    {
      stellwert = 100.0;
    }
    return stellwert;
}
```

The PeakAndPeriod() function monitors the temperature to calculate peak values and periods in relation to a setpoint. If the temperature exceeds the setpoint, it saves the difference between the current temperature and the setpoint as a peak value. As soon as the temperature rises from a lower value to the setpoint, the function records the elapsed time as a period (Tu) and resets the counter. An LED is briefly switched on to mark this moment. The counter i is incremented in each iteration and returns the number of steps since the setpoint was last exceeded. This function is used to determine the values for the parameter calculation.

```
int PeakAndPeriod(float setpoint,float temperature, float *peak, float * Tu)
{
  static float temperature_old = 0;
  static int i=0;
```

```
  if (temperature>setpoint) *peak = temperature - setpoint;
  if (temperature_old < setpoint && temperature >= setpoint)
  {
    digitalWrite(LED1,1);
    *Tu = i;
    i= 0;
    *peak = 0;
    digitalWrite(LED1,0);
  }
  temperature_old = temperature;
  i++;
  return i;
}

// Init hardware
void setup() {
  pinMode(LED_BUILTIN, OUTPUT); //Configuration of the pin as an output
  pinMode(LED1, OUTPUT);        //Configuration of the pin as an output
  // Init UART USB
  Serial.begin(115200);
}

// Main Loop
void loop()
{
  // Place your main code here to execute repeatedly:
  float setpoint = 50;
  float temperature = 0;
  float control_signal = 100.0;
  float hysteresis = 0.0;

  float peak = 0;
  float Tu = 0;

  // Reset PT2 System
  temperature = PT2_Simulation(0,1);
  // toggle LED
  digitalWrite(LED_BUILTIN, !digitalRead(LED_BUILTIN));
  // wait 200 ms
  delay(200);
  for (int time = 0; time < 500; time++)
  {
      // Simulating the temperature control system
      temperature = PT2_Simulation(control_signal,0);
      // Control signal from the two-point controller
```

```
      // control_signal = pid_regler_stell(setpoint,temperature);
      control_signal = pid_regler_diff(setpoint,temperature);

      // control_signal = twoPointController(setpoint,temperature,hysteresis);
/*
      int period = PeakAndPeriod(setpoint,temperature, &peak, &Tu);

      Serial.print(" Period:");
      Serial.print(Tu);                 // Setpoint with one decimal place
      Serial.print(" Peak:");
      Serial.print(peak,2);             // Setpoint with one decimal place
 */
      Serial.print(" Setpoint:");
      Serial.print(setpoint, 1);        // Setpoint with one decimal place
      Serial.print(" Temperature:");
      Serial.print(temperature, 2);     // Temperature with one decimal place
      Serial.print(" ControlSignal:");
      Serial.print(control_signal, 1);  // Control signal with one decimal place
      Serial.println();                 // New line for the next output
      delay(200);
  }
}
```

5.6 Arduino PID controller with real controlled system (ALU block)

The next example shows the temperature curve of a real PID controller for heating an aluminium block measuring 80x30x15 mm as calculated in the previous chapter. The PID parameters were determined using the relay method.

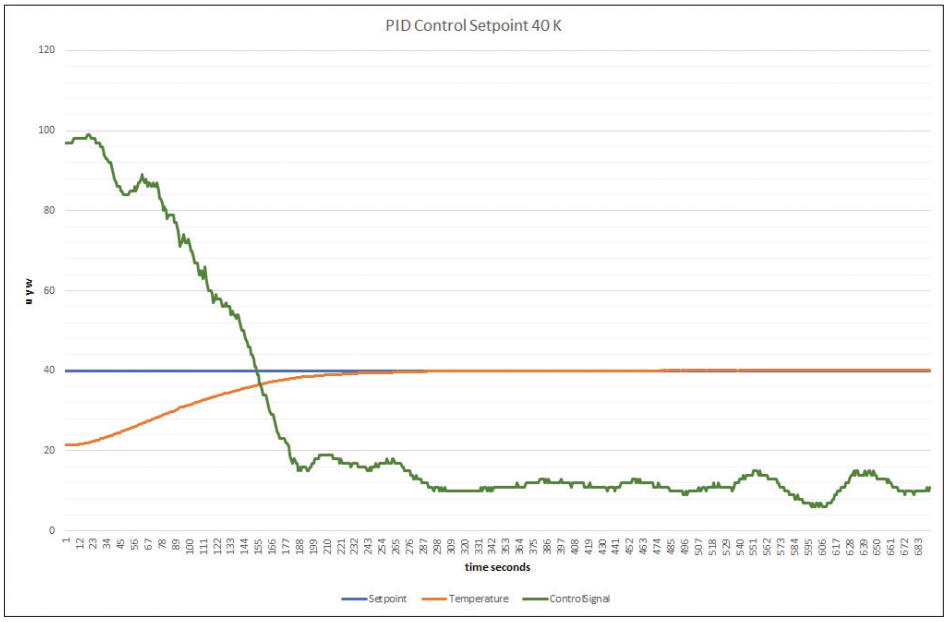


Figure 5.9: PID temperature control

Listing 5.3: Arduino_PhysicalHeater_PID.ino

```
//-----------------------------------------------------------------------
//
//            Arduino PID Physical Heater
//            ===================================
//
//      Control Software for Arduino ESP32 PID Fuzzy Control Board
//      10ms Timer Interrupt Function
//      Init UART2 for Nextion Display
//      Display ADC1 and ADC2 Values on Display NX3224F028_001
//
//      Reading NTC1 Sensor / Set the Setpoint with POT2
//      PID Controller Example
//      Outputting the pulses to RELAIS1 for the heater
//
//
//  Date:     04/07/2024
//  Author:   Josef Bernhardt
//  Hardware: Arduino (ESPRESSIF) ESP32
//
//  File:     Arduino_PhysicalHeater_PID.ino
//-----------------------------------------------------------------------

#include <Arduino.h>
```

```
#include <Ticker.h>

int LED_BUILTIN = 2;

// ESP32 UART 2 Pins
#define RXD2 16
#define TXD2 17

// I2C Pins
#define I2CSCL 22
#define I2CSDA 21

// ESP32 ADC Pins
#define ADC1 32
#define ADC2 33
// Motor Sensor Pins
#define MOTSENS1 15
#define MOTSENS2 34
// Buttons
#define BTN1 14
#define BTN2 27

// Relay and Mosfet (BSP76) Pins
#define RELAIS1 18
#define RELAIS2 19
// LED1
#define LED1 5
// PWM1 and PWM2 Pins
#define PWM1 12
#define PWM2 13
// DAC Outputs 0..5V
#define DAC1 25
#define DAC2 26
// One Wire
#define DS18B20 4

#define FILTER_SIZE 10

// Define 10ms Period
const unsigned long timerInterval = 10;

// Ticker-Objekt for the Timer-Funktion
Ticker timer;

// Counter
int counter10ms=0;
```

```
int counterRelais1 = 0;

// When 1 second is elapsed
bool OneSecond = false;

// Counter for seconds
int CounterSeconds = 0;

// Arrays for NTC10K calculation
float Voltages[18] = {0};
float Temperatures[18] = {0};
int n_ntc = 18;

float VoltageESP32 = 3.3;

/*
Set the voltage at ADC1 with the potentiometer to e.g. 1 volt.
Enter the value voltage1 in the display for the corresponding
value "float ADC[7]". Repeat this with the other voltages
from 100 mV to 3.3 volts.
12 Bit ADC-Values from 0 to 4095
*/
float ADC[] = {0,   0,    123,   249, 496,  817,  1141, 1456, 1778, 2416, 3067,
3922, 4095, 4095, 4095};
// (Ua) in volts values applied to ADC by calibrator
float Ua[] = {0.00, 0.10, 0.20, 0.30, 0.50, 0.75, 1.00, 1.25, 1.50, 2.00, 2.50,
3.00, 3.07, 3.15, 3.30};
int n_adc = 15;
/*
https://www.bernhardt.de/relaymethod/
Calculated values with Relay Method:
Ku = 75.7881
OmegaU = 0.0610
Kp = 34.1047
Ki = 0.3973
Kd = 351.2780
Ti = 85.8333
Td = 10.3000
*/
// Parameter ALU Block Heater 19,2 Watt
#define Kp 34.1047
#define Ki 0.3973
#define Kd 351.2780

void Heat1_ON(int value)
```

```
{
  //Serial.println("ON");
  if (value > 0)
  {
    counterRelais1 = value;
    digitalWrite(RELAIS1,1);
    digitalWrite(LED1,1);
  }
}

void Heat1_OFF()
{
  //Serial.println("OFF");
  digitalWrite(RELAIS1,0);
  digitalWrite(LED1,0);
}

int mapInteger(int value, int fromLow, int fromHigh, int toLow, int toHigh) {
    return (value - fromLow) * (toHigh - toLow) / (fromHigh - fromLow) + toLow;
}

float mapFloat(float x, float in_min, float in_max, float out_min, float out_max)
{
  return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
}

// Linear interpolation
// Input Parameters:
// n = Number of values -1
// x = x Array
// y = y Array
// xn = Value to be calculated
float readvalue_linear(int n,float *x,float *y, float xn)
{
    int j = 0;
    // Determine interval of xn
    while (j < n-1 && xn > x[j+1]) j++;
    // interpolate xn
    return ((y[j+1]-y[j]) / (x[j+1]-x[j])) * (xn-x[j]) + y[j];
}

// Moving Average Filter
float filter_NTC1(float value)
{
    static float values[FILTER_SIZE] = {0};
    float sum = 0;
```

```
    int i;
    for (i=0; i<FILTER_SIZE-1;i++)
    {
      values[i] = values[i+1];
    }
    values[FILTER_SIZE-1] = value;
    for (i=0; i<FILTER_SIZE;i++)
    {
      sum +=values[i];
    }
    return sum / (float)FILTER_SIZE;
}

// Filter of the NTC1 sensor
int filterControlValue(int newValue) {
    // Array for saving the last FILTER_SIZE values
    static int values[FILTER_SIZE] = {0};
    static int index = 0;
    int sum = 0;
    // Updating the value at the current index position
    values[index] = newValue;
    // Increment the index for the next value
    index = (index + 1) % FILTER_SIZE;
    // Calculation of the average of the values in the array
    for (int i = 0; i < FILTER_SIZE; ++i) {
        sum += values[i];
    }
    return sum / FILTER_SIZE;
}

// Function to calculate the resistance of an NTC thermistor based on temperature
float calculateResistance(float T)
{
    // Constants
    const float R25 = 10000.0; // Nominal resistance of the thermistor at 25°C
    const float B = 3950.0;   // B-value of the thermistor see data sheet
    return R25 * exp(B * ((1.0 / (T + 273.15)) - (1.0 / (25 + 273.15))));
}

// Voltage from voltage divider 10k input NTC resistor
float calulateVoltage(float Rt)
{
    float R25 = 10000;
    float UE = VoltageESP32;
    float Ua=(UE*Rt)/(Rt+R25);
    return Ua;
```

```
}

// Init Array with NTC Resist and Temperature
void initArrayNTC()
{
    // Temperature range from -35°C to 135°C.
    int min_temp = -35;
    int max_temp = 135;
    float voltage;
    int loop = sizeof(Voltages) / sizeof(Voltages[0]) - 1;
    // Loop over the temperature range
    for (float temp = min_temp; temp <= max_temp; temp+=10)
    {
        // Calculating the resistance for the current temperature
        float resistance = calculateResistance(temp);
        float voltage = calulateVoltage(resistance);
        // Saving the values for temperature and voltage in arrays
        Temperatures[loop] = temp;
        Voltages[loop] = voltage;
        loop--;
    }
}

// Timer-Funktion
// Called every 10 ms
void timerCallback()
{
  // Inc 10ms Counter
  counter10ms++;
  counterRelais1--;
  // If one Second
  if(counter10ms==100)
  {
     counter10ms = 0;
     // One Second
     OneSecond = true;
  }
  if(counterRelais1==0)
  {
     Heat1_OFF();
  }
}

// Sends Data to Nextion Display in a textbox
void Nextion_Send_Text_to_Textbox(char *textbox,String sfloat)
{
```

```
        Serial2.write(textbox);
        Serial2.write(".txt=\"");
        Serial2.print(sfloat);
        Serial2.write("\"");
        Serial2.write(0xFF);
        Serial2.write(0xFF);
        Serial2.write(0xFF);
}

// PID controller (positioning algorithm)
```

The function implements a PID controller to calculate the output `y` based on the setpoint `w` and the actual value `x`. It computes the current error and applies controller gains q0,q1,q_0, q_1,q0,q1, and q2q_2q2 for proportional, integral, and derivative terms, respectively, using a recursive formula. To ensure stability, the function limits the output value to a range of 0 to 100 and updates historical error and output values for the next iteration. Finally, it returns the calculated control signal as the output.

```
float pid_regler_stell(float w, float x)
{
  float y, q0, q1, q2;
  static float ealt, ealt2, yalt = 0; // Previous error values and output value
  float e = w - x; // Calculate current error
  // Calculate controller weights
  q0 = Kp + Ki + Kd;    // Sum of all controller gains
  q1 = -Kp - 2.0 * Kd;  // Previous error term
  q2 = Kd;              // Derivative term
  // Controller equation
  y = yalt + q0 * e + q1 * ealt + q2 * ealt2;
  // Update previous error values
  ealt2 = ealt;
  ealt = e;
  // Limit output value
  if (y > 100.0)
  {
    y = 100.0;
  }
  if (y < 0.0)
  {
    y = 0.0;
  }
  // Update previous output value
  yalt = y;
  // Return output value
  return y;
}
```

The function `pid_regler_diff` implements a PID controller using a difference equation to compute the output `y` based on the setpoint `w` and the actual value `x`. It calculates the error `e`, updates the integral term with anti-windup logic to limit its contribution, and incorporates the proportional, integral, and derivative terms into the control signal. The output is constrained to the range of 0 to 100 to ensure stability, and the previous error is stored for the next iteration. Finally, the function returns the calculated control signal as a float.

```
// PID controller with difference equation
float pid_regler_diff(float w, float x)
{
  float e;
  float imax = 50;
  static float y,e_alt,isum = 0;

  // Calculate deviation e
  e = w - x;
  isum +=e;
  // Anti Windup
  // Limit integral part
  if (Ki * isum > imax)
  {
    isum = imax / Ki;
  }
  if (Ki * isum < -imax)
  {
    isum = -imax / Ki;
  }
  // Calculate control signal here as a float
  y = Kp * e + Ki * isum + Kd * (e-e_alt);
  // Save deviation for the next iteration
  e_alt = e;
  // Limit control signal
  if (y > 100.0)
  {
    y = 100.0;
  }
  if (y < 0.0)
  {
    y = 0.0;
  }
  return y;
}
```

The two-point controller adjusts the output based on the actual value and the setpoint. If the actual value meets or exceeds the setpoint, the output is set to 0. If the actual value falls below the setpoint, minus the hysteresis value, the output is set to 100. The hysteresis value helps prevent frequent switching of the output due to small fluctuations around the setpoint.

```
// Two-point controller
float twoPointController(float setpoint, float actualValue, float hysteresis)
{
    static float controlOutput = 0.0; // Initialize the static variable
    if (actualValue >= setpoint)
    {
        controlOutput = 0.0;
    }
    else if (actualValue < (setpoint - hysteresis))
    {
        controlOutput = 100.0;
    }
    return controlOutput;
}

// Initializing the hardware
void setup()
{
  // Init USB UART
  Serial.begin(115200);
  // Nextion Display
  // TFTSerial.begin(9600);
  Serial2.begin(9600, SERIAL_8N1, RXD2, TXD2);
  // Init Hardware
  pinMode(LED_BUILTIN, OUTPUT);
  pinMode(LED1, OUTPUT);
  pinMode(RELAIS1, OUTPUT);
  pinMode(RELAIS2, OUTPUT);
  pinMode(BTN1, INPUT);
  pinMode(BTN2, INPUT);

  pinMode(ADC1, INPUT);
  pinMode(ADC2, INPUT);

  Nextion_Send_Text_to_Textbox("t1","VoltageNTC");
  Nextion_Send_Text_to_Textbox("t2","Temp NTC1");
  Nextion_Send_Text_to_Textbox("t3","ControlValue");
  Nextion_Send_Text_to_Textbox("t4","Voltage POT2");
  Nextion_Send_Text_to_Textbox("t5","Setpoint");
  Nextion_Send_Text_to_Textbox("t6","Time");
```

```
  initArrayNTC();

  // Init Timer with 10ms
  timer.attach_ms(timerInterval, timerCallback);
}
```

The `loop()` function executes repeatedly and manages the control logic for a system involving temperature regulation. It reads input voltages from two analog channels (`ADC1` and `ADC2`) and maps the second input to a setpoint value in degrees. The temperature is determined by interpolating the first voltage using a linear mapping function, and a PID controller calculates the control signal based on the difference between the setpoint and the temperature. Every second, the function logs the setpoint, temperature, and control signal to a serial monitor, updates a Nextion display with the values, and toggles an LED for visual feedback. Additionally, the control signal is applied to a heating element with a filtered output to ensure smooth operation.

```
void loop()
{
  // Place your main code here to execute repeatedly:
  float setpoint = 0;
  float temperature = 0;
  int control_signal = 0;
  float hysteresis = 0.0;

  // Read ADC1 value
  int voltage1 = analogRead(ADC1);
  // Read ADC2 value
  int voltage2 = analogRead(ADC2);

  int scaledValue = mapInteger(voltage2, 0, 4095, 25, 60);
  setpoint = (float)scaledValue;

  // Interpolate voltage 1
  float u1 = readvalue_linear(n_adc-1,ADC,Ua,(float)voltage1);
  // Interpolate voltage 2
  // float u2 = readvalue_linear_voltage(n_adc-1,ADC,Ua,(float)voltage2);

  temperature = readvalue_linear(n_ntc-1,Voltages,Temperatures,filter_NTC1(u1));

  // Check if one second has passed
  if (OneSecond)
  {
      // Reset the boolean variable OneSecond
      OneSecond = false;
```

```
      CounterSeconds++;

      // control_signal = (int)pid_regler_stell(setpoint,temperature);
      control_signal = (int)pid_regler_diff(setpoint,temperature);

      // control_signal = twoPointController(setpoint,temperature,hysteresis);
      control_signal = filterControlValue(control_signal);

      // Heat1 10-1000 ms on step 10ms
      Heat1_ON(control_signal);

      // Sends variable to USB UART and Seriell Monitor
      Serial.print("SP:");
      Serial.print(setpoint, 1);    // Setpoint with one decimal place
      Serial.print(" Temp:");
      Serial.print(temperature, 2); // Temperature with one decimal place
      Serial.print(" CV:");
      Serial.print(control_signal); // Control signal with one decimal place
      Serial.println();             // New line for the next output

      // Sends variable to display
      Nextion_Send_Text_to_Textbox("txtT1",String(voltage1));
      Nextion_Send_Text_to_Textbox("txtT2",String(temperature,3));
      Nextion_Send_Text_to_Textbox("txtT3",String(control_signal));
      // Sends variable to display
      Nextion_Send_Text_to_Textbox("txtT4",String(voltage2));
      Nextion_Send_Text_to_Textbox("txtT5",String(setpoint,1));
      Nextion_Send_Text_to_Textbox("txtT6",String(CounterSeconds));
      // Switching LED
      digitalWrite(LED_BUILTIN,!digitalRead(LED_BUILTIN));
  }
  // Additional code for the main loop
}
```

5.7 Arduino PID controller for standing pendulum

This program simulates and controls the behavior of a standing pendulum under the influence of an external force. It uses the equation of motion for a pendulum and a PID controller to stabilize the pendulum. The results of the codeblocks Ansi C program are output both on the console and in a CSV file.

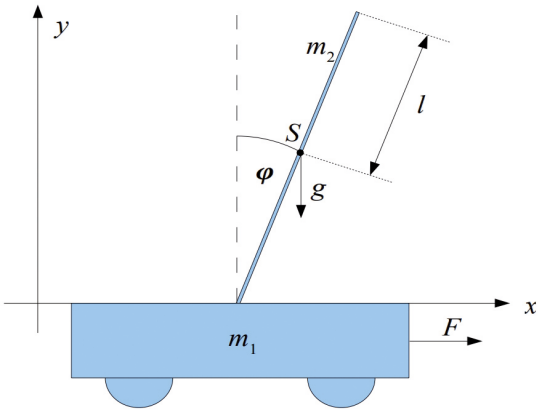


Figure 5.10: Inverted pendulum on a cart

Theoretical background:
The equation of motion of a pendulum is

$$\theta'' = -g / L \cdot \sin(\theta) + f / L$$

Where are:

θ: Angle of the pendulum
g: acceleration due to gravity
L: length of the pendulum
f: external force

The acceleration of the pendulum a results from the sum of the gravitational force and the external force:

$$a = -g / L \cdot \sin(\theta) + f / L$$

For numerical integration, the angular velocity ω and the angle θ are updated using an Euler step:

$$\omega(t+\Delta t) = \omega(t) + a \cdot \Delta t$$
$$\theta(t+\Delta t) = \theta(t) + \omega(t) \cdot \Delta t$$

The angle θ is limited to the range from 0 to 180 degrees to prevent physically unrealistic values.

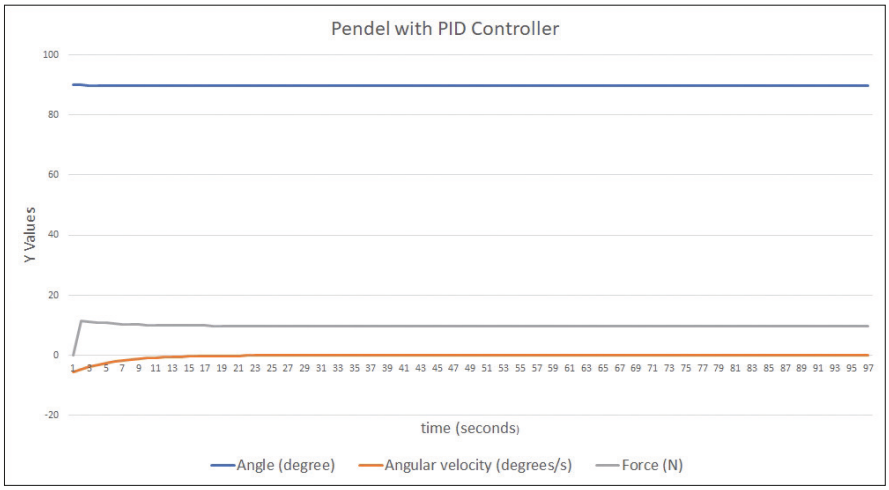


Figure 5.11 Pendulum PID control curve

Program structure, Header and macros, definition of constants and conversion factors:

```
#define G 9.81    // Acceleration due to gravity in m/s^2
#define L 1.0     // Length of the pendulum in metres
#define DT 0.01   // Time increment in seconds
#define PI 3.14159265358979323846
#define DEG_TO_RAD(deg) ((deg) * (PI / 180.0)) // Conversion factor from degrees
to radians
#define RAD_TO_DEG(rad) ((rad) * (180.0 / PI)) // Conversion factor from radians
to degrees
```

Pendulum structure for storing the pendulum status (angle and angular velocity):

```
// Pendulum structure for storing the state
typedef struct {
    double theta;     // Angle oft he pendulum in radians
    double omega;     // Angle velocity in radians per second
} Pendel;
```

Function for updating the pendulum status and calculating the acceleration and updating the angular speed and angle:

```
Pendel pendel;
int gesamtSchritte = 300; // total number of steps
double zielWinkel = DEG_TO_RAD(100.0); // Target angle in radians
double f; // Direction of force
int i;
```

The following function updates the state of the standing pendulum. It calculates the acceleration based on the external force and gravity and then updates the angular velocity and angle of the pendulum with a simple Euler step. The angle is limited to the range of 0 to 180 degrees to prevent physically unrealistic values. When the angle reaches the limits, the The function returns the current angular velocity.

```
double update_pendulum (Pendel *pendel, double f)
{
    // Calculate the acceleration on the pendulum, depending on the external
force
    double beschleunigung = (-G / L) * sin(pendel->theta) + (f / L); // Korrektur
der Formel
    // Update the angular velocity and angle with a simple Euler step
    pendel->omega += beschleunigung * DT;
    pendel->theta += pendel->omega * DT;
    // Limit the angle to the range from 0 to 180 degrees
    if (pendel->theta > DEG_TO_RAD(180.0)) {
        pendel->theta = DEG_TO_RAD(180.0);
        pendel->omega = 0; // Set the speed to 0 when the limit is reached
    } else if (pendel->theta < 0) {
        pendel->theta = 0;
        pendel->omega = 0; // Set the speed to 0 when the limit is reached
    }
    // Return the angular velocity
    return pendel->omega;
}
```

PID control function, calculation of the manipulated variable based on the control deviation:

```
// PID controller
float PID(float error)
{
    // PID-Parameter
    double Kp = 10000.0; // Proportionality factor
    double Ki = 100;     // Integration factor
    double Kd = 100;     // Derivative factor
    // Static variables for PID control
    static float integral = 0.0;
    static float error_old = 0.0;
    // Calculate the integral and derivative terms
    integral += error * DT;
    float derivative = (error - error_old) / DT;
    // PID controller calculation
    float controlValue = Kp * error + Ki * integral + Kd * derivative;
    // Update of the old error
    error_old = error;
```

```
    // Limit the control value to the range from -100 to +100
    if (controlValue > 100.0f) {
        controlValue = 100.0f;
    } else if (controlValue < -100.0f) {
        controlValue = -100.0f;
    }

    return controlValue;
}
main() function
```

Initialization of the pendulum, execution of the simulation and output of the results:

Summary
This program simulates a standing pendulum using an external force calculated by means of a PID controller. The simulation is carried out by numerical integration with an Euler step and the results are output both on the console and in a CSV file.

Listing 5.4: Arduino_PendelExample.ino

```
//--------------------------------------------------------------------
//
//            Arduino Upstanding Pendel Test
//            ==============================
//
//      Control Testprogramm for Arduino Boards
//      Timer Cycle
//
//
//  Date:     16/08/2024
//  Author:   Josef Bernhardt
//  Hardware: Arduino
//
//  File:     Arduino_PendelExample.ino
//--------------------------------------------------------------------
// To calculate the pendulum status, the differential equation for a pendulum is
used.
// The basic form of the equation of motion is:
//
//     θ'' = - (g / L) * sin(θ) + (f / L)
//
// Here, θ is the angle of the pendulum, g is the gravitational acceleration,
// L is the length of the pendulum, and f is the external force acting on the
pendulum.
// The terms in the equation have the following meanings:
```

```
//
// 1. - (g / L) * sin(θ): This term describes the acceleration of the pendulum
due to gravity.
//    It is proportional to the sine function of the angle θ and depends on the
length L and
//    the gravitational acceleration g. For small angles, sin(θ) can be
approximated by θ,
//    which simplifies the calculation.
//
// 2. (f / L): This term describes the acceleration due to the external force f.
//    It is directly proportional to the applied force and inversely proportional
to the length of the pendulum.
//
// The acceleration a of the pendulum results from the sum of the two terms and
is used to update
// the angular velocity (ω) and the angle (θ):
//
//     ω(t+Δt) = ω(t) + a * Δt
//     θ(t+Δt) = θ(t) + ω(t) * Δt
//
// Here, Δt is the time increment used for numerical integration. The Euler step
is a simple method
// for calculating the new values based on the current values and the calculated
acceleration.
//
// To obtain physically realistic values, the angle θ is limited to the range
from 0 to 180 degrees.
// If the angle reaches these limits, the angular velocity is set to 0 to prevent
the pendulum
// from swinging beyond this range.


#include <Arduino.h>
#include <math.h>

#define G 9.81    // Gravitational acceleration in m/s^2
#define L 1.0     // Length of the pendulum in meters
#define DT 0.01   // Time increment in seconds
#define PI 3.14159265358979323846
#define DEG_TO_RAD(deg) ((deg) * (PI / 180.0)) // Conversion factor from degrees
to radians
#define RAD_TO_DEG(rad) ((rad) * (180.0 / PI)) // Conversion factor from radians
to degrees

// Pendulum structure for storing state
typedef struct {
```

```
    double theta;     // Pendulum angle in radians
    double omega;     // Angular velocity in radians per second
} Pendulum;

// The following function updates the state of a stationary pendulum.
// It calculates the acceleration based on the external force and gravity,
// and then updates the angular velocity and angle of the pendulum using a simple
Euler step.
// The angle is limited to the range from 0 to 180 degrees to prevent physically
unrealistic values.
// If the angle reaches these limits, the angular velocity is set to 0 to prevent
the pendulum
// from swinging beyond the permissible range. The function returns the current
angular velocity.

// Function to update the pendulum state
double update_pendulum(Pendulum *pendulum, double f) {
    // Calculate the acceleration on the pendulum, depending on the external
force
    double acceleration = (-G / L) * sin(pendulum->theta) + (f / L); //
Correction of the formula

    // Update angular velocity and angle using simple Euler step
    pendulum->omega += acceleration * DT;
    pendulum->theta += pendulum->omega * DT;

    // Limit the angle to the range from 0 to 180 degrees
    if (pendulum->theta > DEG_TO_RAD(180.0)) {
        pendulum->theta = DEG_TO_RAD(180.0);
        pendulum->omega = 0;  // Set speed to 0 when the limit is reached
    } else if (pendulum->theta < 0) {
        pendulum->theta = 0;
        pendulum->omega = 0;  // Set speed to 0 when the limit is reached
    }

    // Return the angular velocity
    return pendulum->omega;
}

// PID control function
float PID(float error)
{
    // PID parameters
    double Kp = 10000.0; // Proportional factor
    double Ki = 100;   // Integral factor
    double Kd = 100;   // Derivative factor
```

```
    // Static variables for PID control
    static float integral = 0.0;
    static float error_old = 0.0;
    // Calculate integral and derivative terms
    integral += error * DT;
    float derivative = (error - error_old) / DT;
    // PID controller calculation
    float controlValue = Kp * error + Ki * integral + Kd * derivative;
    // Update the old error
    error_old = error;

    // Limit the control value to the range from -100 to +100
    if (controlValue > 100.0f) {
        controlValue = 100.0f;
    } else if (controlValue < -100.0f) {
        controlValue = -100.0f;
    }

    return controlValue;
}

Pendulum pendulum;
int totalSteps = 300; // Total number of steps
double targetAngle = DEG_TO_RAD(100.0); // Target angle in radians
double f; // Force direction
int i;

void setup()
{
    Serial.begin(115200);

    // Set the initial angle to 90 degrees (vertical)
    pendulum.theta = DEG_TO_RAD(90.0); // 90 degrees in radians
    pendulum.omega = 0.0;              // Initial angular velocity: 0.0

    // Output the header
    Serial.println("Step\tTime (s)\tAngle (Deg)\tAngular Velocity (Deg/s)\tForce
(N)");
}

void loop()
{
    for (i = 0; i < totalSteps; ++i)
       {
        // Calculate the current deviation
        // Use the PID function to calculate the force
```

```
        f = PID(targetAngle - pendulum.theta);

        // Update the pendulum
        double omega_deg = update_pendulum(&pendulum, f) * RAD_TO_DEG(1.0);
   // Angular velocity in degrees/s

        // Output the results to the console
        Serial.print("Degree:");
        Serial.print(RAD_TO_DEG(pendel.theta), 2);
        Serial.print(" Omega:");
        Serial.print(omega_deg, 2);
        Serial.print(" Force:");
        Serial.println(f, 2);

        // Wait for the time increment
        delay(DT * 1000);
    }

    // Stop the loop
    while (true) {}
}
```

Chapter 6 • Fuzzy Logic basics

6.1 Fuzzy Logic history

In the 1960s, Professor Lotfi Zadeh from the renowned University of California in Berkeley discovered a crucial gap in traditional Boolean algebra. This algebra was based solely on the binary states 'true' or 'false', but this was inadequate for capturing subtle grey values, which play a role in many applications.

Zadeh, with an eye for innovation, recognized the need to fill this gap and develop a more flexible method to capture intermediate values between states. From this realization, he formulated a groundbreaking concept: fuzzy values, better known as 'fuzzy sets' or membership functions.

Zadeh's groundbreaking approach overcame the traditional view of 'wrong' or 'right' and replaced it with a more flexible and nuanced perspective. Instead of a clear, digital demarcation, his method allowed for a more gradual evaluation of statements. For example, a statement could no longer be considered simply 'wrong' or 'right', but could be evaluated in a more nuanced way, such as '30% right' and '70% wrong'. This enabled a much more precise and realistic representation of complex issues.

An outstanding advantage of fuzzy logic is its ability to integrate expert knowledge and model complex decision-making processes without the need for in-depth mathematical analysis. This significantly reduces development costs and makes it accessible for various applications, from industrial plants to household appliances.

Another feature that makes fuzzy logic attractive is its adaptability to non-linear paths and the ability to work with fuzzy, colloquial terms that occur in many real-life situations. This allows for more precise control and better performance, especially in systems with unpredictable variables.

The core principles of fuzzy logic include fuzzification of input data into fuzzy sets, inference of rules based on these fuzzy sets, and finally defuzzification to transform fuzzy output data into clear, manageable results.

Fuzzification involves the assignment of numerical values to fuzzy terms based on their membership of fuzzy sets. This enables a more precise capture of fuzzy input data, which is particularly useful when linguistic variables are used.

The inference phase evaluates the fuzzy rules defined by experts and generates fuzzy output data based on the combination of these rules. Links such as AND (MIN), OR (MAX) and NOT are used here to logically link fuzzy sets.

Finally, defuzzification takes place, in which fuzzy output data is converted into clear, manageable values for manipulated variables. This can be done using various methods such as center of gravity (COG), center of area (COA), center of maximum (COM) and mean of maximum (MOM), depending on the requirements of the application.

In practice, fuzzy logic is widely used in various industries, including industry, household appliances and traffic engineering such as anti-lock braking systems (ABS) in vehicles. The ability to run on low-cost microcontrollers has further facilitated its proliferation and implementation, leading to a variety of innovations and efficiency gains.

6.2 Fuzzy controller structure

In principle, a fuzzy controller has a similar structure to a PID controller. However, the control value is calculated in the order of fuzzification, inference and defuzzification.

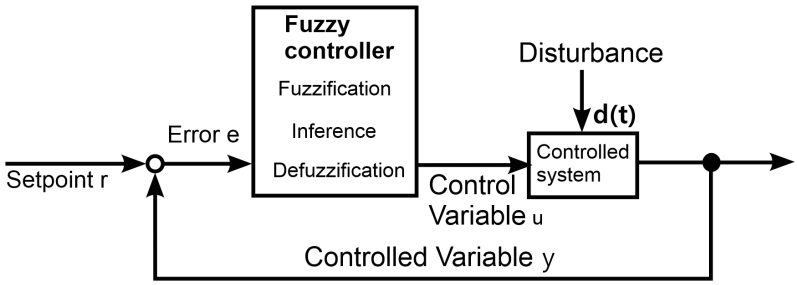


Figure 6.1: Overview of fuzzy controllers

6.3 Fuzzification

Fuzzification is the first step in the process of fuzzy logic, in which clear numerical input data is converted into fuzzy sets This conversion makes it possible to translate linguistic variables such as 'fast' or 'slow' into mathematically recognizable forms Fuzzification takes into account the uncertainty and fuzziness in the input data, which is particularly important when human language or expert knowledge is integrated into the system The results of fuzzification are fuzzy sets that represent the membership of the input data to different linguistic terms An example of fuzzification is the assignment of a temperature of 25°C to the fuzzy sets 'cold', 'warm' and 'hot', depending on the defined membership functions.

6.4 Inference

Inference is the second step in fuzzy logic, where fuzzy rules are applied to the fuzzy sets to generate fuzzy output data. These rules, which are defined by experts or empirical studies, specify how the fuzzy input data is combined to produce fuzzy output data. During inference, various combinations such as AND (MIN), OR (MAX) and NOT are used to logically link the fuzzy sets and generate the fuzzy output data. The inference phase aims to apply the fuzzy rules to make fuzzy decisions that can then be interpreted in a more precise context. An example of inference is the application of fuzzy rules to control an automatic transmission in a vehicle, based on fuzzy input data such as speed and accelerator pedal position.

6.5 Defuzzification

Defuzzification is the final step in fuzzy logic, in which fuzzy output data is converted into clear, manageable values. This conversion is crucial to translate the fuzzy results of the inference phase into precise actions or control commands. There are several methods for defuzzification, including Centre of Gravity (COG), Centre of Area (COA), Centre of Maximum (COM) and Mean of Maximum (MOM). Each of these methods has its own

advantages and disadvantages and is suitable for different applications depending on the specific requirements of the system. An example of defuzzification is the conversion of a fuzzy rule such as 'The heating should be increased moderately' into a clear action such as adjusting the thermostat controller to a specific temperature.

6.6 Fuzzy controller example for a fan

In principle, the structure of a fuzzy controller is similar to that of a PID controller. The control deviation is calculated from the difference between the setpoint value and the current value. The control value is determined from this deviation.

A simple example of controlling a fan used to cool an inverter is presented to show how the control value is calculated.

Firstly, the membership functions, so-called fuzzy sets, are defined in the form of triangles. The control value should be calculated so that the fan starts to run slowly at 20 °C, faster from 30 to 40 °C and at full power from 60°C onwards.

The values for the position and width of the triangles are defined globally in the program. There are two values each for left_open() and right_open() and three values for the triangle. An odd number of triangles is usually used. Fuzzy sets with 5 or 7 triangles are often used for a more precise gradation.

```
// Global arrays for the definitions of the triangular functions
float left_open[] = {20.0, 40.0};
float triangle[] = {20.0, 40.0, 60.0};
float right_open[] = {40.0, 60.0};
```

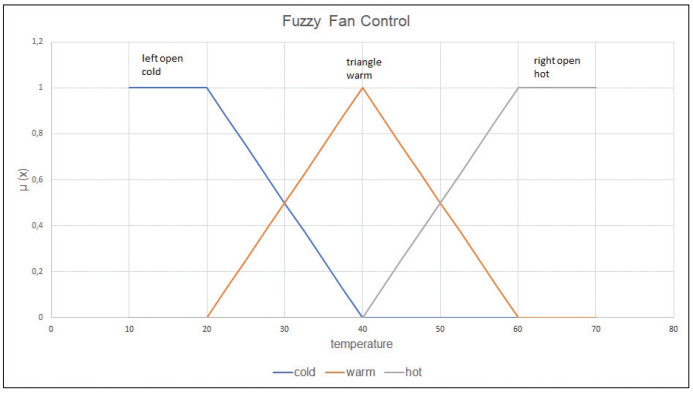


Figure 6.2: Fuzzy Memberships

The program reads in the temperature and uses this value and the fuzzy sets to calculate the membership of the fuzzy triangles (fuzzy sets) for cold, warm and hot.

```
// Calling up the fuzzy controller function to calculate the fan speed
float fan_speed = fuzzy_controller(temperature);
```

The fuzzy_controller() function calculates the associations between the measured temperature and the fuzzy sets.

```
float rule1 = fuzzyLeftOpen(temperature, left_open[0], left_open[1]);
float rule2 = fuzzyTriangle(temperature, triangle[0], triangle[1],triangle[2]);
float rule3 = fuzzyRightOpen(temperature, right_open[0], right_open[1]);
```

The calculation is performed using the left_open(), triangle() and right_open() functions.

Fuzzy Leftopen	Fuzzy Triangle	Fuzzy Rightopen
float fuzzyLeftOpen(float x, float a, float b) { if (x <= a) { return 1.0; } else if (x <= b) { return (b - x) / (b - a); } else { return 0.0; } }	float fuzzyTriangle(float x, float a, float b, float c) { if (x <= a || x >= c) { return 0.0; } else if (x <= b) { return (x - a) / (b - a); } else { return (c - x) / (c - b); } }	float fuzzyRightOpen(float x, float a, float b) { if (x <= a) { return 0.0; } else if (x <= b) { return (x - a) / (b - a); } else { return 1.0; } }

The results for a calculation with temperature values from 17.5 to 62.5°C

Temperature	cold	warm	hot
17,5	1	0	0
20	1	0	0
22,5	0,875	0,125	0
25	0,75	0,25	0
27,5	0,625	0,375	0
30	0,5	0,5	0
32,5	0,375	0,625	0
35	0,25	0,75	0
37,5	0,125	0,875	0
40	0	1	0
42,5	0	0,875	0,125
45	0	0,75	0,25
47,5	0	0,625	0,375
50	0	0,5	0,5
52,5	0	0,375	0,625
55	0	0,25	0,75
57,5	0	0,125	0,875
60	0	0	1
62,5	0	0	1

From these calculated values you can see that at 27.5°C the membership is 0.625 for cold, 0.375 for warm and 0 for hot.

Rule evaluation (knowledge base)

The knowledge base contains information about the value ranges of the measured and control variables; here there is an assignment of how fast the fan should rotate at a certain temperature.

The control values for the associated fuzzy sets, also known as singletons, are defined globally here. Each membership function is assigned a weight, which is the point at which the membership value is at its maximum.

```
float Y[] = {5.0, 50.0, 100.0};
```

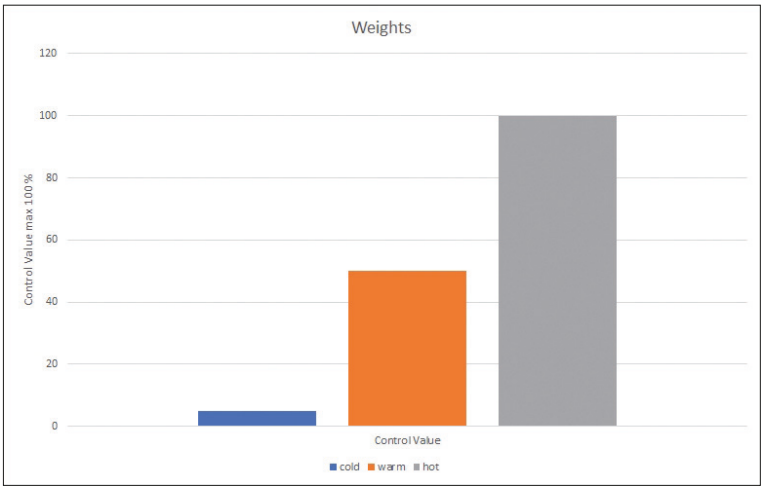


Figure 6.3: Singletons for the control value

If the temperature is cold, the control value should be 5 %.

If the temperature is warm, the control value should be 50 %.

If the temperature is hot, the control value should be 100 %.

Defuzzification
The defuzzification method used in this code is the weighted average of the memberships. However, it is only valid for symmetrical membership functions. This method requires relatively little computing power and is ideal for implementation on microcontrollers. The sum of the weighted values (numerator) divided by the sum of the weights (denominator) is calculated.

Calculation of the weighted average of the rules:

```
float numerator =   (rule1 * Y[0]) + (rule2 * Y[1]) + (rule3 * Y[2]);
float denominator = rule1 + rule2 + rule3;
float controlvalue = numerator / denominator;
```

Here are some results for the calculated manipulated variable:

Temperature - Control Value			
35	37,5	32,5	31,25
40	50,0	37,5	43,75
45	62,5	42,5	56,25
50	75,0	47,5	68,75
55	87,5	52,5	81,25
60	100,0	57,5	93,75

The characteristic curve of this fuzzy controller can be derived from this:

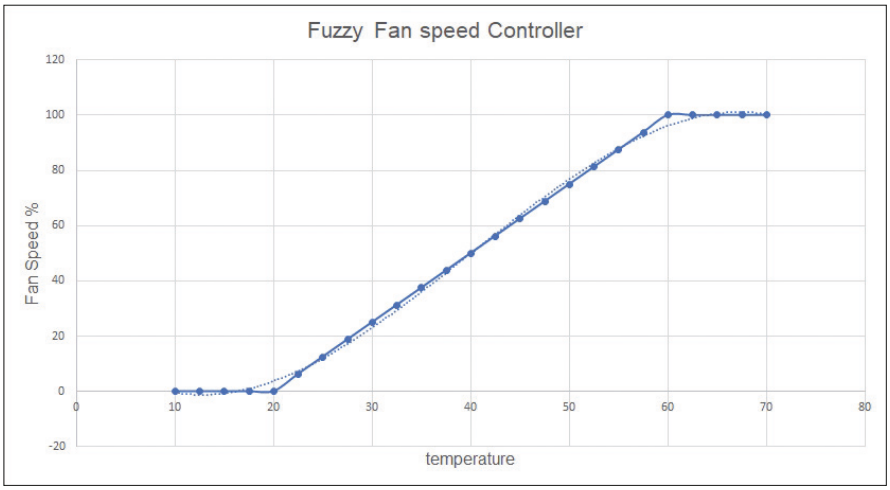


Figure 6.4: Controller characteristic curve

Listing 6.1: Fuzzy Fan Controller:

```
//--------------------------------------------------------------------
//
//            Ansi C Fuzzy Triangle FAN Control
//            =================================
//
//      Control Software Cooling Microinverter
//      Fuzzy Triangle Controller Example
//      Input Temperature
//      Output Fan Value 0 to 100 %
//
//  Date:     07/06/2024
//  Author:   Josef Bernhardt
//  Hardware: PC with Ansi C
//
//  File:     Codeblocks_C_FuzzyFanControl
//--------------------------------------------------------------------

#include <stdio.h>

// Global arrays for the definitions of the triangle functions
float left_open[] = {20.0, 40.0};
float triangle[] = {20.0, 40.0, 60.0};
float right_open[] = {40.0, 60.0};

// Control Values
```

```
float Y[] = {0.0, 50.0, 100.0};

// Fuzzification function - Left open
float fuzzyLeftOpen(float x, float a, float b) {
    if (x <= a)
       {
        return 1.0;
    } else if (x <= b)
       {
        return (b - x) / (b - a);
    } else
       {
        return 0.0;
    }
}

// Fuzzification function - Triangle
float fuzzyTriangle(float x, float a, float b, float c) {
    if (x <= a || x >= c)
       {
    } else if (x <= b)
       {
        return (x - a) / (b - a);
    } else
       {
        return (c - x) / (c - b);
    }
}

// Fuzzification function - Right open
float fuzzyRightOpen(float x, float a, float b) {
    if (x <= a) {
        return 0.0;
    } else if (x <= b)
       {
        return (x - a) / (b - a);
    } else
       {
        return 1.0;
    }
}

// Fuzzy controller for adjusting the fan speed
float fuzzy_controller(float temperature) {
    // Define the membership degrees for the temperature
    float rule1 = fuzzyLeftOpen(temperature,left_open[0],left_open[1]);
```

```
    float rule2 = fuzzyTriangle(temperature,triangle[0],triangle[1],triangle[2]);
    float rule3 = fuzzyRightOpen(temperature,right_open[0],right_open[1]);
    printf("temperature %.1f rule1 %.3f rule2 %.3f rule3 %.3f
",temperature,rule1,rule2,rule3);
    // Calculate the weighted average of the rules
    float numerator =   (rule1 * Y[0]) + (rule2 * Y[1]) + (rule3 * Y[2]);
    float denominator = rule1 + rule2 + rule3;
    // Check if the denominator is not zero
    if (denominator != 0.0) {
        // Calculate if the denominator is not zero
        return numerator / denominator;
    } else {
        // If the denominator is zero, terminate the program
        return 0.0;
    }
}

int main()
{
    printf("temperature;fan_speed\n");
     // Simulated temperature (in Kelvin)
    for (float temperature = 10.0; temperature <= 70.0; temperature += 2.5)
       {
        // Call the fuzzy controller to calculate the fan speed
        float fan_speed = fuzzy_controller(temperature);
        printf("%.1f;%.2f\n", temperature, fan_speed);
    }
    return 0;
}
```

6.7 Control of an inverted pendulum

The inverted pendulum is an interesting system that is frequently investigated in engineering, especially in control engineering and robotics. It involves keeping a pendulum, which would normally hang downwards, upright through targeted movements - similar to balancing a rod on your hand.

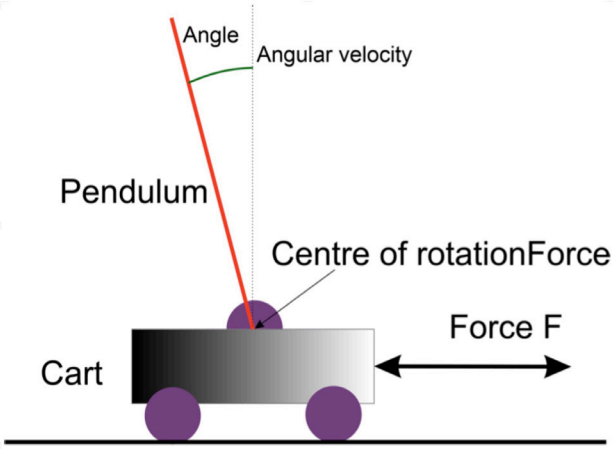


Figure 6.5: Inverted pendulum

This pendulum is attached to a carriage, which must move accordingly to maintain balance. Despite its seemingly simple construction, controlling the inverted pendulum requires advanced control techniques, making it a popular object for experimentation. Particularly exciting is the use of fuzzy logic to enable the stabilization of this unstable, non-linear system.

Simulation of the pendulum:
The pendulum is influenced by the setting value and the elapsed time. A damping factor of 0.9 reduces the angular velocity and thus the friction. The actuating value acts on the pendulum in the opposite direction, which leads to an acceleration of the angular velocity. This speed is then reduced by the damping factor. The angle of the pendulum is continuously updated based on the remaining angular velocity. To avoid extreme positions, the angle is limited to the range from 0 to 180 degrees. If the pendulum reaches one of these limits, both the angle and the angular velocity are set to zero to ensure a stable end position.

Listing 6.2 Arduino_Pendel_FuzzyTriangleController.ino

```
Pendulum update_pendulum(float control_value, float delta_time, Pendulum pendulum)
{
    // Damping factor (between 0 and 1) to reduce the angular velocity
    // Simulates friction/resistance
    float damping = 0.90f;
    // Carriage movement influences the angular velocity
    // Acceleration through carriage movement
    control_value *= -1;
    pendulum.angular_velocity += control_value * delta_time;
    // Attenuates the angular velocity
    pendulum.angular_velocity *= damping;
    // Update the angle based on the angular velocity
    // Angle change due to angular velocity
```

```
    pendulum.angle += pendulum.angular_velocity * delta_time;
    // Limitation of the angle to 0 to 180 degrees
    // Stop at the anchor point
    if (pendulum.angle < 0) {
        pendulum.angle = 0;
        pendulum.angular_velocity = 0;
    } else if (pendulum.angle > 180) {
        pendulum.angle = 180;
        pendulum.angular_velocity = 0;
    }
    return pendulum;
}
```

In this example, a fuzzy controller with five triangles is used to calculate the control value for the carriage.

```
// These arrays are used to represent the degrees of membership
// of an input value in different regions of a fuzzy set.
float Nmax[]  = {-10,-5};       // Values for the left region
float Nsmall[]  = {-10,-5, 0};  // Values for the middle region
float Zero[]  = {-5, 0, 5};     // Values for the middle region
float Psmall[]  = { 0, 5, 10};  // Values for the middle region
float Pmax[]  = { 5, 10};       // Values for the right region

// Output Values
float U[]       = { -30 , -20, 0 , 20, 30 };
```

The fuzzy controller function uses the five declared triangles for a finer gradation.

```
float fuzzy_control(float e)
{
    // Calculate the membership values for the error signal in different
       // regions of the fuzzy sets.
    // Membership value for the left region
       float left_degree_NM   = fuzzyLeftOpen  (e, Nmax[0],  Nmax[1]);
    // Membership value for the middle region N
       float middle_degree_NS = fuzzyTriangle  (e, Nsmall[0],Nsmall[1],Nsmall[2]);
S
    // Membership value for the middle region ZE
       float middle_degree_ZE = fuzzyTriangle  (e, Zero[0],Zero[1],Zero[2]);
    // Membership value for the middle region PS
       float middle_degree_PS = fuzzyTriangle  (e, Psmall[0],Psmall[1],Psmall[2]);
    // Membership value for the right region
       float right_degree_PM  = fuzzyRightOpen (e, Pmax[0], Pmax[1]);

    // Calculate the weighted sum of membership values
```

```
    float numerator   = left_degree_NM * U[0] + middle_degree_NS * U[1] +
                                                middle_degree_ZE * U[2]  + middle_
degree_PS * U[3]
                                                + right_degree_PM * U[4];
    // Sum of membership values
           float denominator = left_degree_NM + middle_degree_NS +
                                                        middle_degree_ZE + middle_
degree_PS +
                                                        right_degree_PM;

    float y; // Control signal

    // Avoid division by zero by checking if the denominator is zero.
    if (denominator != 0)
    {
        // Calculate the control signal as the ratio of the
       // weighted sum to the sum of membership values.
        y = numerator / denominator;
    } else
    {
        // If the denominator is zero, set the control signal to
               // zero to avoid undefined behavior.
        y = 0;
    }
    // Limit the control signal to the range [-100, 100]
    if (y > 100) {
        y = 100;
    } else if (y < -100) {
        y = -100;
    }
    return y; // Return the control signal.
}
```

The result of the simulation.

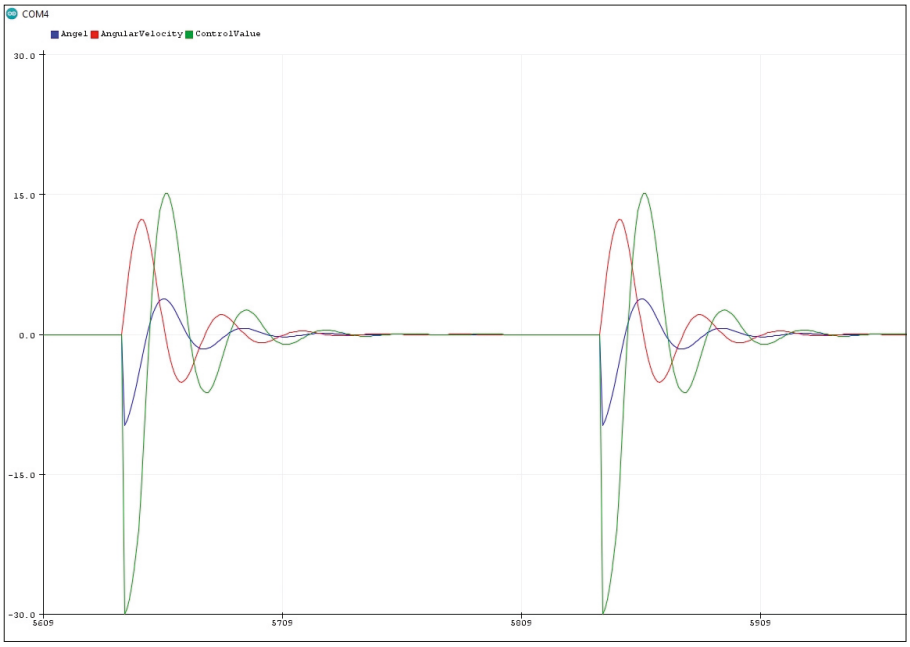


Figure 6.6: Arduino simulation for a vertical pendulum

The main loop begins with the initialization of the pendulum, which starts in an upright position of 80 degrees. The movement of the pendulum is simulated in a for loop with 200 iterations. Within the loop, the LED on the microcontroller is alternately switched on and off. In each step, the fuzzy controller calculates the control value based on the deviation of the pendulum angle from the 90-degree position. Finally, the state of the pendulum is updated, the current angle and speed values as well as the control value are output via the serial interface and the loop pauses briefly for 100 milliseconds.

```
void loop()
{
    Pendulum pendulum = {80.0f, 0.0f};  // Pendulum starts upright at 80 degrees
    float control_value  = 0.0f;        // Input value: Carriage movement (-100
to +100 %)
    float delta_time = 0.1f;            // Time difference in seconds (simulation
step)
    int steps = 200;                    // Number of simulation steps
    int direction = 1;                  // Direction of carriage movement (1 or
-1)

    // Simulate a pendulum movement for a certain number of steps
    for (int i = 0; i < steps; i++)
    {
```

```
        digitalWrite(LED_BUILTIN, !digitalRead(LED_BUILTIN));

        // Fuzzy Controller
        control_value = fuzzy_control(pendulum.angle - 90);

        // Update the pendulum state
        pendulum = update_pendulum(control_value, delta_time, pendulum);

        Serial.print("Angle:");
        Serial.print(pendulum.angle - 90, 2);
        Serial.print(" AngularVelocity:");
        Serial.print(pendulum.angular_velocity, 2);
        Serial.print(" ControlValue:");
        Serial.println(control_value, 2);

        delay(100);
    }
}
```

6.8 Fuzzy sets

In addition to the fuzzy sets used in this example, there are several others. The triangular fuzzy sets and their special form, the trapezoidal fuzzy sets, are used most frequently. Other common fuzzy sets are the Gaussian and Bell fuzzy sets as well as the sigmoid fuzzy sets.

Below you will find an overview of the various fuzzy membership functions and controller transfer functions as well as C-functions for calculating the different memberships. C examples for triangular, trapezoidal, Gaussian, Bell and sigmoid fuzzy sets are also included.

The C-functions in the tables provide an overview of the frequently used fuzzy membership functions 'Left Open' and 'Right Open' represent fuzzy sets that are open on one side. Each function describes how the membership of a value x is calculated relative to certain limits a, b, and c.

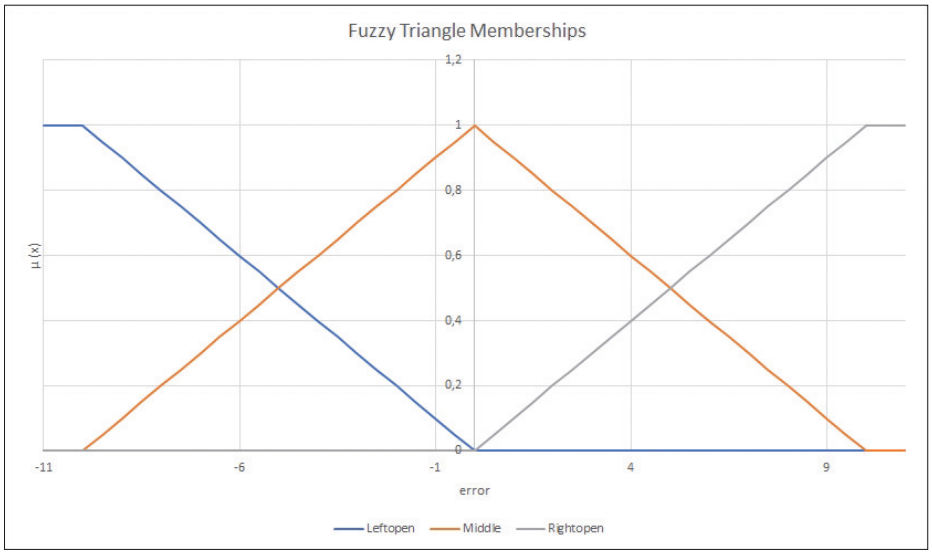


Figure 6.7: Fuzzy membership triangles

Membership Left Left open	Membership Middle Triangle	Membership Right Rightopen
float fuzzyLeftOpen(float x, float a, float b) { if (x <= a) { return 1.0; } else if (x <= b) { return (b - x) / (b - a); } else { return 0.0; } }	float fuzzyTriangle(float x, float a, float b, float c) { if (x <= a || x >= c) { return 0.0; } else if (x <= b) { return (x - a) / (b - a); } else { return (c - x) / (c - b); } }	float fuzzyRightOpen(float x, float a, float b) { if (x <= a) { return 0.0; } else if (x <= b) { return (x - a) / (b - a); } else { return 1.0; } }

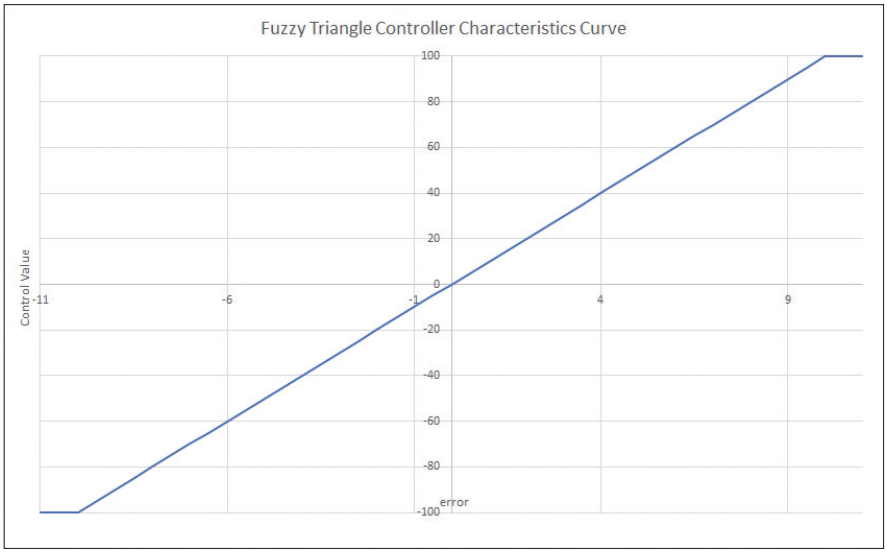


Figure 6.8: Fuzzy controller characteristic curve

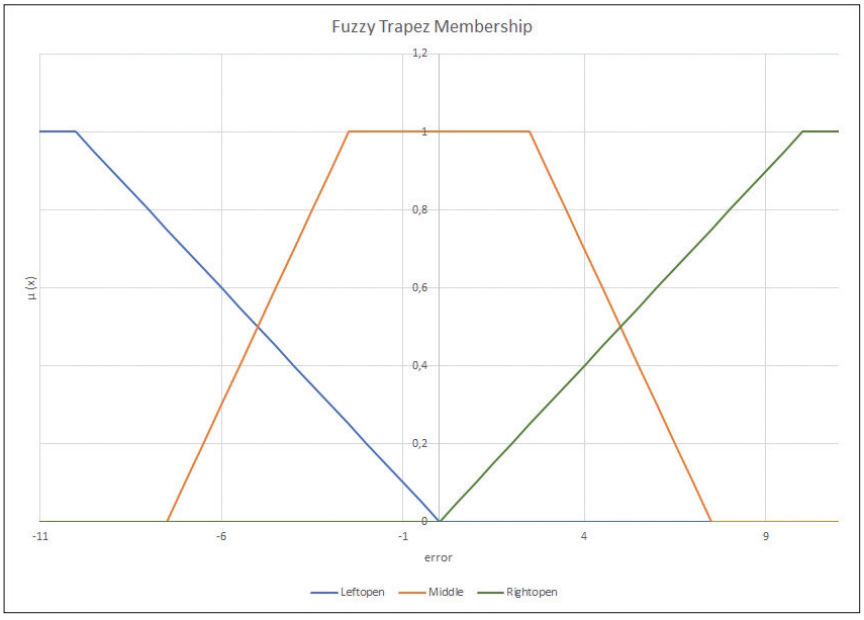


Figure 6.9: Fuzzy membership with trapezoid

Membership Left Left open

```
float fuzzyLeftOpen(float x, float a, float b)

{
    if (x <= a)
{
        return 1.0;
}
else if (x <= b)
{
return (b - x) / (b - a);
}
else
{
        return 0.0;
}
}
```

Membership Middle Trapezoid

```
// Function to calculate the trapezoidal membership function
float trapezoidalMembership(float x, float a, float m1, float m2, float b)
{
    if (x <= m1 - a || x > m2 + b)
    {
        return 0.0;
    }
    else if (x > m1 - a && x <= m1)
    {
        return 1 + (1.0 / a) * (x - m1);
    }
    else if (x > m1 && x <= m2)
    {
        return 1.0;
    }
    else if (x > m2 && x <= m2 + b)
    {
        return 1 - (1.0 / b) *
(x - m2);
    }
    else
    {
        return 0.0;
    }
}
```

Membership Right Rightopen

```
float fuzzyRightOpen(float x, float a, float b)

{
    if (x <= a)
{
        return 0.0;
}
else if (x <= b)
{
    return (x - a) / (b - a);
}
else
{
        return 1.0;
}
}
```

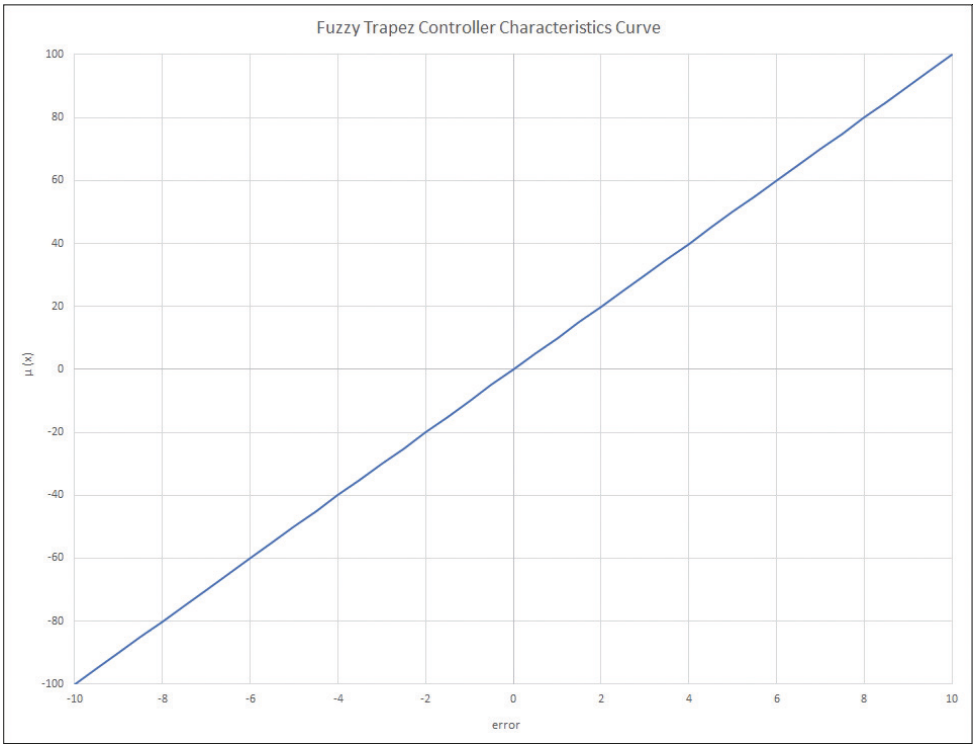


Figure 6.10: Controller characteristic curve

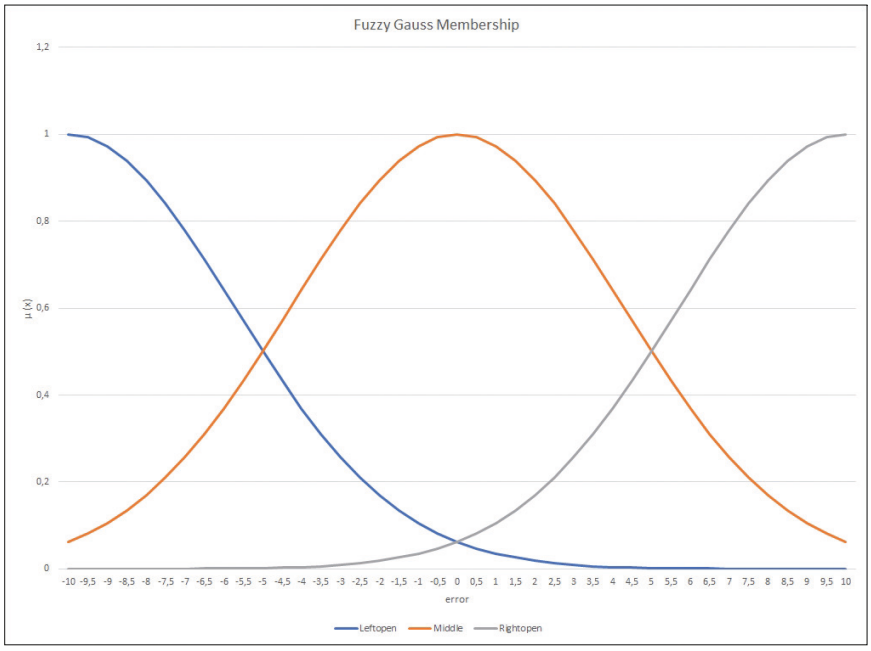


Figure 6.11: Fuzzy Gauss Memberships

Membership Left Left open Gauss	Membership Middle Gauss	Membership Right Right open Gauss
// This function calculates the membership value of a given input // 'x' in a left-open Gaussian membership function. // The left-open Gaussian membership function returns 1.0 for // input values less than the mean, // and the standard Gaussian membership function value otherwise. float gauss_membership_leftopen(float x, float mean, float sigma) { if (x < mean) { return 1.0; // Outside the interval, return 0 } return exp(-0.5 * pow((x - mean) / sigma, 2)); }	// This function calculates the membership value of a given input 'x' // in a standard Gaussian membership function. // The standard Gaussian membership function is continuous and // returns a value based on the input, mean, and standard deviation. float gauss_membership(float x, float mean, float sigma) { return exp(-0.5 * pow((x - mean) / sigma, 2)); }	// This function calculates the membership value of a given input 'x' // in a right-open Gaussian membership function. // The right-open Gaussian membership function returns 1.0 for // input values greater than the mean, // and the standard Gaussian membership function value otherwise. float gauss_membership_rightopen(float x, float mean, float sigma) { if (x > mean) { return 1.0; // Outside the interval, return 0 } return exp(-0.5 * pow((x - mean) / sigma, 2)); }

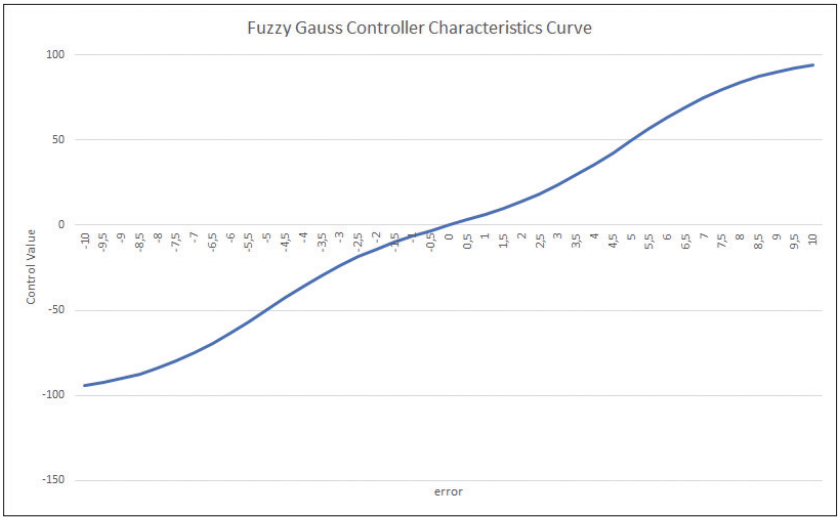


Figure 6.12: Fuzzy Gaussian controller characteristic curve

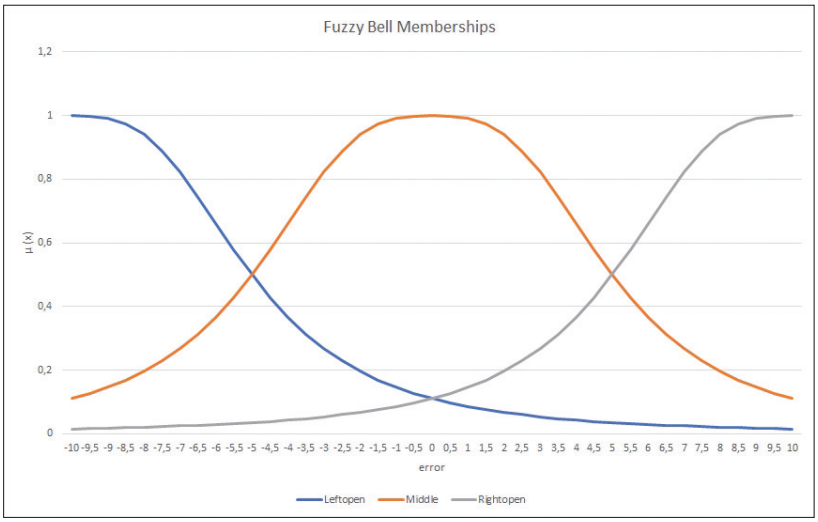


Figure 6.13: Fuzzy Bell Memberships

Membership Left **Left open Bell**	**Membership Middle** **Bell**	**Membership Right** **Right open Bell**
`// Definition of the generalised bell function (bell leftopen)` `float generalized_bell_left(float x, float a, float b, float c)` `{` `  float result;` `  if (b == 0) return 0.0;` `  if (x <= c) return 1.0;` `  result = 1.0 / (1.0 + pow(fabs((x - c) / a), 2 * b));` `  return result;` `}`	`// Definition of the generalised bell function` `float generalized_bell_middle(float x, float a, float b, float c)` `{` `  float result;` `  if (b == 0) return 0.0;` `  result = 1.0 / (1.0 + pow(fabs((x - c) / a), 2 * b));` `  return result;` `}`	`// Definition of the generalised bell function (bell rightopen)` `float generalized_bell_right(float x, float a, float b, float c)` `{` `  float result;` `  if (b == 0) return 0.0;` `  if (x >= c) return 1.0;` `  result = 1.0 / (1.0 + pow(fabs((x - c) / a), 2 * b));` `  return result;` `}`

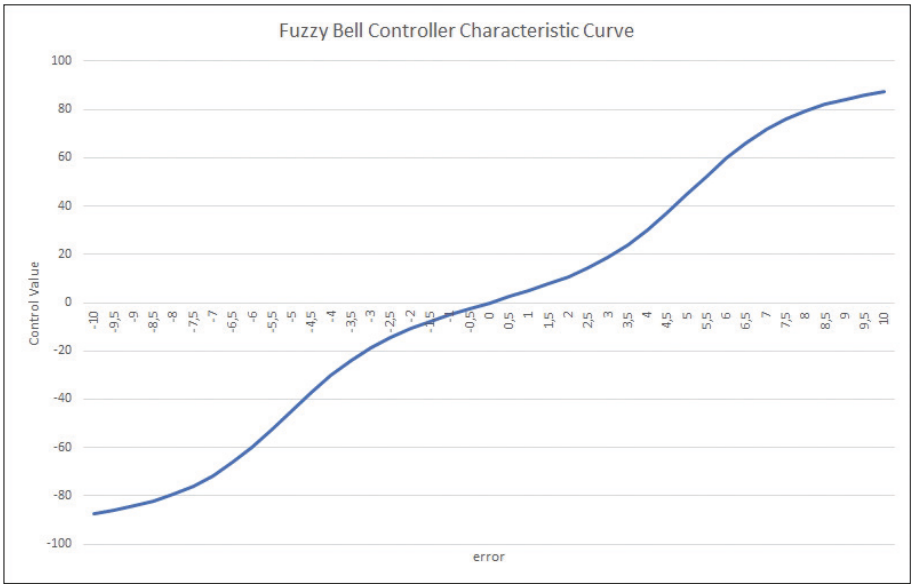


Figure 6.14: Fuzzy Bell controller characteristic curve

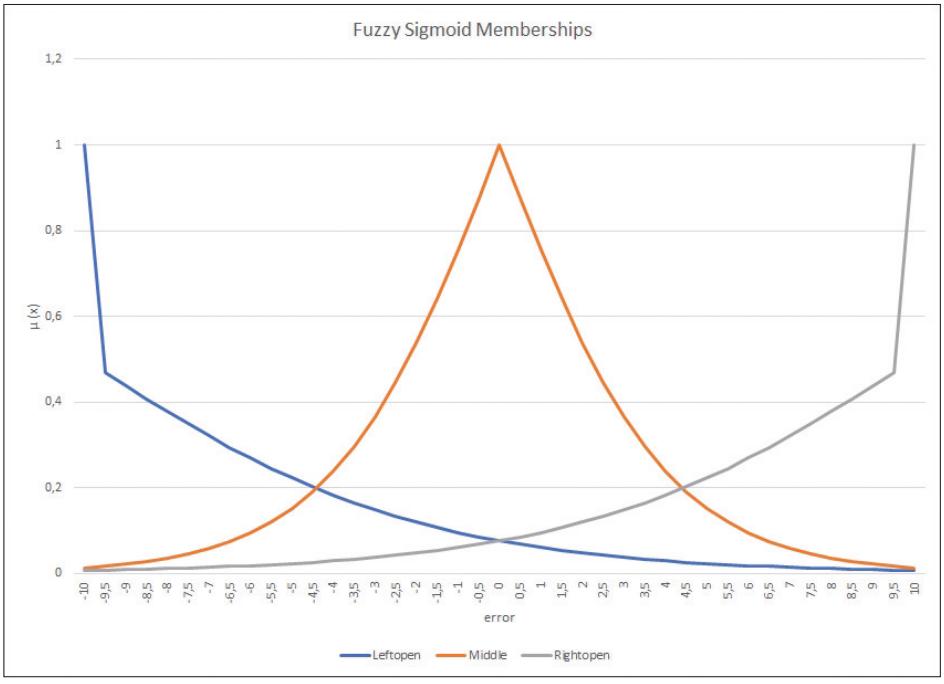


Figure 6.15: Fuzzy Sigmoid Memberships

Membership Left
Left open Sigmoid

```
// This function calculates the membership value of a given input 'x'
// in a left-open Sigmoid membership function.
// The left-open sigmoid membership function returns 1.0 for
// input values less than the mean, and a sigmoid value otherwise.
float fuzzyLeftOpenSigmoid(float x, float mean, float steepness)
{
    if (x <= mean)
    {
        return 1.0;
    } else
    {
        return 1.0 / (1.0 + exp(steepness * (x - mean)));
    }
}
```

Membership Middle
Sigmoid

```
// This function calculates the membership value of a given input 'x'
// in a Sigmoid membership function.
// The sigmoid membership function returns a value between 0.0 and 1.0 based on the input
// and the parameters for the mean and steepness.
float fuzzySigmoid(float x, float mean, float steepness)
{
    float result;
    if (x < mean)
    {
        result = 1.0 / (1.0 + exp(-steepness * (x - mean)));
    } else
    {
        result = 1.0 / (1.0 + exp(steepness * (x - mean)));
    }
    return result * 2;
}
```

Membership Right
Right open Sigmoid

```
// This function calculates the membership value of a given input 'x'
// in a right-open Sigmoid membership function.
// The right-open sigmoid membership function returns 1.0 for
// input values greater than the mean, and a sigmoid value otherwise.
float fuzzyRightOpenSigmoid(float x, float mean, float steepness)
{
    if (x >= mean)
    {
        return 1.0;
    } else {
        return 1.0 / (1.0 + exp(-steepness * (x - mean)));
    }
}
```

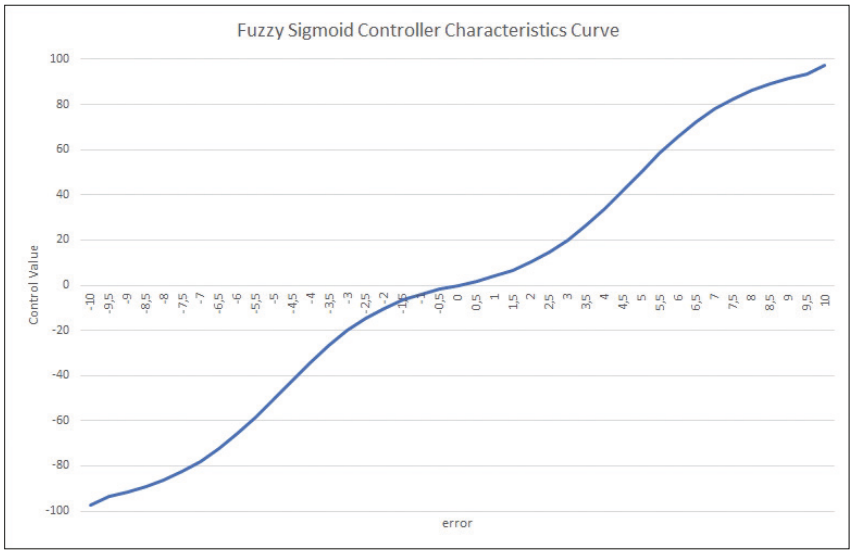


Figure 6.16: Fuzzy sigmoid controller characteristic curve

Examples to test the membership functions are available for the Arduino as well as for Codeblocks C in the download directory.

In the Codeblocks projects there are also functions to save the membership tables in CSV format for Excel.

Complete controller simulations for the PT2 circuit are available in the Arduino programs.

6.9 Simulation of a function with fuzzy logic

The program simulates a fuzzy logic controller that is realized by assigning various input variables to triangular fuzzy sets. The fuzzyTriangle() function calculates the degree of membership of an input value to a triangular set defined by three corner points. These degrees of membership are calculated for seven different triangle areas, which are represented by the arrays middle1 to middle7.

The control value is then calculated as a weighted average of these degrees of membership, whereby the weighting is specified by the values in the array `U`. To ensure numerical stability, a check is made to ensure that the sum of the degrees of membership is not equal to zero to avoid division by zero. The resulting control value is limited to ensure that it remains within a specified range, for example between 0 and 1023.

In addition, the applyDamping() function is used to enable damping of the control value by mixing the current value with the previous one. The savetriangles() function stores the calculated membership levels of the fuzzy sets for a range of input values in a CSV file to allow detailed analysis of the fuzzy logic control.

In the `main` function, the control logic is simulated via a loop in which input values from 0 to 8 are incremented step by step, and the corresponding control value is calculated and output. Finally, the CSV file is closed, and the simulated control values are output to the console.

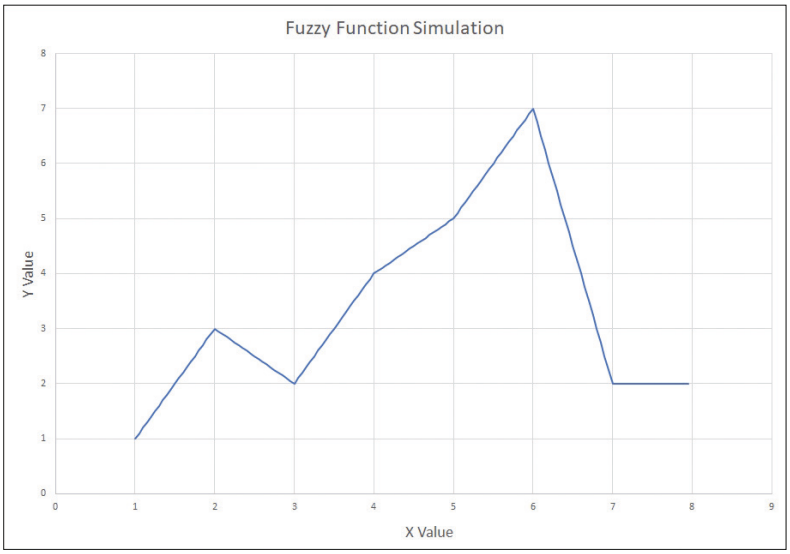


Figure 6.17: Function that is simulated

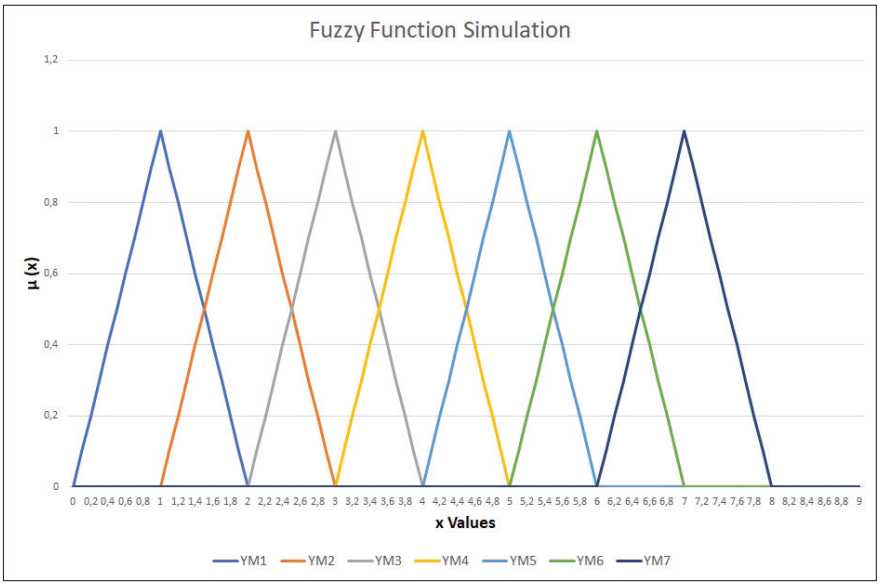


Figure 6.18: Fuzzy Memberships for the function

Program example in CodeBlocks:

```
#include <stdio.h>
#include <math.h>

// These arrays are used to represent the degrees of membership
// of an input value in different regions of a fuzzy set.

// float left[] = { -200, -100 };        // leftopen
float middle1[] = { 0, 1, 2 };
float middle2[] = { 1, 2, 3 };
float middle3[] = { 2, 3, 4 };
float middle4[] = { 3, 4, 5 };
float middle5[] = { 4, 5, 6 };
float middle6[] = { 5, 6, 7 };
float middle7[] = { 6, 7, 8 };
// float right[] = { 100, 200 };        // right open

float U[] = { 1, 3, 2, 4, 5, 7, 2 };    // Anpassung der Ausgangswerte

// This function calculates the membership value of a given input 'x'
// in a left-open Triangle membership function.
// The left-open membership function returns 1.0 for
// input values less than the mean,
// and the standard Triangle membership function value otherwise.
float fuzzyLeftOpen(float x, float a, float b)
{
  if (x <= a)
  {
    return 1.0;
  }
  else if (x <= b)
  {
    return (b - x) / (b - a);
  }
  else
  {
    return 0.0;
  }
}

// This function calculates the membership value of a given input 'x'
// in a standard Triangle membership function.
// The standard Triangle membership function is continuous and
// returns a value based on the input
float fuzzyTriangle(float x, float a, float b, float c)
```

```
{
  if (x <= a || x >= c)
  {
    return 0.0;
  }
  else if (x <= b)
  {
    return (x - a) / (b - a);
  }
  else
  {
    return (c - x) / (c - b);
  }
}

// This function calculates the membership value of a given input 'x'
// in a right-open Triangle membership function.
// The right-open Triangle membership function returns 1.0 for
// input values greater than the mean,
// and the standard Triangle membership function value otherwise.
float fuzzyRightOpen(float x, float a, float b)
{
  if (x <= a)
  {
    return 0.0;
  }
  else if (x <= b)
  {
    return (x - a) / (b - a);
  }
  else
  {
    return 1.0;
  }
}


// damping Factor
float applyDamping(float previousValue, float currentValue, float dampingFactor)
{
  return previousValue * (1.0 - dampingFactor) + currentValue * dampingFactor;
}


/*
 Defuzzification is performed by a weighted sum of the membership values
```

```
 of the linguistic variables. The weighting is determined by the values
 of the membership functions and the associated output values.
 Here's an explanation of how it works:
 1. First, the membership degrees for the linguistic variables
    (left, middle, right) are calculated by applying the respective
    fuzzy functions to the error (e).
 2. Then, the weighted sums of these membership degrees are calculated.
    Each sum is multiplied by the corresponding output value (U).
 3. The weighted sums are added to obtain the numerator for the mean.
 4. The sum of the membership degrees is used as the denominator.
 5. Finally, the mean (y) is calculated by dividing the numerator by the
denominator.
 The mean is used as the final control signal.
 Here's an overview of the calculation:
 - Numerator: left_degree * U[0] + middle_degree1 * U[1] .... + right_degree *
U[4]
 - Denominator: left_degree + middle_degree + right_degree
 - Mean: Numerator / Denominator
 This function calculates the control signal (y) based on the error signal (e)
using fuzzy logic.
*/
float fuzzy_control(float e) {
  // Calculate the membership values for the error signal in different regions of
the fuzzy sets.
  float y;                 // Control signal
  static float y_old = 0;  // Control signal

  // float left_degree = fuzzyLeftOpen(e, left[0], left[1]);
// Membership value for the right region
  float middle_degree1 = fuzzyTriangle(e, middle1[0], middle1[1], middle1[2]);
// Membership value for the middle1 region
  float middle_degree2 = fuzzyTriangle(e, middle2[0], middle2[1], middle2[2]);
// Membership value for the middle2 region
  float middle_degree3 = fuzzyTriangle(e, middle3[0], middle3[1], middle3[2]);
// Membership value for the middle3 region
  float middle_degree4 = fuzzyTriangle(e, middle4[0], middle4[1], middle4[2]);
// Membership value for the middle1 region
  float middle_degree5 = fuzzyTriangle(e, middle5[0], middle5[1], middle5[2]);
// Membership value for the middle2 region
  float middle_degree6 = fuzzyTriangle(e, middle6[0], middle6[1], middle6[2]);
// Membership value for the middle3 region
  float middle_degree7 = fuzzyTriangle(e, middle7[0], middle7[1], middle7[2]);
// Membership value for the middle3 region
  // float right_degree = fuzzyRightOpen(e, right[0], right[1]);
// Membership value for the right region
```

```
  // Calculate the weighted sum of membership values to determine the control
signal.
  float numerator = middle_degree1 * U[0] + middle_degree2 * U[1] + middle_
degree3 * U[2] + middle_degree4 * U[3] + middle_degree5 * U[4] + middle_degree6 *
U[5] + middle_degree7 * U[6] ;  // Weighted sum of membership values
  float denominator = middle_degree1 + middle_degree2 + middle_degree3
+ middle_degree4 + middle_degree5 + middle_degree6 + + middle_degree7 ;
// Sum of membership values
  // Avoid division by zero by checking if the denominator is zero.
  if (denominator != 0) {
    // Calculate the control signal as the ratio of the weighted sum to the sum
of membership values.
    y = numerator / denominator;
  } else {
    // If the denominator is zero, set the control signal to zero to avoid
undefined behavior.
    y = 0;
  }
  if (y > 1023) y = 1023;
  if (y < 0) y = 0;
  // y = applyDamping(y_old, y, 0.75);
  y_old = y;
  return y;  // Return the control signal.
}

void savetriangles()
{
    float e;
    FILE *file = fopen("FuzzyTrianglesFunction.csv", "w"); // Öffnen der Datei im
Schreibmodus
    if (file == NULL) {
        fprintf(stderr, " Error opening the file.\n");
        return 1;
    }
    fprintf(file, "E;YM1;YM2;YM3;YM4;YM5;YM6;YM7\n");
    // Loop for test output and CSV output
    for (e = 0.0; e <= 9.0; e +=0.1)
    {
       // float left_degree = fuzzyLeftOpen(e, left[0], left[1]);
// Membership value for the right region
      float middle_degree1 = fuzzyTriangle(e, middle1[0], middle1[1], middle1[2]);
// Membership value for the middle1 region
      float middle_degree2 = fuzzyTriangle(e, middle2[0], middle2[1], middle2[2]);
// Membership value for the middle2 region
      float middle_degree3 = fuzzyTriangle(e, middle3[0], middle3[1], middle3[2]);
// Membership value for the middle3 region
```

```
      float middle_degree4 = fuzzyTriangle(e, middle4[0], middle4[1], middle4[2]);
// Membership value for the middle1 region
      float middle_degree5 = fuzzyTriangle(e, middle5[0], middle5[1], middle5[2]);
// Membership value for the middle2 region
      float middle_degree6 = fuzzyTriangle(e, middle6[0], middle6[1], middle6[2]);
// Membership value for the middle3 region
      float middle_degree7 = fuzzyTriangle(e, middle7[0], middle7[1], middle7[2]);
// Membership value for the middle3 region
      // float right_degree = fuzzyRightOpen(e, right[0], right[1]);
// Membership value for the right region

      printf("E: %.2f M1: %.2f M2: %.2f M3: %.2f M4: %.2f M5: %.2f M6: %.2f M7:
%.2f\n",e,middle_degree1,middle_degree2,middle_degree3,middle_degree4,middle_
degree5,middle_degree6,middle_degree7);
      fprintf(file,"%.2f;%.2f;%.2f;%.2f;%.2f;%.2f;%.2f;%.2f\n",e,middle_
degree1,middle_degree2,middle_degree3,middle_degree4,middle_degree5,middle_
degree6,middle_degree7);

    }
    fclose(file); // Close File
}


// main programm
int main()
{
    float control_value = 0.0;

    savetriangles();

    for (float xaxis = 0.0; xaxis < 8.0; xaxis +=0.05)
    {
        control_value = fuzzy_control(xaxis);
        printf("X AXIS:%.3f ControlValue:%.3f\n", xaxis, control_value);
     }

    system("pause");
    return 0;
}
```

6.10 Application of a fuzzy function for a temperature profile

A temperature profile is required for soldering SMD (surface mount device) components because this process requires precise thermal conditions to ensure a reliable and high-quality soldered connection. A temperature profile describes the exact timing and control of the temperature during the soldering process.

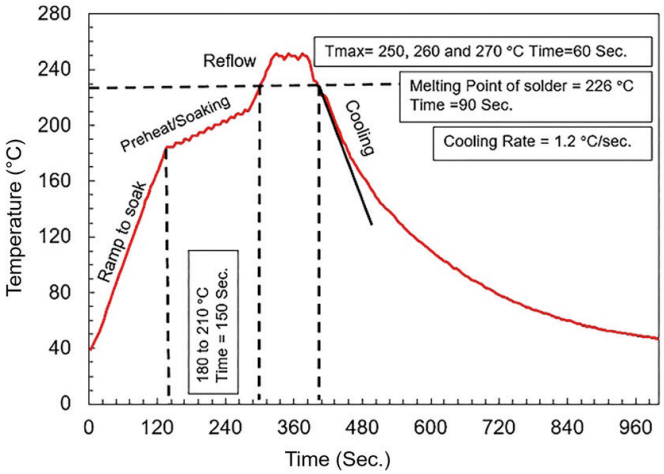


Figure 6.19 Temperature profile for SMD soldering
Quelle: Researchgate.net

This program is very suitable for creating an SMD temperature profile to model the different phases of the soldering process, such as preheating, soaking, reflow and cooling, while allowing flexible and fuzzy transition control between the individual temperature ranges.

Here are the main reasons why a temperature profile is important:

1. Avoiding thermal damage: SMD components are often very small and sensitive to high temperatures. An optimized temperature profile ensures that the components are not heated too quickly, which could lead to thermal damage or mechanical stress.

2. Ensuring soldering quality: Solder has specific melting temperatures at which it produces the best solder joints. A correct temperature profile ensures that the solder melts completely to create a good connection between the component and the PCB and that it then cools correctly to form a solid and permanent connection.

3. Avoid defects (e.g. voids, cold solder joints): If the temperature rises or falls too quickly, defects such as voids or cold solder joints can occur. A precise temperature profile minimizes the risk of such defects by providing a controlled heating and cooling phase.

4. Flux activation: Flux contained in solder paste is activated by heat. It cleans the solder joints by removing oxides to enable a better connection. A temperature profile ensures that the flux is activated correctly and fulfils its task efficiently.

5. Minimization of component displacement: During soldering, SMD components can slip due to surface stresses in the molten solder. A well-adjusted temperature profile helps to minimize these effects and ensure that the components remain in their intended position.

6. Reproducibility: In industrial production, it is important that soldering processes are reproducible to achieve consistent results. A standardized temperature profile makes it possible to precisely control each soldering process and ensure that all PCBs produced are of the same quality.

Overall, a correct temperature profile in SMD soldering ensures that the components and solder joints are thermally optimally treated, maximizing both the reliability and longevity of the solder joints.

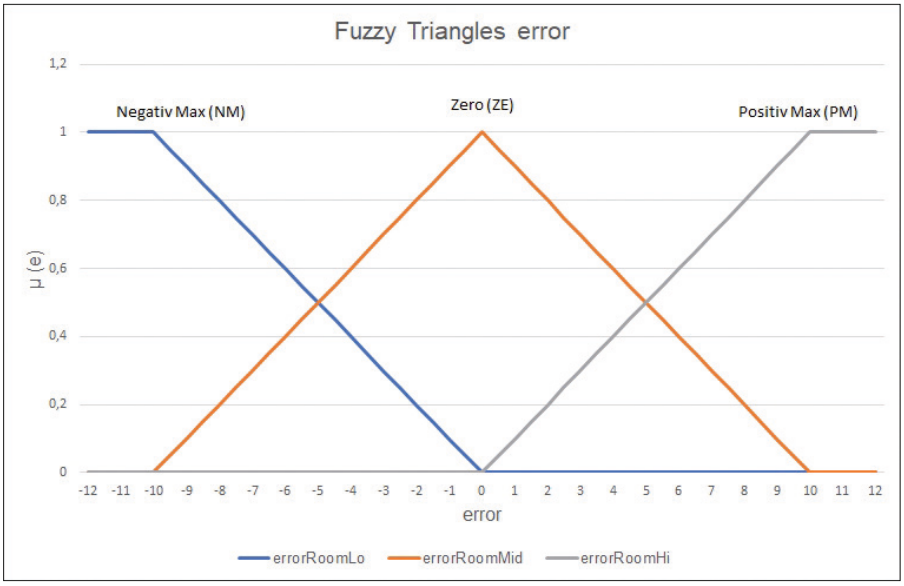
Chapter 7 • Fuzzy Logic temperature controller

7.1 Fuzzy controller with PT2 simulation

In this chapter, two temperature controllers are explained, one with the PT2 simulation and one with a real heating system.

In comparison to the previous example with fan control, the fuzzy sets here work with the temperature deviation (error) instead of the current temperature. This requires the three fuzzy sets 'left open, triangle and right open'.

Here is an illustration of the three sets with the values on the x-axis:



ø=Figure 7.1: Fuzzy triangles for the control deviation

From the fuzzy sets, you can see that the affiliations must be calculated from the control deviation.

With a positive deviation of 7.5 K, for example, the affiliation to set ZE is 0.25 and to set PM 0.75. To set NM 0. This case occurs if the setpoint has not yet been reached.

If there is a negative deviation of -4.0, the set is assigned to NM 0.4, ZE 0.6 and PM 0. This can happen if the temperature rises above the setpoint (overshoot).

The application would also be very suitable for controlling air conditioning systems, controlling Peltier elements and other applications where a negative control value is required.

The settings for the triangles are made here in the program:

```
float left[]    = { -10, 0};      // Values for the left region
float middle[]  = { -10, 0, 10};  // Values for the middle region
float right[]   = { 0, 10};       // Values for the right region
float U[]       = { 0 , 42 , 57 };// Output Values
```

In the loop() function, these lines are responsible for the control.

Here, the PT2 simulation is called up with the control value. The return value is then the current temperature.

```
temperature = PT2_Simulation(control_signal, 0);
```

In this case, the fuzzy controller is called up with the control deviation e= r-y. The return value is the control value for the next call of the PT2 simulation.

```
// Control signal from the Fuzzy Controller
control_signal = fuzzy_control(setpoint - temperature);
```

In the `fuzzy_control()` function first calculates the fuzzy set memberships from the control deviation.

```
// Membership value for the left region
float left_degree   = fuzzyLeftOpen  (e, left[0],  left[1]);
// Membership value for the middle region
float middle_degree = fuzzyTriangle  (e, middle[0], middle[1], middle[2]);
// Membership value for the right region
float right_degree  = fuzzyRightOpen (e, right[0], right[1]);
```

These values are used to calculate the actuating value from the calculated memberships

and the values in `U[] = { 0 , 42 , 57 }`.

```
// Calculate the weighted sum of membership values to determine the control
signal.
float numerator   = left_degree * U[0] + middle_degree * U[1] + right_degree *
U[2];
// Weighted sum of membership values
float denominator = left_degree + middle_degree + right_degree;
// Control signal
float y = numerator / denominator;
```

The result of the controller after some adjustments in U[1] und U[2].

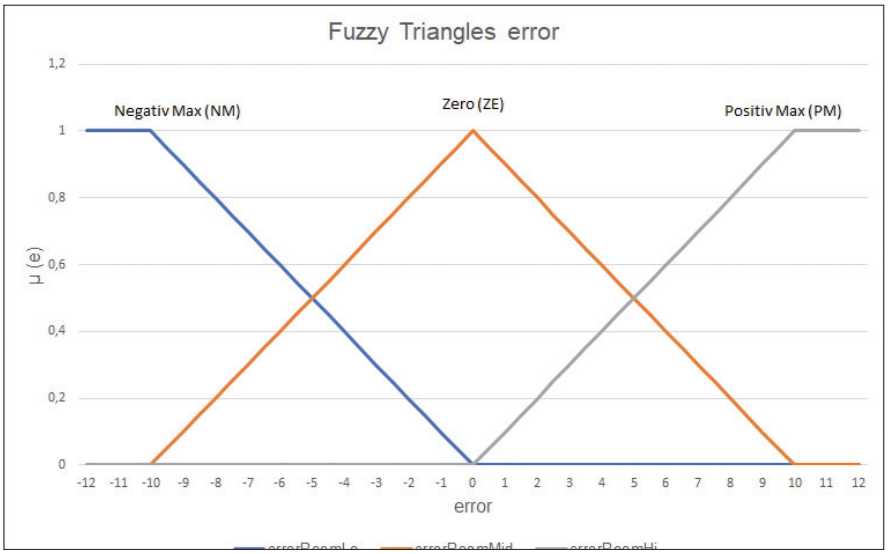


Figure 7.2 Fuzzy temperature control

Listing 7.1: Arduino_PT2_Fuzzy_Triangle.ino

```
//-----------------------------------------------------------------------
//
//            Arduino PT2 Fuzzy Triangle Controller
//            =====================================
//
//      Control Testprogramm for Arduino Boards
//      Fuzzy Triangle Controller Test with PT2 Simulation
//
//
//  Date:     29/04/2024
//  Author:   Josef Bernhardt
//  Hardware: Arduino (ESPRESSIF) ESP32
//
//  File:     Arduino_PT2_Fuzzy_Triangle.ino
//-----------------------------------------------------------------------

// Include libraries
#include <Arduino.h>

// Pin-Definitionen
#define LED_BUILTIN 2 // Pin to which the LED is connected (here GPIO 2 of the
ESP32)
// LED1
#define LED1 5
```

```
// These arrays are used to represent the degrees of membership
// of an input value in different regions of a fuzzy set.
float left[]    = { -10, 0};     // Values for the left region
float middle[]  = { -10, 0, 10}; // Values for the middle region
float right[]   = { 0, 10};      // Values for the right region
// Output Values
float U[]       = { 0 , 42 , 57 };


// Simulating a PT2 control system
float PT2_Simulation(float controlInput, int reset)
{
  // Parameter
  float deltaT = 1.0;
  float ProportionalGain = 1.2;
  float IntegralGain = 1.0;
  float DerivativeGain = 1.0;
  float TimeConstant1 = 100.0;
  float TimeConstant2 = 8.0;
  float TimeConstant3 = 10.0;

  // State variables
  static float output1 = 0;
  static float output2 = 0;
  static float output3 = 0;

  // Reset static Variabel
  if (reset == 1)
  {
    output1 = 0;
    output2 = 0;
    output3 = 0;
  }
  // Model equations, Euler integration
  float d_output1 = (ProportionalGain * controlInput - output1) / TimeConstant1;
  float d_output2 = (IntegralGain * output1 - output2) / TimeConstant2;
  float d_output3 = (DerivativeGain * output2 - output3) / TimeConstant3;

  output1 += d_output1 * deltaT;
  output2 += d_output2 * deltaT;
  output3 += d_output3 * deltaT;

  return output3;
}

// This function calculates the membership value of a given input 'x'
```

```
// in a left-open Triangle membership function.
// The left-open membership function returns 1.0 for
// input values less than the mean,
// and the standard Triangle membership function value otherwise.
float fuzzyLeftOpen(float x, float a, float b) {
  if (x <= a) {
    return 1.0;
  } else if (x <= b) {
    return (b - x) / (b - a);
  } else {
    return 0.0;
  }
}

// This function calculates the membership value of a given input 'x'
// in a standard Triangle membership function.
// The standard Triangle membership function is continuous and
// returns a value based on the input
float fuzzyTriangle(float x, float a, float b, float c) {
  if (x <= a || x >= c) {
    return 0.0;
  } else if (x <= b) {
    return (x - a) / (b - a);
  } else {
    return (c - x) / (c - b);
  }
}

// This function calculates the membership value of a given input 'x'
// in a right-open Triangle membership function.
// The right-open Triangle membership function returns 1.0 for
// input values greater than the mean,
// and the standard Triangle membership function value otherwise.
float fuzzyRightOpen(float x, float a, float b) {
  if (x <= a) {
    return 0.0;
  } else if (x <= b) {
    return (x - a) / (b - a);
  } else {
    return 1.0;
  }
}

// This function uses the Center of Gravity method for defuzzification.
// Defuzzification is performed by a weighted sum of the membership values
// of the linguistic variables. The weighting is determined by the values
```

```
// of the membership functions and the associated output values.
// Here's an explanation of how it works:
// 1. First, the membership degrees for the linguistic variables
//    (left, middle, right) are calculated by applying the respective
//    fuzzy functions to the error (e).
// 2. Then, the weighted sums of these membership degrees are calculated.
//    Each sum is multiplied by the corresponding output value (U).
// 3. The weighted sums are added to obtain the numerator for the mean.
// 4. The sum of the membership degrees is used as the denominator.
// 5. Finally, the mean (y) is calculated by dividing the numerator by the
denominator.
// The mean is used as the final control signal.
// Here's an overview of the calculation:
// - Numerator: left_degree * U[0] + middle_degree * U[1] + right_degree * U[2]
// - Denominator: left_degree + middle_degree + right_degree
// - Mean: Numerator / Denominator
// This function calculates the control signal (y) based on the error signal (e)
using fuzzy logic.
float fuzzy_control(float e)
{
// Calculate the membership values for the error signal in different regions of
the fuzzy sets.
// Membership value for the left region
float left_degree   = fuzzyLeftOpen  (e, left[0],  left[1]);
// Membership value for the middle region
float middle_degree = fuzzyTriangle  (e, middle[0], middle[1], middle[2]);
// Membership value for the right region
float right_degree  = fuzzyRightOpen (e, right[0], right[1]);

// Calculate the weighted sum of membership values to determine the control
signal.
float numerator   = left_degree * U[0] + middle_degree * U[1] + right_degree *
U[2]; // Weighted sum of membership values
float denominator = left_degree + middle_degree + right_degree;
// Sum of membership values
float y; // Control signal

  // Avoid division by zero by checking if the denominator is zero.
  if (denominator != 0) {
// Calculate the control signal as the ratio of the weighted sum to the sum of
membership values.
    y = numerator / denominator;
  } else {
// If the denominator is zero, set the control signal to zero to avoid undefined
behavior.
    y = 0;
```

```
  }
  return y; // Return the control signal.
}

// Init Hardware
void setup() {
  pinMode(LED_BUILTIN, OUTPUT); // Configure pin as output
  pinMode(LED1, OUTPUT); // Configure pin as output

  // Init UART USB
  Serial.begin(115200);
}

/*
Control variable: y
Setpoint : r or w
Error: e
Control signal: u
*/
void loop()
{
  // Place your main code here to execute repeatedly:
  float setpoint = 50;
  float temperature = 0;
  float control_signal = 0.0;
  float hysteresis = 0.0;

  // Reset PT2 System
  temperature = PT2_Simulation(0, 1);
  // toggle LED
  digitalWrite(LED_BUILTIN, !digitalRead(LED_BUILTIN));
  // wait 200 ms
  delay(200);
  for (int time = 0; time < 400; time++)
  {

    // if (time > 200) setpoint = 60;

    // Simulating the temperature control system
    digitalWrite(LED1, 1);

    temperature = PT2_Simulation(control_signal, 0);

    // Control signal from the Fuzzy Controller
    control_signal = fuzzy_control(setpoint - temperature);
```

```
    Serial.print(" Setpoint:");
    Serial.print(setpoint, 1);         // Setpoint with one decimal place
    Serial.print(" Temperature:");
    Serial.print(temperature, 1);      // Temperature with one decimal place
    Serial.print(" ControlSignal:");
    Serial.print(control_signal, 1);   // Control signal with one decimal place
    Serial.println();                  // New line for the next output
    digitalWrite(LED1, 0);
    delay(50);

  }
}
```

7.2 Fuzzy controller with the ESP32 and real heating

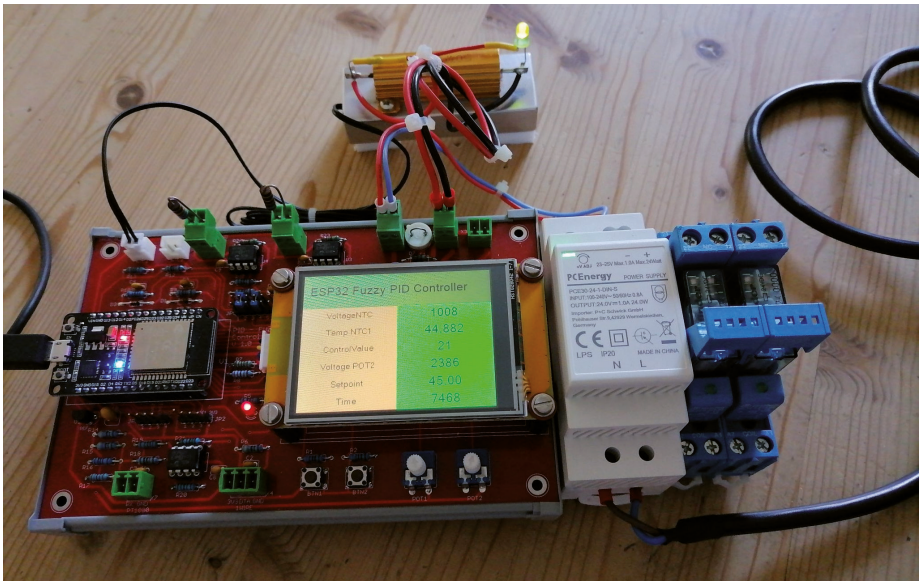


Figure 7.3 Fuzzy and PID ESP32 Board

This practical example is about real heating. An aluminium block measuring 80 x 30 x 15 mm is heated with a 30 Ohm 50 Watt resistor. At 24 V, the maximum power is P = U2 / R = 19.2 watts. An NTC 10 K sensor is used for temperature measurement. The examples for an exact measurement have already been explained in Chapter 2 and Chapter 4.

In order to enable a finer gradation of the heating power, the fuzzy sets have been extended by two triangles compared to the previous example with the PT2 simulation. There are also two additional entries in U[] for calculating the control values.

```
// These arrays are used to represent the degrees of membership
// of an input value in different regions of a fuzzy set.
float NM[]   = {-6,-3};      // Values for the left region
float NS1[]  = {-6,-3, 0};   // Values for the middle region
float ZE[]   = {-3, 0, 3};   // Values for the middle region
float PS1[]  = { 0, 3, 6};   // Values for the middle region
float PM[]   = { 3, 6};      // Values for the right region

// Output Values
float U[]       = { 0 , 0, 18 , 40, 100 };
```

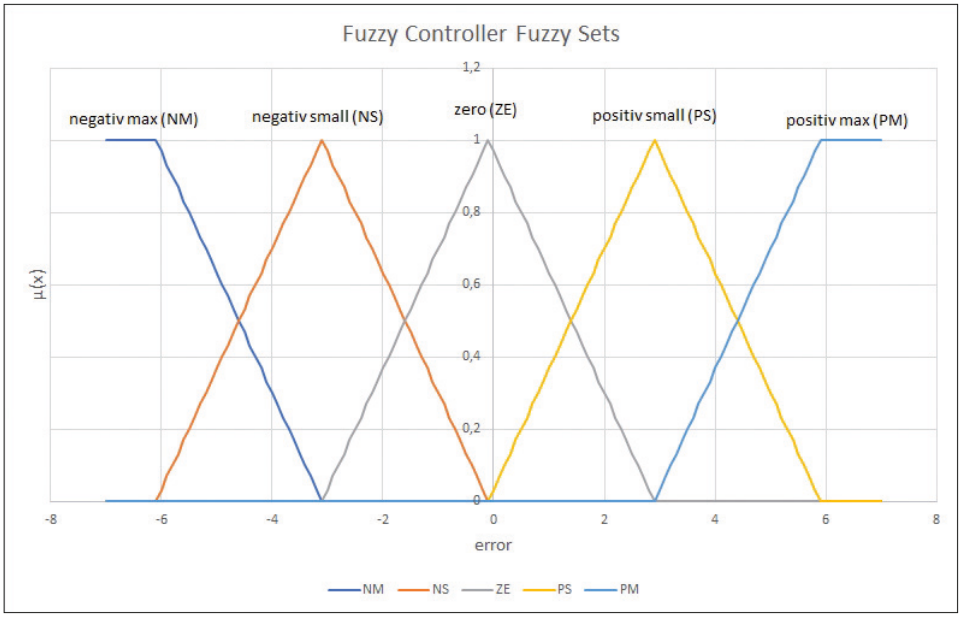


Figure 7.4 Fuzzy Memberships temperature controller

The finer gradation with the additional fuzzy sets also affects the fuzzy memberships temperature controller characteristic curve of the controller.

The calculation of the fuzzy sets:

```
// Membership value for the left region
float left_degree_NM   = fuzzyLeftOpen  (e, NM[0],  NM[1]);
// Membership value for the middle region NS
float middle_degree_NS = fuzzyTriangle  (e, NS1[0],NS1[1],NS1[2]);
// Membership value for the middle region ZE
float middle_degree_ZE = fuzzyTriangle  (e, ZE[0],ZE[1],ZE[2]);
// Membership value for the middle region PS
float middle_degree_PS = fuzzyTriangle  (e, PS1[0],PS1[1],PS1[2]);
```

```
// Membership value for the right region
float right_degree_PM  = fuzzyRightOpen (e, PM[0], PM[1]);

// Calculate the weighted sum of membership values
float numerator   = left_degree_NM * U[0] + middle_degree_NS * U[1] + middle_
degree_ZE * U[2]  + middle_degree_PS * U[3]  + right_degree_PM * U[4];
// Sum of membership values
float denominator = left_degree_NM + middle_degree_NS+ middle_degree_ZE + middle_
degree_PS  + right_degree_PM;
```

The characteristic curve of the controller for a control deviation range of -7 to +7 K. The more flat curve from -3 to +3 is due to the lower values in U[] for the ZE range.

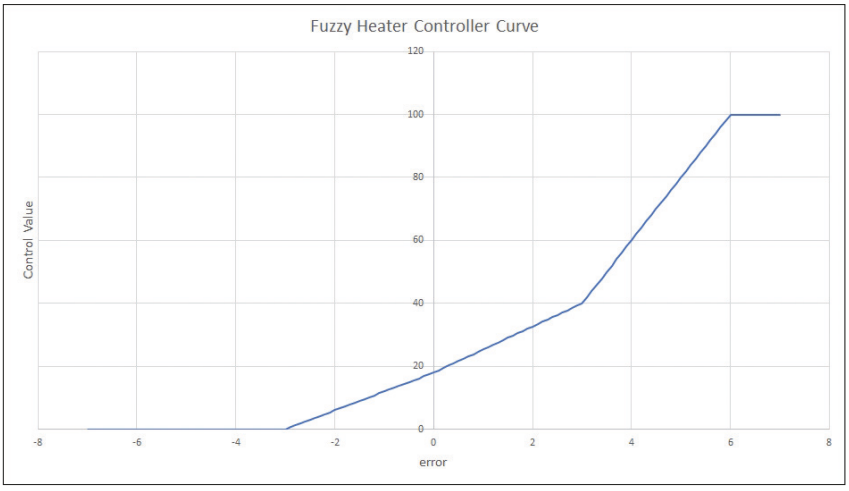


Figure 7.5 Temperature controller characteristic curve

Temperature curve at a setpoint value of 50 K. From a temperature of 44 °C, the controller begins to reduce the output and stabilises at approx. 20 % of the 19.2 watts.

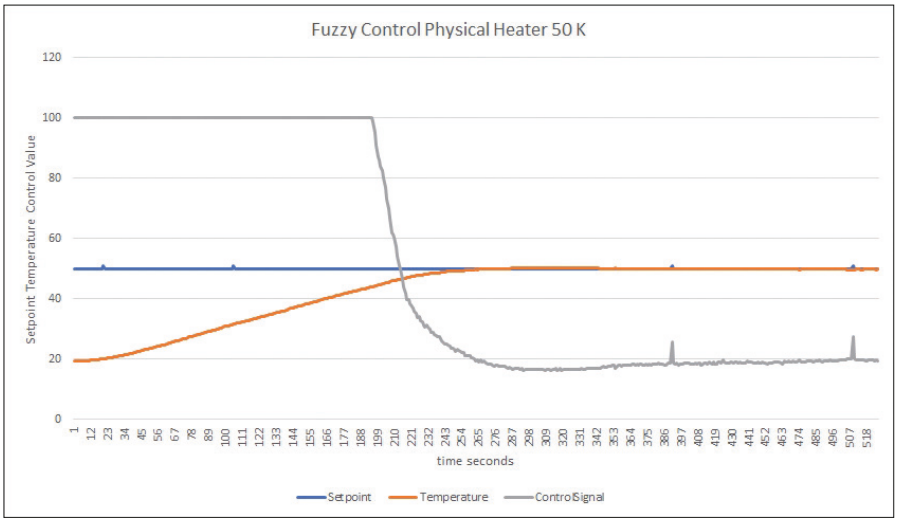


Figure 7.6 Heating temperature curve

With this controller, disturbances in the system are also quickly corrected.

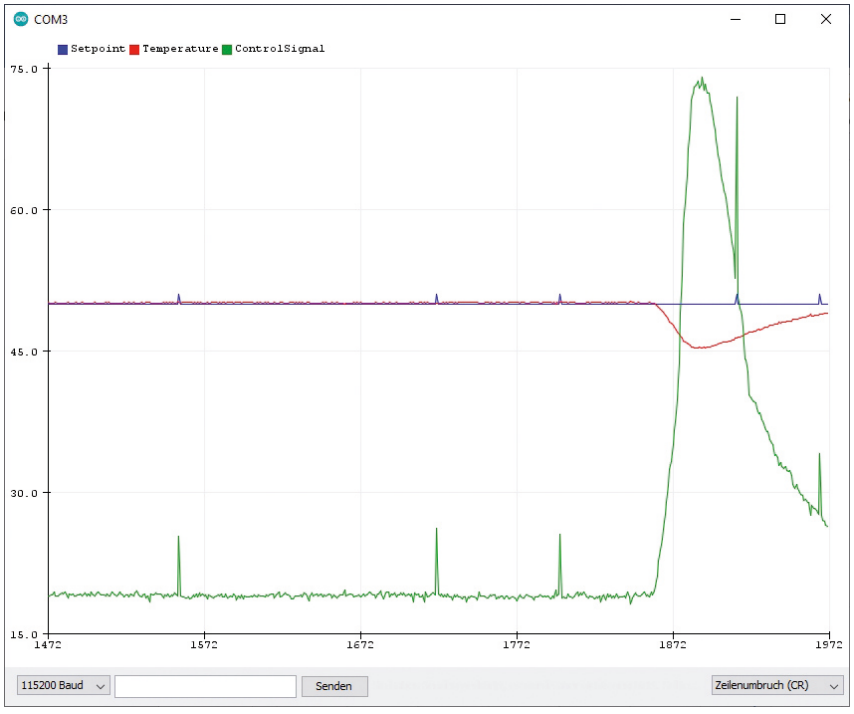


Figure 7.7 Behaviour in case of malfunctions

Listing 7.2: Arduino_PhysicalHeater_Fuzzy_Triangle.ino

```
//-------------------------------------------------------------------
//
//            Arduino Fuzzy Triangle Physical Heater
//            ======================================
//
//      Control Software for Arduino ESP32 PID Fuzzy Control Board
//      10ms Timer Interrupt Function
//      Init UART2 for Nextion Display
//      Display ADC1 and ADC2 Values on Display NX3224F028_001
//
//      Reading NTC1 Sensor / Set the Setpoint with POT2
//      Fuzzy Triangle Controller Example
//      Outputting the pulses to RELAIS1 for the heater
//
//
//  Date:     12/06/2024
//  Author:   Josef Bernhardt
//  Hardware: Arduino (ESPRESSIF) ESP32
//
//  File:     Arduino_PhysicalHeater_Fuzzy_Triangle.ino
//-------------------------------------------------------------------
#include <Arduino.h>
#include <Ticker.h>

int LED_BUILTIN = 2;

// ESP32 UART 2 Pins
#define RXD2 16
#define TXD2 17

// I2C Pins
#define I2CSCL 22
#define I2CSDA 21

// ESP32 ADC Pins
#define ADC1 32
#define ADC2 33
// Motor Sensor Pins
#define MOTSENS1 15
#define MOTSENS2 34
// Buttons
#define BTN1 14
#define BTN2 27
```

```
// Relay and Mosfet (BSP76) Pins
#define RELAIS1 18
#define RELAIS2 19
// LED1
#define LED1 5
// PWM1 and PWM2 Pins
#define PWM1 12
#define PWM2 13
// DAC Outputs 0..5V
#define DAC1 25
#define DAC2 26
// One Wire
#define DS18B20 4

#define FILTER_SIZE 10

// Define 10ms Period
const unsigned long timerInterval = 10;

// Ticker-Objekt for the Timer-Funktion
Ticker timer;

// Counter
int counter10ms=0;

int counterRelais1 = 0;

// When 1 second is elapsed
bool OneSecond = false;

// Counter for seconds
int CounterSeconds = 0;

// Arrays for NTC10K calculation
float Voltages[18] = {0};
float Temperatures[18] = {0};
int n_ntc = 18;

/*
Set the voltage at ADC1 with the potentiometer to e.g. 1 volt.
Enter the value voltage1 in the display for the corresponding
value "float ADC[7]". Repeat this with the other voltages
from 100 mV to 3.3 volts.
12 Bit ADC-Values from 0 to 4095
*/
float ADC[] = {0,   0,    123,   249, 496,  817,  1141, 1456, 1778, 2416, 3067,
```

```
3922, 4095, 4095, 4095};
// (Ua) in volts values applied to ADC by calibrator
float Ua[] = {0.00, 0.10, 0.20, 0.30, 0.50, 0.75, 1.00, 1.25, 1.50, 2.00, 2.50,
3.00, 3.07, 3.15, 3.30};
int n_adc = 15;


float VoltageESP32 = 3.3;

// Resistor is 30 Ohm 50 W Voltage at 24 Volt

// These arrays are used to represent the degrees of membership
// of an input value in different regions of a fuzzy set.
float Nmax[]  = {-6,-3};         // Values for the left region
float Nsmall[]  = {-6,-3, 0};    // Values for the middle region
float Zero[]  = {-3, 0, 3};      // Values for the middle region
float Psmall[]  = { 0, 3, 6};    // Values for the middle region
float Pmax[]  = { 3, 6};         // Values for the right region

// Output Values
float U[]       = { 0 , 0, 20 , 40, 100 };

// This function calculates the membership value of a given input 'x'
// in a left-open Triangle membership function.
// The left-open membership function returns 1.0 for
// input values less than the a and 0 if greater b,
// and the standard Triangle membership function value otherwise.
float fuzzyLeftOpen(float x, float a, float b) {
    if (x <= a) {
        return 1.0;
    } else if (x <= b) {
        return (b - x) / (b - a);
    } else {
        return 0.0;
    }
}

// This function calculates the membership value of a given input 'x'
// in a standard Triangle membership function.
// The standard Triangle membership function is continuous and
// returns a value based on the input
float fuzzyTriangle(float x, float a, float b, float c) {
    if (x <= a || x >= c) {
        return 0.0;
    } else if (x <= b) {
        return (x - a) / (b - a);
```

```
    } else {
        return (c - x) / (c - b);
    }
}

// This function calculates the membership value of a given input 'x'
// in a right-open Triangle membership function.
// The right-open Triangle membership function returns 1.0 for
// input values greater than b,
// and the standard Triangle membership function value otherwise.
float fuzzyRightOpen(float x, float a, float b) {
    if (x <= a) {
        return 0.0;
    } else if (x <= b) {
        return (x - a) / (b - a);
    } else {
        return 1.0;
    }
}

// This function uses the Center of Gravity method for defuzzification.
// Defuzzification is performed by a weighted sum of the membership values
// of the linguistic variables. The weighting is determined by the values
// of the membership functions and the associated output values.
// Here's an explanation of how it works:
// 1. First, the membership degrees for the linguistic variables
//    (left, middle, right) are calculated by applying the respective
//    fuzzy functions to the error (e).
// 2. Then, the weighted sums of these membership degrees are calculated.
//    Each sum is multiplied by the corresponding output value (U).
// 3. The weighted sums are added to obtain the numerator for the mean.
// 4. The sum of the membership degrees is used as the denominator.
// 5. Finally, the mean (y) is calculated by dividing the numerator by the
denominator.
// The mean is used as the final control signal.
// Here's an overview of the calculation:
// - Numerator: left_degree * U[0] + middle_degree * U[1] + right_degree * U[2]
// - Denominator: left_degree + middle_degree + right_degree
// - Mean: Numerator / Denominator
// This function calculates the control signal (y) based on the error signal (e)
using fuzzy logic.
float fuzzy_control(float e)
{
    // Calculate the membership values for the error signal in different regions
of the fuzzy sets.
    float left_degree_NM   = fuzzyLeftOpen  (e, Nmax[0],  Nmax[1]);     //
```

```
Membership value for the left region
    float middle_degree_NS = fuzzyTriangle  (e, Nsmall[0],Nsmall[1],Nsmall[2]);
// Membership value for the middle region NS
    float middle_degree_ZE = fuzzyTriangle  (e, Zero[0],Zero[1],Zero[2]); //
Membership value for the middle region ZE
    float middle_degree_PS = fuzzyTriangle  (e, Psmall[0],Psmall[1],Psmall[2]);
// Membership value for the middle region PS
    float right_degree_PM  = fuzzyRightOpen (e, Pmax[0], Pmax[1]);      //
Membership value for the right region

    // Calculate the weighted sum of membership values
    float numerator   = left_degree_NM * U[0] + middle_degree_NS * U[1] + middle_
degree_ZE * U[2]  + middle_degree_PS * U[3]  + right_degree_PM * U[4];
    // Sum of membership values
    float denominator = left_degree_NM + middle_degree_NS + middle_degree_ZE +
middle_degree_PS  + right_degree_PM;                 // Sum of membership values

    float y; // Control signal

    // Avoid division by zero by checking if the denominator is zero.
    if (denominator != 0) {
        // Calculate the control signal as the ratio of the weighted sum to the
sum of membership values.
        y = numerator / denominator;
    } else {
        // If the denominator is zero, set the control signal to zero to avoid
undefined behavior.
        y = 0;
    }
    return y; // Return the control signal.
}

void Heat1_ON(int value)
{
  //Serial.println("ON");
  if (value > 0)
  {
    counterRelais1 = value;
    digitalWrite(RELAIS1,1);
    digitalWrite(LED1,1);
  }
}

void Heat1_OFF()
{
  //Serial.println("OFF");
```

```
  digitalWrite(RELAIS1,0);
  digitalWrite(LED1,0);
}

int mapInteger(int value, int fromLow, int fromHigh, int toLow, int toHigh) {
    return (value - fromLow) * (toHigh - toLow) / (fromHigh - fromLow) + toLow;
}

float mapFloat(float x, float in_min, float in_max, float out_min, float out_max)
{
  return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
}

// Linear interpolation
// Input Parameters:
// n = Number of values -1
// x = x Array
// y = y Array
// xn = Value to be calculated
float readvalue_linear(int n,float *x,float *y, float xn)
{
    int j = 0;
    // Determine interval of xn
    while (j < n-1 && xn > x[j+1]) j++;
    // interpolate xn
    return ((y[j+1]-y[j]) / (x[j+1]-x[j])) * (xn-x[j]) + y[j];
}

// Function to calculate the resistance of an NTC thermistor based on temperature
float calculateResistance(float T)
{
    // Constants
    const float R0 = 10000.0; // Nominal resistance of the thermistor at 25°C
    const float B = 3950.0;   // B-value of the thermistor see data sh
    return R0 * exp(B * ((1.0 / (T + 273.15)) - (1.0 / (25 + 273.15))));
}

// Voltage from voltage divider 10k input NTC resistor
float calulateVoltage(float Rt)
{
    float R25 = 10000;
    float UE = VoltageESP32;
    float Ua=(UE*Rt)/(Rt+R25);
    return Ua;
}
```

```
// Filter of NTC1 Sensor
float filterNTC_1(float newValue) {
    // Array for saving the last FILTER_SIZE values
    static float values[FILTER_SIZE] = {0};
    static int index = 0;
    float sum = 0;
    // Update value at the current index position
    values[index] = newValue;
    // Increment the index for the next value
    index = (index + 1) % FILTER_SIZE;
    // Calculation of the average of the values in the array
    for (int i = 0; i < FILTER_SIZE; ++i) {
        sum += values[i];
    }
    return sum / FILTER_SIZE;
}

// Init Array with NTC Resist and Temperature
void initArrayNTC()
{
    // Temperature range from -35°C to 135°C
    int min_temp = -35;
    int max_temp = 135;
    float voltage;
    int loop = sizeof(Voltages) / sizeof(Voltages[0]) - 1;
    // Loop over the temperature range
    for (float temp = min_temp; temp <= max_temp; temp+=10)
    {
        // Calculating the resistance for the current temperature
        float resistance = calculateResistance(temp);
        float voltage = calulateVoltage(resistance);
        // Saving the values for temperature and voltage in arrays
        Temperatures[loop] = temp;
        Voltages[loop] = voltage;
        loop--;
    }
}

// Timer-Funktion
// Called every 10 ms
void timerCallback()
{
  // Inc 10ms Counter
  counter10ms++;
  counterRelais1--;
  // If one Second
```

```
  if(counter10ms==100)
  {
     counter10ms = 0;
     // One Second
     OneSecond = true;
  }
  if(counterRelais1==0)
  {
     Heat1_OFF();
  }
}

// Sends Data to Nextion Display in a textbox
void Nextion_Send_Text_to_Textbox(char *textbox,String sfloat)
{
        Serial2.write(textbox);
        Serial2.write(".txt=\"");
        Serial2.print(sfloat);
        Serial2.write("\"");
        Serial2.write(0xFF);
        Serial2.write(0xFF);
        Serial2.write(0xFF);
}

// Initializing the hardware
void setup()
{
  // Init USB UART
  Serial.begin(115200);
  // Nextion Display
  // TFTSerial.begin(9600);
  Serial2.begin(9600, SERIAL_8N1, RXD2, TXD2);
  // Init Hardware
  pinMode(LED_BUILTIN, OUTPUT);
  pinMode(LED1, OUTPUT);
  pinMode(RELAIS1, OUTPUT);
  pinMode(RELAIS2, OUTPUT);
  pinMode(BTN1, INPUT);
  pinMode(BTN2, INPUT);

  pinMode(ADC1, INPUT);
  pinMode(ADC2, INPUT);

  Nextion_Send_Text_to_Textbox("t1","VoltageNTC mV");
  Nextion_Send_Text_to_Textbox("t2","Temp NTC1");
  Nextion_Send_Text_to_Textbox("t3","ControlValue %");
```

```
  Nextion_Send_Text_to_Textbox("t4","Voltage POT2 mV");
  Nextion_Send_Text_to_Textbox("t5","Setpoint");
  Nextion_Send_Text_to_Textbox("t6","Time");

  initArrayNTC();

  // Init Timer with 10ms
  timer.attach_ms(timerInterval, timerCallback);

 /*
  // Output of
  for (float x=-7.0; x<=7.0; x+=0.1)
  {
    float control_signal = fuzzy_control(x);
    Serial.print(" E:");
    Serial.print(x, 2); // Control signal with one decimal place
    Serial.print(" U:");
    Serial.println(control_signal, 2); // Control signal with one decimal place
  }
*/

}

void loop()
{
  // Place your main code here to execute repeatedly:
  float setpoint;
  float temperature = 0;
  float control_signal = 0.0;

  // Read ADC1 value
  int voltage1 = analogRead(ADC1);
  // Read ADC2 value
  int voltage2 = analogRead(ADC2);

  int scaledValue = map(voltage2, 0, 4095, 25, 60);
  setpoint = (float)scaledValue;

  // Interpolate voltage 1
  float u1 = readvalue_linear(n_adc-1,ADC,Ua,(float)voltage1);

  temperature = readvalue_linear(n_ntc-1,Voltages,Temperatures,u1);

  // Check if one second has passed
  if (OneSecond)
  {
```

```
      // Reset the boolean variable OneSecond
      OneSecond = false;
      CounterSeconds++;

      // Control signal from Fuzzy Controller
      control_signal = fuzzy_control(setpoint-temperature);

      // Heat1 10-1000 ms on step 10ms
      Heat1_ON(control_signal);

      // control_signal = twoPointController(setpoint,temperature,hysteresis);
      // Sends variable to USB UART and Seriell Monitor
      Serial.print("SP:");
      Serial.print(setpoint);         // Setpoint with one decimal place
      Serial.print(" TMP:");
      Serial.print(temperature,2);    // Temperature with one decimal place
      Serial.print(" CV:");
      Serial.print(control_signal,1); // Control signal with one decimal place
      Serial.println();               // New line for the next output

      // Sends variable to display
      Nextion_Send_Text_to_Textbox("txtT1",String(voltage1));
      Nextion_Send_Text_to_Textbox("txtT2",String(temperature));
      Nextion_Send_Text_to_Textbox("txtT3",String(control_signal));
      // Sends variable to display
      Nextion_Send_Text_to_Textbox("txtT4",String(voltage2));
      Nextion_Send_Text_to_Textbox("txtT5",String(setpoint));
      Nextion_Send_Text_to_Textbox("txtT6",String(CounterSeconds));
      // Switching LED
      digitalWrite(LED_BUILTIN,!digitalRead(LED_BUILTIN));
 }
 // Additional code for the main loop
}
```

7.3 Fuzzy examples with Gauss, Bell and Sigmoid

Further examples of fuzzy controllers with the Bell and Gauss membership functions are also available for the Arduino. Both as Arduino versions for real heating and as PT2 simulations.

Chapter 8 • Fuzzy logic temperature controller with two inputs

8.1 Temperature controller with two inputs

In comparison to the fan control example in Chapter 6, the fuzzy sets work with the current temperature difference (error) and the temperature deviation (derror). This requires the appropriate three fuzzy sets 'left open, triangle and right open'.

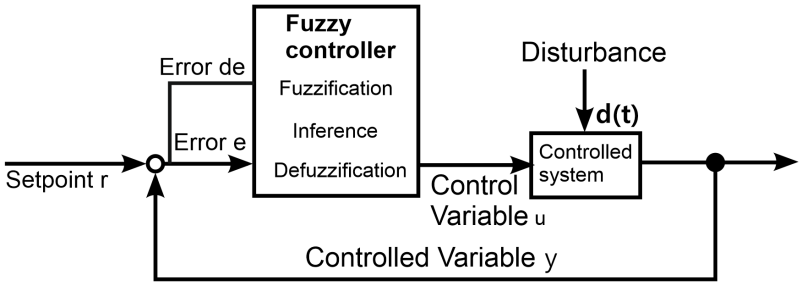


Figure 8.1: Overview of fuzzy controller with two inputs

This is a graphic of the membership functions for the control deviation. The range can be seen on the X-axis and goes from -8 to +8.

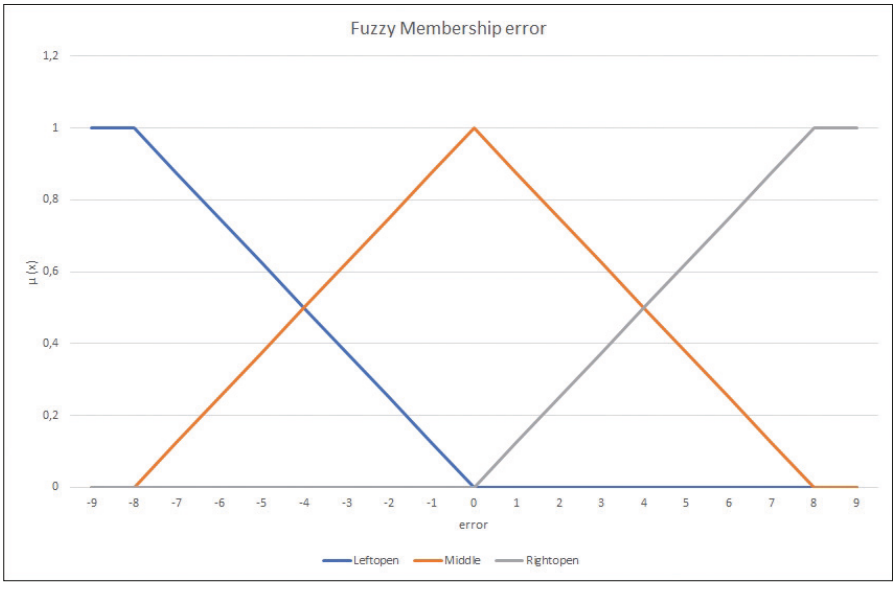


Figure 8.2: Fuzzy Membership error

The following is a figure of the membership functions for the change in the control deviation. Here the range is much smaller and lies between -0.4 and +0.4.

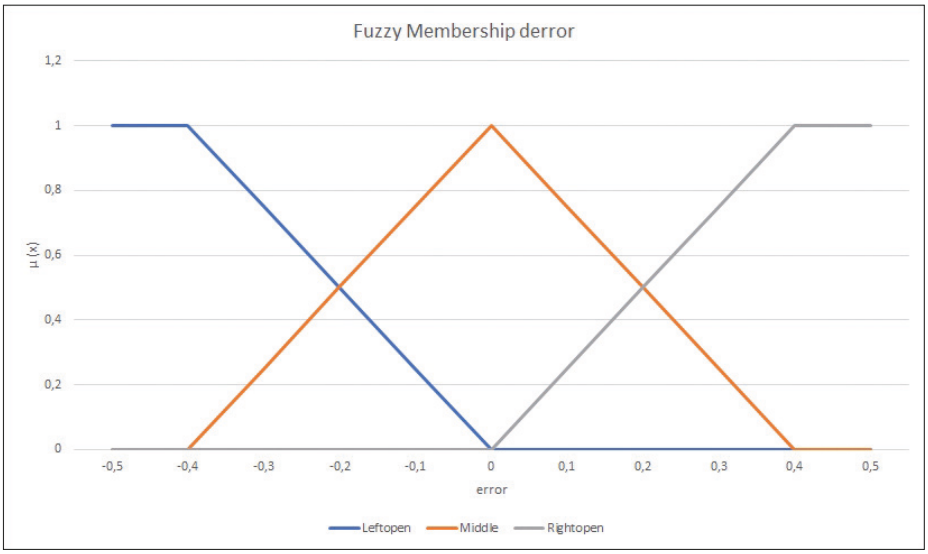


Figure 8.3: Fuzzy Membership derror

Fuzzification:
Fuzzification is first performed for the control deviation (error) and then for the change in the control deviation (derror). The calculated values are then processed with an AND operation.

Listing 8.1 Codeblocks_C_PT2_TempErrorDError

```
// Fuzzification of temperature
    float errorHi  = fuzzyRightOpen(error, 0, 8);         // Error positive
initially
    float errorMid = fuzzyTriangle(error, -8, 0, 8);      // Error 0
    float errorLo  = fuzzyLeftOpen(error, -8, 0);         // Error negative after
overshoot

    // Fuzzification of temperature change
    float dErrorHi =  fuzzyRightOpen(dError, 0, 0.4);     // Delta pos
(temperature rising)
    float dErrorMid = fuzzyTriangle(dError, -0.4, 0, 0.4);// Delta 0
    float dErrorLo =  fuzzyLeftOpen(dError, -0.4, 0);     // Delta neg
(temperature
```

Evaluation of the rules:

```
    If error is high and dError is high, control value is high
    If error is high and dError is medium, control value is very high
    If error is high and dError is low, control value is very high
```

```
If error is medium and dError is high, control value is medium
If error is medium and dError is medium, control value is medium
If error is medium and dError is low, control value is medium

If error is low and dError is high, control value is very low
If error is low and dError is medium, control value is low
If error is low and dError is low, control value is medium
```

The controller function combines the data with the MIN function. With an OR link, it would be the MAX function.

With two inputs, each with three assignment functions, there are nine links.

In a table it looks like this:

	dError: high	dError: medium	dError: low
Error: high	Control: high	Control: very high	Control: very high
Error: medium	Control: medium	Control: medium	Control: medium
Error: low	Control: very low	Control: low	Control: medium

Calculation of the rules:

```
float rule1 = fmin(errorHi, dErrorHi);     // Control MID
float rule2 = fmin(errorHi, dErrorMid);    // Control HIGH
float rule3 = fmin(errorHi, dErrorLo);     // Control VERY HIGH
float rule4 = fmin(errorMid,dErrorHi);     // Control MID
float rule5 = fmin(errorMid,dErrorMid);    // Control MID
float rule6 = fmin(errorMid,dErrorLo);     // Control MID
float rule7 = fmin(errorLo, dErrorHi);     // Control VERY LOW
float rule8 = fmin(errorLo, dErrorMid);    // Control LOW
float rule9 = fmin(errorLo, dErrorLo);     // Control MID
```

Defuzzification:

Defining the control values for defuzzification:

```
// Define control values for the rules
const float VERYHIGH = 100.0;
const float HIGH = 60.0;
const float MID = 40.0;
const float LOW = 35.0;
const float VERYLOW = 33.0;
```

The defuzzification method used in this code is the weighted average of the memberships. This method requires relatively little computing power and is ideal for implementation on microcontrollers. The sum of the weighted values (numerator) divided by the sum of the weights (denominator) is calculated.

```
    // Calculate the weighted average of the rules
    float numerator =   (rule1 * MID) + (rule2 * HIGH) + (rule3 * VERYHIGH) +
                        (rule4 * MID) + (rule5 * MID) + (rule6 * MID) +
                        (rule7 * VERYLOW) + (rule8 * LOW) + (rule9 * MID);
    float denominator = rule1 + rule2 + rule3 + rule4 + rule5 + rule6 + rule7 +
rule8 + rule9;
    float result = 0;
    result = numerator / denominator;
```

For a controller with two inputs, the controller characteristic can no longer be represented with a simple XY characteristic. A 3D representation is required here.

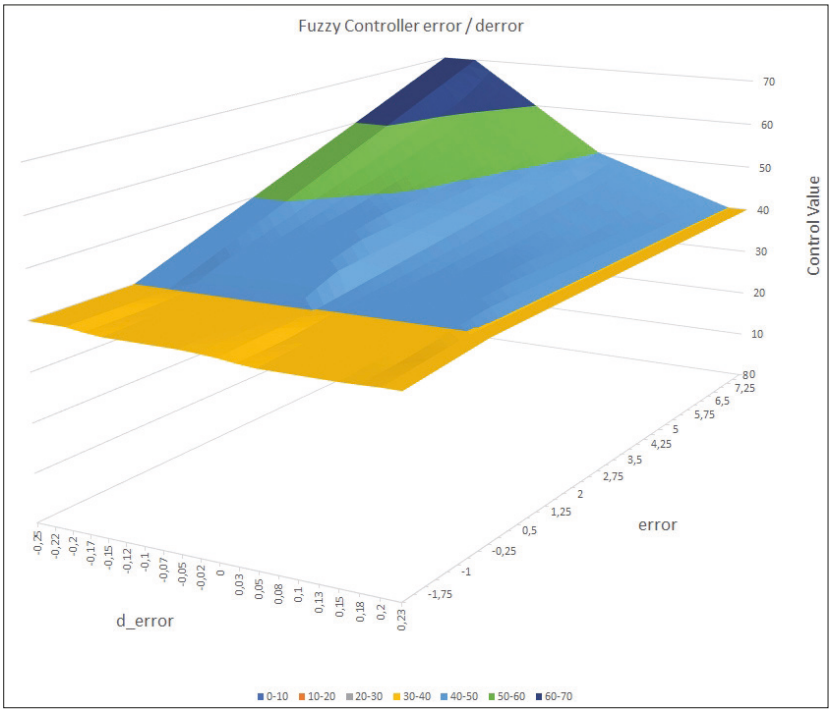


Figure 8.4 3D Chart Fuzzy Controller

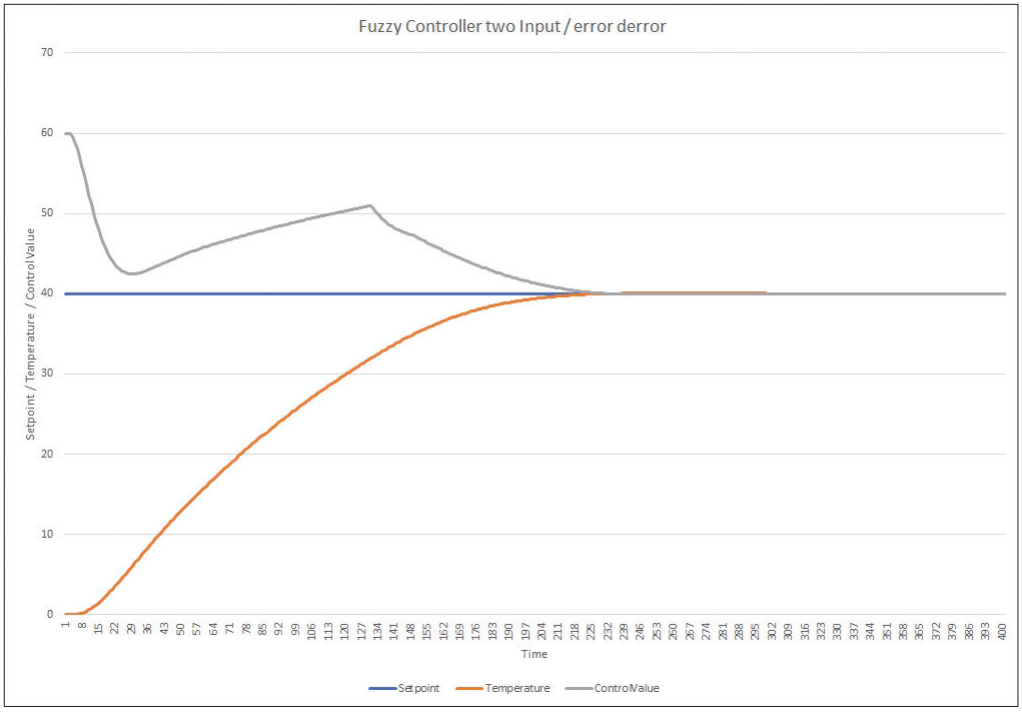


Figure 8.5: Fuzzy Regler Simulation

8.2 Fuzzy control example for a heating control system

Another example is the recording of two temperatures for a fuzzy heating controller. In this example, the outdoor temperature and the room temperature are recorded. In standard heating controllers, the flow temperature is determined using a table or an equation.

The flow temperature is calculated using the following equation:

VT_Soll = RT_Soll + Level – (Slope * DAR * (1.4347 + 0.021 * DAR + 247.9 * 10-6 * DAR^2))

Listing 8.2: Codeblocks C Flow temperatureHeating

```
/*

VT_Soll = RT_Soll + Level – (Slope * DAR * (1.4347 + 0.021 * DAR + 247.9 * 10-6 *
DAR^2))

VTSoll is Flow temperature setpoint
RTSoll is Room setpoint
DAR is Mixed outside temperature - RTSoll
*/
```

```
#include <stdio.h>
#include <math.h>

// Temperature Bad Abbach 7.2.2024  17 o'clock to 15 o'clock every 2 hours
float temperature_profile[] = {8, 6, 5, 5, 3, 2, 1, 4, 7, 8, 8, 9};

double Calculate_Flow_Temperature(double RTSoll, double DAR, double Level, double
Slope)
{
    return RTSoll + Level - (Slope * DAR * (1.4347 + 0.021 * DAR + 247.9e-6 * DAR
* DAR));
}

int main() {
    double RTSoll = 22.0;           // Room setpoint
    double OutsideTemperature = 0;  // Outside temperature
    double DAR =0;                  // Mixed outside temperature - Room setpoint
    double Level = 5.0;             // Level
    double Slope = 1.2;             // Slope

    printf("\nTRoom= 22 K\n");
    for (int i = 0; i < 12; i++)
    {
        OutsideTemperature = temperature_profile[i];
        DAR = OutsideTemperature - RTSoll;
        double VTSoll = Calculate_Flow_Temperature(RTSoll, DAR, Level, Slope);
        printf("Outside Temperature: %.2f Flow Temperature:
%.2f\n",OutsideTemperature, VTSoll);

    }

    printf("\nTRoom= 18 K\n");
    RTSoll = 18.0;
    for (int i = 0; i < 12; i++)
    {
        OutsideTemperature = temperature_profile[i];
        DAR = OutsideTemperature - RTSoll;
        double VTSoll = Calculate_Flow_Temperature(RTSoll, DAR, Level, Slope);
        printf("Outside Temperature: %.2f Flow Temperature:
%.2f\n",OutsideTemperature, VTSoll);

    }
    return 0;
}
```

In the following example, the OutdoorTemp and IndoorTemp membership functions are used for the fuzzy controller. The rules for the heating are as follows.

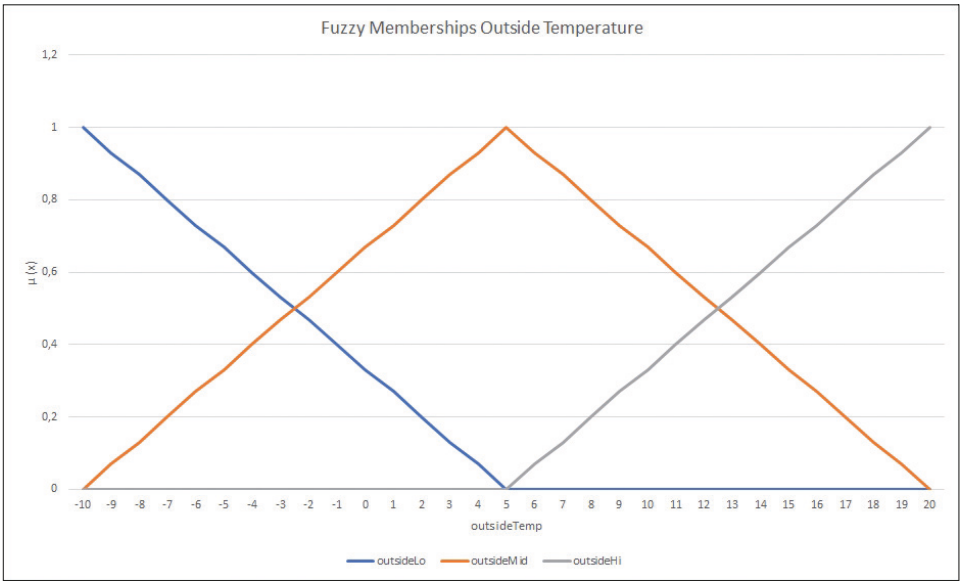


Figure 8.6: Fuzzy membership outdoor temperature

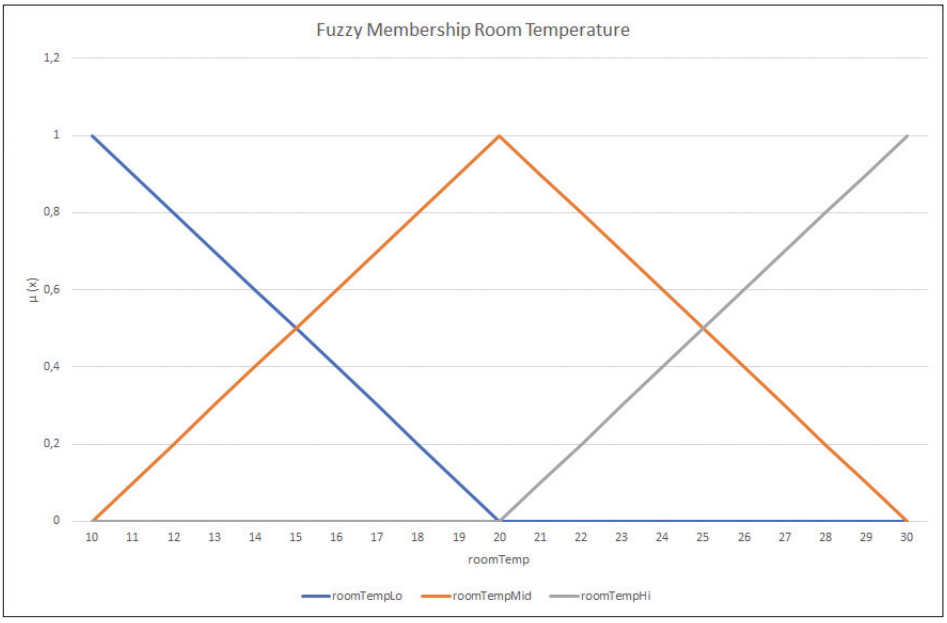


Figure 8.7: Fuzzy membership room temperature

Outdoor	Room	Radiators
cold	cool	full
cold	comfortable	full
cold	warm	half
comfortable	cool	half
comfortable	comfortable	half
comfortable	warm	off
hot	cool	off
hot	comfortable	off
hot	warm	off

The control value for the flow temperature is calculated here from the two input variables.

As in the previous programme, there is a 3D chart for displaying the control value from the two input variables.

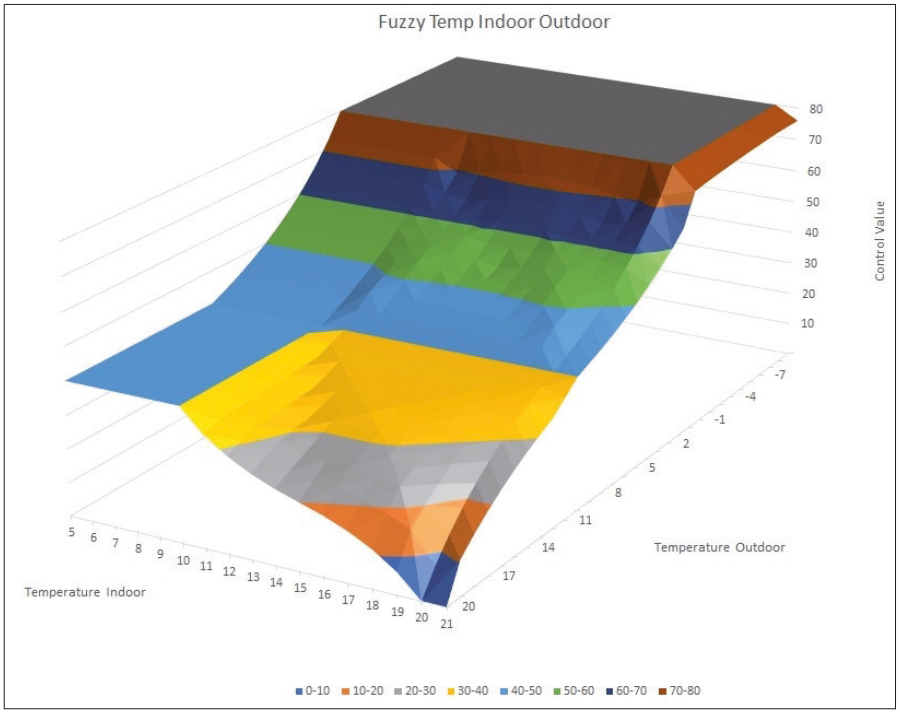


Figure 8.8: 3D chart fuzzy controller

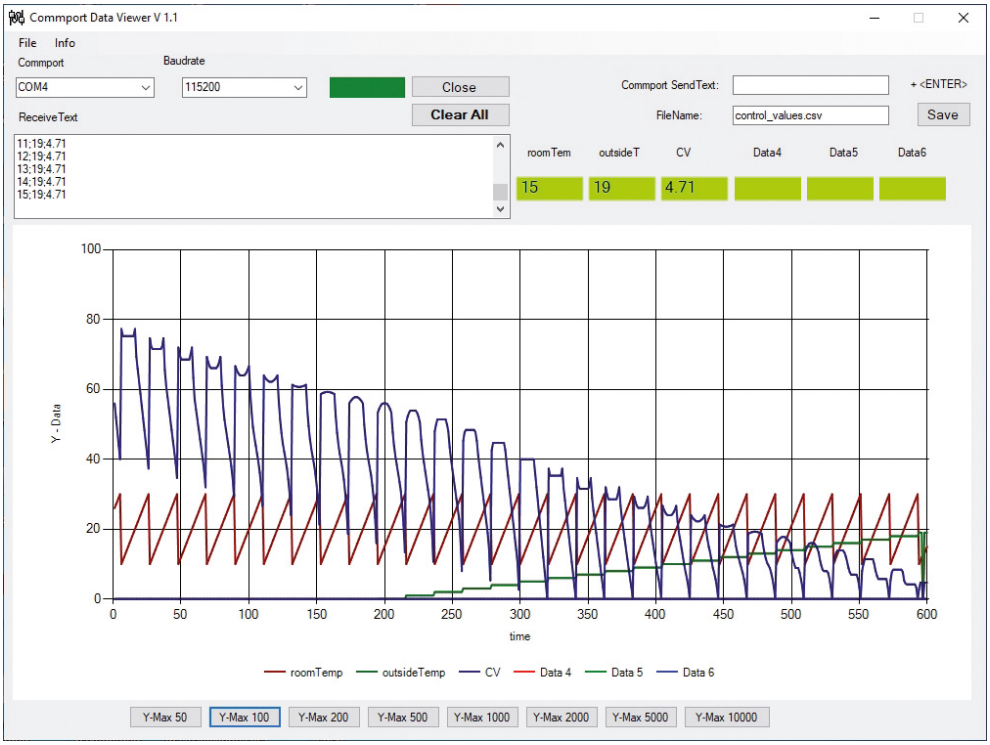


Figure 8.9: Arduino controller simulation

Listing 8.3: Arduino_PT2_Fuzzy_Triangle_IndoorOutdoor.ino

```
//---------------------------------------------------------------------
//
//            Arduino PT2 Fuzzy Triangle Controller
//            =====================================
//
//      Control Testprogramm for Arduino Boards
//      Fuzzy Triangle Controller Test
//      Two Input Fuzzy Control with Indoor Temp and Outdoor Temp
//
//  Date:     23/07/2024
//  Author:   Josef Bernhardt
//  Hardware: Arduino (ESPRESSIF) ESP32 and Arduino Nano
//
//  File:     Arduino_PT2_Fuzzy_Triangle_InddorOutdoor.ino
//---------------------------------------------------------------------
// Include Arduino Library
#include <Arduino.h>

// Pin-Definition
```

```
#define LED_BUILTIN 2
// LED1
#define LED1 5

// Output
#define FULL 80.0
#define HALF 40.0
#define OFF   0.0

// Fuzzification function - Left open
float fuzzyLeftOpen(float x, float a, float b)
{
    if (x <= a)
       {
    return 1.0;
  }
       else if (x <= b)
       {
     return (b - x) / (b - a);
  }
       else
       {
    return 0.0;
  }
}

// Fuzzification function - Triangle
float fuzzyTriangle(float x, float a, float b, float c)
{
    if (x <= a || x >= c)
    {
      return 0.0;
    }
    else if (x <= b)
    {
      return (x - a) / (b - a);
    } else
    {
      return (c - x) / (c - b);
    }
}

// Fuzzification function - Right open
float fuzzyRightOpen(float x, float a, float b) {
    if (x <= a)
       {
```

```
    return 0.0;
  }
  else if (x <= b)
      {
    return (x - a) / (b - a);
  }
  else
  {
    return 1.0;
  }
}
```

The FuzzyControl() function implements a fuzzy controller to control the temperature. The controller uses two input values: the room temperature and the outside temperature. Firstly, these input values are fuzzified by dividing them into three fuzzy sets: low, medium and high. Nine rules are then defined that describe the dependency between the room and outdoor temperatures to determine the control value. Each rule results in a rating (FULL, HALF, OFF), which is combined with the corresponding fuzzy sets. Finally, the result is calculated as a weighted average of the rule ratings. The code also handles the special case of a division by zero and returns the control value that controls the heating.

```
// Function for the rules of the fuzzy controller
float FuzzyControl(float roomTemp, float outsideTemp)
{
    // Fuzzification of internal temperature
    float roomTempLo = fuzzyLeftOpen(roomTemp, 10, 20);
    float roomTempMid = fuzzyTriangle(roomTemp, 10, 20, 30);
    float roomTempHi = fuzzyRightOpen(roomTemp, 20, 30);

    // Fuzzification of outside temperature
    float outsideLo = fuzzyLeftOpen(outsideTemp, -10, 5);
    float outsideMid = fuzzyTriangle(outsideTemp, -10, 5, 20);
    float outsideHi = fuzzyRightOpen(outsideTemp, 5, 20);

    // Define rules
    float rule1 = fmin(outsideLo, roomTempLo);   // FULL
    // Define rules
    float rule2 = fmin(outsideLo, roomTempMid);  // FULL
    // Define rules
    float rule3 = fmin(outsideLo, roomTempHi);   // HALF
    // Define rules
    float rule4 = fmin(outsideMid, roomTempLo);  // HALF
    // Define rules
    float rule5 = fmin(outsideMid, roomTempMid); // HALF
    // Define rules
    float rule6 = fmin(outsideMid, roomTempHi);  // OFF
```

```
    // Define rules
    float rule7 = fmin(outsideHi, roomTempLo);   // OFF
    // Define rules
    float rule8 = fmin(outsideHi, roomTempMid);  // OFF
    // Define rules
    float rule9 = fmin(outsideHi, roomTempHi);   // OFF

       // Calculation of the weighted average of the rules
    float numerator = (rule1 * FULL) + (rule2 * FULL) + (rule3 * HALF) + (rule4 *
HALF) + (rule5 * HALF) + (rule6 * OFF) + (rule7 * OFF) + (rule8 * OFF) + (rule9 *
OFF);
    float denominator = rule1 + rule2 + rule3 + rule4 + rule5 + rule6 + rule7 +
rule8 + rule9;

    float result; //  = numerator / denominator;

    if (denominator != 0)
    {
        result = numerator / denominator;
    }
    else
    {
      // Handle division by zero case
      // Set a default value or handle it as per your requirements
      result = 0;
    }
    return result;
}


// Init Hardware
void setup()
{
  pinMode(LED_BUILTIN, OUTPUT);   // Configure pin as output
  pinMode(LED1, OUTPUT);          // Configure pin as output
  Serial.begin(115200);           // Init UART USB
}

void loop()
{
  // Place your main code here to execute repeatedly:
  float roomTemp = 0;
  float outsideTemp = 0;

  // toggle LED
  digitalWrite(LED_BUILTIN, !digitalRead(LED_BUILTIN));
```

```
  // wait 200 ms
  delay(2000);


  for (outsideTemp = -10; outsideTemp < 21; outsideTemp += 1.0)
  {
    for (roomTemp = 10; roomTemp < 31; roomTemp += 1.0)
    {
      float control_value = FuzzyControl(roomTemp, outsideTemp);
      Serial.print("roomTemp:");
      Serial.print(roomTemp, 1);        // Setpoint with one decimal place
      Serial.print(" outsideTemp:");
      Serial.print(outsideTemp, 1);     // Temperature with one decimal place
      Serial.print(" CV:");
      Serial.print(control_value, 2);   // Control signal with one decimal place
      Serial.println();                 // New line for the next output
      delay(100);
      digitalWrite(LED1, 0);
      delay(100);
    }
  }
}
```

There are many other programme examples with two input variables, such as temperature and pressure for industrial gas heating systems.

The inverted pendulum with angular deviation and rate of change of the deviation.

Chapter 9 • Fuzzy controller examples with a DC motor

9.1 DC motor basics

DC motors with pulse output are widely used as actuators in robotics and industrial control systems. These motors enable precise and dynamic motion control by using digital pulses to regulate the speed. In robotics, DC motors are often used to drive joints and grippers, where they make it possible to perform movements with high precision and speed.

Industrial control systems also utilize this technology for conveyor belts, automatic door systems and other applications where fast and reliable positioning is required. By precisely detecting pulses, these motors can be customized to the exact requirements of a particular task, making them ideal for applications that require both accuracy and efficiency.

In this book, these motors are used for speed control as well as position control.



Figure 9.1: DC motor with Hall sensor

Technical data of the motor:

Description	Value
Voltage	12 V
Manufacturer	Hililand1so9xuaf28-05 SKU-GS20205 JGA25-371 100RPM
No-load speed	10000 1/min
Gear ratio	1:50
Pulses per revolution	11
Pulses per revolution at the shaft	550
Torque	16 kG/cm
Length of connection cable	20 cm
Length of drive shaft	15 mm

Diameter of drive shaft	5 mm
Dimensions (without shaft)	40 * 36 * 125 mm 1,57 * 1,41 * 4,92 Zoll
Weight	100 g

Connecting cables:

Description	Color
Motor positive power supply (+)	Red
Motor negative power supply (-)	White
Encoder positive power supply (+)	Blue
Encoder negative power supply (-)	Black
Pulse 1	Yellow (11 pulses per revolution)
Pulse 2	Green (11 pulses per revolution)

Hall sensors measure the speed of a motor by detecting magnetic fields. These sensors consist of a thin semiconductor strip that is located in the magnetic field of the magnet on the motor shaft. When the magnet moves past the sensor, the Hall effect in the semiconductor generates an electrical pulse that is proportional to the strength of the magnetic field. Hall sensors are often installed in the motor together with permanent magnets near the rotor shaft, which rotate with the shaft. The sensors register the change in the magnetic field when the rotor shaft and the magnets attached to it pass by and convert these changes into electrical pulses. These pulses are used to measure the speed of the motor and determine the direction of rotation.



Figure 9.2: Hall sensors and connections

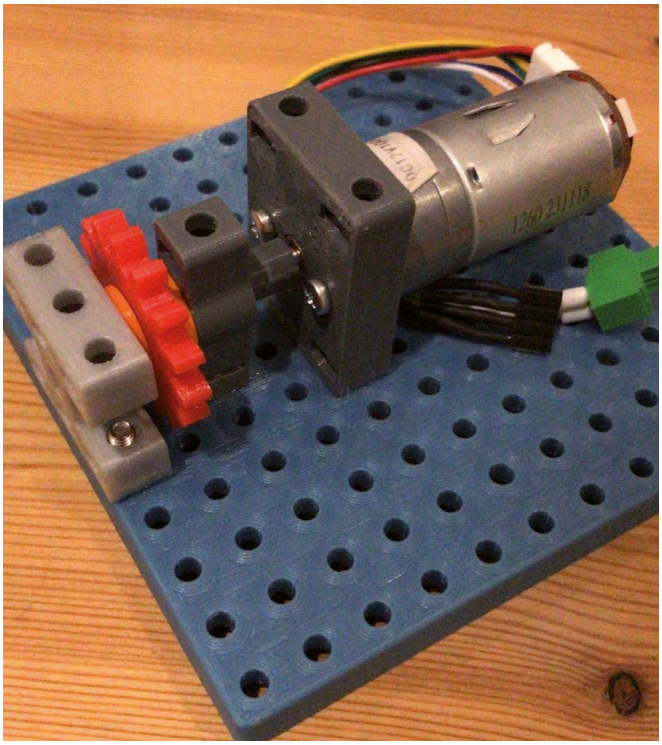


Figure 9.3: Test setup of the DC motor

9.2 Control of the motor with PWM (speed control)

Listing 9.1: Arduino_Motor_PWM_ADC.ino

In this first example with the DC motor, it is about an initial test of the DC motor. This is not a controller, but a simple control with a PWM signal. The PWM output of the ESP32 controls the BSP 76 with one of the two PWM outputs.

Initialize the hardware:

```
// ESP32 ADC Pins
#define ADC1 32
#define ADC2 33

// PWM1 and PWM2 Pins
#define PWM1 12
#define PWM2 13
// use 10 or 12 bit precision for LEDC timer
#define PWM_RESOLUTION 10
// use 1000 Hz as a LEDC base frequency
#define PWM_FREQUENZ 1000
```

In setup():

```
  pinMode(ADC1, INPUT);
  pinMode(ADC2, INPUT);

  Nextion_Send_Text_to_Textbox("t1","Voltage 1");
  Nextion_Send_Text_to_Textbox("t2"," PWM Value 1 ");
  Nextion_Send_Text_to_Textbox("t3"," ");
  Nextion_Send_Text_to_Textbox("t4","Voltage 2");
  Nextion_Send_Text_to_Textbox("t5"," PWM Value 2 ");
  Nextion_Send_Text_to_Textbox("t6"," Seconds");

  // Sets up a channel (0-15), a PWM duty cycle frequency, and a PWM resolution
(1 - 16 bits)
  ledcAttach(PWM1, PWM_FREQUENZ, PWM_RESOLUTION);
  ledcAttach(PWM2, PWM_FREQUENZ, PWM_RESOLUTION);
```

The voltage is read out on the ESP32 board via ADC1. The 12-bit value from 0 to 4095 is converted into a PWM value from 0 to 1023 and output.

In loop():

```
  // Read ADC1 value
  int voltage1 = analogRead(ADC1);
  // Read ADC2 value
  int voltage2 = analogRead(ADC2);
  // Interpolate voltage 1
  int scaledValue1 = map(voltage1, 0, 4095, 0, 1023);
  int scaledValue2 = map(voltage2, 0, 4095, 0, 1023);

  ledcWrite(PWM1, scaledValue1);
  ledcWrite(PWM2, scaledValue2);
```

9.3 Motor control with PWM and speed measurement

Listing 9.2: Arduino_Motor_Speed_Measurement.ino

Hardware description:

The magnetic speed measurement of small direct current motors (DC motors) with two signal outputs is typically carried out using a Hall effect sensor or a magnetic encoder system. Here is a general description of the structure and mode of operation:

Components:

1. Magnet: A small magnet is attached to the motor shaft. This magnet generates a magnetic field that is detected by the sensors.

2. Hall effect sensors: Two Hall effect sensors are positioned near the magnet. These sensors detect changes in the magnetic field when the magnet rotates.

3. Signal processing unit: An electronic unit that processes the signals from the Hall effect sensors and calculates the speed of the motor.

How it works

1. Magnetic field change: When the motor shaft rotates, the magnetic field changes in the vicinity of the Hall effect sensors.

2. Signal generation: Each Hall effect sensor generates a digital signal (usually a square wave signal) when it detects a change in the magnetic field. The frequency of these signals is proportional to the speed of the motor.

3. Phase shift: As there are two Hall effect sensors positioned at a certain distance from each other, there is a phase shift between the two signals. This phase shift is used to determine the direction of rotation of the motor.

4. Signal processing: The signals from the two Hall effect sensors are sent to the signal processing unit. This unit counts the number of pulses per time unit (e.g. pulses per second) to calculate the speed. It also analyses the phase shift of the two signals to determine the direction of rotation.

The speed is measured using the two inputs MOT1 and MOT2. The inputs for the two motor sensor lines are configured as interrupt inputs. An interrupt is triggered on a rising edge and the function is called which increments the pulseCount variable.

```
#define PWM1 12
#define PWM2 13

#define MOTSENS1 15
#define MOTSENS2 34

// Motor Pulses
volatile int pulseCount = 0;
volatile bool direction = false;
int lastPulseCount = 0;
int pulseDiff = 0;

// Sense Pins from Motor Encoder
pinMode(MOTSENS1, INPUT_PULLUP);
pinMode(MOTSENS2, INPUT_PULLUP);
// Interrupt für die Motor Impulse
attachInterrupt(digitalPinToInterrupt(MOTSENS1), handleSensorInterrupt1, RISING);
```

The handleSensorInterrupt1() function is called by a positive edge of the MOTSENS1 input pin and runs parallel to the main programme. It determines the direction of rotation of the motor by checking the state of the MOTSENS2 pin in comparison to MOTSENS1: If the state of MOTSENS2 is different to that of MOTSENS1, the direction of rotation is set as 1 (clockwise); otherwise, it is interpreted as -1 (anti-clockwise). The function also increments the pulseCount variable to count the number of pulses measured. These pulses are used to calculate the speed or position of the motor. As this function is triggered by an interrupt signal, it is marked as IRAM_ATTR to ensure that it is executed in fast RAM and can therefore react quickly to interrupts.

```
// This function is called when the state of the input pin MOTSENS1 changes.
// It is triggered by an interrupt and runs concurrently with the main program
void IRAM_ATTR handleSensorInterrupt1() {
  // Determine rotation direction based on the state of MOTSENS2.
  direction = (digitalRead(MOTSENS2) != digitalRead(MOTSENS1)) ? 1 : -1;
  // Increase pulse count.
  pulseCount++;
}
```

In the loop() function, the speed of the motor is calculated every 100 milliseconds and output serially. If the flag flag100ms is set, it is reset to false and the difference in the number of pulses since the last cycle (pulseDiff) is calculated by subtracting the current pulse count (pulseCount) from the previous count (lastPulseCount). This difference is used to calculate the speed of the motor in revolutions per minute (RPM), whereby the conversion is based on the pulse count and the time. The calculated control value is then output to the motor via the PWM signal to set the speed. In addition, the current setpoint, the calculated speed (rpm) and the control value are output on the serial interface and an LED is switched to visually indicate the status of the loop.

```
  // Control Loop 100ms rpm calulating
  if (flag100ms)
  {
      flag100ms=false;
      // Pulse count difference since the last cycle
      pulseDiff = pulseCount - lastPulseCount;
      // Update the variables for the next cycle
      lastPulseCount = pulseCount;
      // pulseDiff = movingAverageFilter(pulseDiff);
      // Motor Speed in revolutions per minute (RPM)
      rpm = (pulseDiff * 600) / 11;

      ledcWrite(PWM2, controlvalue);

      Serial.print("SP:");  Serial.print(setpoint);
      Serial.print(" RPM:");Serial.print(rpm);
      Serial.print(" CV:"); Serial.println(controlvalue);  // 0.. 1023
```

```
    digitalWrite(LED1,!digitalRead(LED1));
}
```

Output to the Nextion display takes place every second.

```
// Update Nextion Display
if (flag1sec)
{
    flag1sec=false;

    Nextion_Send_Text_to_Textbox("txtT1",String(pulseDiff));
    Nextion_Send_Text_to_Textbox("txtT2",String(rpm));
    Nextion_Send_Text_to_Textbox("txtT3",String(controlvalue));
    Nextion_Send_Text_to_Textbox("txtT4",String(adcValue2));
    Nextion_Send_Text_to_Textbox("txtT5",String(setpoint));
    Nextion_Send_Text_to_Textbox("txtT6",String(pulseCount));

    digitalWrite(LED_BUILTIN,!digitalRead(LED_BUILTIN));
}
```

The table with the control value from 0 to 1023 and the speed. You can see immediately that the motor only starts to rotate from a PWM value of approx. 200.

Control Value	Rotational Speed
0	0
100	0
150	0
200	763
250	1800
300	2509
400	3545
500	4200
600	4520
700	4850
800	5018
900	5236
1023	5618

Table: PWM - Speed values

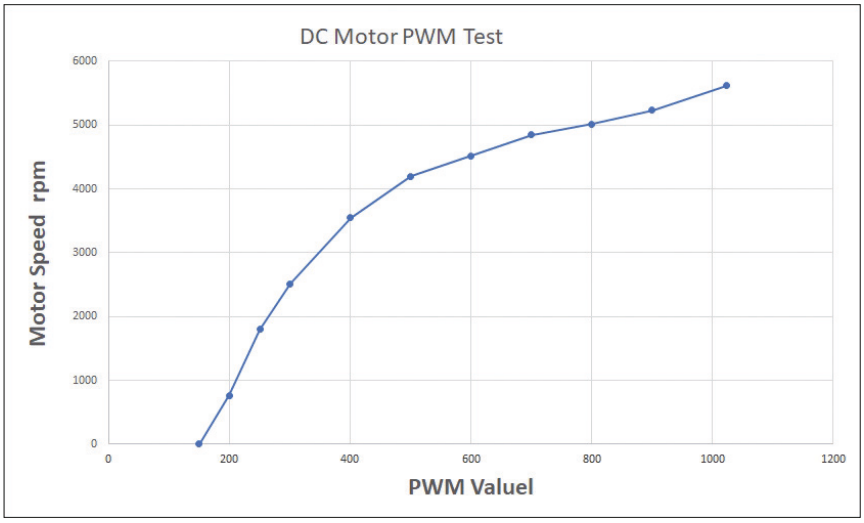


Figure 9.4: Motor characteristic curve of the DC motor

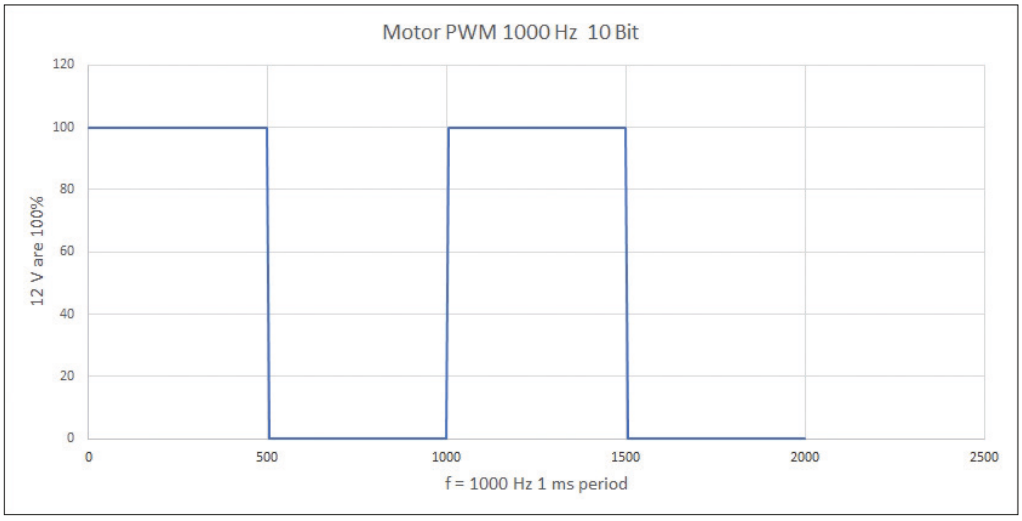


Figure 9.5: PWM output

The next section describes the precise speed control of the motor.

9.4 Motor speed control with Fuzzy Logic

The control software for the direct current motor (DC motor) is based on the Fuzzy Logic principle, which enables precise and smooth adjustment of the motor speed to the desired setpoint. This control is particularly useful for applications where a constant and stable speed is required, even under varying load conditions.

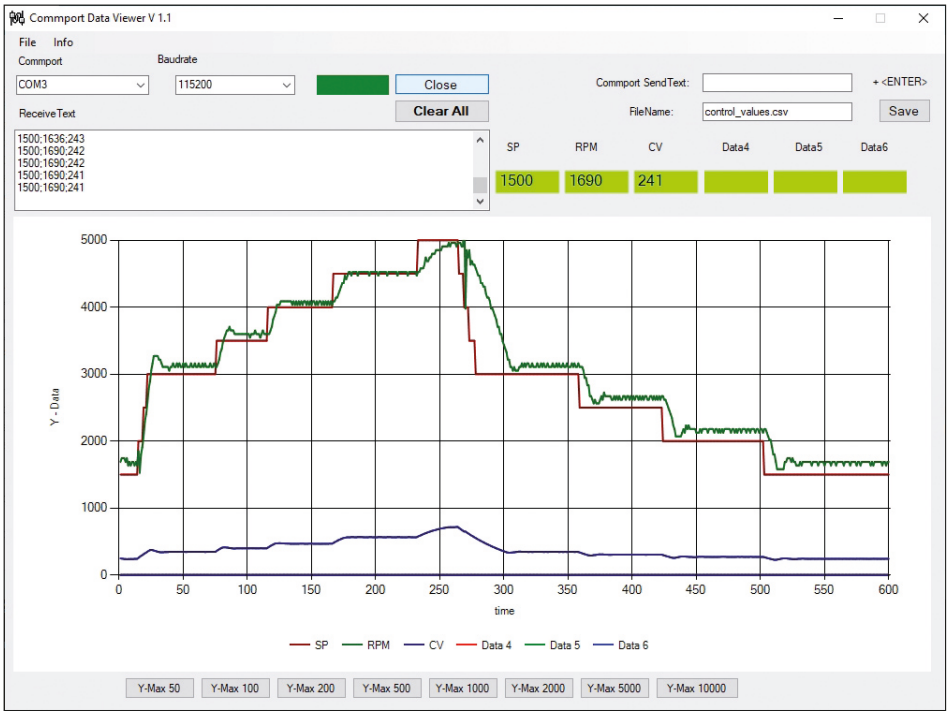


Figure 9.6: Speed curve of the motor control

How the fuzzy logic controller works:

The following method uses the center of gravity method for defuzzification. During defuzzification, a weighted sum of the membership values of the linguistic variables is calculated. The weighting is done by the values of the membership functions and the corresponding output values. Here is an explanation of how this procedure works:

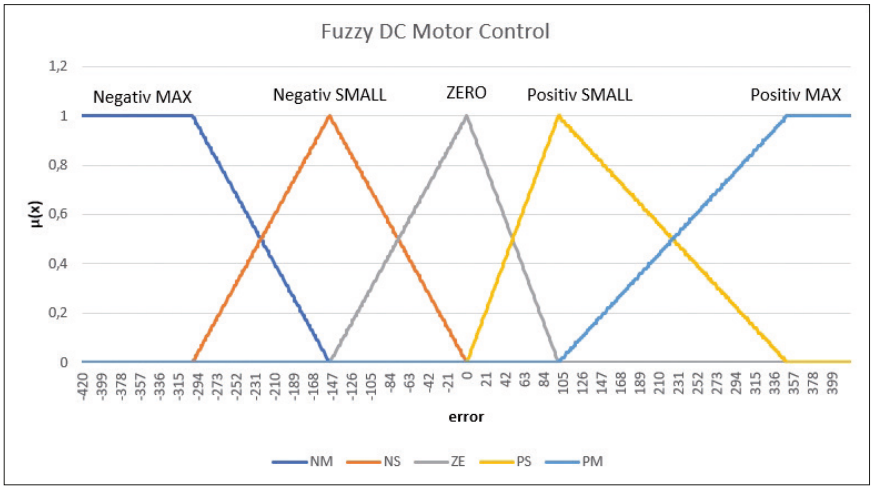


Figure 9.7: Fuzzy memberships of the motor controller

1. First, the degrees of membership for the linguistic variables (left, center, right) are calculated by applying the corresponding fuzzy functions to the error (e).

2. Then the weighted sums of these degrees of membership are calculated. Each sum is multiplied by the corresponding output value (U[x]).

3. The weighted sums are added to obtain the numerator for the mean.

4. The sum of the degrees of membership is used in the denominator.

5. Finally, the mean (y) is calculated by dividing the numerator by the denominator.

The mean value serves as the final control signal. Here is an overview of the calculation:

- Numerator: left * U[0] + center * U[1] + right * U[2]
- Denominator: left + center + right
- Average value: numerator / denominator

This method calculates the control signal (y) based on the error signal (e) using fuzzy logic.

The speed is set here using the BTN1 and BTN2 buttons. BTN1 for increasing the setpoint and BTN2 for decreasing the setpoint. The value is set to 0 by pressing both buttons.

Listing 9.3: Arduino_Motor_Speed_Control.ino

```
//------------------------------------------------------------------
//
//            Arduino ESP32 DC Motor Example with PWM
//            =======================================
//
//      Motor Control Software for Arduino ESP32 PID Fuzzy Control Board
//
//      Setpoint with POT 2 Value
//
//      Read Pulses per 100 ms and RPM
//
//      Outputting PWM2 to Motor // Getriebe 1 : 50
//
//  Date:     25/5/2024 11.03
//  Author:   Josef Bernhardt
//  Hardware: Arduino (ESPRESSIF) ESP32
//
//  File:     Arduino_Motor_Speed_Control.ino
//------------------------------------------------------------------
#include <Arduino.h>
```

```
#include <Ticker.h>

// ESP32 UART 2 Pins
#define RXD2 16
#define TXD2 17

// ESP32 ADC Pins
#define ADC1 32
#define ADC2 33

#define LED_BUILTIN 2

#define RELAIS1 18
#define RELAIS2 19

#define PWM1 12
#define PWM2 13

#define MOTSENS1 15
#define MOTSENS2 34

// Buttons
#define BTN1 14
#define BTN2 27

// LED1
#define LED1 5
/*
float Kp = 0.24;
float Ki = 1.2;
float Kd = 0.012;
*/
/*
float Kp = 2.6;
float Ki = 0.0;
float Kd = 0.00;

float Kp = 0.0;
float Ki = 0.0;
float Kd = 0.0;
*/

float Kp = 0.34;
float Ki = 1.2;
float Kd = 0.012;
```

```
// use 10 or 12 bit precision for LEDC timer
#define PWM_RESOLUTION 10
// use 1000 Hz as a LEDC base frequency
#define PWM_FREQUENZ 1000

// Define the duration between timer calls (in milliseconds)
const unsigned long timerInterval = 10;  // 10 ms Intervall
volatile int counter1s = 0;
volatile bool flag1sec = false;

volatile int counter100ms = 0;
volatile bool flag100ms = false;

// Motor Pulses
volatile int pulseCount = 0;
int lastPulseCount = 0;
int pulseDiff = 0;

int setpoint = 0;

#define MIN(a, b) ((a) > (b) ? (b) : (a))
#define MAX(a, b) ((a) > (b) ? (a) : (b))

volatile int direction = 0;  // 0 for stop, 1 for forwards, -1 for backwards

// Ticker object for the timer function
Ticker timer;

// These arrays are used to represent the degrees of membership
// of an input value in different regions of a fuzzy set.

float left[] = { -300, -150 };           // leftopen
float middle1[] = { -300, -150, 0 };     // negativ small
float middle2[] = { -150, 0, 100 };      // zero
float middle3[] = { 0, 100, 350 };       // positiv small
float right[] = { 100, 350 };            // right open

float U[] = { 0, 300, 600, 900, 1023 };  // Adjustment of the output values
```

This function calculates the degree of membership of a given input value x in a left-open triangular membership function. The left open membership function returns a value of 1.0 if the input value is less than the value a. For values greater than or equal to this value, the value is calculated according to the standard form of the triangular membership function.

```
float fuzzyLeftOpen(float x, float a, float b)
{
  if (x <= a)
  {
    return 1.0;
  }
  else if (x <= b)
  {
    return (b - x) / (b - a);
  }
  else
  {
    return 0.0;
  }
}
```

This function calculates the degree of membership of a given input value x in a standard triangular membership function. The standard triangular membership function is continuous and returns a value based on the input value.

If x is less than or equal to a or greater than or equal to c, the function returns a membership degree of 0.0.

If x is between a and b, the degree of membership increases linearly from 0.0 to 1.0.

If x is between b and c, the degree of membership decreases linearly from 1.0 to 0.0.

```
float fuzzyTriangle(float x, float a, float b, float c)
{
  if (x <= a || x >= c)
  {
    return 0.0;
  }
  else if (x <= b)
  {
    return (x - a) / (b - a);
  }
  else
  {
    return (c - x) / (c - b);
  }
}
```

This function calculates the degree of membership of a given input value `x` in a right open triangle membership function. The right open triangular membership function always returns 1.0 for input values greater than the mean.

If x is less than or equal to a, the function returns a membership degree of 0.0.

If x is between a and b, the degree of membership increases linearly from 0.0 to 1.0.

If x is greater than b, the degree of membership remains constant at 1.0.

```
float fuzzyRightOpen(float x, float a, float b)
{
  if (x <= a)
  {
    return 0.0;
  }
  else if (x <= b)
  {
    return (x - a) / (b - a);
  }
  else
  {
    return 1.0;
  }
}
```

The exponential moving average (EMA) is a filter that calculates a weighted average of current and previous values, whereby the more recent values are weighted more heavily. In contrast to the simple moving average (SMA), where all values are weighted equally, the EMA takes more recent data into account exponentially. The damping factor controls how strongly the current value influences the smoothed value: a small damping factor leads to a slower adjustment, while a large damping factor enables a faster reaction. The previous value is saved statically to ensure continuous smoothing over several function calls. The function finally returns the damped value, which is a mixture of the current and the previous value.

```
float applyDamping(float currentValue)
{
  // Variable declarations
  float result;
  float dampingFactor = 0.02;     // Damping factor
  static float previousValue = 0; // Previous value, stored statically
  // Calculation of the damped value
  result = previousValue * (1.0 - dampingFactor) + currentValue * dampingFactor;
  // Update the previous value
  previousValue = result;
  // Return the damped value
  return result;
}
```

The FuzzyControl(float w, float x) function uses the center of gravity method for defuzzification to calculate a control signal based on the control deviation. Firstly, the error e is calculated as the difference between the setpoint w and the actual value x.

Then the degrees of membership are calculated for different fuzzy sets representing different regions of the error range. These degrees of membership are used to calculate weighted sums of the control values. The numerator of the calculation is formed by the weighted sum of the degrees of membership multiplied by the corresponding control values, while the denominator is the sum of the degrees of membership.

The control signal is calculated by dividing the numerator by the denominator and ensuring that the value is within the permissible range of 0 to 1023. Finally, the control value is processed and returned by an attenuation function.

```
int FuzzyControl(float w, float x) {
  // Calculate the membership values for the error signal in different regions of
the fuzzy sets.
  float e = w - x;
  float y;                 // Control signal

  float left_degree = fuzzyLeftOpen(e, left[0], left[1]);
// Membership value for the right region
  float middle_degree1 = fuzzyTriangle(e, middle1[0], middle1[1], middle1[2]);
// Membership value for the middle1 region
  float middle_degree2 = fuzzyTriangle(e, middle2[0], middle2[1], middle2[2]);
// Membership value for the middle2 region
  float middle_degree3 = fuzzyTriangle(e, middle3[0], middle3[1], middle3[2]);
// Membership value for the middle3 region
  float right_degree = fuzzyRightOpen(e, right[0], right[1]);
// Membership value for the right region

  // Calculate the weighted sum of membership values to determine the control
signal.
  float numerator = left_degree * U[0] + middle_degree1 * U[1] + middle_degree2
* U[2] + middle_degree3 * U[3] + right_degree * U[4];  // Weighted sum of
membership values
  float denominator = left_degree + middle_degree1 + middle_degree2 + middle_
degree3 + right_degree;                              // Sum of membership values
  // Avoid division by zero by checking if the denominator is zero.
  if (denominator != 0) {
    // Calculate the control signal as the ratio of the weighted sum to the sum
of membership values.
    y = numerator / denominator;
  } else {
    // If the denominator is zero, set the control signal to zero to avoid
undefined behavior.
```

```
    y = 0;
  }
  if (y > 1023) y = 1023;
  if (y < 0) y = 0;
  y = applyDamping(y);
  return (int)y;  // Return the control signal.
}
```

The FuzzyControlMotor() function uses fuzzy logic to control the motor and adjusts the control value (controlvalue) based on the difference between the setpoint and the actual speed (rpm). The function first calculates the difference between the setpoint and the actual speed. With a defined damping factor of 0.5, the setpoint is gently adjusted if the setpoint changes significantly to avoid sudden changes. The function then calculates the degrees of membership for small and large errors to determine how strong the adjustment of the control value should be. The adjustment is made based on these degrees of membership and a maximum adjustment value. The calculated control value is limited to a valid range of 0 to 1023 to ensure that it remains within the acceptable range. Finally, the current setpoint is saved to be used as a reference in the next cycle.

```
int FuzzyControlMotor(int setpoint, int rpm)
{
  static int controlvalue = 0;
  static int previous_setpoint = 0;
  int error = setpoint - rpm;
  // Damping factor for smooth adjustment of the setpoint
  float dampingFactor = 0.5;
  // Fuzzy Logic Parameter
  float maxError = 1000.0;
  float maxAdjustment = 30.0;
  // Calculation of the degrees of membership
  float smallErrorMembership = fmin(1.0, fabs(error) / 100.0);
  float largeErrorMembership = fmin(1.0, (fabs(error) - 100.0) / (maxError
- 100.0));
  // Aggregation of the rule base
  float adjustment = smallErrorMembership * maxAdjustment * 0.02 +
largeErrorMembership * maxAdjustment;
  if (error < 0) {
    adjustment = -adjustment;
  }
  // Smooth adjustment of the setpoint
  int setpointChange = setpoint - previous_setpoint;
  if (abs(setpointChange) > 10) {
    setpoint = previous_setpoint + (setpointChange * dampingFactor);
  }
  // Adjustment of the control value
  controlvalue += (int)adjustment;
```

```
  // Limiting the control value to the valid range
  if (controlvalue > 1023) {
    controlvalue = 1023;
  } else if (controlvalue < 0) {
    controlvalue = 0;
  }
  // Saving the current setpoint
  previous_setpoint = setpoint;
  return controlvalue;
}

// Timer-Funktion
void timerCallback() {
  // Timer Period is 10ms
  counter1s++;
  if (counter1s >= 100) {
    flag1sec = true;
    counter1s = 0;
  }
  counter100ms++;
  if (counter100ms == 10) {
    flag100ms = true;
    counter100ms = 0;
  }
}

void IRAM_ATTR handleSensorInterrupt1() {
  // Determine rotation direction based on the state of MOTSENS2.
  direction = (digitalRead(MOTSENS2) != digitalRead(MOTSENS1)) ? 1 : -1;
  // Increase pulse count.
  pulseCount++;
}

// Send Values ti Nextion Display
void Nextion_Send_Text_to_Textbox(char *textbox, String sfloat) {
  Serial2.write(textbox);
  Serial2.write(".txt=\"");
  Serial2.print(sfloat);
  Serial2.write("\"");
  Serial2.write(0xFF);
  Serial2.write(0xFF);
  Serial2.write(0xFF);
}

unsigned char read_btn1() {
  unsigned char key = 0;
```

```
  unsigned char btn;
  static unsigned char btn_old = 0;  // Set initial value to 0 since the default
state of the button is not pressed (logically inverted)
  btn = digitalRead(BTN1);           // Using digitalRead to read the state of
the button
                                     // Start of button check Detect rising edge
  if (btn && (!btn_old))             // If the current state of the button is
HIGH and the previous state was LOW
  {
    // Debouncing
    delay(20);
    // Read again
    btn = digitalRead(BTN1);
    // If still HIGH, return 1
    if (btn) {
      key = 1;
    }
  }
  // Remember old value
  btn_old = btn;
  return key;
}

unsigned char read_btn2() {
  unsigned char key = 0;
  unsigned char btn;
  static unsigned char btn_old = 0;  // Set initial value to 0 since the default
state of the button is not pressed (logically inverted)
  btn = digitalRead(BTN2);           // Using digitalRead to read the state of
the button
                                     // Start of button check Detect rising edge
  if (btn && (!btn_old))             // If the current state of the button is
HIGH and the previous state was LOW
  {
    // Debouncing
    delay(20);
    // Read again
    btn = digitalRead(BTN2);
    // If still HIGH, return 1
    if (btn) {
      key = 1;
    }
  }
  // Remember old value
  btn_old = btn;
  return key;
```

```
}

void setup() {
  // USB Port
  Serial.begin(115200);
  // Nextion Display
  Serial2.begin(9600, SERIAL_8N1, RXD2, TXD2);

  pinMode(BTN1, INPUT);
  pinMode(BTN2, INPUT);

  Nextion_Send_Text_to_Textbox("t1", "Pulses");
  Nextion_Send_Text_to_Textbox("t2", "RPM 1/min");
  Nextion_Send_Text_to_Textbox("t3", "ControlValue");
  Nextion_Send_Text_to_Textbox("t4", "ADC2 ");
  Nextion_Send_Text_to_Textbox("t5", "Setpoint ");
  Nextion_Send_Text_to_Textbox("t6", " ");

  // Setup timer and attach timer to a led pin
  ledcAttach(PWM1, PWM_FREQUENZ, PWM_RESOLUTION);
  ledcAttach(PWM2, PWM_FREQUENZ, PWM_RESOLUTION);

  // Set the timer function with the desired duration of 10ms
  timer.attach_ms(timerInterval, timerCallback);
  // Blink LED
  pinMode(LED_BUILTIN, OUTPUT);
  pinMode(LED1, OUTPUT);
  // Pins for the motor pulses
  pinMode(MOTSENS1, INPUT_PULLUP);
  pinMode(MOTSENS2, INPUT_PULLUP);
  // Interrupt for motor pulses
  attachInterrupt(digitalPinToInterrupt(MOTSENS1), handleSensorInterrupt1,
RISING);
}

void loop()
{

  int controlvalue = 0;
  int setpointold = 0;
  int rpm = 0;

  int adcValue1 = analogRead(ADC1);  // Read Voltage from GPIO 32
  int adcValue2 = analogRead(ADC2);  // Read Voltage from GPIO 33

  // scale voltage to 0..50
```

```
// setpoint = map(adcValue2, 0, 4095, 0, 50);
// setpoint steps of 100
// setpoint *= 100;

if (read_btn1())
{
  if (setpoint < 5000 ) setpoint +=500;
}

if (read_btn2())
{
  if (setpoint > 500 ) setpoint -=500;
}

if (digitalRead(BTN1) && digitalRead(BTN2))
{
  setpoint = 0;
}

// Control Loop 100ms
if (flag100ms)
{
  flag100ms = false;
  // Pulse count difference since last cycle
  pulseDiff = pulseCount - lastPulseCount;
  // Update the variables for the next cycle
  lastPulseCount = pulseCount;

  // Motor Speed in revolutions per minute (RPM)
  rpm = (pulseDiff * 600) / 11;

  if (setpoint > 50)
  {
    // Standard Fuzzy Control with triangles
    controlvalue = FuzzyControl((float)setpoint,(float)rpm);
    // New Fuzzy Example
    // controlvalue = FuzzyControlMotor(setpoint, rpm);
    ledcWrite(PWM2, controlvalue);
  }
  else
  {
    ledcWrite(PWM2, 0);
  }

  Serial.print("SP:");
  Serial.print(setpoint);
```

```
    Serial.print(" RPM:");
    Serial.print(rpm);
    Serial.print(" CV:");
    Serial.println(controlvalue);  // 0.. 1023
    digitalWrite(LED1, !digitalRead(LED1));
  }

  // Update Nextion Display
  if (flag1sec) {
    flag1sec = false;

    Nextion_Send_Text_to_Textbox("txtT1", String(pulseDiff * 10));
    Nextion_Send_Text_to_Textbox("txtT2", String(rpm));
    Nextion_Send_Text_to_Textbox("txtT3", String(controlvalue));
    Nextion_Send_Text_to_Textbox("txtT4", String(adcValue2));
    Nextion_Send_Text_to_Textbox("txtT5", String(setpoint));
    Nextion_Send_Text_to_Textbox("txtT6", "");
    digitalWrite(LED_BUILTIN, !digitalRead(LED_BUILTIN));
  }
}
```

9.5 Motor position control

A direct current motor (DC motor) can be used for precise position control by using speed pulses as feedback to determine the current position of the motor.

1. Basic principle of position control

Position control aims to bring the rotational position of the motor to a specified setpoint and maintain it there. This is achieved by continuously adjusting the motor movement to correct deviations from the setpoint.

2. Speed pulses as feedback

Encoder: A rotary encoder is attached to the motor shaft and generates pulses when the shaft rotates. These pulses are proportional to the speed of the motor and can be used to calculate the angular position travelled.

Counting the pulses: The number of pulses generated is counted to determine the travelled rotation of the motor. Each pulse corresponds to a certain angle, depending on the resolution of the encoder.

3. Determination of the current position

By integrating the counted pulses over time, the current position of the motor can be determined.

Example: If the encoder generates 1000 pulses per revolution, each pulse represents a rotation of 0.36 degrees. If 500 pulses are counted, the shaft has rotated by 180 degrees.

4th control loop for position control: Setpoint (target position): The desired position to which the motor is to be moved.

Actual value (current position): The current position of the motor, determined by the counted pulses.

Control deviation: The difference between the setpoint and actual value.

Controller: Based on the control deviation, the controller calculates a control signal that is sent to the motor.

5. Motor control
The control signal calculated by the controller is used to adjust the voltage and current supply to the motor. This controls the speed and direction of rotation of the motor.

Example: If the current position falls short of the setpoint, the controller increases the motor power to bring the motor to the desired position more quickly.

6. Feedback loop
The motor movement generates new pulses from the encoder, which are counted and analyzed again.

This continuous process ensures that the motor is always corrected in the direction of the target position.

To summarize, position control of a DC motor with speed pulses enables precise control through continuous feedback and adjustment of the motor movement. The combination of encoder, pulse counting and control loop ensures that the motor reaches and maintains the desired position.

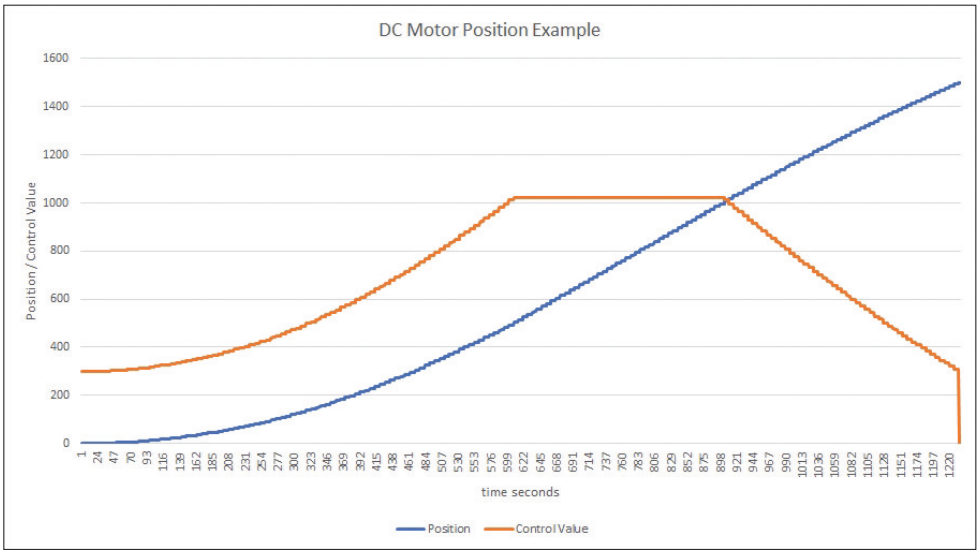


Figure 9.8: DC Motor Positionssteuerung

Description of the software:
The simple function void `void MotorPositionControlSimple(int setpoint)` controls the motor at full power. The void `void MotorPositionControlFuzzy(int setpoint)` function uses a fuzzy triangle to calculate the control value based on the current pulseCount and error.

With these adjustments, the control value for the rise and fall is calculated using an external fuzzy triangle function.

Listing 9.4: Arduino_Motor_Position_Control.ino

```
//-----------------------------------------------------------------------
//
//            Arduino ESP32 DC Motor Example with PWM
//            =======================================
//
//      Control Software for Arduino ESP32 PID Fuzzy Control Board
//
//      Control Motor Position with values from POT 2
//
//      Read Pulses with Motsens1
//
//      Outputting PWM2 to Motor BTN1 is the Start Button
//
//  Date:     15/07/2024
//  Author:   Josef Bernhardt
//  Hardware: Arduino (ESPRESSIF) ESP32
```

```
//
//  File:    Arduino_Motor_Position_Control.ino
//---------------------------------------------------------------------
#include <Arduino.h>
#include <Ticker.h>

// ESP32 UART 2 Pins
#define RXD2 16
#define TXD2 17

// ESP32 ADC Pins
#define ADC1 32
#define ADC2 33

#define LED_BUILTIN  2

#define RELAIS1 18
#define RELAIS2 19

#define PWM1 12
#define PWM2 13

#define MOTSENS1 15
#define MOTSENS2 34

// Buttons
#define BTN1 14
#define BTN2 27
// LED1
#define LED1 5

// use 10 or 12 bit precision for LEDC timer
#define PWM_RESOLUTION 10
// use 1000 Hz as a LEDC base frequency
#define PWM_FREQUENZ 1000

// Define the duration between timer calls (in milliseconds)
const unsigned long timerInterval = 10; // 10 ms Intervall
volatile int counter5s = 0;
volatile bool flag5sec = false;

// Motor Pulses
volatile int pulseCount = 0;

// Ticker-Objekt für die Timer-Funktion
Ticker timer;
```

```
unsigned char read_btn1()
{
    unsigned char key = 0;
    unsigned char btn;
    static unsigned char btn_old = 0; // Set initial value to 0 since the default
state of the button is not pressed (logically inverted)
    btn = digitalRead(BTN1);          // Using digitalRead to read the state of
the button
                                      // Start of button check Detect rising edge
    if (btn && (!btn_old))            // If the current state of the button is
HIGH and the previous state was LOW
    {
        // Debouncing 30ms
        delay(30);
        // Read again
        btn = digitalRead(BTN1);
        // If still HIGH, return 1
        if (btn)
        {
            key = 1;
        }
    }
    // Remember old value
    btn_old = btn;
    return key;
}


// Timer-Funktion
void timerCallback()
{
  // Timer Period is 10ms
  counter5s++;
  if (counter5s >= 500)
  {
    flag5sec = true;
    counter5s = 0;
  }
}

// This function is called when the state of the input pin MOTSENS1 changes.
// It is triggered by an interrupt and runs concurrently with the main program
void IRAM_ATTR handleSensorInterrupt1() {
  // Determine rotation direction based on the state of MOTSENS2.
  // Increase pulse count.
```

```
  if (digitalRead(MOTSENS2)==true) pulseCount++;
}

void MotorPositionControlSimple(int setpoint)
{
      int error;
      int controlvalue = 500;
      digitalWrite(LED1,1);
      while (true)
      {
        error = setpoint - pulseCount;
        ledcWrite(PWM2, controlvalue);
        // Nextion_Send_Text_to_Textbox("txtT6",String(pulseCount));
        if (error <= 0)
        {
          ledcWrite(PWM2, 0);
          Nextion_Send_Text_to_Textbox("txtT6",String(pulseCount));
          Serial.print("Pulse:");  Serial.println(pulseCount);
          digitalWrite(LED1,0);
          return;
        }
      }
}
```

The `fuzzyTriangle()` function calculates an output value based on a triangular fuzzy logic curve. If the input value is less than or equal to `inputMin`, the function returns `outputMin`. If the input value is greater than or equal to `inputMax`, `outputMax` is returned. If the input value is in the interval between `inputMin` and `inputMax`, the function calculates an interpolated output value.

```
// Fuzzy Triangle
int fuzzyTriangle(int input, int inputMin, int inputMax, int outputMin, int
outputMax) {
    if (input <= inputMin) {
        return outputMin;
    } else if (input >= inputMax) {
        return outputMax;
    } else {
        return outputMin + ((input - inputMin) * (outputMax - outputMin)) /
(inputMax - inputMin);
    }
}
```

The `MotorPositionControlFuzzy()` function controls the position of a motor by comparing the setpoint (setpoint) with the current value (pulseCount) and calculating the error. At the beginning, the control value of the motor is set to 300 and an LED is activated. During the

course of control, the motor's setpoint is dynamically adjusted, either by the fuzzyTriangle function for an error range or directly to 1023 if the error is greater than 500. As soon as the error is zero or negative, the motor is stopped, the LED is switched off and the current value is output via an interface. During operation, debug information is output serially to monitor the number of pulses and the actuating value.

```
// Motor Position Control
void MotorPositionControlFuzzy(int setpoint) {
    int error;
    int controlvalue = 300; // Initialer Stellwert
    digitalWrite(LED1, 1);
    while (true) {
        error = setpoint - pulseCount;
        if (pulseCount < 500) {
            // Fuzzy triangle for the initial range (increase from 300 to 1023)
            controlvalue = fuzzyTriangle(pulseCount, 0, 500, 300, 1023);
        } else if (error > 500) {
            controlvalue = 1023;
        } else if (error > 0 && error <= 500) {
            // Fuzzy triangle for the end range (drop from 1023 to 300)
            controlvalue = fuzzyTriangle(error, 0, 500, 300, 1023);
        } else {
            controlvalue = 0;
        }
        // Send control value to the motor
        ledcWrite(PWM2, controlvalue);
        // Debug information (for test purposes)
        Serial.print("Pulse:");Serial.print(pulseCount);
        Serial.print(" CV:");  Serial.println(controlvalue);
        // Nextion_Send_Text_to_Textbox("txtT6", String(pulseCount));
        // If the error is 0 or negative, stop the motor
        if (error <= 0) {
            ledcWrite(PWM2, 0);
            Nextion_Send_Text_to_Textbox("txtT6", String(pulseCount));
            Serial.print("Pulse:");  Serial.println(pulseCount);
            digitalWrite(LED1, 0);
            return;
        }
    }
}
// Send Values to Nextion Display
void Nextion_Send_Text_to_Textbox(char *textbox,String sfloat)
{
        Serial2.write(textbox);
        Serial2.write(".txt=\"");
        Serial2.print(sfloat);
```

```
        Serial2.write("\"");
        Serial2.write(0xFF);
        Serial2.write(0xFF);
        Serial2.write(0xFF);
}

// Init hardware
void setup()
{
  // USB Port
  Serial.begin(115200);
  // Nextion Display
  Serial2.begin(9600, SERIAL_8N1, RXD2, TXD2);

  Nextion_Send_Text_to_Textbox("t1","");
  Nextion_Send_Text_to_Textbox("t2","");
  Nextion_Send_Text_to_Textbox("t3","ControlValue");
  Nextion_Send_Text_to_Textbox("t4","ADC2 ");
  Nextion_Send_Text_to_Textbox("t5","Setpoint ");
  Nextion_Send_Text_to_Textbox("t6","PulseCount");

  // Setup timer and attach timer to a led pin
  ledcAttach(PWM1, PWM_FREQUENZ, PWM_RESOLUTION);
  ledcAttach(PWM2, PWM_FREQUENZ, PWM_RESOLUTION);

  ledcWrite(PWM1, 0);
  ledcWrite(PWM2, 0);

  // Set the timer function with the desired duration in 10 ms steps
  timer.attach_ms(timerInterval, timerCallback);
  // Blink LED
  pinMode(LED_BUILTIN, OUTPUT);
  pinMode(LED1, OUTPUT);
  // Pins for the pulses from the motor
  pinMode(MOTSENS1, INPUT_PULLUP);
  pinMode(MOTSENS2, INPUT_PULLUP);
  pinMode(BTN1, INPUT);
  pinMode(BTN2, INPUT);
  // Set up interrupt function for the motor pulses
  attachInterrupt(digitalPinToInterrupt(MOTSENS1), handleSensorInterrupt1,
CHANGE);

}
```

In the main program, the analogue value of a pin (ADC2) is read continuously within an endless loop and converted into a value from 0 to 100. This value is then scaled to 0 to 5000 by multiplying it by 50 to calculate the setpoint. When a button is pressed (recognized by read_btn1()), a series of actions takes place: The text in a textbox on the Nextion display is deleted, the pulseCount variable is reset to 0 and the `MotorPositionControlExternTriangle()` function is called to move the motor to the calculated position. Several text boxes on the Nextion display are updated with current values, including the analogue input value, the calculated setpoint and a control value. In addition, an LED is regularly switched on and off every 250 milliseconds to provide visual feedback. The code thus ensures that the display and control values are continuously updated and that the motor moves based on the user input.

```
void loop()
{
  int setpoint = 0;
  int controlvalue = 0;
  int error = 0;
  float controlValueFloat = 0;
  int i = 0;

  // int adcValue1 = analogRead(ADC1); // Reading the analogue input value at
GPIO 32
  // int adcValue2 = analogRead(ADC2); // Reading the analogue input value at
GPIO 33

  Nextion_Send_Text_to_Textbox("txtT1","");
  Nextion_Send_Text_to_Textbox("txtT2","");
  Nextion_Send_Text_to_Textbox("txtT3","");
  Nextion_Send_Text_to_Textbox("txtT4","");
  Nextion_Send_Text_to_Textbox("txtT5","");
  Nextion_Send_Text_to_Textbox("txtT6","");

  while(true)
  {

      int adcValue2 = analogRead(ADC2);

      // Scaling of the voltage for the setpoint in steps of 50
      setpoint = (int)map(adcValue2, 0, 4095, 0, 100); setpoint *=50;

      // Starte Move to Position
      if (read_btn1())
      {
          // Clear TextBox
          Nextion_Send_Text_to_Textbox("txtT6","");
          pulseCount = 0;
```

```
          MotorPositionControlExternTriangle(setpoint);
          // MotorPositionControlSimple(setpoint);
      }

      ledcWrite(PWM2, 0);
      Nextion_Send_Text_to_Textbox("txtT1",String(0));
      Nextion_Send_Text_to_Textbox("txtT2",String(0));
      Nextion_Send_Text_to_Textbox("txtT3",String(controlvalue));
      Nextion_Send_Text_to_Textbox("txtT4",String(adcValue2));
      Nextion_Send_Text_to_Textbox("txtT5",String(setpoint));
      // Nextion_Send_Text_to_Textbox("txtT6",String(pulseCount));

      digitalWrite(LED_BUILTIN,!digitalRead(LED_BUILTIN));
      delay(250);

  }
}
```

Chapter 10 • Sliding mode controller

The sliding mode control was developed in the 1950s by Emelyanov and Itkis. This method was further developed and popularized by the work of Vadim Utkin in the 1970s. Vadim Utkin is considered one of the main pioneers in this field and has contributed significantly to the theory and application of the sliding mode controller.

10.1 General description of the sliding mode controller

A sliding mode controller (SMR) is a non-linear control approach that is particularly effective in systems with uncertainties and disturbances. This control method forces the system to remain on a predefined sliding surface, making it robust against model uncertainties and external disturbances.

The basic idea of the sliding mode controller is to use a discontinuous control signal that dynamically guides the system towards the sliding surface and keeps it on it. This achieves a sharp separation between the control phases, enabling a fast and precise response to changes in system behavior.

A key feature of the sliding mode controller is its ability to shape the behavior of a system so that it reaches and remains in the desired state. This is achieved by causing the system to 'slide' on a sliding surface defined by a linear combination of the control error and its derivative.

The robustness of the sliding mode controller against disturbances and uncertainties makes it particularly attractive for applications in the automotive industry, robotics, aerospace and many other areas where reliability and precision are crucial.

Effect of $c1$ and $c2$ on the sliding mode controller:
In a sliding mode controller (SMR), the sliding surface s is a function of the control error e and its derivative $\dot{e}$. In this case, the sliding surface is defined as $s = c1 \cdot e + c2 \cdot \dot{e}$. The constants $c1$ and $c2$ play a decisive role in the design of the control behavior. Here are the main effects of $c1$ and $c2$:

Stability and dynamics of the system:

- $c1$: This coefficient determines the influence of the control error e on the sliding surface. A higher value of $c1$ causes the controller to react more strongly to deviations from the setpoint. This can increase the convergence speed, but can also lead to greater sensitivity to measurement noise.

- $c2$: This coefficient determines the influence of the derivative of the control error $\dot{e}$ on the sliding surface. A higher value of $c2$ increases the response of the controller to rapid changes in the control error. This improves the ability of the controller to dampen rapid disturbances but can also lead to a higher control amplitude and thus to greater stress on the actuator.

Compromise between speed and robustness:
A suitable choice of $c1$ and $c2$ can achieve a compromise between control speed (how quickly the system reacts to disturbances and changes) and robustness (how robust the system is against uncertainties and disturbances). A high value for $c1$ could make the system faster, but less robust against noise. A high value for $c2$ could make the system more robust against disturbances but lead to a stronger oscillation.

Tracking performance:
The weighting of e and $\dot{e}$ by $c1$ and $c2$ influences how well the controller follows the reference signal. A good choice of $c1$ and $c2$ ensures that the controller efficiently follows the desired path without becoming unstable or oscillating.

Robustness against model uncertainties:
In sliding mode controllers, robustness against model uncertainties and external disturbances is one of the main strengths. The right choice of $c1$ and $c2$ can improve the controller's ability to deal with such uncertainties. For example, a higher weighting on $\dot{e}$ (through $c2$) can help the system to react quickly to unforeseen changes and stabilize itself.

The control law: u=-K·sign(s):
The expression u=-K·sign(s) describes the control law of a sliding mode regulator (SMR). To understand this better, we should look at the individual components and their meanings:

Sliding Mode Regulator:
A sliding mode regulator is a special type of regulator that is used to influence the dynamics of a system by forcing the system to stay on a certain sliding surface. This sliding surface is defined by the equation s=$c1$·e+$c2$·$\dot{e}$, where e is the error (the difference between the desired and the actual state) and $\dot{e}$ is the derivative of this error.

Sliding surface s:
The sliding surface s is a linear combination of the error e and its derivative $\dot{e}$. The constants $c1$ and $c2$ determine how heavily each of these two components is weighted.

Control law u = -K · sign(s):
u: This is the control signal that is applied to the system to regulate it.

K: This is a positive gain constant that determines the strength of the control signal.

sign(s): This is the signum function of s, which is defined as follows:

sign(s) = +1 if s > 00 if s = 0-1if s<0

-K·sign(s): This control law means that the control signal u is proportional to the gain K and the sign of s. It will always act in the direction that brings the system back onto the sliding surface.

Mode of operation:
If the system is above the sliding surface ($s>0$), the control signal u becomes negative ($u=-K\cdot(+1)=-K$), which pushes the system down towards the sliding surface.

If the system is below the sliding surface ($s<0$), the control signal u becomes positive ($u=-K\cdot(-1)=+K$), which pushes the system upwards in the direction of the sliding surface.

If the system is exactly on the sliding surface ($s=0$), the control signal u is zero ($u=-K\cdot 0=0$).

Robustness and chattering:
A major advantage of the sliding mode controller is its robustness against model uncertainties and external disturbances, as the control law reacts strongly to the sign of the error and quickly returns the system to the sliding surface.

A disadvantage is the so-called 'chattering', a phenomenon in which the control signal changes very quickly between positive and negative values when the system is close to the sliding surface. This can lead to undesirable vibrations and wear in real systems. To reduce chattering, modified control laws or filters are often used.

To summarize, the control law $u=-K\cdot\text{sign}(s)$ is a simple but effective means of bringing a system to the desired sliding surface and keeping it there, where K determines the strength of the control signal and sign(s) determines the direction of the control.

10.2 Sliding mode controller example with the PT2 system simulation

Listing 10.1: Arduino_PT2_Sliding_Mode_Controller.ino

```
// c1: Influences the effect of the current error on the sliding surface.
// Higher values lead to a faster system response.
// c2: Influences the effect of the error change on the sliding surface.
// Higher values lead to a more aggressive system response.
float slidingModeController(float setpoint, float temperature)
{
    float eta = 100.0;
    float c1 = 0.1;
    float c2 = 1.25; // C2 = 3.5 is OK too
    float control_value = 0;
    float sliding_surface;
    static float previous_error = 0;
    float error = setpoint - temperature;
    // Calculate the sliding surface
    // sliding_surface = lambda * error;
    sliding_surface = c1 * error + c2 * (error - previous_error);
    previous_error = error;
    // Determine the control input based on the sliding surface
    if (sliding_surface > 0)
    {
```

```
        control_value = eta; // Heating
    }
    else if (sliding_surface < 0)
    {
        control_value = -eta; // Cooling
    }
    // no negativ Values
    if (control_value < 0) control_value = 0;
    return control_value;
}
```

The result looks similar to a two-point controller, but you can see that the controller works much more accurately and does not overshoot when raising.

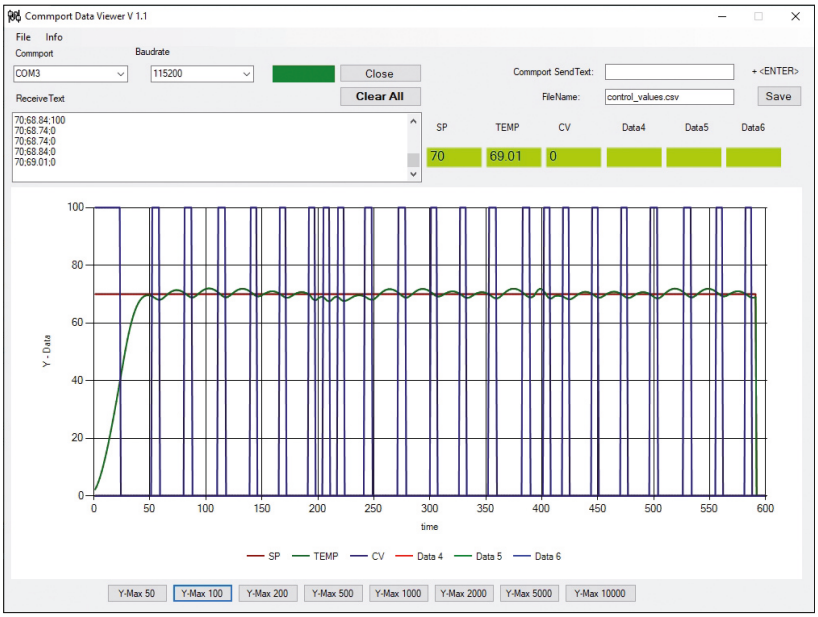


Figure 10.1: Simulation of the SMC controller

If a continuous output variable is required, this version with calculation of the control value using an EXP function is suitable.

```
float slidingModeControllerEXP(float setpoint, float temperature)
{
    float eta = 100.0;
    float c1 = 0.1;
    float c2 = 1.25; // C2 = 3.5 is OK too
    float control_value;
    float sliding_surface;
    static float previous_error = 0;
```

```
    float error = setpoint - temperature;
    // Calculate the sliding surface
    sliding_surface = c1 * error + c2 * (error - previous_error);
    previous_error = error;
    // Determine the control input based on the sliding surface
    // Use a continuous control law instead of a binary one
    // Logistic function for smooth output
    control_value = eta * (1.0 / (1.0 + exp(-sliding_surface)));
    // Ensure control value is within bounds
    if (control_value < 0) control_value = 0;
    if (control_value > 100) control_value = 100;
    return control_value; // Returns pulse duration in milliseconds
}
```

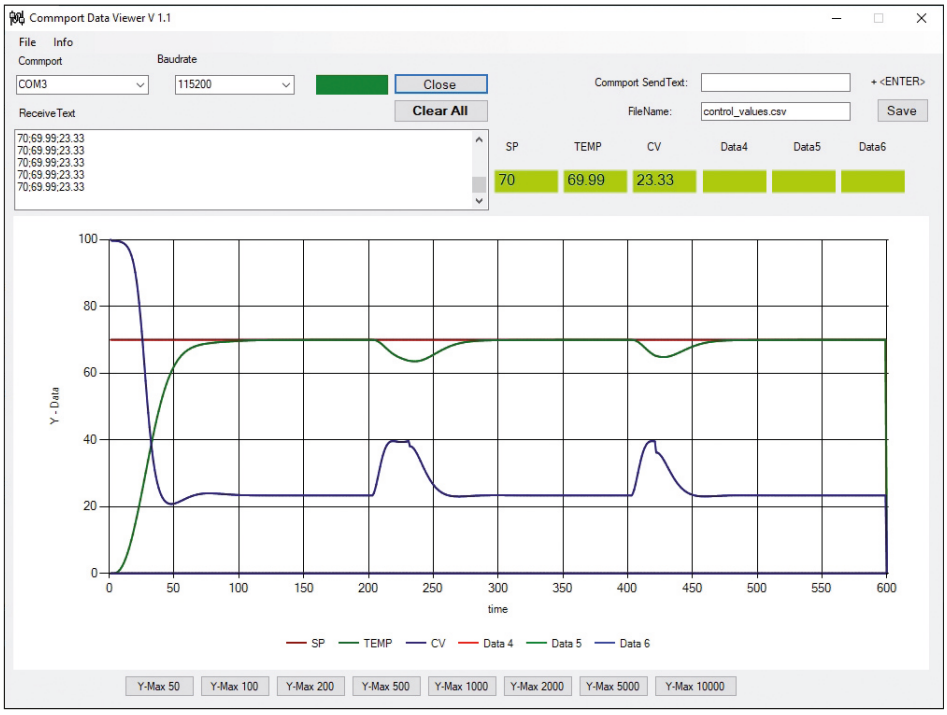


Figure 10.2: Simulation of disturbances

When testing this function, it was noticed that the temperature was higher than set. The value changes depending on the desired control value. A test run with temperatures from 50 K to 100 K allowed the following values to be determined. With an interpolation, the exact value can now also be determined for intermediate values such as 74.3.

```
float CorrectionValue = 0.0;
// 50 K
// float CorrectionValue = 16.08;
// 60 K
// float CorrectionValue = 13.85;
// 70 K
// float CorrectionValue = 11.88;
// 80 K
// float CorrectionValue = 10.11;
// 90 K
// float CorrectionValue = 8.47;
// 100 K
// float CorrectionValue = 6.93;
```

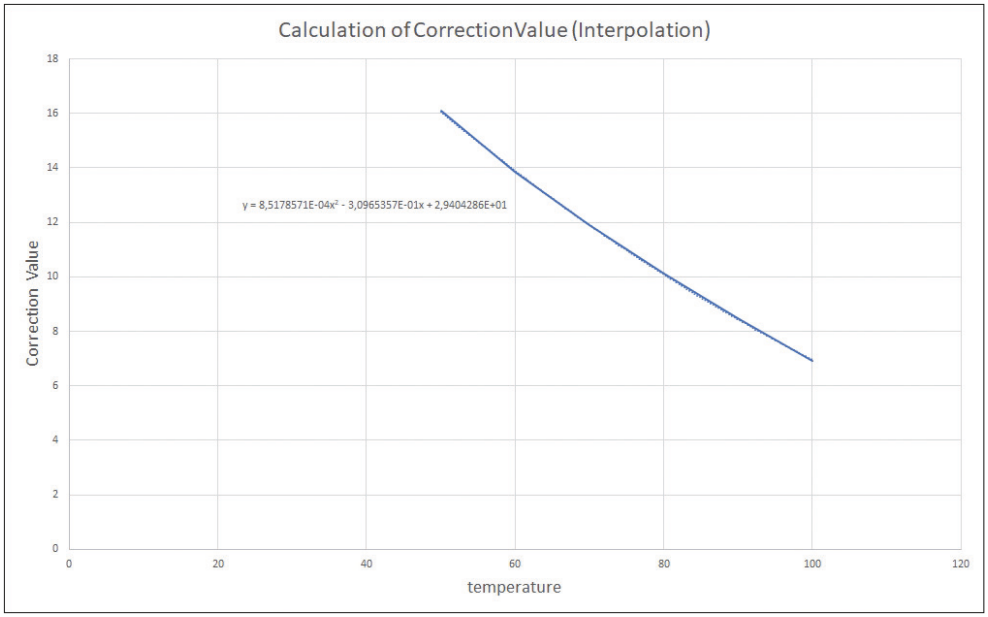


Figure 10.3: Korrektur Chart

Here is the function for customization in Ansi C for the Arduino:

```
// Function for calculating y based on the given quadratic equation
float calculateY(float x) {
    // Coefficients of the quadratic equation
    float a = 8.5178571E-04;
    float b = -3.0965357E-01;
    float c = 2.9404286E+01;
    // Calculation of y
    float y = a * x * x + b * x + c;
```

```
    return y;
}
```

This value is also used when the controller function is called up. This achieves a high accuracy of a few 1/00 K over the entire range.

```
  // uncomment for slidingModeControllerEXP()
  CorrectionValue = calculateY(setpoint);

  for (int i = 0; i < 600; i++)
  {
      // Simulating the temperature control system
      temperature = PT2_Simulation(control_signal,0);

      // Control signal from the two-point controller
      // control_signal = twoPointController(setpoint,temperature,hysteresis);

      // Control signal from the sliding mode controller
      // control_signal = slidingModeController(setpoint,temperature+2.5);
      control_signal =
slidingModeControllerEXP(setpoint,temperature+CorrectionValue);

      Serial.print("SP:");
      Serial.print(setpoint, 1);        // Setpoint with one decimal place
      Serial.print(" TEMP:");
      Serial.print(temperature, 2);     // Temperature with two decimal places
      Serial.print(" CV:");
      Serial.print(control_signal, 2);  // Control signal with two decimal places
      Serial.println(); // New line for the next output
      delay(150);

      if (i>200 && i<230) temperature = PT2_Simulation(0,0);
      if (i>400 && i<420) temperature = PT2_Simulation(0,0);

  }
```

10.3 Sliding mode controller example with real system

Another example can be found under Arduino_Slide_PhysicalHeater.ino for the real heating of the aluminum block from chapter 4:

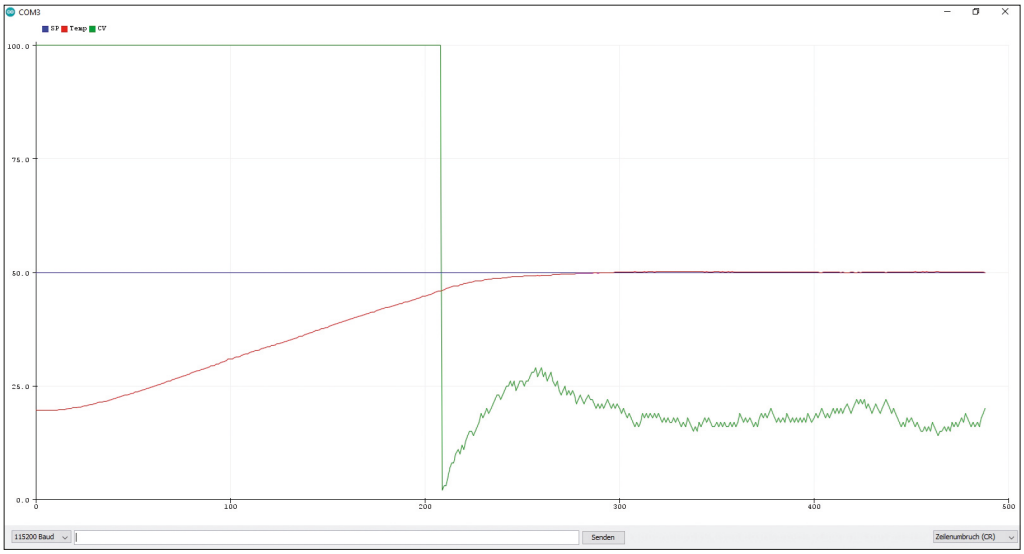


Figure 10.4: Temperature chart heating

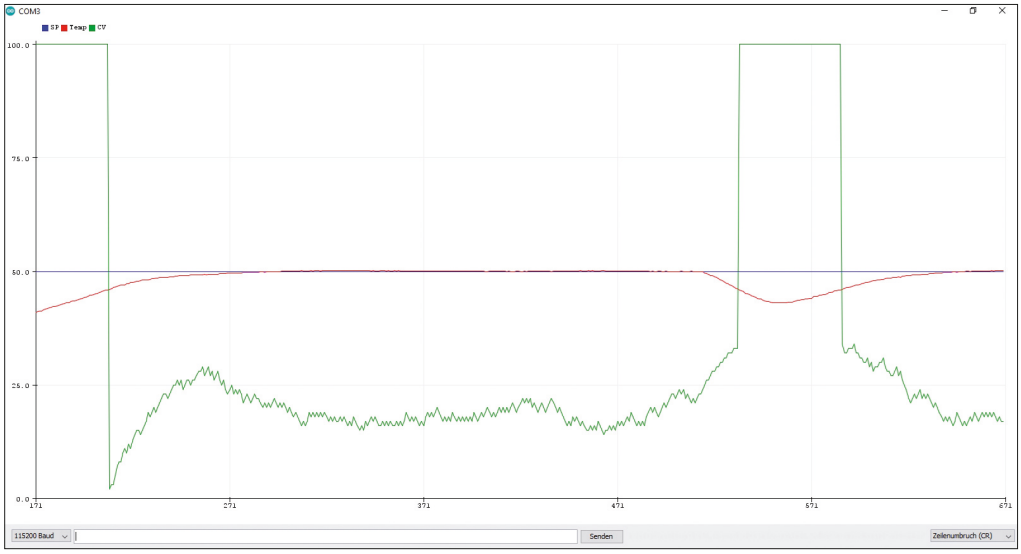


Figure 10.5: Temperature chart disturbances

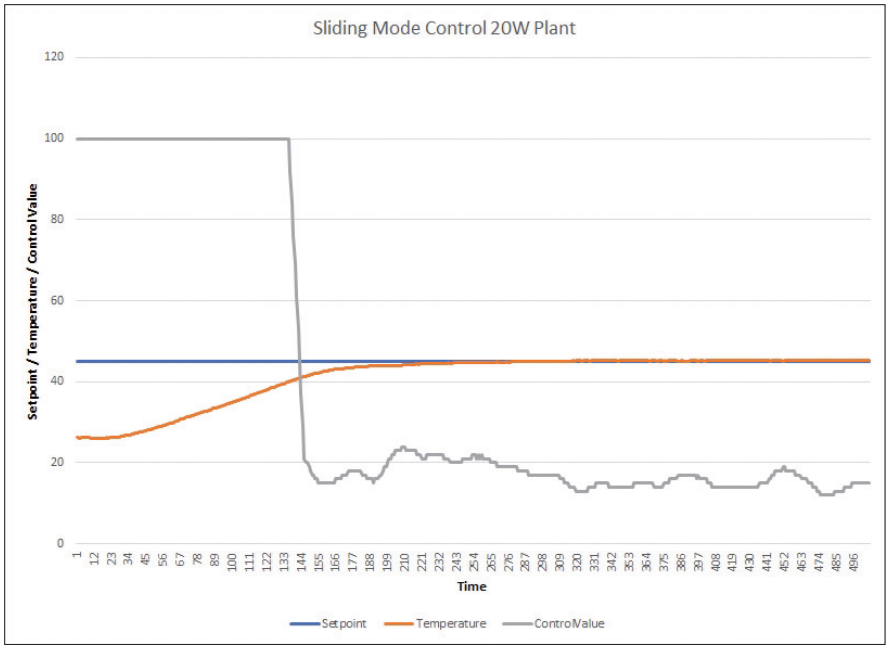


Figure 10.6: Temperature chart heating

Physical temperature controller with sliding mode programme description:

Listing 10.2: Arduino_Slide_PhysicalHeater.ino

```
//-----------------------------------------------------------------------
//
//            Arduino Sliding Mode Physical Heater
//            ============_=======================
//
//      Control Software for Arduino ESP32 PID Fuzzy Control Board
//      10ms Timer Interrupt Function
//      Init UART2 for Nextion Display
//      Display ADC1 and ADC2 Values on Display NX3224F028_001
//
//      Reading NTC1 Sensor with ADC1 / Set the Setpoint with POT2 on ADC2
//      Sliding Mode Controller Example
//      Outputting the pulses to Pin RELAIS1 for the heater
//
//
//  Date:     05/07/2024
//  Author:   Josef Bernhardt
//  Hardware: Arduino (ESPRESSIF) ESP32
//
//  File:     Arduino_PhysicalHeater_SlidingModeControl.ino
```

```
//-----------------------------------------------------------------------
#include <Arduino.h>
#include <Ticker.h>

int LED_BUILTIN = 2;

// ESP32 UART 2 Pins
#define RXD2 16
#define TXD2 17

// I2C Pins
#define I2CSCL 22
#define I2CSDA 21

// ESP32 ADC Pins
#define ADC1 32
#define ADC2 33
// Motor Sensor Pins
#define MOTSENS1 15
#define MOTSENS2 34
// Buttons
#define BTN1 14
#define BTN2 27

// Relay and Mosfet (BSP76) Pins
#define RELAIS1 18
#define RELAIS2 19
// LED1
#define LED1 5
// PWM1 and PWM2 Pins
#define PWM1 12
#define PWM2 13
// DAC Outputs 0..5V
#define DAC1 25
#define DAC2 26
// One Wire
#define DS18B20 4

#define FILTER_SIZE 10

// Define 10ms Period
const unsigned long timerInterval = 10;

// Ticker-Objekt for the Timer-Funktion
Ticker timer;
```

```
// Counter
int counter10ms=0;

int counterRelais1 = 0;

// When 1 second is elapsed
bool OneSecond = false;

// Counter for seconds
int CounterSeconds = 0;

float surface = 0;

// Arrays for NTC10K calculation
float Voltages[18] = {0};
float Temperatures[18] = {0};
int n_ntc = 18;

float VoltageESP32 = 3.3;
```

To check the calibration of the ADC, the voltage at ADC1 is set to e.g. 1 volt using a potentiometer and the displayed value of voltage1 is entered in the display for the corresponding value in ADC[7]. This is repeated for other voltages from 100 mV to 3.3 V. The 12-bit ADC values range from 0 to 4095 and are stored in the ADC[] array. The array Ua[] contains the corresponding voltage values in volts. The number of measuring points is set to 15 by n_adc.

```
float ADC[] = {0,   0,    123,   249, 496,  817,  1141, 1456, 1778, 2416, 3067,
3922, 4095, 4095, 4095};
// (Ua) in volts values applied to ADC by calibrator
float Ua[] = {0.00, 0.10, 0.20, 0.30, 0.50, 0.75, 1.00, 1.25, 1.50, 2.00, 2.50,
3.00, 3.07, 3.15, 3.30};
int n_adc = 15;

void Heat1_ON(int value)
{
  //Serial.println("ON");
  if (value > 0)
  {
    counterRelais1 = value;
    digitalWrite(RELAIS1,1);
    digitalWrite(LED1,1);
  }
}

void Heat1_OFF()
```

```
{
  //Serial.println("OFF");
  digitalWrite(RELAIS1,0);
  digitalWrite(LED1,0);
}

int mapInteger(int value, int fromLow, int fromHigh, int toLow, int toHigh) {
    return (value - fromLow) * (toHigh - toLow) / (fromHigh - fromLow) + toLow;
}

float mapFloat(float x, float in_min, float in_max, float out_min, float out_max)
{
  return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
}
```

The function `readvalue_linear(int n, float *x, float *y, float xn)` performs a linear interpolation to calculate the y-value for a given x-value `xn`. Firstly, the interval in which `xn` lies is determined by searching through the values in the array `x`. Linear interpolation is then performed within this interval to calculate the corresponding y-value. The result is based on the gradient between the neighboring points in the array. The calculated y-coordinate is then returned.

```
// Linear interpolation
// Input Parameters:
// n = Number of values -1
// x = x Array
// y = y Array
// xn = Value to be calculated
float readvalue_linear(int n,float *x,float *y, float xn)
{
    int j = 0;
    // Determine interval of xn
    while (j < n-1 && xn > x[j+1]) j++;
    // interpolate xn
    return ((y[j+1]-y[j]) / (x[j+1]-x[j])) * (xn-x[j]) + y[j];
}
```

The `filter_NTC1(float value)` function implements a moving average filter for the NTC1 sensor. It shifts all values in the `values` array one position to the left to make room for the new value at the end of the array. The new value is then inserted in the last position of the array. The sum of all values in the array is then calculated. Finally, the average of the values is calculated and returned as a filtered result.

```
// Moving Average Filter
float filter_NTC1(float value)
{
    static float values[FILTER_SIZE] = {0};
    float sum = 0;
    int i;
    for (i=0; i<FILTER_SIZE-1;i++)
    {
      values[i] = values[i+1];
    }
    values[FILTER_SIZE-1] = value;
    for (i=0; i<FILTER_SIZE;i++)
    {
      sum +=values[i];
    }
    return sum / (float)FILTER_SIZE;
}
```

The `filterControlValue(int newValue)` function implements a moving average filter for the NTC1 sensor. It saves the last `FILTER_SIZE` values in a static array `values` and updates this with the new value at the current index position each time it is called. The index is incremented cyclically to always overwrite the oldest value in the array. The average of all values in the array is then calculated. The function returns this average to provide the filtered result.

```
// Filter for the NTC1 sensor
int filterControlValue(int newValue) {
    // Array to store the last FILTER_SIZE values
    static int values[FILTER_SIZE] = {0};
    static int index = 0;
    int sum = 0;
    // Update the value at the current index position
    values[index] = newValue;
    // Increment the index for the next value
    index = (index + 1) % FILTER_SIZE;
    // Calculate the average of the values in the array
    for (int i = 0; i < FILTER_SIZE; ++i) {
        sum += values[i];
    }
    return sum / FILTER_SIZE;
}
```

The function `calculateResistance(float T)` calculates the resistance of an NTC thermistor based on the temperature `T`. It uses the Steinhart-Hart equation, taking into account the nominal resistance `R0` at 25°C and the B-value of the thermistor. The result is the resistance of the thermistor at the given temperature.

```
// Function to calculate the resistance of an NTC thermistor based on temperature
float calculateResistance(float T)
{
    // Constants
    const float R0 = 10000.0; // Nominal resistance of the thermistor at 25°C
    const float B = 3950.0;   // B-value of the thermistor, see datasheet
    return R0 * exp(B * ((1.0 / (T + 273.15)) - (1.0 / (25 + 273.15))));
}
```

The function `calulateVoltage(float Rt)` calculates the voltage across a voltage divider in which the NTC resistor `Rt` is connected in series with a fixed resistor of 10k Ohm. The voltage `Ua` is calculated using the input voltage `UE` (VoltageESP32) and the resistor `Rt`. The result is the voltage that drops across the NTC resistor.

```
// Voltage from voltage divider 10k input NTC resistor
float calulateVoltage(float Rt)
{
    float R25 = 10000;
    float UE = VoltageESP32;
    float Ua=(UE*Rt)/(Rt+R25);
    return Ua;
}
```

The `initArrayNTC()` function initializes two arrays that store the resistance values of an NTC thermistor and the corresponding temperatures. It runs through the temperature range from -35°C to 135°C in steps of 10°C and calculates the corresponding resistance and voltage for each temperature. These calculated values are then stored in the 'Temperatures' and 'Voltages' arrays. The `loop` counter ensures that the values are stored backwards in the arrays.

```
// Initialize array with NTC resistance and temperature
void initArrayNTC()
{
    // Temperature range from -35°C to 135°C
    int min_temp = -35;
    int max_temp = 135;
    float voltage;
    int loop = sizeof(Voltages) / sizeof(Voltages[0]) - 1;
    // Loop through the temperature range
    for (float temp = min_temp; temp <= max_temp; temp += 10)
    {
        // Calculate the resistance for the current temperature
        float resistance = calculateResistance(temp);
        float voltage = calulateVoltage(resistance);
        // Store the values for temperature and voltage in arrays
        Temperatures[loop] = temp;
```

```
        Voltages[loop] = voltage;
        loop--;
    }
}

// Timer-Funktion
// Called every 10 ms
void timerCallback()
{
  // Inc 10ms Counter
  counter10ms++;
  counterRelais1--;
  // If one Second
  if(counter10ms==100)
  {
     counter10ms = 0;
     // One Second
     OneSecond = true;
  }
  if(counterRelais1==0)
  {
     Heat1_OFF();
  }
}

// Sends Data to Nextion Display in a textbox
void Nextion_Send_Text_to_Textbox(char *textbox,String sfloat)
{
        Serial2.write(textbox);
        Serial2.write(".txt=\"");
        Serial2.print(sfloat);
        Serial2.write("\"");
        Serial2.write(0xFF);
        Serial2.write(0xFF);
        Serial2.write(0xFF);
}
```

The exponential moving average (EMA) is a filter that calculates a weighted average of current and previous values, whereby the more recent values are weighted more heavily. In contrast to the simple moving average (SMA), where all values are weighted equally, the EMA takes more recent data into account exponentially. The damping factor controls how strongly the current value influences the smoothed value: a small damping factor leads to a slower adjustment, while a large damping factor enables a faster reaction. The previous value is saved statically to ensure continuous smoothing over several function calls. The function finally returns the damped value, which is a mixture of the current and the previous value.

```
float applyDamping(float currentValue)
{
  // Variable declarations
  float result;
  float dampingFactor = 0.05;     // Damping factor
  static float previousValue = 0; // Previous value, stored statically
  // Calculation of the damped value
  result = previousValue * (1.0 - dampingFactor) + currentValue * dampingFactor;
  // Update the previous value
  previousValue = result;
  // Return the damped value
  return result;
}
```

The slidingModeController() function implements a sliding mode controller to control the temperature. It calculates the error between the setpoint and the current temperature and determines the sliding surface using the parameters c1 and c2. Based on the sliding surface, a decision is made as to whether the heating is switched on (if the sliding surface is positive) or no action is taken (if the sliding surface is negative). The calculated control value is further adjusted by a damping function, and if the error is greater than 5.0, it is set to 100. Finally, the adjusted value is returned.

```
float slidingModeController(float setpoint, float temperature)
{
    float eta = 40.0;
    float c1 = 0.025;
    float c2 = 1.0;
    float control_value = 0;
    float sliding_surface = 0;
    static float previous_error = 0;

    float error = setpoint - temperature;
    // Calculate the sliding surface
    sliding_surface = c1 * error + c2 * (error - previous_error);
    // Monitoring sliding surface
    surface = sliding_surface*100.0;
    // Determine the control input based on the sliding surface
    if (sliding_surface > 0)
    {
        control_value = eta;      // Heating
    }
    else
    {
        // control_value = -eta;  // Cooling
        control_value = 0;        // No action
    }
```

```
    control_value = applyDamping(control_value);
    previous_error = error;
    if (error > 5.0) control_value = 100;
    return control_value;
}

// Initializing the hardware
void setup()
{
  // Init USB UART
  Serial.begin(115200);
  // Nextion Display
  // TFTSerial.begin(9600);
  Serial2.begin(9600, SERIAL_8N1, RXD2, TXD2);
  // Init Hardware
  pinMode(LED_BUILTIN, OUTPUT);
  pinMode(LED1, OUTPUT);
  pinMode(RELAIS1, OUTPUT);
  pinMode(RELAIS2, OUTPUT);
  pinMode(BTN1, INPUT);
  pinMode(BTN2, INPUT);

  pinMode(ADC1, INPUT);
  pinMode(ADC2, INPUT);

  Nextion_Send_Text_to_Textbox("t1","VoltageNTC");
  Nextion_Send_Text_to_Textbox("t2","Temp NTC1");
  Nextion_Send_Text_to_Textbox("t3","ControlValue");
  Nextion_Send_Text_to_Textbox("t4","Voltage POT2");
  Nextion_Send_Text_to_Textbox("t5","Setpoint");
  Nextion_Send_Text_to_Textbox("t6","Surface * 100");

  initArrayNTC();

  // Init Timer with 10ms
  timer.attach_ms(timerInterval, timerCallback);

}
```

The `loop()` function continuously executes the main logic of the programme. It reads the analogue values from two ADC channels and uses these values to calculate the setpoint and the current temperature. The voltage from the first ADC channel is interpolated and the resulting temperature is calculated using a filtered value. When one second has elapsed, the control value is calculated by the sliding mode controller and used to control the heater. In addition, values are output via the serial interface and sent to a Nextion display. An LED is switched at regular intervals to indicate operation. The code thus enables regular

updating and control based on the measured and calculated values.

```
void loop()
{
  // Place your main code here to execute repeatedly:
  float setpoint = 0;
  float temperature = 0;
  int control_signal = 0;
  float hysteresis = 0.0;

  // Read ADC1 value
  int voltage1 = analogRead(ADC1);
  // Read ADC2 value
  int voltage2 = analogRead(ADC2);
  // setpoint between 25 and 60 K
  int scaledValue = mapInteger(voltage2, 0, 4095, 25, 60);
  setpoint = (float)scaledValue;

  // Interpolate voltage 1
  float u1 = readvalue_linear(n_adc-1,ADC,Ua,(float)voltage1);
  // Interpolate voltage 2
  // float u2 = readvalue_linear_voltage(n_adc-1,ADC,Ua,(float)voltage2);

  // temperature = readvalue_linear(n_ntc-1,Voltages,Temperatures,u1);
  temperature = readvalue_linear(n_ntc-1,Voltages,Temperatures,filter_NTC1(u1));

  // Check if one second has passed
  if (OneSecond)
  {
      // Reset the boolean variable OneSecond
      OneSecond = false;
      CounterSeconds++;

      control_signal = slidingModeController(setpoint,temperature);

      // control_signal = filterControlValue(control_signal);

      // Heat1 10-1000 ms on step 10ms
      Heat1_ON(control_signal);

      // Sends variable to USB UART and Seriell Monitor
      Serial.print("SP:");
      Serial.print(setpoint, 1);    // Setpoint with one decimal place
      Serial.print(" Temp:");
      Serial.print(temperature, 2); // Temperature with one decimal place
      Serial.print(" CV:");
```

```
      Serial.print(control_signal); // Control signal with one decimal place
      Serial.println();             // New line for the next output

      // Sends variable to display
      Nextion_Send_Text_to_Textbox("txtT1",String(voltage1));
      Nextion_Send_Text_to_Textbox("txtT2",String(temperature,3));
      Nextion_Send_Text_to_Textbox("txtT3",String(control_signal));
      // Sends variable to display
      Nextion_Send_Text_to_Textbox("txtT4",String(voltage2));
      Nextion_Send_Text_to_Textbox("txtT5",String(setpoint));
      // Nextion_Send_Text_to_Textbox("txtT6",String(CounterSeconds));
      Nextion_Send_Text_to_Textbox("txtT6",String(surface,3));

      // Switching LED
      digitalWrite(LED_BUILTIN,!digitalRead(LED_BUILTIN));
  }
  // Additional code for the main loop
}
```

10.4 Optimization of the parameters $c1$ and $c2$ with fuzzy logic

Introduction

Fuzzy logic controllers (FLC) provide a robust method for optimizing parameters in a sliding mode controller (SMC). Here we discuss how FLC can dynamically adjust $c1$ and $c2$ based on the error e and its derivative $\dot{e}$.

Example of fuzzy rules:

The following is a simplified example of fuzzy rules that adapt $c1$ and $c2$ based on linguistic variables of e and $\dot{e}$:

Example of Fuzzy-Rules			
e	$\dot{e}$	$c1$	$c2$
NB (Negativ big)	NB (Negativ big)	L (big)	S (small)
NB (Negativ big)	Z (Zero)	M (middle)	S (small)
Z (Zero)	Z (Zero)	M (middle)	M (middle)
PB (Positiv big)	PB (Positiv big)	L (big)	S (small)

Implementation in the sliding mode controller:

The implementation of the optimized values of $c1$ and $c2$ in the sliding mode controller includes

1. Continuous monitoring of e and $\dot{e}$.
2. Using FLC to determine optimal $c1$ and $c2$ based on current values of e and $\dot{e}$.

3. Update the sliding surface equation $s=c1 \cdot e+c2 \cdot \dot{e}$ with the newly optimized values.

4. Calculation of the control signal $u=-K \cdot \text{sign}(s)$ using the adjusted parameters.

Summary
The combination of fuzzy logic and sliding mode control provides a robust and adaptable approach to improving control performance. Fuzzy logic enables real-time optimization of $c1$ and $c2$, allowing the system to respond effectively to uncertainties and disturbances.

Explanations:

1. Initialization of the fuzzy rules: The rules determine how c1 and c2 are adjusted based on the linguistic variables of the error e and its derivative edot.

2. Initialization of the parameters and variables: Determining the initial values for the variables and constants.

3. Monitoring of e and edot: Measurement of the current error and its derivative.

4. Adjustment of c1 and c2: Determination of the optimum values for c1 and c2 based on the current values of e and edot.

5. Updating the sliding surface equation: Calculation of the sliding surface s with the new optimized values.

6. Calculation of the control signal: Determination of the control signal u using the adjusted parameters.

7. Application of the control signal: Implementation of the calculated control signal on the system.

This pseudo code describes the basic functionality of a fuzzy logic controller for dynamically adjusting the parameters in a sliding mode controller and offers a robust method for control optimization.

Listing 10.4: Dynamic adjustment of a sliding mode controller (pseudo code)

```
INITIALIZE FLC rules for c1 and c2 based on linguistic variables of e and ė
    RULE: IF e == NB AND ė == NB THEN c1 = L AND c2 = S
    RULE: IF e == NB AND ė == Z THEN c1 = M AND c2 = S
    RULE: IF e == Z AND ė == Z THEN c1 = M AND c2 = M
    RULE: IF e == PB AND ė == PB THEN c1 = L AND c2 = S

INITIALIZE parameters and variables
    e = 0      // error
    e_dot = 0  // derivative of error
```

```
    c1 = 1          // initial value of c1
    c2 = 1          // initial value of c2
    s = 0           // sliding surface
    u = 0           // control signal
    K = some_value  // constant for control signal calculation

LOOP until termination condition is met
    // Step 1: Monitor error and its derivative
    e = measure_error()
    e_dot = measure_error_derivative()

    // Step 2: Use FLC to determine optimal c1 and c2 based on current values of
e and e˙
    IF e == NB AND e_dot == NB THEN
        c1 = L
        c2 = S
    ELSE IF e == NB AND e_dot == Z THEN
        c1 = M
        c2 = S
    ELSE IF e == Z AND e_dot == Z THEN
        c1 = M
        c2 = M
    ELSE IF e == PB AND e_dot == PB THEN
        c1 = L
        c2 = S
    END IF

    // Step 3: Update sliding surface equation s = c1 * e + c2 * e_dot with the
new optimized values
    s = c1 * e + c2 * e_dot

    // Step 4: Calculate control signal u = -K * sign(s) using the adjusted
parameters
    u = -K * sign(s)

    // Apply control signal to the system
    apply_control_signal(u)

    // Optional: Log or display values for monitoring and debugging
    log_values(e, e_dot, c1, c2, s, u)

END LOOP

FUNCTION measure_error()
    // Code to measure the current error
    RETURN measured_error
```

```
FUNCTION measure_error_derivative()
    // Code to measure the derivative of the current error
    RETURN measured_error_derivative

FUNCTION sign(value)
    IF value > 0 THEN
        RETURN 1
    ELSE IF value < 0 THEN
        RETURN -1
    ELSE
        RETURN 0
    END IF

FUNCTION apply_control_signal(u)
    // Code to apply the control signal to the system
END FUNCTION

FUNCTION log_values(e, e_dot, c1, c2, s, u)
    // Code to log or display the values for monitoring and debugging
END FUNCTION
```

Chapter 11 • Neural networks

11.1 History of neural networks

The development of neural networks began in the 1950s with the revolutionary work of scientists such as Frank Rosenblatt, who developed the perceptron - an early model of an artificial neuron. Rosenblatt's perceptron was able to perform simple classification tasks and thus laid the foundation for later advances in the field of artificial intelligence. In the following decades, however, these early models were challenged by the discovery of limitations such as the inability to solve non-linear problems.

The 1960s brought significant developments by researchers such as Marvin Minsky and Seymour Papert, who discussed the limitations of the early perceptron models in their book 'Perceptrons'. Their work led to a temporary decline in interest in neural networks, which became known as 'AI Winter', a period of reduced funding and research in artificial intelligence.

In the 1980s, research was revitalized by the rediscovery of the backpropagation algorithm, pioneered by Geoffrey Hinton, Yann LeCun and Ronald J. Williams. The backpropagation algorithm made it possible to efficiently minimize the error function in multilayer networks and opened up new possibilities for training complex neural networks. Through their work, Hinton and LeCun contributed significantly to the further development of feedforward networks.

In addition, John Hopfield made important contributions to the theory of neural networks in the 1980s with his Hopfield networks, which served as a model for associative memories. Judea Pearl's work on the development of Bayesian networks and Yoshua Bengio's advances in the field of deep neural networks are also significant milestones that have contributed to the performance of feedforward networks today.

These developments led to a renaissance in the field of artificial intelligence, and modern feedforward networks have since established themselves as a powerful tool in pattern recognition and machine learning. Their ability to learn and generalize complex non-linear relationships has made them a central element in the research and application of artificial intelligence.

Neural feedforward networks are a fascinating class of artificial neural networks that are able to recognize complex patterns and make predictions. These networks consist of multiple layers of neurons, with each layer connected in only one direction - from the input layer, through one or more hidden layers, to the output layer.

The connection between neural feedforward networks and fuzzy logic in control engineering is particularly exciting. Neural networks can learn to capture complex relationships in data, while fuzzy logic provides flexible and interpretable decision rules. Together, these approaches can enable powerful control systems that are both adaptive and robust in the face of uncertainty. The integration of fuzzy logic into neural networks opens up new possibilities for the development of intelligent control systems that can adapt dynamically to changing conditions and make precise decisions.

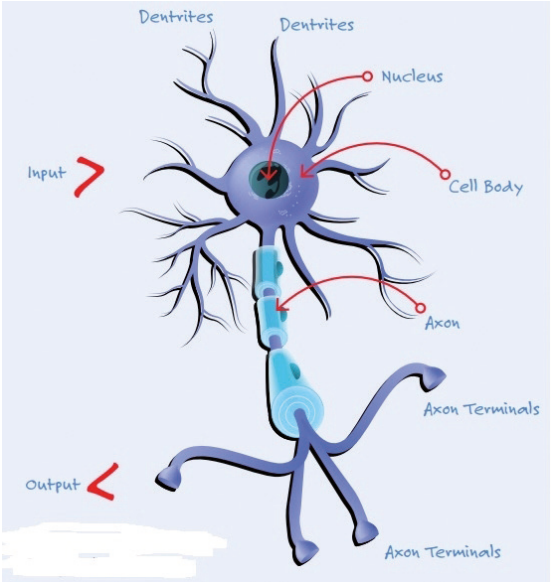


Figure 11.1: Biological neuron

11.2 Biological neuron

Neural networks are inspired by the biological networks in the human brain, as they mimic the basic principles of neural information processing. Like biological neurons, artificial neurons receive signals, process them and pass them on to other neurons, dynamically adjusting the strength of the connections, or weights. This structure and functionality enables neural networks to learn complex patterns and make decisions based on the learnt data, similar to how the brain processes and stores information.

11.3 Basics

The functioning of neural networks is based on precise mathematical models that enable the adaptation and calculation of network outputs. Each artificial neuron in the network is defined by mathematical equations that weigh, sum, and transform the inputs using an activation function. The training process of a network, in which the weights are adjusted so that the network provides optimal predictions or classifications, is carried out using iterative optimization techniques such as the backpropagation algorithm. These mathematical models and calculations are crucial for the performance and accuracy of the network and enable precise control of information processing within the system.

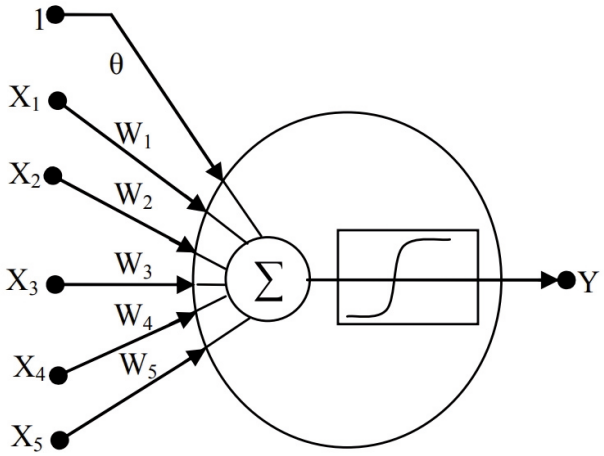


Figure 11.2: Mathematical model of a neuron

To understand the calculation in a neuron with two inputs, we look at the entire process of information processing. First, the neuron's inputs are weighted and summed before an activation function is applied to calculate the final output.

The calculation begins with the weighing and summation of the inputs:

$$z = x1 * w1 + x2 * w2 + b,$$

where x1 and x2 are the inputs, w1 and w2 are the weights and b is the bias term.

The bias value in a neural network is an additional parameter that is added to the weighted sum of the inputs to increase the flexibility of the model. It allows the neuron to make a shift in the activation function and thus react better to the input data, especially if all weights were zero. The bias allows the neuron to generate an output even if the inputs themselves are not sufficient to activate the activation function. This helps to improve the performance of the network and increase its learning ability.

11.4 Activation functions

The result is then transformed by the sigmoid activation function, which looks as follows:

$$\sigma(z) = 1 / (1 + e^{(-z)}).$$

This function converts the calculated value into an output between 0 and 1.

Here is a calculation of the Sigmoid and Tanh values for an input value of -8 to +8.

The Tanh function is used if the output values should also be negative.

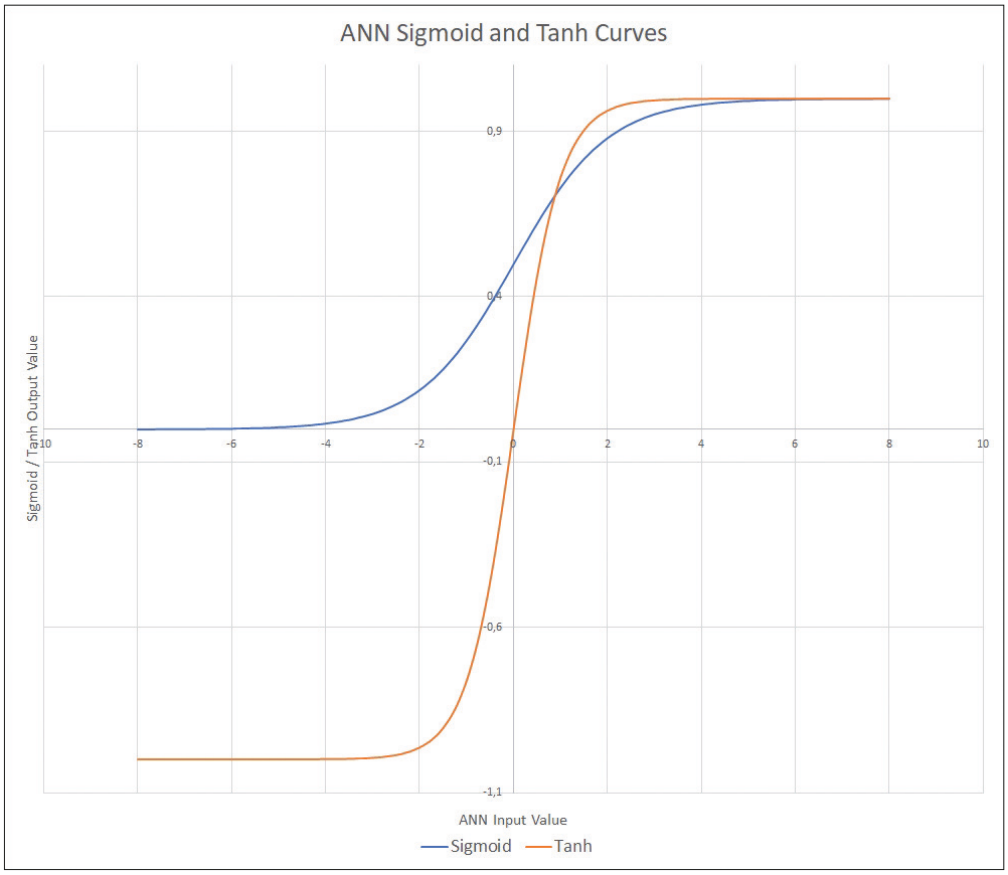


Figure 11.3: Sigmoid and Tanh transfer function

The calculation of Sigmoid and Tanh in AnsiC:

```
// Sigmoid function
double sigmoid(double x)
{
    return 1.0 / (1.0 + exp(-x));
}

// Tanh function
double tanh_function(double x)
{
    return tanh(x);
}

// Loop over values from -8 to +8 in steps of 1
for (input = -8.0; input <= 8.0; input += 0.1)
{
```

```
    // Sigmoid calculation
    double sigmoid_result = sigmoid(input);
    // Tanh- calculation
    double tanh_result = tanh_function(input);

    // Output of the results
    printf("NET: %.3f ", input);
    printf("Sigmoid = %.3f " , sigmoid_result);
    printf("Tanh = %f\n" , tanh_result);

}
```

11.5 The perceptron

The programme Arduino_GradientDescentPerceptronDE.ino implements a simple perceptron that learns the AND function using the gradient descent. It initializes the weights of the perceptron and the training data and then iteratively adjusts the weights over several epochs. The perceptron uses a sigmoid activation function to generate outputs between 0 and 1 based on the weighted sum of the inputs. The training function updates the weights based on the error between the predicted and expected outputs for each training example. After training, the programme outputs the trained weights and tests the perceptron against the training data, displaying the results.

```
// Gradient descent to update the weights based on the error
void gradient_descent(float predicted, float target, float input1, float input2)
{
  float error = target - predicted; // Calculation of the error
  float gradient_w1 = -error * sigmoid_derivative(predicted) * input1; //
Gradient for weight w1
  float gradient_w2 = -error * sigmoid_derivative(predicted) * input2; //
Gradient for weight w2

  // Updating the weights with the calculated learning rate
  w1 -= learning_rate * gradient_w1;
  w2 -= learning_rate * gradient_w2;
}
```

11.6 Feedforward networks

Feedforward networks can consist of multiple inputs, outputs and multiple hidden layers, which increases their flexibility and performance. Each layer in the network processes the data in a different way, passing the inputs through the hidden layers and ultimately aggregating them into the outputs. This architecture allows the network to learn complex relationships and patterns in the data and make accurate predictions.

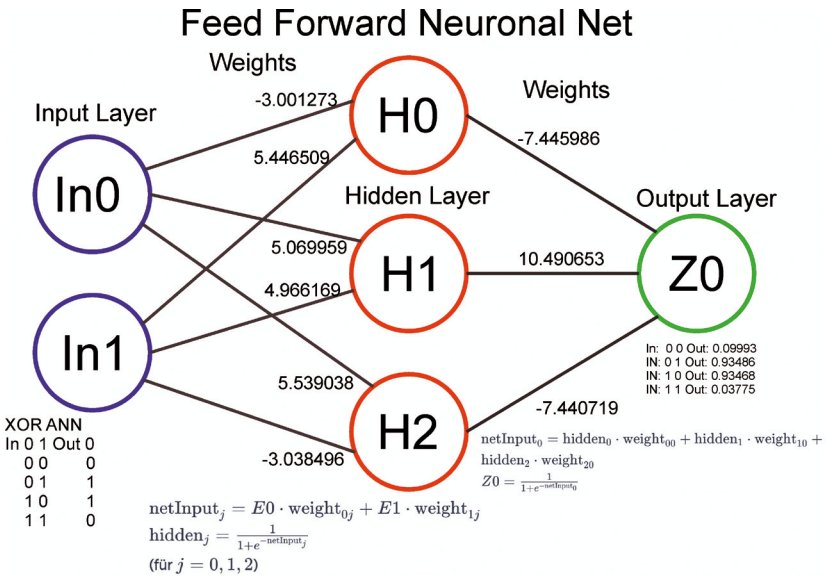


Figure 11.4: Feedforward Net

To practically illustrate the calculations of a trained neural network for the XOR problem, we use a prepared Excel spreadsheet. This table already contains the weights and bias values that were determined by training the network. With this data, we can now calculate the outputs step by step based on various input values and understand how the network works in a specific application.

	A	B	C	D	E	F	G	H	I
1	Neuronal Net with Excel								
2									
3	Input		WeightIH		Hidden Layer			Output Layer	
4									
5				Hidden0	Hidden 0 Out	Weigtht 0			
6			Weight 0 0	-2,26639859	0,093944312	-7,445986			
7			-3,001123						
8	IN0		Weight 1 0						Result
9	1,1		5,446509	Hidden1	Hidden 1 Out	Weigtht 1			
10	IN1		Weight 0 1	6,52052701	0,998529274	10,490653	2,3648141	Output	0,9141046
11	0,19		5,069959						
12			Weight 1 1						
13			4,966169	Hidden2	Hidden 2 Out	Weigtht 2			
14			Weight 0 2	5,51562756	0,995992721	-7,440719			
15			5,539038						
16			Weight 1 2						
17			-3,038496						

Figure 11.5: Calculation of an ANN using Excel

IN0	IN1	XOR (OUT)
0	0	0
0	1	1
1	0	1
1	1	0

Explanation of the table:

- **IN0** and **IN1** are the input values.
- **XOR (OUT)** shows the result of the XOR operation for the given input values.

In the XOR table:

- If both inputs are 0, the result is 0.
- If exactly one of the inputs is 1, the result is 1.
- If both inputs are 1, the result is 0.

If the values IN0 and IN1, which serve as inputs for the neural network, are changed, these changes influence the calculations within the network and therefore also the result at the output. The neural network processes the inputs through its various layers - starting with the input layer, through the hidden layers, to the output layer. Each input is multiplied by the associated weights and further processed by the activation functions in the hidden layers. The resulting values are then used to calculate the output. By adjusting the IN0 and IN1 values, we can observe how the output of the network changes, which gives direct feedback on how the network reacts to different inputs and how well it is able to capture and calculate the underlying XOR pattern.

The intelligence of a neural network lies largely in the weights, as these determine the strength and direction of the connections between neurons. By adjusting the weights during the training process, the network learns to recognize complex patterns and make precise predictions.

Here is an extract from the programme for the calculation of a network in the Ansi C language:

```
void forwardPropagation(NeuralNetwork *network, float input[MAX_INPUT_NEURONS],
float hidden[MAX_HIDDEN_NEURONS], float output[MAX_OUTPUT_NEURONS]) {
    for (int j = 0; j < MAX_HIDDEN_NEURONS; ++j)
    {
        float netInput = 0.0;
        for (int i = 0; i < MAX_INPUT_NEURONS; ++i)
        {
            netInput += input[i] * network->inputToHiddenWeights[i][j];
        }
        hidden[j] = sigmoid(netInput);
    }
```

```
    for (int k = 0; k < MAX_OUTPUT_NEURONS; ++k)
    {
        float netInput = 0.0;
        for (int j = 0; j < MAX_HIDDEN_NEURONS; ++j)
        {
            netInput += hidden[j] * network->hiddenToOutputWeights[j][k];
        }
        output[k] = sigmoid(netInput);
    }
}
```

11.7 Training the neural network

The backpropagation algorithm, often abbreviated as 'backprop', represents a fundamental advance in the field of neural networks and machine learning. The basic idea of this algorithm was first formulated in the 1960s, but it was the refinement in the 1980s by researchers such as Geoffrey Hinton, Yann LeCun and Ronald J. Williams that established its importance for the training of neural networks. Backpropagation is based on the principle of gradient descent to minimize the error function of a network by adjusting the weights in the neurons.

The function of the backpropagation algorithm is to backpropagate the error of the network resulting from the difference between the actual and desired outputs through the network. The error is propagated layer by layer from the output layer through the hidden layers to the input layer. The algorithm calculates the gradients of the weights with respect to the error and uses this information to update the weights and thus improve the accuracy of the prediction.

This process is iterative and allows the network to gradually optimize its weights, leading to a continuous improvement in model accuracy. Backpropagation has proven to be extremely effective, especially for complex feedforward networks with multiple layers. This method allows networks to learn not only simple linear relationships, but also to capture complex non-linear patterns. The introduction and further development of backpropagation have contributed significantly to the current performance of neural networks, which are used in a variety of machine learning and artificial intelligence applications.

Step 1: Initialization
All weights `ω` and threshold values `b` are set to small random values.

Step 2: Calculation of the net output `yk`
First, the net input `net` of the neurons in layer `j` is calculated:

$$netj = \Sigma\omega ij * xi$$

Where `ωij` is the weighting between neuron `i` and neuron `j`, and `xi` is the input signal at input `i`. The output of the neurons in layer `j` is therefore:

yj = ϕ(netj - bj)

Where `ϕ` is the activation function. The net input of the neurons in layer `k` is:

netk = Σωjk * yj

And the net output is

yk = ϕ(netk - bk)

Step 3: Determining the error at the grid output

δk = (tk - yk) * ϕ›(netk - bk)

The variable `tk` is the target output of the output euron `k`.

Step 4: Error feedback

δj = ϕ›(netj - bj) * Σωjk * δk

Step 5: Learning
The weights are calculated as follows

ωjk = ωjk + η * yj * δk
ωij = ωij + η * xi * δj

The threshold values are calculated as follows

bk = bk - η * δk
bj = bj - η * δj

Step 6: End of epoch
As long as there are still training patterns available, step 2 is continued.

Step : End of training
The quadratic error is calculated by E = 1/2 * Σ(tk - yk)^2. As long as E is greater than an error determined at the beginning and the maximum number of epochs has not yet been reached, the process continues at step 2. Epoch refers to one run of a training pattern through the MLP.

ω (small omega) → ωjk
η (small Eta) → η
ϕ (small Phi) → ϕ(netj - bj)
δ (small delta) → δk, δj
tk (small tau) → tk
τ (small tau) → τk

Here is a pseudocode for the backpropagation algorithm:

```
while there exists a pattern p in P such that totalError > errorThreshold do
    # Input the pattern p into the network and compute the output y
    float input[MAX_INPUT_NEURONS], hidden[MAX_HIDDEN_NEURONS],
output[MAX_OUTPUT_NEURONS];
    for (int i = 0; i < MAX_INPUT_NEURONS; ++i) {
        input[i] = trainingPatterns[p][i];
    }
    forwardPropagation(network, input, hidden, output);

    # For each output neuron k
    for (int k = 0; k < MAX_OUTPUT_NEURONS; ++k) {
        if output[k] == targetOutput[k] then
            # Output is correct, no weight adjustment needed
            continue; # No weight adjustment
        else
            if output[k] == 0 then
                # Increase weights if the neuron outputs 0
                for (int j = 0; j < MAX_HIDDEN_NEURONS; ++j) {
                    # Increase weight to k by η * hidden[j]
                    network->hiddenToOutputWeights[j][k] += η * hidden[j];
                }
            end if

            if output[k] == 1 then
                # Decrease weights if the neuron outputs 1
                for (int j = 0; j < MAX_HIDDEN_NEURONS; ++j) {
                     # Decrease weight to k by η * hidden[j]
                     network->hiddenToOutputWeights[j][k] -= η * hidden[j];
                }
            end if
        end if
    end for

    # Compute the totalError for the current pattern
    float patternError = 0.0;
    for (int k = 0; k < MAX_OUTPUT_NEURONS; ++k) {
        patternError += 0.5 * pow(targetOutput[k] - output[k], 2);
    }
    totalError += patternError;

    backPropagation(network, input, hidden, output, targetOutput);
 end while
```

11.8 ANN example XOR

To illustrate the versatile applications of neural networks, we will look at some concrete examples realized on the Arduino platform. First, we show how neural networks can be used to solve the classic XOR problem to illustrate the basic principles of classification and pattern recognition. We then demonstrate how a color sensor can be used in combination with a neural network to evaluate and classify colors. Finally, we look at function approximation, in which a neural network is used to simulate a controller characteristic curve and thus precisely reproduce the behavior of a control system. These examples provide a comprehensive insight into the flexibility and performance of neural networks and show how they can be used in various technical and scientific scenarios.

In this Arduino example Arduino_ANN_OneHiddenLayer.ino, a simple neural network with a hidden layer is implemented on an ESP32 microcontroller to learn logical operations such as AND, OR and XOR. First, the network is initialized with random weights and the sigmoid activation function is used to activate the neurons. The code performs both forward and backward propagation: During forward propagation, the network outputs are calculated, and during backward propagation, the weights are adjusted to minimize the errors. The training data is used for logical operations (AND, OR, XOR) to train the network and the results are displayed before and after training. Finally, the results are monitored in an infinite loop to continuously check the performance of the network.

```
//--------------------------------------------------------------------
//
//            Arduino Neuronal Net One Hidden layer
//            ============_=======================
//
//      ANN Software for Arduino ESP32 PID Fuzzy Control Board
//
//  Date:     31/07/2024
//  Author:   Josef Bernhardt
//  Hardware: Arduino (ESPRESSIF) ESP32
//
//  File:     Arduino_ANN_OneHiddenLayer.ino
//--------------------------------------------------------------------
#include <Arduino.h>

#define MAX_PATTERNS 4
#define MAX_INPUT_NEURONS 2
#define MAX_HIDDEN_NEURONS 5
#define MAX_OUTPUT_NEURONS 1

#define LED 5

typedef struct {
    float inputToHiddenWeights[MAX_INPUT_NEURONS][MAX_HIDDEN_NEURONS];
    float hiddenToOutputWeights[MAX_HIDDEN_NEURONS][MAX_OUTPUT_NEURONS];
```

```
} NeuralNetwork;

void initializeNetwork(NeuralNetwork *network) {
    for (int i = 0; i < MAX_INPUT_NEURONS; ++i) {
        for (int j = 0; j < MAX_HIDDEN_NEURONS; ++j) {
            network->inputToHiddenWeights[i][j] = ((float)random(1000) / 1000.0)
- 0.5;
        }
    }

    for (int i = 0; i < MAX_HIDDEN_NEURONS; ++i) {
        for (int j = 0; j < MAX_OUTPUT_NEURONS; ++j) {
            network->hiddenToOutputWeights[i][j] = ((float)random(1000) / 1000.0)
- 0.5;
        }
    }
}

float sigmoid(float x) {
    return 1.0 / (1.0 + exp(-x));
}

void forwardPropagation(NeuralNetwork *network, float input[MAX_INPUT_NEURONS],
float hidden[MAX_HIDDEN_NEURONS], float output[MAX_OUTPUT_NEURONS]) {
    for (int j = 0; j < MAX_HIDDEN_NEURONS; ++j)
    {
        float netInput = 0.0;
        for (int i = 0; i < MAX_INPUT_NEURONS; ++i)
        {
            netInput += input[i] * network->inputToHiddenWeights[i][j];
        }
        hidden[j] = sigmoid(netInput);
    }

    for (int k = 0; k < MAX_OUTPUT_NEURONS; ++k)
    {
        float netInput = 0.0;
        for (int j = 0; j < MAX_HIDDEN_NEURONS; ++j)
        {
            netInput += hidden[j] * network->hiddenToOutputWeights[j][k];
        }
        output[k] = sigmoid(netInput);
    }
}

void backPropagation(NeuralNetwork *network, float input[MAX_INPUT_NEURONS],
```

```
float hidden[MAX_HIDDEN_NEURONS], float output[MAX_OUTPUT_NEURONS], float
targetOutput[MAX_OUTPUT_NEURONS]) {


    // Backpropagation for the output layer
    for (int k = 0; k < MAX_OUTPUT_NEURONS; ++k)
    {
        float error = targetOutput[k] - output[k];
        float outputGradient = output[k] * (1.0 - output[k]);

        for (int j = 0; j < MAX_HIDDEN_NEURONS; ++j)
        {
            network->hiddenToOutputWeights[j][k] += 0.1 * error * outputGradient
* hidden[j];
        }
    }

    // Backpropagation for the hidden layer
    for (int j = 0; j < MAX_HIDDEN_NEURONS; ++j)
    {
        float hiddenError = 0.0;

        for (int k = 0; k < MAX_OUTPUT_NEURONS; ++k)
        {
            hiddenError += (targetOutput[k] - output[k]) * output[k] * (1.0 -
output[k]) * network->hiddenToOutputWeights[j][k];
        }

        float hiddenGradient = hidden[j] * (1.0 - hidden[j]);

        for (int i = 0; i < MAX_INPUT_NEURONS; ++i)
        {
            network->inputToHiddenWeights[i][j] += 0.1 * hiddenError *
hiddenGradient * input[i];
        }
    }
    // digitalWrite(LED, LOW);
}

void trainNetwork(NeuralNetwork *network, float trainingPatterns[MAX_PATTERNS]
[MAX_INPUT_NEURONS], float targetOutputs[MAX_PATTERNS][MAX_OUTPUT_NEURONS]) {
    int maxEpochs = 50000;
    float errorThreshold = 0.005;

    for (int epoch = 0; epoch < maxEpochs; ++epoch) {
        float totalError = 0.0;
```

```
        for (int p = 0; p < MAX_PATTERNS; ++p)
        {
            float input[MAX_INPUT_NEURONS];
            float hidden[MAX_HIDDEN_NEURONS];
            float output[MAX_OUTPUT_NEURONS];

            for (int i = 0; i < MAX_INPUT_NEURONS; ++i)
            {
                input[i] = trainingPatterns[p][i];
            }

            forwardPropagation(network, input, hidden, output);

            float patternError = 0.0;
            for (int k = 0; k < MAX_OUTPUT_NEURONS; ++k)
            {
                patternError += 0.5 * pow(targetOutputs[p][k] - output[k], 2);
            }
            totalError += patternError;

            backPropagation(network, input, hidden, output, targetOutputs[p]);
        }

        if (totalError < errorThreshold)
        {
            Serial.println("Training successful!");
            break;
        }

        if (epoch % 1000 == 0) {
            Serial.print("Epoch: ");
            Serial.print(epoch + 1);
            Serial.print(", Error: ");
            Serial.println(totalError, 7);
            digitalWrite(LED, !digitalRead(LED));
        }
    }
}

void testNetwork(NeuralNetwork *network, float inputPatterns[MAX_PATTERNS][MAX_
INPUT_NEURONS]) {
    for (int p = 0; p < MAX_PATTERNS; ++p)
    {
        float input[MAX_INPUT_NEURONS];
        float hidden[MAX_HIDDEN_NEURONS];
```

```
        float output[MAX_OUTPUT_NEURONS];

        for (int i = 0; i < MAX_INPUT_NEURONS; ++i)
        {
            input[i] = inputPatterns[p][i];
        }

        forwardPropagation(network, input, hidden, output);
        Serial.print("Input: ");
        Serial.print(input[0]);
        Serial.print(" ");
        Serial.print(input[1]);
        Serial.print(", Output: ");
        Serial.println(output[0]);
    }
}

void setup()
{
    Serial.begin(115200);
    delay(5000);
    pinMode(LED, OUTPUT);
}

void loop()
{

    NeuralNetwork network_and;
    NeuralNetwork network_or;
    NeuralNetwork network_xor;

    float trainingPatterns_and[MAX_PATTERNS][MAX_INPUT_NEURONS] = {{0, 0}, {0,
1}, {1, 0}, {1, 1}};
    float targetOutputs_and[MAX_PATTERNS][MAX_OUTPUT_NEURONS] = {{0}, {0}, {0},
{1}};

    float trainingPatterns_or[MAX_PATTERNS][MAX_INPUT_NEURONS] = {{0, 0}, {0, 1},
{1, 0}, {1, 1}};
    float targetOutputs_or[MAX_PATTERNS][MAX_OUTPUT_NEURONS] = {{0}, {1}, {1},
{1}};

    float trainingPatterns_xor[MAX_PATTERNS][MAX_INPUT_NEURONS] = {{0, 0}, {0,
1}, {1, 0}, {1, 1}};
    float targetOutputs_xor[MAX_PATTERNS][MAX_OUTPUT_NEURONS] = {{0}, {1}, {1},
{0}};
```

```
    initializeNetwork(&network_and);
    initializeNetwork(&network_or);
    initializeNetwork(&network_xor);

    Serial.println("Before Training AND:");
    testNetwork(&network_and, trainingPatterns_and);
    trainNetwork(&network_and, trainingPatterns_and, targetOutputs_and);

    Serial.println("Before Training OR:");
    testNetwork(&network_or, trainingPatterns_or);
    trainNetwork(&network_or, trainingPatterns_or, targetOutputs_or);

    Serial.println("Before Training XOR:");
    testNetwork(&network_xor, trainingPatterns_xor);
    trainNetwork(&network_xor, trainingPatterns_xor, targetOutputs_xor);

    while (1)
    {
      digitalWrite(LED, HIGH);

      Serial.println("\nAfter Training AND:");
      testNetwork(&network_and, trainingPatterns_and);

      Serial.println("\nAfter Training OR:");
      testNetwork(&network_or, trainingPatterns_or);

      Serial.println("\nAfter Training XOR:");
      testNetwork(&network_xor, trainingPatterns_xor);


      digitalWrite(LED, LOW);
      delay(500);


    }

}
```

11.9 Fuzzy Gauss controller with neural network

This project deals with the development and testing of a controller using the Arduino software and a PT2 simulation. The focus is on generating a control characteristic using a fuzzy Gaussian controller and using this characteristic to generate weights for a neural network. This neural network has one input, five hidden neurons and one output.

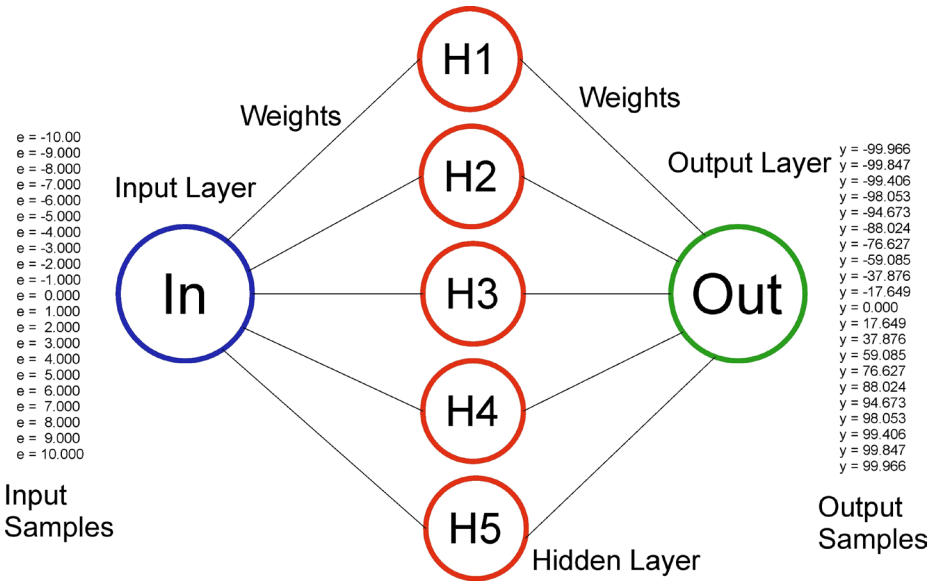


Figure 11.6: Simulation of a function

In the first step, the characteristic curve of the fuzzy Gaussian controller is created using a PT2 simulation. This characteristic curve serves as the basis for generating the weightings for the neural network. The process of determining the weights is carried out in the 'Codeblocks_C_FuzzyGauss_TrainNetworkWeights' project.

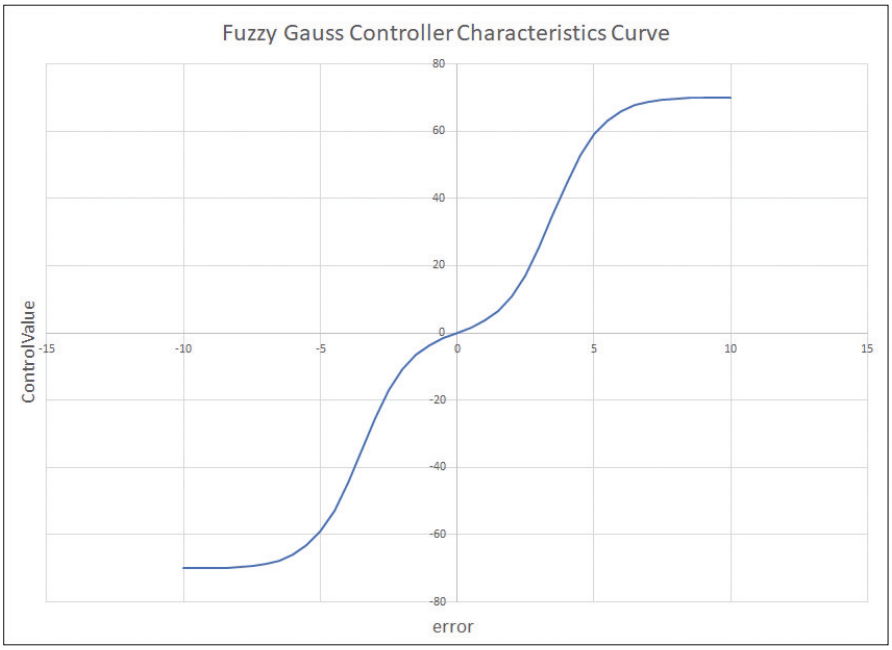


Figure 11.7: ANN Fuzzy Gauss controller

Once the weightings have been successfully determined, they are integrated into an Arduino programme that tests the controller together with a PT2 simulation. This program checks the performance of the neural network using forward propagation and analyses the controller performance in the PT2 environment. The results of this test provide valuable insights into the efficiency and accuracy of the controller.

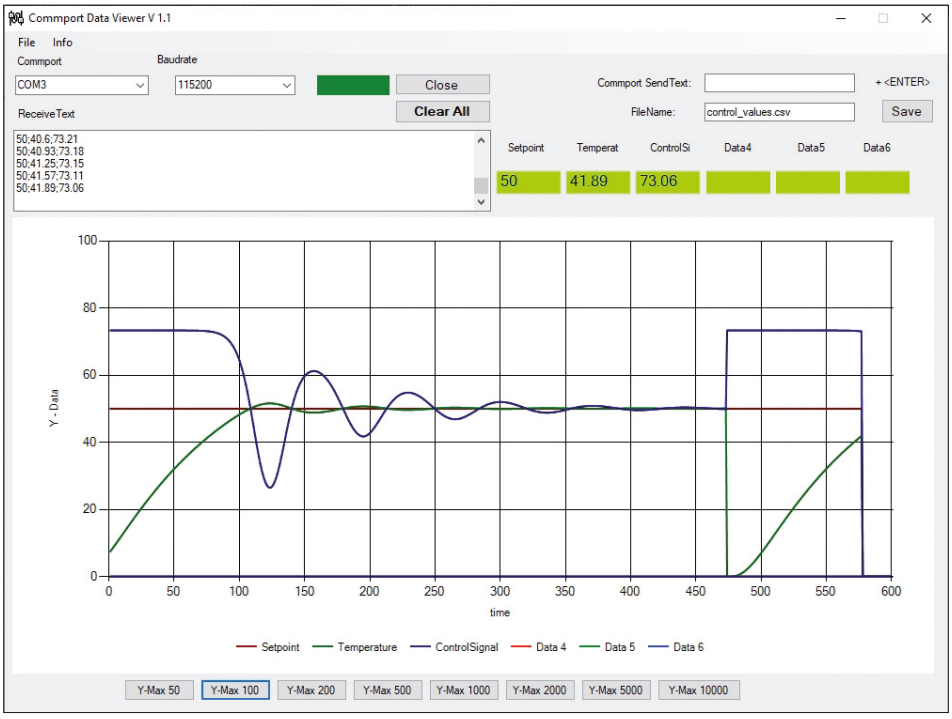


Figure 11.8: ANN controller with PT2 route

In addition, the project 'Codeblocks_C_FuzzyGauss_MembershipOutput' enables the output of the membership functions of the fuzzy Gaussian controller. This provides a deeper insight into how the controller works and its influence on the control characteristic.

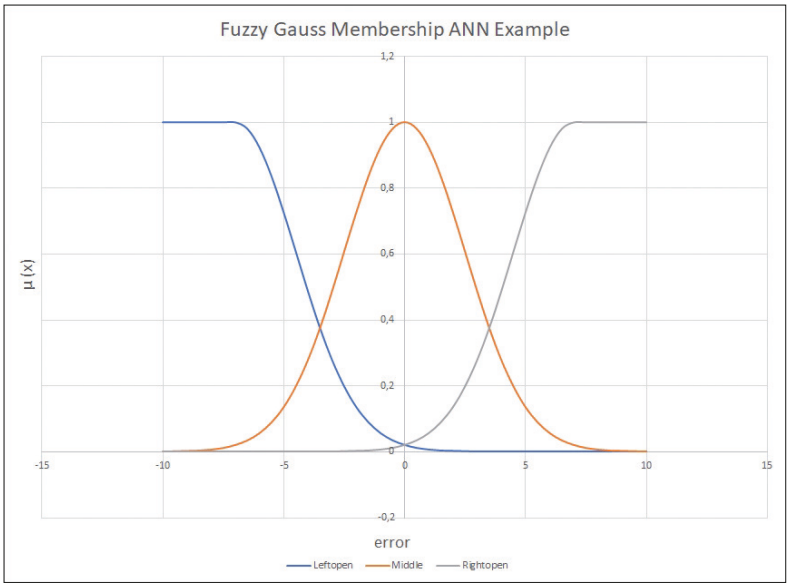


Figure 11.9: ANN Fuzzy Gauss Membership

This project combines the strengths of fuzzy logic with neural networks and modern simulation methods to develop and test innovative and efficient control strategies. It provides a detailed guide to creating, customizing and testing a controller system and demonstrates the practical application of the concepts developed in the Arduino environment.

Listing 11.1: Arduino_ANN_FuzzyGaussExample.ino

```
//---------------------------------------------------------------------
//
//            Arduino Neuronal Net One Hidden layer
//            =====================================
//
//      Control Software for Arduino ESP32 PID Fuzzy Control Board
//      10ms Timer Interrupt Function
//      Fuzzy Gauss Test with weigts from Codeblocks
//
//      Codeblocks_C_FuzzyGauss_ControlCurveAndPT2TestNew
//      Codeblocks_C_FuzzyGauss_TrainNetworkWeights
//
//  Date:     07/08/2024
//  Author:   Josef Bernhardt
//  Hardware: Arduino (ESPRESSIF) ESP32
//
//  File:     Arduino_ANN_FuzzyGaussExample.ino
//---------------------------------------------------------------------
#include <Arduino.h>
```

```
#define MAX_PATTERNS 21
#define MAX_INPUT_NEURONS 1
#define MAX_HIDDEN_NEURONS 5
#define MAX_OUTPUT_NEURONS 1

// Pin Definitions
#define LED_BUILTIN 2 // Pin to which the LED is connected (here GPIO 2 of the
ESP32 )
// LED1
#define LED1 5

typedef struct {
    float inputToHiddenWeights[MAX_INPUT_NEURONS][MAX_HIDDEN_NEURONS];
    float hiddenToOutputWeights[MAX_HIDDEN_NEURONS][MAX_OUTPUT_NEURONS];
} NeuralNetwork;

NeuralNetwork network;
```

Initialize the weights with the trained values from the CodeBlocks program Codeblocks_C_ANN_Fuzzy_Gauss.

```
// Function to initialize the weights of a neural network
void initializeNetworkWithTrainedValues(NeuralNetwork *network)
{
    // Input to Hidden Layer weights
    network->inputToHiddenWeights[0][0]= -0.4119999;
    network->inputToHiddenWeights[0][1]= -0.2670075;
    network->inputToHiddenWeights[0][2]= -0.4113148;
    network->inputToHiddenWeights[0][3]= 0.2680264;
    network->inputToHiddenWeights[0][4]= 0.2677623;
    // Hidden to Output Layer weights
    network->hiddenToOutputWeights[0][0]= 0.6987867;
    network->hiddenToOutputWeights[1][0]= -0.7752706;
    network->hiddenToOutputWeights[2][0]= 1.1442827;
    network->hiddenToOutputWeights[3][0]= 1.0650208;
    network->hiddenToOutputWeights[4][0]= 0.9391873;
}

// tanh if negativ
float tanh_activation(float x) {
    // Compute the hyperbolic tangent (tanh) activation function for the given
input x
    return tanh(x);
}
```

The forwardPropagationFloatValue() function works here as a fuzzy controller. It calculates the output of a neural network for a given input value. First, the input value (control deviation) is multiplied by the weights of the network to calculate the activations in the hidden layer using the tanh activation function. Then the values from the hidden layer are used to calculate the activations in the output layer, again using the tanh activation function. Finally, the result of the first neuron in the output layer is multiplied by 100 and returned.

```
float forwardPropagationFloatValue(NeuralNetwork *network, float input) {
    float hidden[MAX_HIDDEN_NEURONS];
    float output[MAX_OUTPUT_NEURONS];
    // Calculation of the values in the hidden layer
    for (int j = 0; j < MAX_HIDDEN_NEURONS; ++j)
    {
        float netInput = input * network->inputToHiddenWeights[0][j];
        hidden[j] = tanh_activation(netInput);  // Apply activation function
    }
    // Calculation of the values in the output layer
    for (int k = 0; k < MAX_OUTPUT_NEURONS; ++k)
    {
        float netInput = 0.0;
        for (int j = 0; j < MAX_HIDDEN_NEURONS; ++j)
        {
            netInput += hidden[j] * network->hiddenToOutputWeights[j][k];
        }
        output[k] = tanh_activation(netInput);  // Apply activation function
    }
    return output[0]*100.0;  // Return of the result of the output layer
}

// Simulating a PT2 control system
float PT2_Simulation(float controlInput, int reset)
{
    // Parameter
    float deltaT = 1.0;
    float ProportionalGain = 1.0;
    float IntegralGain = 1.0;
    float DerivativeGain = 1.0;
    float TimeConstant1 = 100.0;
    float TimeConstant2 = 8.0;
    float TimeConstant3 = 10.0;

    // State variables
    static float output1 = 0;
    static float output2 = 0;
    static float output3 = 0;
```

```
    // Reset static Variabel
    if (reset == 1)
    {
        output1 = 0;
        output2 = 0;
        output3 = 0;
    }
    // Model equations, Euler integration
    float d_output1 = (ProportionalGain * controlInput - output1) / TimeConstant1;
    float d_output2 = (IntegralGain * output1 - output2) / TimeConstant2;
    float d_output3 = (DerivativeGain * output2 - output3) / TimeConstant3;

    output1 += d_output1 * deltaT;
    output2 += d_output2 * deltaT;
    output3 += d_output3 * deltaT;

    return output3;
}


void setup()
{
    Serial.begin(115200);
    delay(5000);
    pinMode(LED1, OUTPUT);

    initializeNetworkWithTrainedValues(&network);

}
```

A simple temperature control process is simulated in the main loop loop(). First, the value of the PT2 system is reset and the integrated LED is toggled, followed by a short delay of 200 milliseconds. Then a loop begins that runs through 500 iterations, calculating the simulated temperature of the system in each iteration. The control signal for the system is calculated based on the difference between the setpoint and the actual temperature using the forwardPropagationFloatValue() function. Throughout the process, the setpoint, temperature and control signal are continuously output to the serial interface to monitor the progress of the simulation.

```
// Main Loop
void loop()
{
  // Place your main code here to execute repeatedly:
  float setpoint = 50;
  float temperature = 0;
```

```
  float control_signal = 0.0;
  float hysteresis = 0.0;

  // Reset PT2 System
  temperature = PT2_Simulation(0,1);
  // toggle LED
  digitalWrite(LED_BUILTIN, !digitalRead(LED_BUILTIN));
  // wait 200 ms
  delay(200);
  for (int time = 0; time < 500; time++)
  {
      // Simulating the temperature control system
      temperature = PT2_Simulation(control_signal,0);
      // Control signal from the two-point controller

      control_signal = forwardPropagationFloatValue(&network,
setpoint-temperature+4.5);

      Serial.print("Setpoint:");
      Serial.print(setpoint, 1);       // Setpoint with one decimal place
      Serial.print(" Temperature:");
      Serial.print(temperature, 2);    // Temperature with one decimal place
      Serial.print(" ControlSignal:");
      Serial.print(control_signal, 2); // Control signal with one decimal place
      Serial.println(); // New line for the next output
      delay(200);
  }
}
```

11.10 Color sensor evaluation with neural network

Example of pattern recognition with a neural network. Reading a color sensor with the Arduino: Arduino_Nano_TCS74725_Example.ino

The AMS Color Sensor TCS74725 is a sophisticated sensor that communicates with an Arduino Nano via the I2C bus (Inter-Integrated Circuit). On this microcontroller, pins A4 and A5 are used for the I2C connection, which are also labelled as pins 4 (SDA) and 5 (SCL) on the Nano. When using other Arduino models, these pins may vary, so it is important to check the specific I2C pins for each model. The sensor provides a precise measurement of light intensity known as the 'clear' value. This value indicates the total amount of light detected by the sensor without specific filtering by color channels such as red, green or blue. The 'clear' value is particularly useful for determining the general brightness of the environment or the intensity of light sources and thus forms the basis for a variety of applications where light conditions need to be precisely recorded.

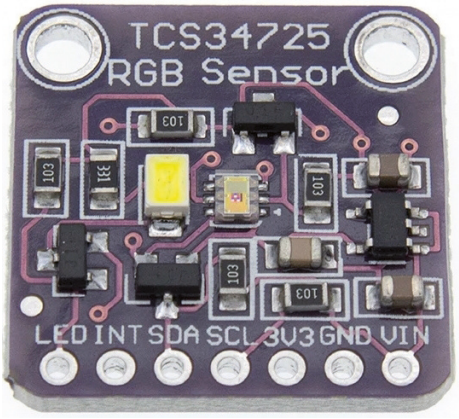


Figure 11.10: AMS Color Sensor Platine

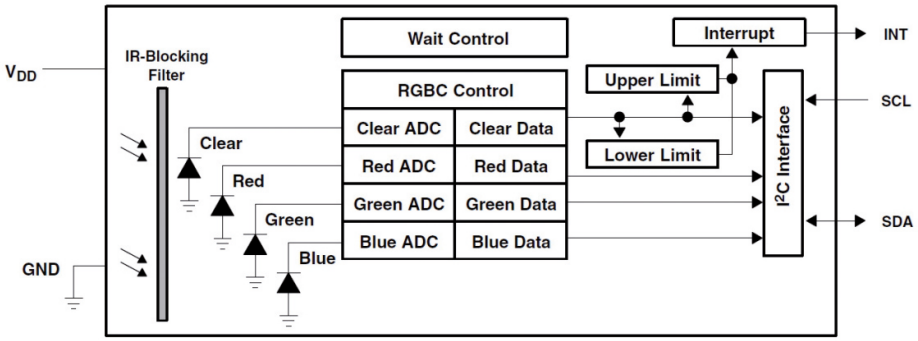


Figure 11.11: Internal sensor circuit

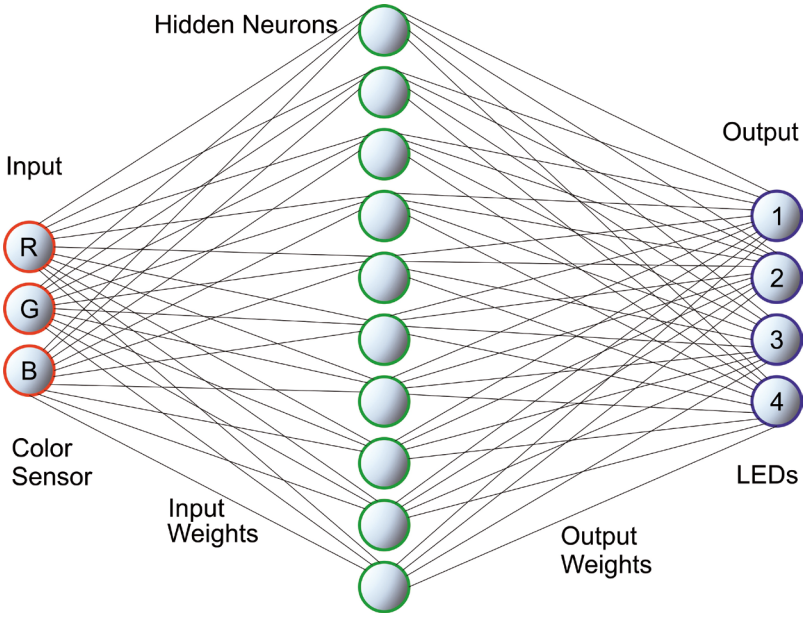


Figure 11.12: ANN network for AMS Color

Program description for the TCS34725 color sensor:
This program controls a microcontroller system that uses a color sensor and a Nextion display to detect and display colors. It initializes the serial communication for debugging and the Nextion display, configures LEDs, relays and buttons, and checks the availability of the color sensor. The color sensor is then initialized and switched on, and initial values are set on the display.

In the main loop, a neural network for color recognition is initialized and trained with predefined color patterns. The color sensor continuously reads the red, green and blue color values. These values are transferred to the neural network, which processes the color values and delivers the results.

The color values and results are displayed both on the serial monitor and on the Nextion display. If no specific color is detected, the message 'NO MATCH' is displayed.

Listing 8.5.2: Arduino_ANN_ColorSensorTCS34725.ino

```
//---------------------------------------------------------------------
//
//            Arduino Color Sensor Reader TCS34725
//            ====================================
//
//      ANN Software for Arduino ESP32 PID Fuzzy Control Board
//      10ms Timer Interrupt Function
//      Init UART2 for Nextion Display
```

```
//      Display RGB Values on Display NX3224F028_001
//      Color Sensor on I2C
//
//      Training of a Neuronal Net with 3 Inputs and 4 Outputs
//
//      Calculating Output with forward Propagation
//
//  Date:     04/08/2024
//  Author:   Josef Bernhardt
//  Hardware: Arduino (ESPRESSIF) ESP32
//
//  File:     Arduino_ANN_ColorSensorTCS34725.ino
//-----------------------------------------------------------------------

#include <Arduino.h>
#include <Wire.h>

// Color Sensor
#define TCS34725_ADDRESS 0x29
#define TCS34725_CONTROL 0x0F
#define TCS34725_ATIME 0x01
#define COMMAND_BIT     (0x80)
#define ENABLE_REGISTER (0x00)
#define ENABLE_PON      (0x01)
#define ENABLE_AEN      (0x02)
#define RGBC_REGISTER   (0x14)

#define I2C_FREQUENCY  400000

// ANN
#define MAX_PATTERNS 4
#define MAX_INPUT_NEURONS 3
#define MAX_HIDDEN_NEURONS 10
#define MAX_OUTPUT_NEURONS 4
#define MAX_EPOCHS 10000
#define LEARNINGRATE 0.5
#define ERRORTHRESHOLD 0.001

int LED_BUILTIN = 2;

// ESP32 UART 2 Pins
#define RXD2 16
#define TXD2 17

// I2C Pins
#define I2CSDA 21
```

```
#define I2CSCL 22

// ESP32 ADC Pins
#define ADC1 32
#define ADC2 33
// Motor Sensor Pins
#define MOTSENS1 15
#define MOTSENS2 34
// Buttons
#define BTN1 14
#define BTN2 27

// Relay and Mosfet (BSP76) Pins
#define RELAIS1 18
#define RELAIS2 19
// LED1
#define LED1 5
// PWM1 and PWM2 Pins
#define PWM1 12
#define PWM2 13
// DAC Outputs 0..5V
#define DAC1 25
#define DAC2 26
// One Wire
#define DS18B20 4

//---------------------------------ANN-------------------------

typedef struct {
    float inputToHiddenWeights[MAX_INPUT_NEURONS][MAX_HIDDEN_NEURONS];
    float hiddenToOutputWeights[MAX_HIDDEN_NEURONS][MAX_OUTPUT_NEURONS];
} NeuralNetwork;

void initializeNetwork(NeuralNetwork *network) {
    for (int i = 0; i < MAX_INPUT_NEURONS; ++i) {
        for (int j = 0; j < MAX_HIDDEN_NEURONS; ++j) {
            network->inputToHiddenWeights[i][j] = ((float)random(1000) / 1000.0)
- 0.5;
        }
    }

    for (int i = 0; i < MAX_HIDDEN_NEURONS; ++i) {
        for (int j = 0; j < MAX_OUTPUT_NEURONS; ++j) {
            network->hiddenToOutputWeights[i][j] = ((float)random(1000) / 1000.0)
- 0.5;
        }
```

```
    }
}

float sigmoid(float x) {
    return 1.0 / (1.0 + exp(-x));
}
```

The forwardPropagation() function calculates the output of the neural network based on the inputs. First, for each neuron in the first hidden layer, the value is calculated by multiplying the inputs by the weights of the network and summing them up. This value is processed by the sigmoid activation function to obtain the activation of the neurons in the hidden layer. These calculated activations are then used as inputs for the next layer. In the next step, the outputs of the network are calculated by multiplying the values of the hidden layer by the weights to the output and adding them up. The sigmoid function is also used here to determine the final output values of the network.

```
void forwardPropagation(NeuralNetwork *network, float input[MAX_INPUT_NEURONS],
float hidden[MAX_HIDDEN_NEURONS], float output[MAX_OUTPUT_NEURONS])
{
    for (int j = 0; j < MAX_HIDDEN_NEURONS; ++j) {
        float netInput = 0.0;
        for (int i = 0; i < MAX_INPUT_NEURONS; ++i) {
            netInput += input[i] * network->inputToHiddenWeights[i][j];
        }
        hidden[j] = sigmoid(netInput);
    }

    for (int k = 0; k < MAX_OUTPUT_NEURONS; ++k) {
        float netInput = 0.0;
        for (int j = 0; j < MAX_HIDDEN_NEURONS; ++j) {
            netInput += hidden[j] * network->hiddenToOutputWeights[j][k];
        }
        output[k] = sigmoid(netInput);
    }
}
```

The backPropagation() function adjusts the weights of a neural network to minimize the difference between the actual and desired outputs. First, the error is calculated for each neuron in the output layer by using the difference between the expected target value and the actual network output. The gradient of the sigmoid activation function is used to update the weights between the hidden layer and the output layer using a fixed learning rate factor (LEARNINGRATE). The error in the neurons of the hidden layer is then calculated by summing the error amounts of the output layer. Again, the gradient of the sigmoid function is used to adjust the weights between the input layer and the hidden layer.

```
void backPropagation(NeuralNetwork *network, float input[MAX_INPUT_NEURONS],
float hidden[MAX_HIDDEN_NEURONS], float output[MAX_OUTPUT_NEURONS], float
targetOutput[MAX_OUTPUT_NEURONS]) {
    // Backpropagation for the output layer
    for (int k = 0; k < MAX_OUTPUT_NEURONS; ++k) {
        float error = targetOutput[k] - output[k];
        // Calculate the error for the output neuron
        float outputGradient = output[k] * (1.0 - output[k]);
        // Update the weights between the hidden layer and the output layer
        for (int j = 0; j < MAX_HIDDEN_NEURONS; ++j) {
            network->hiddenToOutputWeights[j][k] += LEARNINGRATE * error *
outputGradient * hidden[j];
        }
    }
    // Backpropagation for the hidden layer
    for (int j = 0; j < MAX_HIDDEN_NEURONS; ++j) {
        float hiddenError = 0.0;
        // Accumulate the contribution of each output neuron to the hidden layer
error
        for (int k = 0; k < MAX_OUTPUT_NEURONS; ++k) {
            hiddenError += (targetOutput[k] - output[k]) * output[k] * (1.0 -
output[k]) * network->hiddenToOutputWeights[j][k];
        }
        // Calculate the gradient for the hidden neuron using the sigmoid
activation function
        float hiddenGradient = hidden[j] * (1.0 - hidden[j]);
        // Update the weights between the input layer and the hidden layer
        for (int i = 0; i < MAX_INPUT_NEURONS; ++i) {
            network->inputToHiddenWeights[i][j] += LEARNINGRATE * hiddenError *
hiddenGradient * input[i];
        }
    }
}
```

The trainNetwork() function trains a neural network using the backpropagation algorithm. At the beginning, the maximum epochs and the error threshold for convergence are defined. Each training cycle iterates through all training patterns, passing the inputs to the network to calculate the outputs. The error for each pattern is calculated as the mean square error and added to the total error of the epoch. After updating the weights by backpropagation, a check is made to see if the total error is below the threshold; if so, the training is considered successful and terminated. During training, progress reports are regularly sent to the serial interface.

```
void trainNetwork(NeuralNetwork *network, float trainingPatterns[MAX_PATTERNS]
[MAX_INPUT_NEURONS], float targetOutputs[MAX_PATTERNS][MAX_OUTPUT_NEURONS]) {
    // Set the maximum number of training epochs and the error threshold for
convergence.
    int maxEpochs = MAX_EPOCHS;
    float errorThreshold = ERRORTHRESHOLD;
    char s[10] = {0};

    // Iterate through training epochs.
    for (int epoch = 0; epoch < maxEpochs; ++epoch) {
        // Initialize the total error for the current epoch.
        float totalError = 0.0;

        // Iterate through each training pattern.
        for (int p = 0; p < MAX_PATTERNS; ++p) {
            // Initialize arrays to store input, hidden, and output values.
            float input[MAX_INPUT_NEURONS];
            float hidden[MAX_HIDDEN_NEURONS];
            float output[MAX_OUTPUT_NEURONS];

            // Copy the input pattern for the current training example.
            for (int i = 0; i < MAX_INPUT_NEURONS; ++i) {
                input[i] = trainingPatterns[p][i];
            }
            // Perform forward propagation to calculate output values.
            forwardPropagation(network, input, hidden, output);

            // Calculate the error for the current pattern using mean squared
error.
            float patternError = 0.0;
            for (int k = 0; k < MAX_OUTPUT_NEURONS; ++k) {
                patternError += 0.5 * pow(targetOutputs[p][k] - output[k], 2);
            }
            // Accumulate the total error for the epoch.
            totalError += patternError;

            // Perform backpropagation to update the weights.
            backPropagation(network, input, hidden, output, targetOutputs[p]);
        }

        // Check if the total error is below the convergence threshold.
        if (totalError < errorThreshold)
        {
            // printf("Training successful!\n");
            Serial.println("Training successful!\n");
```

```
            // Exit the training loop.
            break;
        }

        // Print training progress every 10 epochs.
        if (epoch % 100 == 0)
        {
            // Display epoch and total error on Serial Monitor
            Serial.print(„Epoch: „);
            Serial.print(epoch);
            Serial.print(„  „);
            Serial.print(„Error: „);
            Serial.println(totalError, 5);  // Outputs the error with 5 decimal
places
            Nextion_Send_Text_to_Textbox("txtT4",String(totalError, 6));

        }
    }
}

//--------------TCS34725 Functions-----------------------

// This function writes a single byte to a specific register on the TCS34725
sensor.
void writeRegister(uint8_t address, uint8_t reg, uint8_t value) {
  Wire.beginTransmission(address);  // Begin I2C transmission to the given
address
  Wire.write(reg);                  // Write the register address
  Wire.write(value);                // Write the value to the register
  Wire.endTransmission();           // End I2C transmission
}

// This function initializes the TCS34725 sensor by setting the integration time,
gain, and enabling the sensor.
void initColorSensor() {
  // Set the integration time (e.g., 153ms)
  writeRegister(TCS34725_ADDRESS, TCS34725_ATIME, 0xC0);
  // Set the gain (e.g., 4x gain)
  writeRegister(TCS34725_ADDRESS, TCS34725_CONTROL, ENABLE_PON);
  // Enable the color sensor (Power ON)
  writeRegister(TCS34725_ADDRESS, ENABLE_REGISTER | COMMAND_BIT, ENABLE_PON);
  delay(3); // Waiting time for stabilisation of the sensor
  writeRegister(TCS34725_ADDRESS, ENABLE_REGISTER | COMMAND_BIT, ENABLE_PON |
ENABLE_AEN);
  delay(10); // Wait for the sensor to stabilize
}
```

The `readColor()` function reads the color data from the RGBC sensor via the I²C interface and saves the values in the transferred variables. First, an I²C transmission to the sensor is initiated and the command to read the color data is sent. The connection is kept open to request the required 8 bytes of sensor data. The received data is stored in a buffer and then the individual color data for red, green, blue and clear is combined into 16-bit values. These values are then written to the transferred variables r,g,b,c.

```
void readColor(uint16_t *r, uint16_t *g, uint16_t *b, uint16_t *c)
{
  Wire.beginTransmission(TCS34725_ADDRESS);                    // Begin I2C
transmission to the sensor
  Wire.write(RGBC_REGISTER | COMMAND_BIT);                     // Write the
command to read the color data
  Wire.endTransmission(false);                                 // End I2C
transmission but keep the connection active
  Wire.requestFrom(TCS34725_ADDRESS, (uint8_t)8, true);        // Request 8
bytes of data from the sensor

  uint8_t buf[8];                                              // Buffer to
hold the data
  for (uint8_t i = 0; i < 8; i++) {
    buf[i] = Wire.read();                        // Read the data into the buffer
  }
  // Combine the two bytes for each color channel into a 16-bit value
  *c = (uint16_t)buf[1] << 8 | buf[0];                         // Clear data
  *r = (uint16_t)buf[3] << 8 | buf[2];                         // Red data
  *g = (uint16_t)buf[5] << 8 | buf[4];                         // Green data
  *b = (uint16_t)buf[7] << 8 | buf[6];                         // Blue data
}

// This function checks if the TCS34725 sensor is available on the I2C bus.
bool isSensorAvailable() {
  Wire.beginTransmission(TCS34725_ADDRESS);  // Begin I2C transmission to the
sensor
  if (Wire.endTransmission() == 0) {         // If the transmission ends with no
error
    return true;                             // The sensor is available
  } else {
    return false;                            // The sensor is not available
  }
}


void TCS_LED_ON()
{
    digitalWrite(RELAIS1,1);
```

```
    digitalWrite(LED1,1);
}

void TCS_LED_OFF()
{
  //Serial.println("OFF");
  digitalWrite(RELAIS1,0);
  digitalWrite(LED1,0);
}

// Sends Data to Nextion Display in a textbox
void Nextion_Send_Text_to_Textbox(char *textbox,String sfloat)
{
        Serial2.write(textbox);
        Serial2.write(".txt=\"");
        Serial2.print(sfloat);
        Serial2.write("\"");
        Serial2.write(0xFF);
        Serial2.write(0xFF);
        Serial2.write(0xFF);
}
```

The ColorCalcNetwork() function calculates the color based on the RGB values that are passed to the neural network as input patterns. First, the input values are copied into an array and the network calculation forwardPropagation() is performed to calculate the outputs. The highest value in the output layer of the network is then checked. According to the highest value, a color name (such as 'NONE', 'RED', 'GREEN' or 'BLUE') is sent to a text box on a Nextion display. If none of the outputs exceeds the threshold value of 0.9, 'NO MATCH' is displayed as a fallback.

```
// Calculate Color from RGB Values
void ColorCalcNetwork(NeuralNetwork *network, float
inputPatterns[MAX_INPUT_NEURONS])
{
      float input[MAX_INPUT_NEURONS];
      float hidden[MAX_HIDDEN_NEURONS];
      float output[MAX_OUTPUT_NEURONS];
      char s[10] = {0};
      for (int i = 0; i < MAX_INPUT_NEURONS; ++i)
      {
         input[i] = inputPatterns[i];
      }
      forwardPropagation(network, input, hidden, output);

      if (output[0] > 0.9)  {
          Nextion_Send_Text_to_Textbox("txtT6","NONE");
```

```
            return;
        }
        if (output[1] > 0.9)  {
            Nextion_Send_Text_to_Textbox("txtT6","RED");
            return;
        }
        if (output[2] > 0.9)  {
            Nextion_Send_Text_to_Textbox("txtT6","GREEN");
            return;
        }
        if (output[3] > 0.9)  {
            Nextion_Send_Text_to_Textbox("txtT6","BLUE");
            return;
        }
        // Fallback
        else {
            Nextion_Send_Text_to_Textbox("txtT6","NO MATCH");
        }
}

// Initializing the hardware
void setup()
{
  // Init USB UART
  Serial.begin(115200);
  // TFT Serial for Nextion Display
  Serial2.begin(9600, SERIAL_8N1, RXD2, TXD2);

  pinMode(LED_BUILTIN, OUTPUT);
  pinMode(LED1, OUTPUT);
  pinMode(RELAIS1, OUTPUT);
  pinMode(RELAIS2, OUTPUT);
  pinMode(BTN1, INPUT);
  pinMode(BTN2, INPUT);
  TCS_LED_OFF();

  // Init Hardware
  Wire.begin(I2CSDA,I2CSCL);

  // Check Color Sensor
  if (!isSensorAvailable()) {
    Serial.println("Color Sensor not available!\n");
    while (1);
  }

  initColorSensor();
```

```
  TCS_LED_ON();

  Nextion_Send_Text_to_Textbox("t1","RED");
  Nextion_Send_Text_to_Textbox("t2","GREEN");
  Nextion_Send_Text_to_Textbox("t3","BLUE");
  Nextion_Send_Text_to_Textbox("t4","CLEAR");
  Nextion_Send_Text_to_Textbox("t5","Color");
  Nextion_Send_Text_to_Textbox("t6","Time");
}
```

The loop() function first initialize a neural network for color recognition and trains it with defined training patterns and target outputs. In a continuous loop, the current color values are read in from a sensor and stored in an array. These color values are then processed by the neural network analysis ColorCalcNetwork() to determine the recognized color. At the same time, the color values and other variables are displayed via the serial interface and on a Nextion display. Finally, the built-in LED is periodically switched on and off to provide visual feedback.

```
void loop()
{
  // Place your main code here to execute repeatedly:
  uint16_t clear, red, green, blue;
  //  init Neuronal Net
  NeuralNetwork network_color;
  // Network with Random Numbers
  initializeNetwork(&network_color);

  float ColorPatterns_color[MAX_INPUT_NEURONS] =  {0, 0, 0};

  float trainingPatterns_color[MAX_PATTERNS][MAX_INPUT_NEURONS] = {{1840, 1820,
1760},  {1920, 750, 680}, {1220, 3500, 1560}, {1650, 6020, 8030}};

  float targetOutputs_color[MAX_PATTERNS][MAX_OUTPUT_NEURONS] = {{ 1, 0, 0, 0 },
{ 0, 1, 0, 0 }, { 0, 0, 1, 0}, { 0, 0, 0, 1 }};

  trainNetwork(&network_color, trainingPatterns_color, targetOutputs_color);

  while(true)
  {
      readColor(&red, &green, &blue, &clear);
      ColorPatterns_color[0] = red;
      ColorPatterns_color[1] = green;
      ColorPatterns_color[2] = blue;
      ColorCalcNetwork(&network_color,ColorPatterns_color);

      // Sends variable to USB UART and Seriell Monitor
```

```
      Serial.print("RED:");
      Serial.print(red);          // Display Color
      Serial.print(" GREEN:");
      Serial.print(green);        // Display Color
      Serial.print(" BLUE:");
      Serial.print(blue);         // Display Color
      Serial.println();           // New line for the next output

      // Sends variable to display
      Nextion_Send_Text_to_Textbox("txtT1",String(red));
      Nextion_Send_Text_to_Textbox("txtT2",String(green));
      Nextion_Send_Text_to_Textbox("txtT3",String(blue));
      // Sends variable to display
      Nextion_Send_Text_to_Textbox("txtT4",String(clear));
      Nextion_Send_Text_to_Textbox("txtT5","");
      // Nextion_Send_Text_to_Textbox("txtT6",String(CounterSeconds));
      // Switching LED
      digitalWrite(LED_BUILTIN,!digitalRead(LED_BUILTIN));
      delay(200);
  }
}
```

11.11 Training a neural network with two hidden layers

This chapter introduces the implementation and training of a neural network with two hidden layers to solve the XOR problem. The neural network is trained to recognize complex patterns in the input data by adjusting the weights so that the output matches the desired targets. The training process involves initializing the weights, which are randomly chosen between -1 and 1 to ensure optimal starting conditions. A key component of the network is forward propagation, where the input data is propagated through the layers of the network to calculate the output.

The forward propagation function performs this calculation and is used in the main programme to determine the network output for different input combinations. The sigmoid activation function and its derivative play a central role in calculating the neuron activations and adjusting the weights during training. The training process is realised by the train_ network function, which trains the network with the input and output patterns of the XOR problem. After the training is completed, the network performance is checked by the test_ network function, which compares the calculated outputs with the expected targets. This method ensures that the network is able to recognize the underlying patterns correctly. The chapter provides a detailed description of the implementation and illustrates the concepts with practical examples and code.

In addition, the trained network can be used to check the results in real time. The inputs are provided by two buttons (`BTN1` and `BTN2`), whose status (`HIGH` or `LOW`) is transferred to the network. The result of the forward propagation is used to control an LED (`LED1`): If the result is above 0.9, the LED is switched on, otherwise it is switched off.

This practical check makes it possible to test and visualize the performance of the network directly on the buttons and the LED.

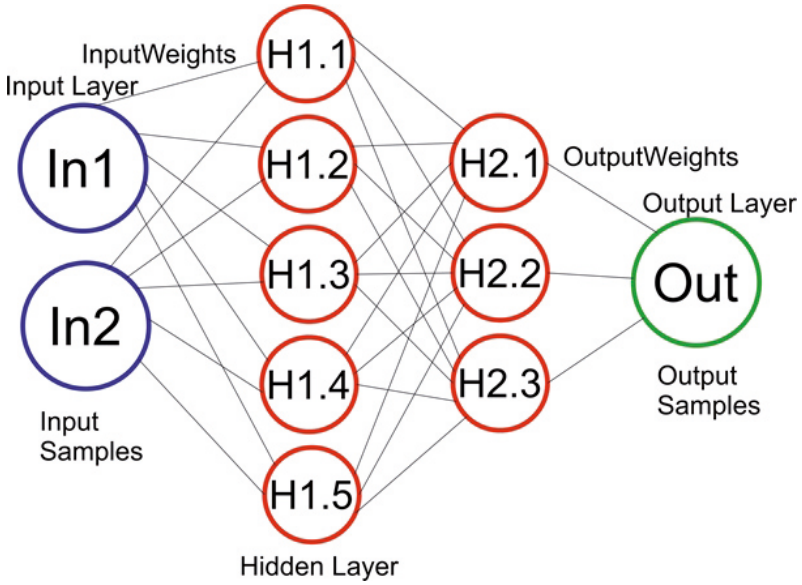


Figure 11.13: ANN with two Hidden Layer

Outputs of the programme:

```
40001 epochs 0.0026922 total_error
41001 epochs 0.0026439 total_error
42001 epochs 0.0025982 total_error
43001 epochs 0.0025545 total_error
44001 epochs 0.0025126 total_error
Training completed after 44312 epochs.
```

Testing the trained network:

```
Input: 0 0, Output: 0.01, Target: 0
Input: 0 1, Output: 0.99, Target: 1
Input: 1 0, Output: 0.99, Target: 1
Input: 1 1, Output: 0.01, Target: 0
```

Hidden Layer 1 Weights:

```
          j=0   |    j=1   |    j=2   |
i=0   0.2481614 | 0.4624176 | 0.4527809 |
i=1   0.2111865 | -0.3501117 | -0.5864201 |
i=2   -0.5525772 | 0.2785741 | -0.0804271 |
i=3   4.7717438 | -4.9274030 | -2.6861346 |
i=4   4.4282727 | -4.1995506 | 2.2122681 |
```

Hidden Layer 2 Weights:

```
      j=0   |   j=1   |   j=2   |   j=3   |   j=4   |   j=5   |
i=0   0.0198693 | 0.6823708 | 0.0334172 | 4.9224834 | -4.2964525 | 1.5263215 |
i=1   -0.3436544 | 0.2543070 | 0.1580395 | 0.1180307 | -2.0611918 | 0.6064123 |
i=2   -1.7025173 | 0.7684172 | -1.1430445 | -4.1578183 | 3.8163869 | -0.2331006 |
```

Output Layer Weights:

```
      j=0   |   j=1   |   j=2   |   j=3   |
i=0   7.0260234 | 1.6490515 | -6.5299678 | -0.4783683 |
```

In extension, the trained network can be used to check the results in real time. The inputs are provided by two buttons (BTN1 and BTN2), whose status (HIGH or LOW) is transferred to the network. The result of the forward propagation is used to control an LED (LED1): If the result is above 0.9, the LED is switched on, otherwise it is switched off. This practical check makes it possible to test and visualize the performance of the network directly on the buttons and the LED.

Listing 11.3: Arduino_ANN_TwoHiddenLayer.ino

```
//----------------------------------------------------------------------
//
//            Arduino Neuronal Net
//            ====================
//
//      Testprogramm for Arduino ESP32 PID Fuzzy Control Board
//      Neuronal Net with 2 Hidden Layers and XOR
//
//  Date:     29/06/2024
//  Author:   Josef Bernhardt
//  Hardware: Arduino (ESPRESSIF) ESP32
//
//  File:     Arduino_ANN_TwoHiddenLayer.ino
//----------------------------------------------------------------------
```

This programme for the Arduino implements a neural network with two hidden layers to solve the XOR problem.

```
#include <math.h>

#define INPUT_PATTERN 4
#define INPUT_SIZE 2
#define HIDDEN1_SIZE 5
#define HIDDEN2_SIZE 3
#define OUTPUT_SIZE 1
#define LEARNING_RATE 0.1
#define MAX_EPOCHS 50000
```

```
#define ERROR_THRESHOLD 0.01

// Buttons
#define BTN1 14
#define BTN2 27

// Relay and Mosfet (BSP76) Pins
#define RELAIS1 18
#define RELAIS2 19
// LED1
#define LED1 5
#define LED_BUILTIN 2
// Neural network weights
float hidden1_weights[HIDDEN1_SIZE][INPUT_SIZE + 1];
float hidden2_weights[HIDDEN2_SIZE][HIDDEN1_SIZE + 1];
float output_weights[OUTPUT_SIZE][HIDDEN2_SIZE + 1];

// Training with XOR example
float inputs[INPUT_PATTERN][INPUT_SIZE] = {{0, 0}, {0, 1}, {1, 0}, {1, 1}};
float targets[INPUT_PATTERN] = {0, 1, 1, 0};

// Function prototypes
float sigmoid(float x);
float sigmoid_derivative(float x);
void initialize_weights();
void train_network(float inputs[INPUT_PATTERN][INPUT_SIZE], float
targets[INPUT_PATTERN]);
float forward_propagation_pins(float input1, float input2);
void test_network(float inputs[INPUT_PATTERN][INPUT_SIZE], float
targets[INPUT_PATTERN]);
void forward_propagation(float input[INPUT_SIZE], float hidden1[HIDDEN1_SIZE],
float hidden2[HIDDEN2_SIZE], float output[OUTPUT_SIZE]);
void backpropagation(float input[INPUT_SIZE], float hidden1[HIDDEN1_SIZE], float
hidden2[HIDDEN2_SIZE], float output[OUTPUT_SIZE], float target);
void print_weights(float hidden1_weights[HIDDEN1_SIZE][INPUT_SIZE + 1],
                   float hidden2_weights[HIDDEN2_SIZE][HIDDEN1_SIZE + 1],
                   float output_weights[OUTPUT_SIZE][HIDDEN2_SIZE + 1]);
```

The setup() function is executed once when the program is started to carry out the necessary configuration. Firstly, serial communication is activated with a baud rate of 115200 in order to output debug information. The pins for the internal LED, an additional LED, two relays and two buttons are then configured. The weights of the neural network are then initialized, and the network is trained with the input data and target values provided. The trained network is then tested, and the results are displayed. Finally, the weight values of the various network layers are output to the serial console to check the configuration after training.

```
void setup() {
  Serial.begin(115200);
  // Initialize weights

  pinMode(LED_BUILTIN, OUTPUT);
  pinMode(LED1, OUTPUT);
  pinMode(RELAIS1, OUTPUT);
  pinMode(RELAIS2, OUTPUT);
  pinMode(BTN1, INPUT);
  pinMode(BTN2, INPUT);

  initialize_weights();
  train_network(inputs, targets);

  // Testing the trained network
  test_network(inputs, targets);

  // Display Weights
  print_weights(hidden1_weights, hidden2_weights, output_weights);

}
```

The loop() function continuously tests a neural network and outputs the results to the serial console. First, the function forward_propagation_pins() is called four times with the input combinations of the XOR problem (0,0; 0,1; 1,0; 1,1). The calculated outputs of the network are displayed together with the input values on the serial console. The states of two buttons (pins BTN1 and BTN2) are then read and transferred to the neural network as inputs. The result of the network then controls an LED: if the value is above 0.9, the LED is switched on, otherwise it remains off. At the end of each iteration, there is a delay of 500 milliseconds before the cycle starts again. This enables continuous checking and display of the network outputs as well as simple visualization of the results via the LED.

```
void loop() {

  float result;

  // Testoutput
  result = forward_propagation_pins(0, 0);
  Serial.print("IN1: ");Serial.print(0);Serial.print(" IN2: ");Serial.
print(0);Serial.print(" OUT1: ");Serial.println(result,3);

  result = forward_propagation_pins(0, 1);
  Serial.print("IN1: ");Serial.print(0);Serial.print(" IN2: ");Serial.
print(1);Serial.print(" OUT1: ");Serial.println(result,3);

  result = forward_propagation_pins(1, 0);
```

```
  Serial.print("IN1: ");Serial.print(1);Serial.print(" IN2: ");Serial.
print(0);Serial.print(" OUT1: ");Serial.println(result,3);

  result = forward_propagation_pins(1, 1);
  Serial.print("IN1: ");Serial.print(1);Serial.print(" IN2: ");Serial.
print(1);Serial.print(" OUT1: ");Serial.println(result,3);

  bool btn1 = digitalRead(BTN1);
  bool btn2 = digitalRead(BTN2);

  result = forward_propagation_pins(btn1, btn2);

  if (result > 0.9) digitalWrite(LED1,1); else digitalWrite(LED1,0);

  Serial.println();

  delay(500);

}
```

The sigmoid activation function and its derivative are used to calculate the output of the neurons.

```
// Sigmoid activation function
float sigmoid(float x) {
  return 1 / (1 + exp(-x));
}

// Derivative of sigmoid activation function
float sigmoid_derivative(float x) {
  return x * (1 - x);
}
```

The weights of the neural network are initialized with random numbers, which is a proven method to prevent the network from getting stuck in a suboptimal solution.

```
// Initialize weights with random values between -1 and 1
void initialize_weights() {
  int i, j;
  for (i = 0; i < HIDDEN1_SIZE; i++) {
    for (j = 0; j < INPUT_SIZE + 1; j++) {
      hidden1_weights[i][j] = ((float)rand() / RAND_MAX) * 2 - 1;
    }
  }
  for (i = 0; i < HIDDEN2_SIZE; i++) {
    for (j = 0; j < HIDDEN1_SIZE + 1; j++) {
```

```
      hidden2_weights[i][j] = ((float)rand() / RAND_MAX) * 2 - 1;
    }
  }
  for (i = 0; i < OUTPUT_SIZE; i++) {
    for (j = 0; j < HIDDEN2_SIZE + 1; j++) {
      output_weights[i][j] = ((float)rand() / RAND_MAX) * 2 - 1;
    }
  }
}
```

Training is performed by the train_network() function, which trains the network with the specified input and output patterns. After training, the network is tested with the test_network() function, which outputs the network output for the specified input patterns.

```
// Train the neural network
void train_network(float inputs[INPUT_PATTERN][INPUT_SIZE], float targets[INPUT_
PATTERN]) {
  int epoch, i;
  for (epoch = 0; epoch < MAX_EPOCHS; epoch++) {
    float total_error = 0;
    for (i = 0; i < INPUT_PATTERN; i++) {
      float hidden1[HIDDEN1_SIZE];
      float hidden2[HIDDEN2_SIZE];
      float output[OUTPUT_SIZE];
      forward_propagation(inputs[i], hidden1, hidden2, output);
      backpropagation(inputs[i], hidden1, hidden2, output, targets[i]);
      total_error += fabs(targets[i] - output[0]);
     }
    total_error /= INPUT_PATTERN; // Average error per example

      if (epoch % 1000 == 0)
      {
        // Testoutput
        Serial.print(epoch + 1);
        Serial.print(" epochs ");

        Serial.print(total_error / INPUT_PATTERN,7);
        Serial.println(" total_error");
        digitalWrite(LED_BUILTIN,!digitalRead(LED_BUILTIN));
     }

    if (total_error < ERROR_THRESHOLD) {
      Serial.print("Training completed after ");
      Serial.print(epoch + 1);
      Serial.println(„ epochs.");
      break;
```

```
    }
  }
  digitalWrite(LED_BUILTIN,0);
}
```

The forward_propagation_pins() function calculates the output of a neural network by implementing forward propagation. First, the input value is propagated through the first hidden layer (Hidden Layer 1) by multiplying the input by the weights and then adding a bias. The result is then passed through the sigmoid activation function to calculate the activations in this layer. In the next step, the second hidden layer (Hidden Layer 2) is calculated by multiplying the activations from the first hidden layer by the weights and adding a bias. Again, the sigmoid function is applied to the result to calculate the activations of the second hidden layer. Finally, the output layer is calculated by multiplying the activations from the second hidden layer by the output weights and adding a bias. The result of the output layer is processed by the sigmoid function and returned as output. This function implements the forward pass of the neural network to calculate an output for given inputs.

```
float forward_propagation_pins(float input1, float input2) {
  int i, j;
  float hidden1[HIDDEN1_SIZE];
  float hidden2[HIDDEN2_SIZE];
  float output[OUTPUT_SIZE];
  float sum = 0;


  // Compute Hidden Layer 1
  for (i = 0; i < HIDDEN1_SIZE; i++) {
    float sum = input1 * hidden1_weights[i][0] + input2 * hidden1_weights[i][1] +
hidden1_weights[i][2]; // Bias
    hidden1[i] = sigmoid(sum);
  }
  // Compute Hidden Layer 2
  for (i = 0; i < HIDDEN2_SIZE; i++) {
    float sum = 0;
    for (j = 0; j < HIDDEN1_SIZE; j++) {
      sum += hidden1[j] * hidden2_weights[i][j];
    }
    sum += hidden2_weights[i][HIDDEN1_SIZE]; // Bias
    hidden2[i] = sigmoid(sum);
  }
  // Compute Output Layer
  for (i = 0; i < OUTPUT_SIZE; i++) {
    sum = 0;
    for (j = 0; j < HIDDEN2_SIZE; j++) {
      sum += hidden2[j] * output_weights[i][j];
    }
```

```
    sum += output_weights[i][HIDDEN2_SIZE]; // Bias
    //output[i] = sigmoid(sum);
  }
  return sigmoid(sum);
}
```

The test_network() function tests the neural network using a series of input patterns and targets. First, the input patterns are passed through the network in sequence by calling the forward_propagation() function to calculate the output of the network for each input pattern. The intermediate results are stored in the hidden layers before the final output is generated. Once the output has been calculated, the input pattern, the calculated output and the corresponding target value are output via the serial interface to enable a comparison between the actual and expected values. This function is useful to check how well the network responds to the given input patterns after training and whether it can predict the correct target values.

```
// Test the neural network
void test_network(float inputs[INPUT_PATTERN][INPUT_SIZE], float targets[INPUT_
PATTERN]) {
  int i;
  Serial.println("Testing the trained network:");
  for (i = 0; i < INPUT_PATTERN; i++) {
    float hidden1[HIDDEN1_SIZE];
    float hidden2[HIDDEN2_SIZE];
    float output[OUTPUT_SIZE];
    forward_propagation(inputs[i], hidden1, hidden2, output);
    Serial.print("Input: ");
    Serial.print((int)inputs[i][0]);
    Serial.print(" ");
    Serial.print((int)inputs[i][1]);
    Serial.print(", Output: ");
    Serial.print(output[0], 2);
    Serial.print(", Target: ");
    Serial.println((int)targets[i]);
  }
}
```

First, the first hidden layer (Hidden Layer 1) is calculated by multiplying the input values by the corresponding weights and then adding a bias. The result is then sent through the sigmoid activation function to calculate the activation of the neurons in the first hidden layer. The second hidden layer (Hidden Layer 2) is then calculated by multiplying the activations of the first hidden layer by the weights of Hidden Layer 2 and adding a bias. Again, the sigmoid function is applied to calculate the activation of the neurons in the second hidden layer. Finally, the output layer is calculated by multiplying the activations of the second hidden layer by the output weights and adding a bias. The result is sent through the sigmoid activation function to calculate the final output values of the network.

```
// Perform forward propagation through the neural network
void forward_propagation(float input[INPUT_SIZE], float hidden1[HIDDEN1_SIZE],
float hidden2[HIDDEN2_SIZE], float output[OUTPUT_SIZE]) {
  int i, j;
  // Compute Hidden Layer 1
  for (i = 0; i < HIDDEN1_SIZE; i++) {
    float sum = 0;
    for (j = 0; j < INPUT_SIZE; j++) {
      sum += input[j] * hidden1_weights[i][j];
    }
    sum += hidden1_weights[i][INPUT_SIZE]; // Bias
    hidden1[i] = sigmoid(sum);
  }
  // Compute Hidden Layer 2
  for (i = 0; i < HIDDEN2_SIZE; i++) {
    float sum = 0;
    for (j = 0; j < HIDDEN1_SIZE; j++) {
      sum += hidden1[j] * hidden2_weights[i][j];
    }
    sum += hidden2_weights[i][HIDDEN1_SIZE]; // Bias
    hidden2[i] = sigmoid(sum);
  }
  // Compute Output Layer
  for (i = 0; i < OUTPUT_SIZE; i++) {
    float sum = 0;
    for (j = 0; j < HIDDEN2_SIZE; j++) {
      sum += hidden2[j] * output_weights[i][j];
    }
    sum += output_weights[i][HIDDEN2_SIZE]; // Bias
    output[i] = sigmoid(sum);
  }
}
```

The backpropagation() function updates the weights of a neural network based on the error between the calculated output and the target value. First, the error in the output layer is calculated by multiplying the difference between the target value and the output by the derivative of the sigmoid activation function. Then the error for the second hidden layer (Hidden Layer 2) is calculated by tracing the error of the output layer back to the neurons of the second hidden layer. The error for the first hidden layer (hidden layer 1) is then calculated by transferring the error from hidden layer 2 to hidden layer 1. After the errors in all layers have been calculated, the weights for the output layer are updated by adjusting them proportionally to the error and the activations of the previous layer. Then the weights for hidden layer 2 are updated in a similar way. Finally, the weights for hidden layer 1 are updated based on the input value and the calculated error of the first hidden layer. In each layer, the bias weight is also adjusted accordingly. The function thus implements the backpropagation algorithm to train the neural network.

```
// Perform backpropagation to update weights based on error
void backpropagation(float input[INPUT_SIZE], float hidden1[HIDDEN1_SIZE], float
hidden2[HIDDEN2_SIZE], float output[OUTPUT_SIZE], float target) {
  int i, j;
  float output_error[OUTPUT_SIZE];
  float hidden2_error[HIDDEN2_SIZE];
  float hidden1_error[HIDDEN1_SIZE];

  // Compute error in Output Layer
  for (i = 0; i < OUTPUT_SIZE; i++) {
    output_error[i] = (target - output[i]) * sigmoid_derivative(output[i]);
  }

  // Compute error in Hidden Layer 2
  for (i = 0; i < HIDDEN2_SIZE; i++) {
    float sum = 0;
    for (j = 0; j < OUTPUT_SIZE; j++) {
      sum += output_error[j] * output_weights[j][i];
    }
    hidden2_error[i] = sum * sigmoid_derivative(hidden2[i]);
  }

  // Compute error in Hidden Layer 1
  for (i = 0; i < HIDDEN1_SIZE; i++) {
    float sum = 0;
    for (j = 0; j < HIDDEN2_SIZE; j++) {
      sum += hidden2_error[j] * hidden2_weights[j][i];
    }
    hidden1_error[i] = sum * sigmoid_derivative(hidden1[i]);
  }

  // Update weights for Output Layer
  for (i = 0; i < OUTPUT_SIZE; i++) {
    for (j = 0; j < HIDDEN2_SIZE; j++) {
      output_weights[i][j] += LEARNING_RATE * output_error[i] * hidden2[j];
    }
    output_weights[i][HIDDEN2_SIZE] += LEARNING_RATE * output_error[i]; // Bias
  }

  // Update weights for Hidden Layer 2
  for (i = 0; i < HIDDEN2_SIZE; i++) {
    for (j = 0; j < HIDDEN1_SIZE; j++) {
      hidden2_weights[i][j] += LEARNING_RATE * hidden2_error[i] * hidden1[j];
    }
    hidden2_weights[i][HIDDEN1_SIZE] += LEARNING_RATE * hidden2_error[i]; // Bias
  }
```

```
  // Update weights for Hidden Layer 1
  for (i = 0; i < HIDDEN1_SIZE; i++) {
    for (j = 0; j < INPUT_SIZE; j++) {
      hidden1_weights[i][j] += LEARNING_RATE * hidden1_error[i] * input[j];
    }
    hidden1_weights[i][INPUT_SIZE] += LEARNING_RATE * hidden1_error[i]; // Bias
  }
}
```

The function print_weights() outputs the weights of the neural network in a structured form via the serial interface. First, the weights of the first hidden layer (Hidden Layer 1) are displayed. The indices of the input neurons (including the bias) are displayed in the header, followed by the respective weight values for each neuron in this layer. The same process is then performed for the second hidden layer (Hidden Layer 2). Here, the weight values for each connection between the neurons of the first and second hidden layer, including the bias, are also output. Finally, the weights of the output layer are displayed in the same way, showing the connections between the second hidden layer and the output layer. This feature facilitates debugging by visualizing the current weight values of the neural network in a readable form.

```
void print_weights(float hidden1_weights[HIDDEN1_SIZE][INPUT_SIZE + 1],
                   float hidden2_weights[HIDDEN2_SIZE][HIDDEN1_SIZE + 1],
                   float output_weights[OUTPUT_SIZE][HIDDEN2_SIZE + 1]) {
  int i, j;

  Serial.println("Hidden Layer 1 Weights:");
  Serial.print("     ");
  for (j = 0; j < INPUT_SIZE + 1; j++) {
    Serial.print("     j=");
    Serial.print(j);
    Serial.print("   |");
  }
  Serial.println();
  for (i = 0; i < HIDDEN1_SIZE; i++) {
    Serial.print("i=");
    Serial.print(i);
    Serial.print("  ");
    for (j = 0; j < INPUT_SIZE + 1; j++) {
      Serial.print(" ");
      Serial.print(hidden1_weights[i][j], 7);
      Serial.print(" |");
    }
    Serial.println();
  }
```

```
  Serial.println("\nHidden Layer 2 Weights:");
  Serial.print("     ");
  for (j = 0; j < HIDDEN1_SIZE + 1; j++) {
    Serial.print("     j=");
    Serial.print(j);
    Serial.print("   |");
  }
  Serial.println();
  for (i = 0; i < HIDDEN2_SIZE; i++) {
    Serial.print("i=");
    Serial.print(i);
    Serial.print("  ");
    for (j = 0; j < HIDDEN1_SIZE + 1; j++) {
      Serial.print(" ");
      Serial.print(hidden2_weights[i][j], 7);
      Serial.print(" |");
    }
    Serial.println();
  }

  Serial.println("\nOutput Layer Weights:");
  Serial.print("     ");
  for (j = 0; j < HIDDEN2_SIZE + 1; j++) {
    Serial.print("     j=");
    Serial.print(j);
    Serial.print("   |");
  }
  Serial.println();
  for (i = 0; i < OUTPUT_SIZE; i++) {
    Serial.print("i=");
    Serial.print(i);
    Serial.print("  ");
    for (j = 0; j < HIDDEN2_SIZE + 1; j++) {
      Serial.print(" ");
      Serial.print(output_weights[i][j], 7);
      Serial.print(" |");
    }
    Serial.println();
  }
}
```

Chapter 12 • Fuzzy controller for feeding electricity into the domestic grid

12.1 Introduction

To use solar power generated during the day efficiently and make it available at night, here is a low-cost solution using standard components and an ESP32 (Arduino) controller. This solution makes it possible to store excess solar power in a LiFePO4 battery, which is then available during the night hours. With the help of a PID or fuzzy controller, the power is precisely controlled and fed into the grid. The current output of the three phases is continuously monitored and the micro-inverter is controlled according to the calculated output value to ensure optimum feed-in.

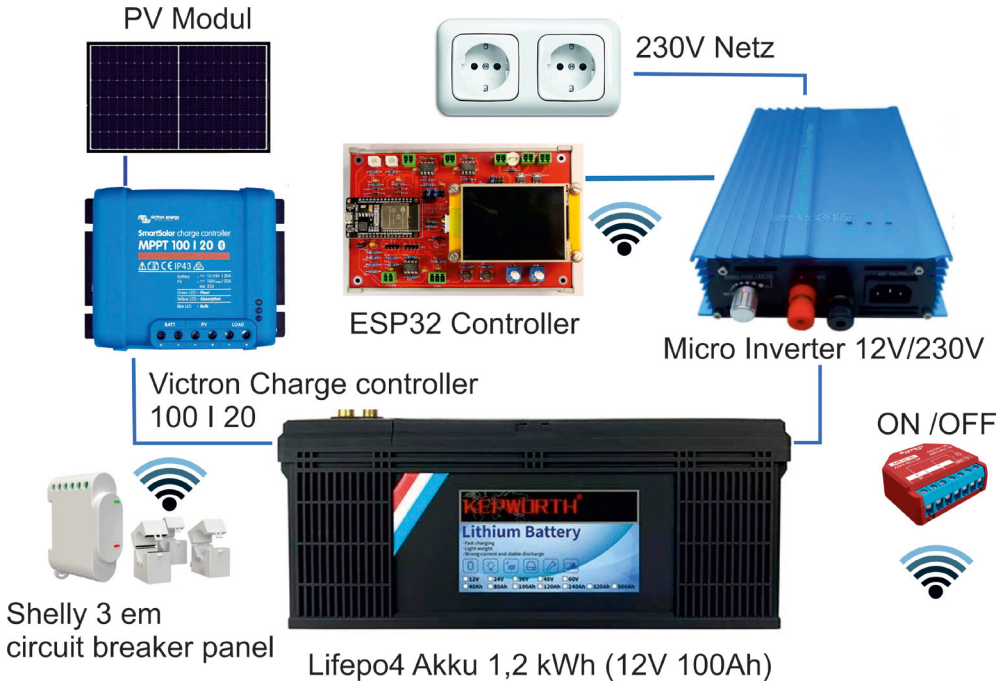


Figure 12.1: System for zero feed-in

12.2 Shelly 3EM Energy Meter

The actual power is recorded with a Shelly 3EM power meter:



Figure 12.2: Shelly 3EM WLAN energy meter

The Shelly 3EM is a powerful energy meter that has been specially developed for three-phase power grids. It enables real-time monitoring of energy consumption in households and industrial applications. With WLAN support, the Shelly 3EM can be easily integrated into the internal network. The measuring accuracy of the device is 1%, which ensures precise recording of power consumption. It can measure up to 120 A per phase, making it ideal for various applications. The Shelly 3EM also supports the monitoring of solar power generation. Users can view current consumption and historical data via the mobile app. The Shelly 3EM offers a reliable solution for monitoring energy consumption in real time and enables easy integration into modern smart home environments. The Technische Daten des Shelly 3EM device also enables consumers to be switched.

Technical data	Values
Power supply	110-230V ±10%, 50/60Hz
Maximum current per phase	120 A
WLAN	802.11 b/g/n @ 2.4 GHz
Accuracy	±1%
Operating humidity	30% - 85%
Operating temperature	-40°C bis 40°C
Maximum load	3x10A pro Kanal
Communication protocols	HTTP, MQTT, CoAP, TCP/UDP
App support	Shelly App (iOS, Android)
Dimensions	73 mm x 57 mm x 22 mm

Table 12.1: Technical data of the Shelly 3EM:

The measured values are queried via this http request for the individual three phases:

http://192.168.178.40/emeter/0
http://192.168.178.40/emeter/1
http://192.168.178.40/emeter/2

The Shelly 3EM responds t this answer for phase 0:


```
{
    "power": 30.98,
    "pf": 0.66,
    "current": 0.2,
    "voltage": 228.99,
    "is_valid": true,
    "total": 661247.1,
    "total_returned": 0
}
```


The Shelly 3EM can be read out completely with the following http request:

192.168.178.40/status

Here is an extract of the answer with the most important data:


```
     "emeters": [
        {
            "power": 21.97,
            "pf": 0.6,
            "current": 0.16,
            "voltage": 230.87,
            "is_valid": true,
            "total": 661262.7,
            "total_returned": 0
        },
        {
            "power": -299.66,
            "pf": -0.97,
            "current": 1.33,
            "voltage": 231.29,
            "is_valid": true,
            "total": 431236.4,
            "total_returned": 569726.4
        },
        {
            "power": -26.34,
            "pf": -0.29,
            "current": 0.39,
            "voltage": 230.34,
            "is_valid": true,
            "total": 1379409.6,
            "total_returned": 327835.9
        }
    ]
```

12.3 Measurements with the Shelly Plus 1PM

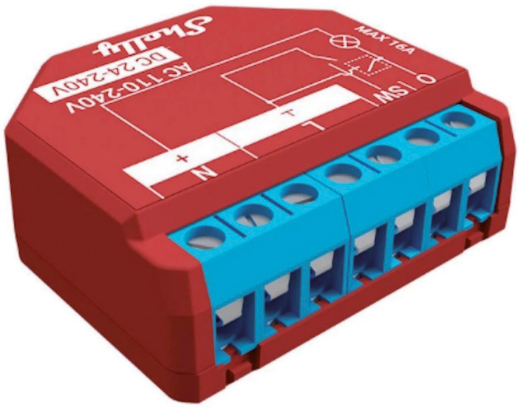


Figure 12.3: Shelly Plus 1PM Switch

The Shelly Plus 1PM is a compact, intelligent smart home switch that has been specially developed for monitoring and controlling electrical devices with a voltage of 230V. It can switch up to 16 A of current and offers precise measurement of voltage, current and power as well as work (kWh). With support for Wi-Fi connections, the Shelly Plus 1PM can be easily integrated into existing smart home systems and can be controlled via a user-friendly app or web interface. The switch is also able to create automatic controls and schedules to optimize energy consumption and increase comfort. It offers protection functions such as overload and overvoltage protection to ensure the safety of the connected devices. Its compact design allows it to be installed in standard EU switch boxes, making it easy to integrate into existing electrical systems. The Shelly Plus 1PM also supports http request to read out the measured values with an ESP32. An example for reading out several Shellys can be found in the Arduino_ESP32_Shelly1PMPlusReader directory.

Property	Value
Model	Shelly Plus 1PM
Type	Smart switch with energy measurement
Max. Switch Current	16 A
Max. Switching Power	3680 W (at 230 V)
Max. Voltage	110-230 V AC (Alternating Current)
Current Measurement Range	0-16 A
Energy Consumption Measurement	Yes, including kWh and power data
Power Factor Measurement	Yes
Total Energy Consumption	Up to 999,999 kWh
Switchable Load	16 A / 3680 W
WLAN	IEEE 802.11 b/g/n

Property	Value
WLAN Frequency	2.4 GHz
Operating Conditions	-10°C to 40°C
Dimensions	42 x 42 x 17 mm
Mounting Type	Flush mount
Compatibility	iOS, Android (Shelly App)
Control	App, Web interface, HTML Requests
Energy Monitoring	Real-time and historical consumption data
Integration	Amazon Alexa, Google Assistant
Connectivity	WLAN, HTTP, MQTT
Safety Features	Overload protection, Surge protection
Status Indicators	LED status indicator
Certifications	CE, RoHS

Table 11.2: Technische Daten des Shelly 1PM Plus:

This is an HTML call to read out the measured values:

```
// URLs für die Shelly Plus 1PM-Geräte
String serverNamePM1 = "http://192.168.178.63/rpc/Shelly.GetStatus";
```

You can test this very easily with a browser by entering the following line as the URL:

http://192.168.178.63/rpc/Shelly.GetStatus

The response is an extensive JSON string containing the measured values.

Listing 11.1: Shelly1PMPlus reading:

```
// Implementation of the handleShellyPlusData function
void handleShellyPlusData(String serverPath, float* PowerArr, float* byMinuteArr,
String onColor, String offColor)
{
  HTTPClient http;
  // Connect to the specified URL
  http.begin(serverPath.c_str());
  // Send HTTP GET request
  int httpResponseCode = http.GET();
  if (httpResponseCode > 0) {
    // Store the response as a string
    String payload = http.getString();
    // Process JSON string
```

```
    DynamicJsonDocument jsonDoc(2048);
    DeserializationError err = deserializeJson(jsonDoc, payload);
    if (err) {
      Serial.print("Error parsing: ");
      Serial.println(err.c_str());
      http.end();  // Free resources
      return;
    }
    // Extract JSON values
    JsonObject aenergy = jsonDoc["switch:0"]["aenergy"];
    JsonObject temperatur = jsonDoc["switch:0"]["temperature"];
    // Fill PowerArr with relevant values
    PowerArr[0] = jsonDoc["switch:0"]["voltage"];       // Voltage
    PowerArr[1] = jsonDoc["switch:0"]["current"];       // Current
    PowerArr[2] = jsonDoc["switch:0"]["apower"];        // Power
    PowerArr[3] = aenergy["total"];                     // Total energy
    PowerArr[4] = temperatur["tC"];                     // Temperature in Celsius
    // Extract by_minute array and store it in the passed byMinuteArr array
    JsonArray byMinute = aenergy["by_minute"];
    for (size_t i = 0; i < byMinute.size() && i < 3; i++) {  // Assumption: max.
3 elements in the array
      byMinuteArr[i] = byMinute[i];
    }
    // Change background color based on switch state (not relevant for this demo)
    bool StateSW = jsonDoc["switch:0"]["output"];
    if (StateSW) {
      // Nextion_Send_Color_to_Textbox(onColor);   // If the switch is on
    } else {
      // Nextion_Send_Color_to_Textbox(offColor);  // If the switch is off
    }
  } else {
    // Error handling for the HTTP request
    Serial.print("Error code: ");
    Serial.println(httpResponseCode);
  }
  // Free resources
  http.end();
}
```

Here is the output for a Shelly:

```
Reading data from PM1 Shelly:
PM1 Shelly Daten:   Voltage: 227.10 V   Current: 1.307 A   Power: 293.00 W
Total Energy: 841.387 kWh   Temperature: 54.00 °C
  by_minute:     Value 1: 0.271 Wh     Value 2: 1.351 Wh     Value 3: 1.342 Wh
```

12.4 Charge controller - Victron Smart Solar 100 I 20

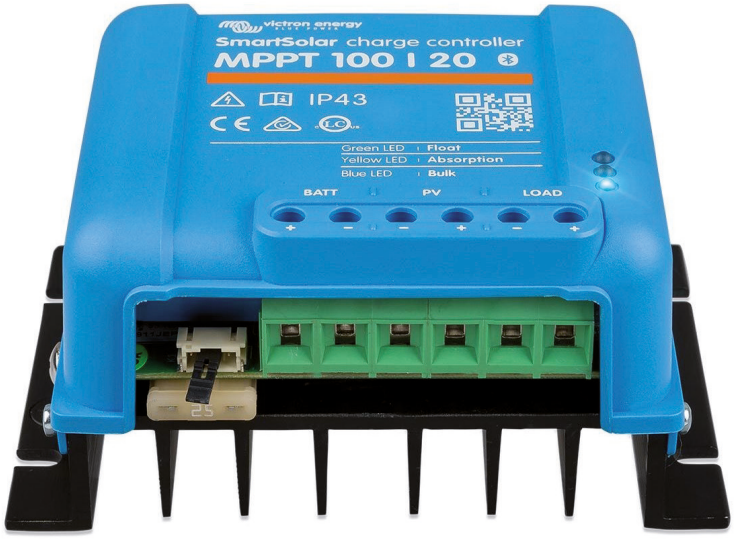


Figure 12.4: Victron Smart Solar charge controller

The Victron SmartSolar MPPT 100/20 is a powerful and efficient charge controller designed to control and optimize solar charging processes in photovoltaic systems. Here is a detailed description of its main features and technical data:

Area of application: For use in solar energy systems, especially in motorhomes, boats and small off-grid systems.

Main characteristics

1. MPPT technology: The SmartSolar MPPT 100/20 uses Maximum Power Point Tracking technology to get the maximum energy out of the solar modules, even in changing light conditions. MPPT is an algorithm that continuously determines the optimum point for energy yield, which means that the charge controller always draws the best possible power from the solar modules. This is particularly important when solar irradiation varies, for example when the sky is cloudy or at different times of day. The efficient conversion and maximization of solar power ensures that the batteries are charged quickly and fully. MPPT technology thus significantly improves the overall performance and efficiency of the solar system, resulting in better use of solar energy and lower operating costs.

2. High efficiency: With an efficiency of up to 98%, the controller optimizes energy conversion and minimizes losses, resulting in better performance of the entire solar system.

3. Bluetooth connectivity: The integrated Bluetooth receiver enables easy monitoring and configuration of the controller via the VictronConnect app on a smartphone or tablet. This enables user-friendly and detailed monitoring of the state of charge and system performance.

4. Versatile battery compatibility: The regulator is compatible with various battery types, including lead-acid, gel, AGM, and lithium batteries, making it flexible and versatile in different applications.

5. Temperature compensation: To extend the service life of the batteries, the charge is adjusted depending on the battery temperature.

6. Robust design: With a degree of protection of IP43, the controller is protected against splash water and dust, making it suitable for use in demanding environments.

7. Easy installation: The connections are designed for simple and safe wiring. The compact design allows easy integration into various installation environments.

8. High input voltage: The controller can support solar strings with up to 100 V input voltage, which offers more flexibility when planning solar systems.

The Victron SmartSolar MPPT 100/20 is an excellent choice for anyone who needs a reliable and efficient solar charging system that offers easy monitoring and customization while ensuring high performance and flexibility in various applications.

Property	Value
Max. Input Voltage	100 V DC (Maximum input voltage)
Max. Output Current	20 A
Max. Charging Power	260 W at 12 V, 520 W at 24 V
MPPT Range	15 V to 100 V
Efficiency	Up to 98%
Solar Input Range	Compatible with 12 V or 24 V systems
Battery Voltage	12 V or 24 V
Battery Types	Lead-acid, Gel, AGM, Lithium (LiFePO4)
Temperature Range	-40°C to +60°C
Temperature Compensation	Yes
Connection Terminals	High-quality screw terminals
Communication	Bluetooth (SmartSolar App) and UART TTL
Dimensions	200 x 130 x 50 mm
Weight	1.5 kg

Property	Value
Protection Class	IP43 (Protection against splash water from all directions)
Certifications	CE, RoHS

Table 12.4: Technical data of the Victron charge controller:

The charge controller can also be connected to the PC via a USB TTL cable; the Victron Connect software for the PC is available free of charge. With the UART interface, a connection to the ESP32 would also be possible to read out the current state of charge.

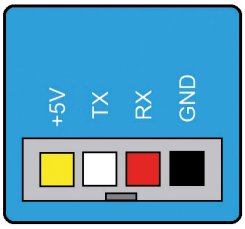


Figure 12.5: Victron TTL UART Interface

PID	Product ID
FW	Firmware Version
SER#	Serial Number
V	Battery Voltage
I	Battery Current (negative during discharge)
VPV	Solar Panel Voltage
PPV	Solar Module Power
CS	Charge Controller Operating Status (see table)
MPPT	MPPT Tracker Status (0 = off, 1 = voltage or current limited, 2 = active)
OR	Off Reason
ERR	Error Code
LOAD	Load Output Status (active, inactive)
IL	Load Output Current
H19	Total Yield
H20	Yield Today
H21	Maximum Solar Power Today
H22	Yield Yesterday
H23	Maximum Solar Power Yesterday
HSDS	Daily Counter (0 to 364)

Table 12.4: Data currently output by a SmartSolar 75/15 with firmware 1.56:

12.5 Windows software SharpDevelop_Shelly3emWinform

A Windows programme is also available for monitoring the Shelly 3EM the 1PM Plus. The programme can be modified and recompiled with Sharp Develop or Visual Studio to adapt the IP addresses.

This allows data to be saved as a CSV for Excel for later analysis. It is also possible to use it to switch loads or inverters that are connected to it.

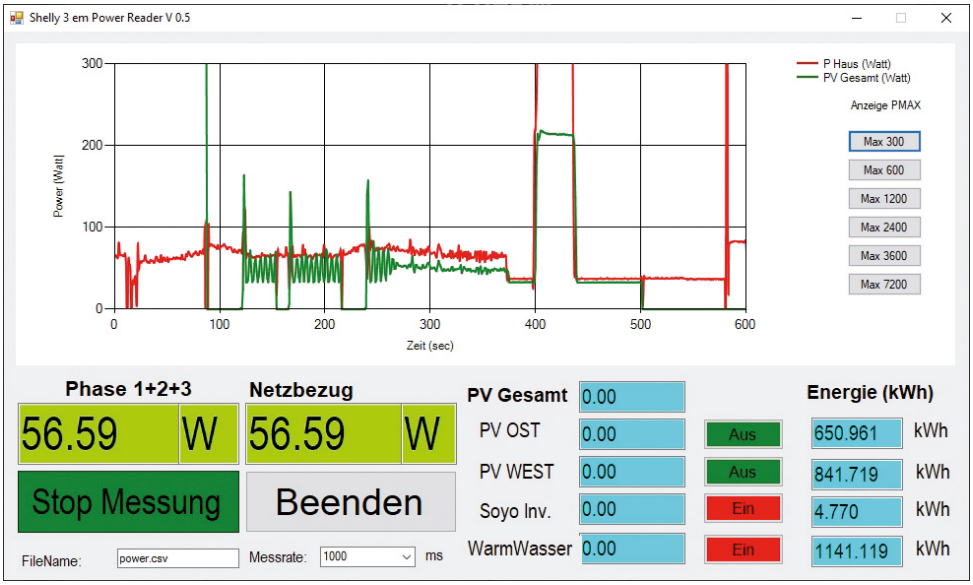


Figure 12.6: Windows PV Monitor for the Shellys

12.6 Soyo Micro Inverter for feeding into the grid

Feeding into the grid is done with the Soyo Micro Grid Inverter and a 12 V / 100 AH LiFePO4 battery.

This Soyo inverter is compatible with 12 V batteries and can be connected directly to them. Versions with 24 V and 48 V are also available. In 12 V operation, it offers adjustable power in the range from 30 W to 250 W. To protect the life of the battery, the inverter has a low battery voltage cut-out, which prevents the battery from discharging completely. The efficiency of the inverter depends on the input voltage, with efficiency decreasing at a voltage close to 11 V. It is therefore advisable to keep the voltage above this value to ensure optimum efficiency.

The microinverter converts the battery voltage (DC) into alternating current (AC) and feeds this directly into the power grid. In doing so, the microinverter automatically synchronizes with the grid frequency and voltage to ensure that the generated current is in phase with the grid current. As soon as synchronization is complete, the microinverter gradually increases the output voltage until the power of the solar module corresponds to the set specification.



Figure 12.7: Soyo Micro Grid Inverter

To use this inverter to feed variable power into the domestic grid, the POTI is removed and a voltage in the range of 0 to 5 V is applied to its pins, which corresponds to a feed-in power of 30 W to 250 W.

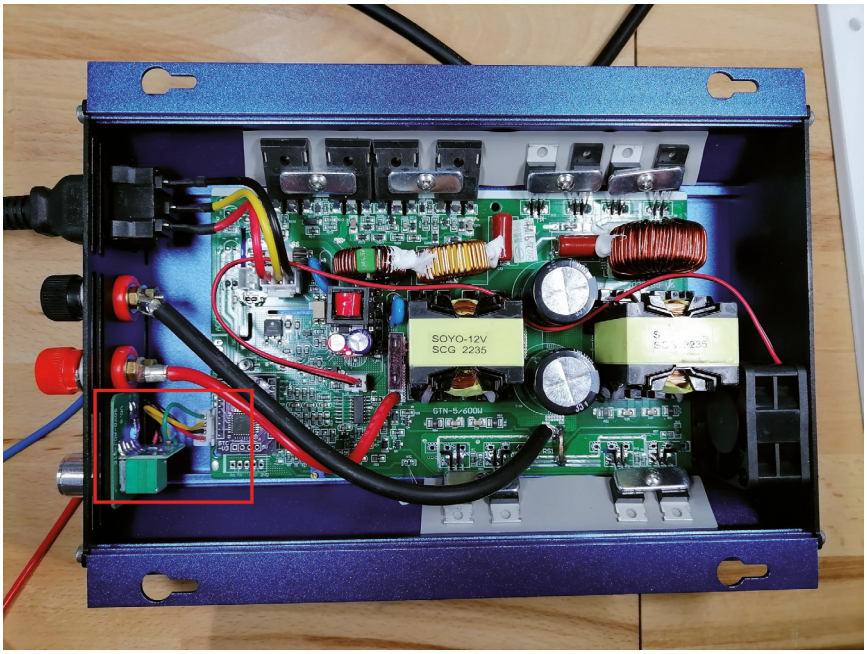


Figure 12.8: Connection for the external feed

Color	Description
Green	Off +5V On 0V (GND)
Yellow	0..5 Volt for feed-in power
Black	GND
Red	+5 Volt

Table 12.5: Pin layout for potentiometer interface:

12.7 ESP32 PID Fuzzy Control Board for closed-loop control

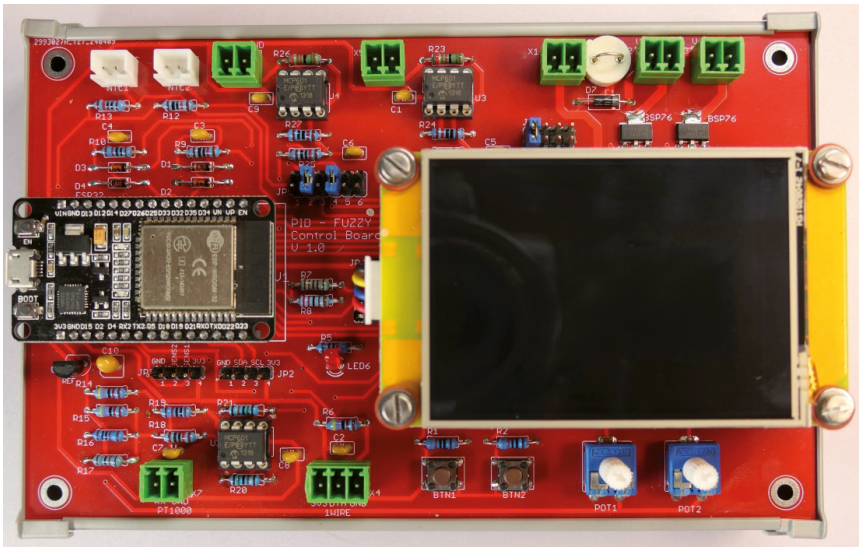


Figure 12.9: ESP32 Fuzzy PID Board

The readout of the performance data from the Shelly 3EM takes place via http Get Requests to the IP address of the device.

192.168.178.40/emeter/0

The response is a string in JSON format as in this example:

```
{
    "power": 13.2,
    "pf": 0.49,
    "current": 0.12,
    "voltage": 228.74,
    "is_valid": true,
    "total": 660366.2,
    "total_returned": 0
}
```

This response string is analyzed using the JSON libraries and the important values are extracted.

```
#include <ArduinoJson.h>
#include <Arduino_JSON.h>
```

Description of the software:
The given programme in the function `void loop()` consists of several main tasks that are executed in different sections. Here is a detailed explanation of the process:

1. Touchscreen evaluation (Nextion Display)
The programme checks whether data has been received via the serial port `Serial2` (this indicates user input via the touch display).

If data has been received, the character is read and analyzed using a 'switch' command. Based on the character received (`P`, `M`, `S`), corresponding messages are output via `Serial.println()`. This could be used, for example, to operate buttons on the display.

2. Control of relays and LEDs (Soyo Control)
The programme monitors a control state (state_control) and the status of a button (read_btn1()).

Depending on the current state (0 or 1), the relay is switched when the button is pressed to connect the Soyo inverter to the battery and the LED is switched on or off.

This control logic also switches a Shelly device on or off via an API call depending on the relay status.

3. Periodic tasks (once per second)
This block is executed once per second, controlled by a timer logic (`millis()`).

LED1 (status indicator) is toggled within this block.

Several Shelly devices (Shelly 3EM and Shelly 1PM) are read out to monitor the energy consumption or the feed-in in different phases (L1, L2, L3). This data is stored in variables and used to calculate power values.

The power values from the phases are totalled and adjusted to correct negative values that the Shelly 3EM outputs when one or more phases are fed in.

4. Calculations
Based on the power data read out, the total power of the house and the total PV power are calculated.

If there is a grid connection (positive power in the house) and the controller is switched on, a control value is calculated, which is controlled (PID or fuzzy logic) and sent to the system.

If there is no mains supply, the control value is set to zero.

5. Data display
The calculated values (e.g. power in the phases, PV power, actuating value) are formatted as strings and sent to the Nextion display to be shown there.

At the same time, these values are also output on the serial monitor to provide feedback on the current system status.

Overall, it is a programme that enables both real-time monitoring and control of energy flows in a system consisting of several solar installations, Shelly devices and a Soyo inverter. The programme takes on the task of monitoring these devices, regulating the feed-in power and outputting all relevant data to a display and the serial monitor.

Listing 12.2: Arduino_ESP32_Shelly3em_SoyoInverter.ino

```
void loop()
{

  // Analyse Touch Display Nextion if necessary
  if (Serial2.available())
  {
    char receivedChar = Serial2.read();
    switch (receivedChar)
    {
      case 'P': Serial.println("Button1");break;
      case 'M': Serial.println("Button2");break;
      case 'S': Serial.println("Button3");break;
      default:/*Serial.println("Character not valid");*/break;
    }
  }

  // Soyo Control on off
  switch (state_control)
       {
    case 0: if (read_btn1()) { state_control = 1; digitalWrite(LED1, 1);
digitalWrite(RELAIS1, 1); SwitchShellyPlus1pmOnOff(RelaisOnSoyo); } break;
    case 1: if (read_btn1()) { state_control = 0; digitalWrite(LED1, 0);
digitalWrite(RELAIS1, 0); SwitchShellyPlus1pmOnOff(RelaisOffSoyo);} break;
  }

  // call every second
  if ((millis() - lastTime) > timerDelay)
  {
    lastTime = millis();
    // LED Blink
```

```
    digitalWrite(LED_BUILTIN, !digitalRead(LED_BUILTIN));

    // Shelly 3 em read out L1 to L3 and decode JSON strin
    ReadShelly3em(0);
    ReadShelly3em(1);
    ReadShelly3em(2);

    // Read the three Shelly 1 pm and decode the JSON string
    ReadShellyPlus1pmGarage();
    ReadShellyPlus1pmHaus();
    ReadShellyPlus1pmSoyo();

    // Correct negative values in the individual phases
    PowerArr3em[phaseGarage] += PowerArr1pm_Garage[2];
    PowerArr3em[phaseHaus] += PowerArr1pm_Haus[2];
    PowerArr3em[phaseHaus] += PowerArr1pm_Soyo[2];

    // Calculate the total power output of the Shelly 3 em
    PowerArr3em[3]=PowerArr3em[0]+PowerArr3em[1]+PowerArr3em[2];

    // Calculate the total feed-in
    p_power_pv_gesamt = PowerArr1pm_Garage[2] + PowerArr1pm_Haus[2]  +
PowerArr1pm_Soyo[2];

       // Calculate house power, subtract the PV power
       // due to negative values of the individual phases of the Shelly 3em
       // feed-in with - sign / reference + sign
    p_power_home = PowerArr3em[3] - p_power_pv_gesamt;

       // Convert the values into strings for display and output
    powerphase1   = String(PowerArr3em[0], 2);
    powerphase2   = String(PowerArr3em[1], 2);
    powerphase3   = String(PowerArr3em[2], 2);
    powerphase123 = String(PowerArr3em[3], 2);

    power_pv_Garage = String(PowerArr1pm_Garage[2]);
    temp_pv_Garage  = String(PowerArr1pm_Garage[4]);

    power_pv_Haus   = String(PowerArr1pm_Haus[2]);
    temp_pv_Haus    = String(PowerArr1pm_Haus[4]);

    power_pv_Soyo   = String(PowerArr1pm_Soyo[2]);

    power_pv_Gesamt = String(p_power_pv_gesamt,2);

    power_home      = String(p_power_home);
```

```
    // If mains supply is available and controller ON
    if (p_power_home > 0 && state_control )
    {
      // ControlValue = pid_regler(0, p_power_home);
      ControlValue = fuzzy_control(0, p_power_home);
      SendControlValue(ControlValue);
    }
    else
    {
      ControlValue = 0;
      SendControlValue(ControlValue);
    }

    String strControlValue = String(ControlValue);

    // Send Data to Nextion Display
    Nextion_Send_Text_to_Textbox("txtT1",powerphase123);
    Nextion_Send_Text_to_Textbox("txtT2",power_home);
    Nextion_Send_Text_to_Textbox("txtT3",power_pv_Garage);
    Nextion_Send_Text_to_Textbox("txtT4",power_pv_Haus);
    Nextion_Send_Text_to_Textbox("txtT5",power_pv_Soyo);
    Nextion_Send_Text_to_Textbox("txtT6",strControlValue);

    // Send Data to serial Monitor
    Serial.print("PWR-3EM:");
    Serial.print(powerphase123);
    Serial.print(" PWR-Home:");
    Serial.print(power_home);
    Serial.print(" PV-EAST:");
    Serial.print(power_pv_Garage);
    Serial.print(" PV-WEST:");
    Serial.print(power_pv_Haus);
    Serial.print(" PV-Soyo:");
    Serial.print(power_pv_Soyo);
    Serial.print(" ControlValue:");
    Serial.println(strControlValue);
  }
}
```

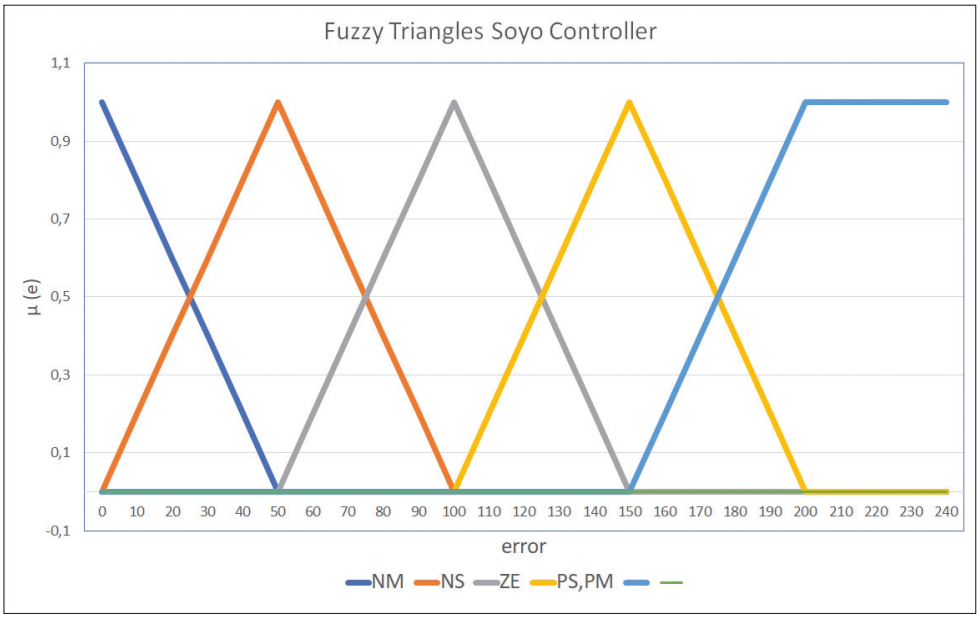


Figure 12.10: Fuzzy triangles for the controller

```
// These arrays are used to represent the degrees of membership
// of an input value in different regions of a fuzzy set.
float Nmax[]  =   {0,50};           // Values for the left region
float Nsmall[]  = {0,50, 100};      // Values for the middle region
float Zero[]  =   {50, 100, 150};   // Values for the middle region
float Psmall[]  = { 100, 150, 200}; // Values for the middle region
float Pmax[]  =   { 150, 200};      // Values for the right region

// Output Values
float U[]       = { 0 , 60, 120 , 170, 255 };
```

12.8 Fuzzy controller function

The fuzzy_control(float w, float x) function calculates a control signal using fuzzy logic. Firstly, the error e is determined as the difference between w and -x, which is the power to be fed in. Based on this error, the membership values for different areas (left, NS, ZE, PS, right) are calculated using the fuzzy functions. The weighted sum of these membership values forms the numerator, while the sum of all membership values forms the denominator. The control signal y is calculated as the quotient of the numerator and denominator. However, if the denominator is zero, y is set to zero to avoid division by zero. Finally, y is returned as a byte variable to output the control value.

```
byte fuzzy_control(float w, float x)
{
      // Calculate the error e
    float e = w - (x * -1.0);
```

```
// Calculate the membership values for the error signal in different regions of
// the fuzzy sets.
       // Membership value for the left region
       float left_degree_NM   = fuzzyLeftOpen  (e, Nmax[0],  Nmax[1]);
       // Membership value for the middle region NS
       float middle_degree_NS = fuzzyTriangle  (e, Nsmall[0],Nsmall[1],Nsmall[2]);
       // Membership value for the middle region ZE
       float middle_degree_ZE = fuzzyTriangle  (e, Zero[0],Zero[1],Zero[2]);
       // Membership value for the middle region PS
       float middle_degree_PS = fuzzyTriangle  (e, Psmall[0],Psmall[1],Psmall[2]);
       // Membership value for the right region
       float right_degree_PM  = fuzzyRightOpen (e, Pmax[0], Pmax[1]);

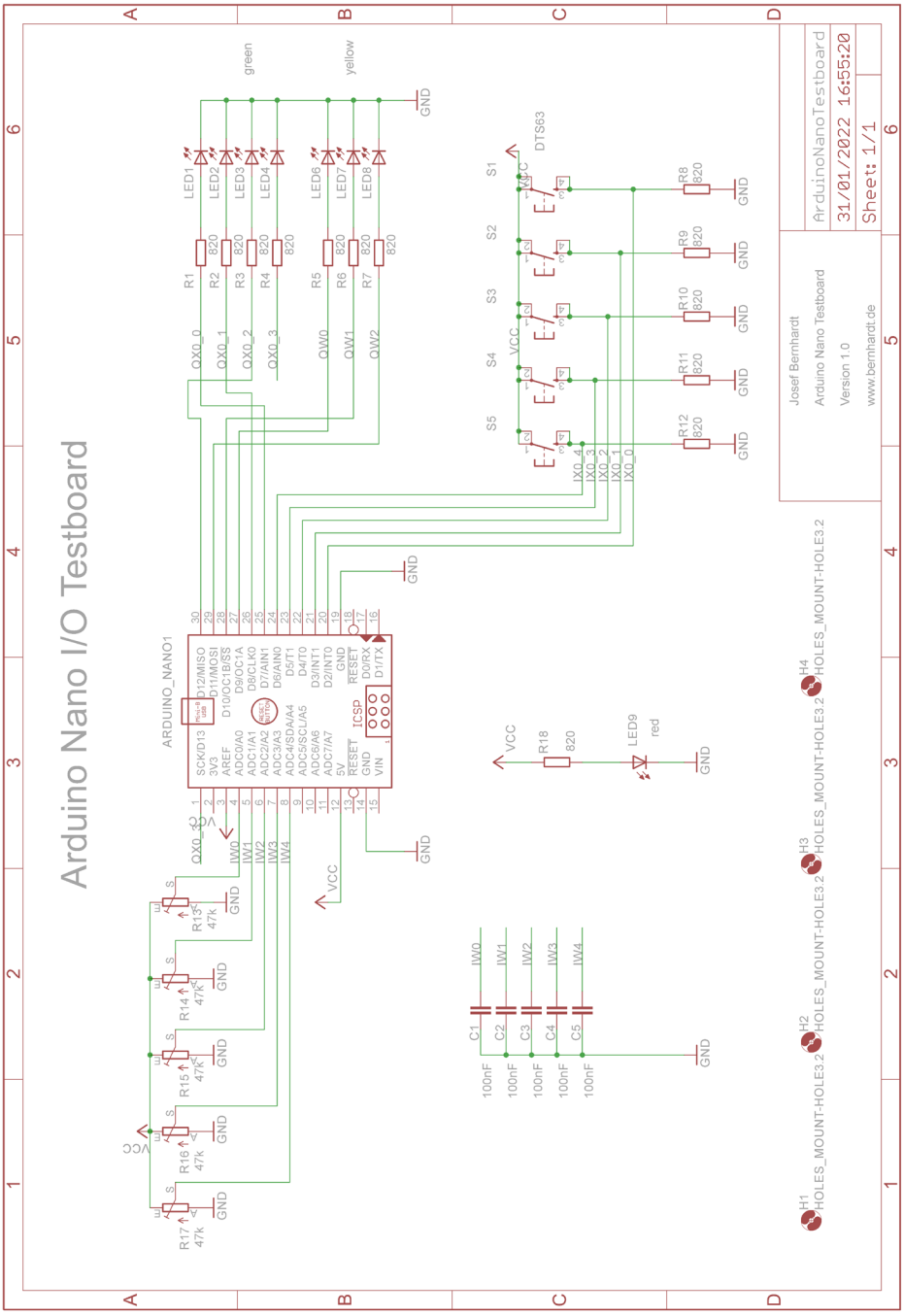
    // Calculate the weighted sum of membership values
    float numerator   = left_degree_NM * U[0] + middle_degree_NS * U[1] +
                                                middle_degree_ZE * U[2]  + middle_
degree_PS * U[3]
                                                + right_degree_PM * U[4];

    // Sum of membership values
    float denominator = left_degree_NM + middle_degree_NS +
                                                middle_degree_ZE + middle_degree_PS
                                                + right_degree_PM;
    // Control signal
    float y;

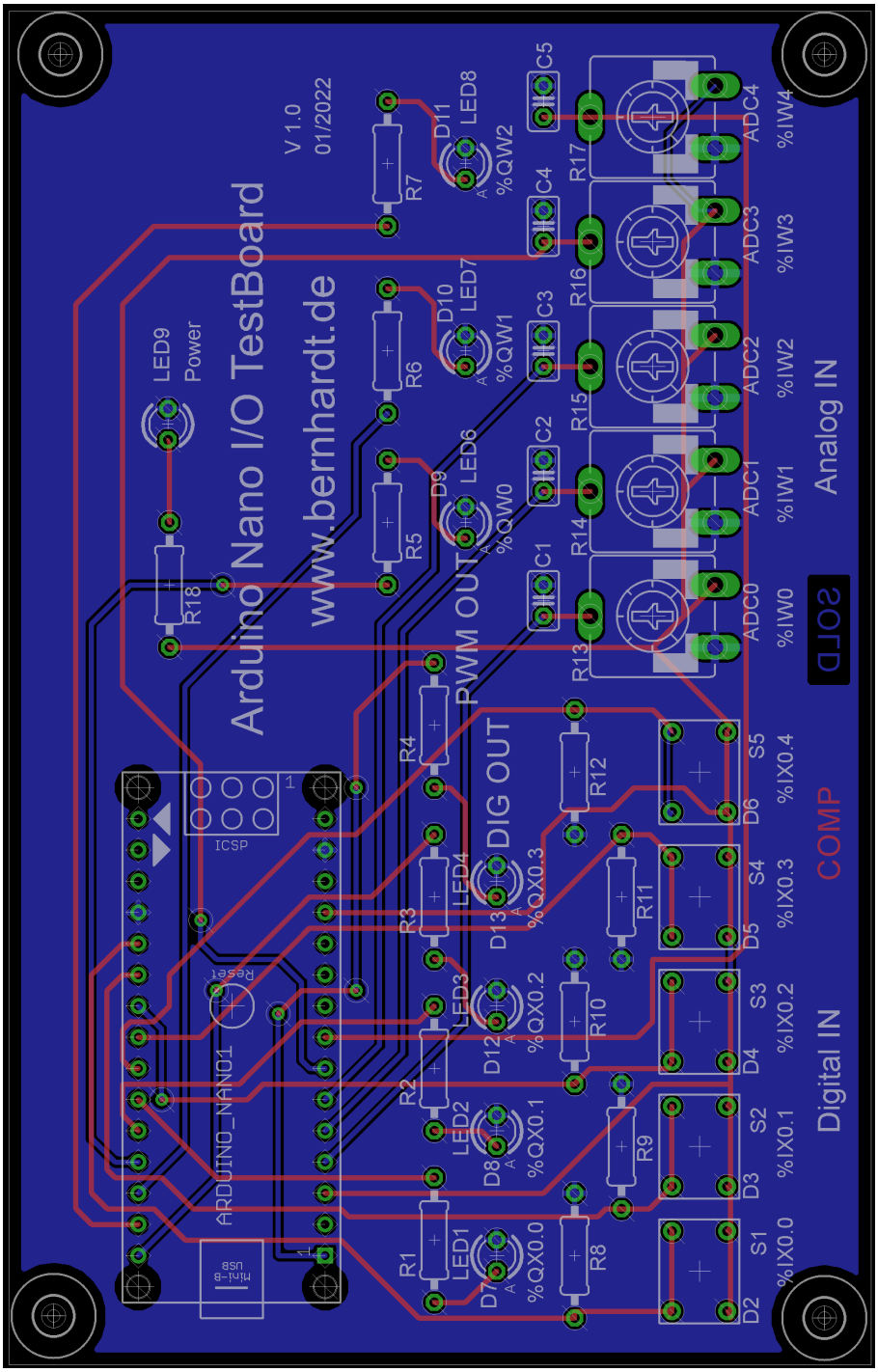
    // Avoid division by zero by checking if the denominator is zero.
    if (denominator != 0) {
        // Calculate the control signal as the ratio of the
        // weighted sum to the sum of membership values.
        y = numerator / denominator;
    } else {
        // If the denominator is zero, set the control signal
       // to zero to avoid undefined behavior.
        y = 0;
    }
    return (byte)y; // Return the control signal.
}
```

Appendix A - Circuit layouts and parts list

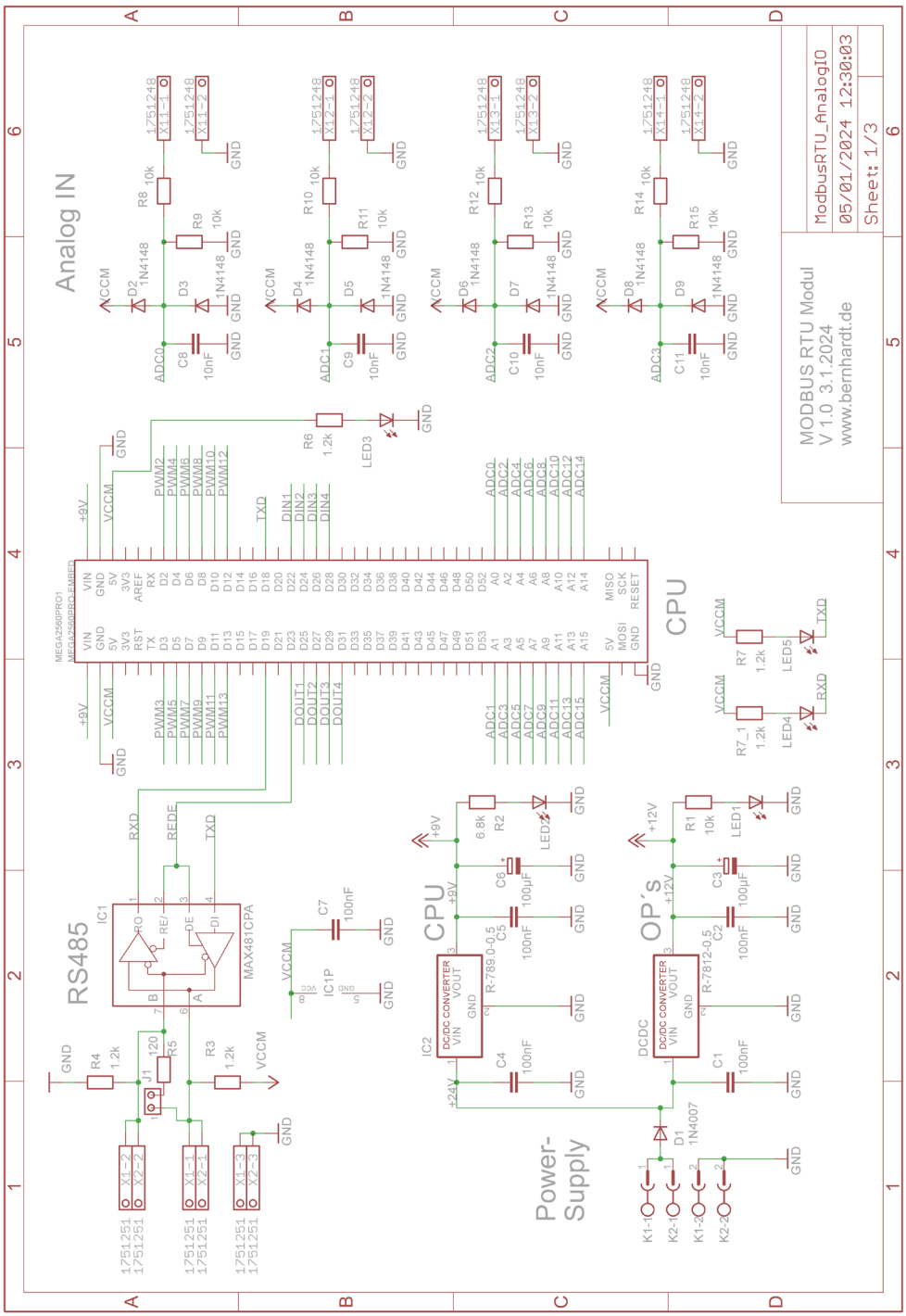
Arduino Nano Test Board Circuit



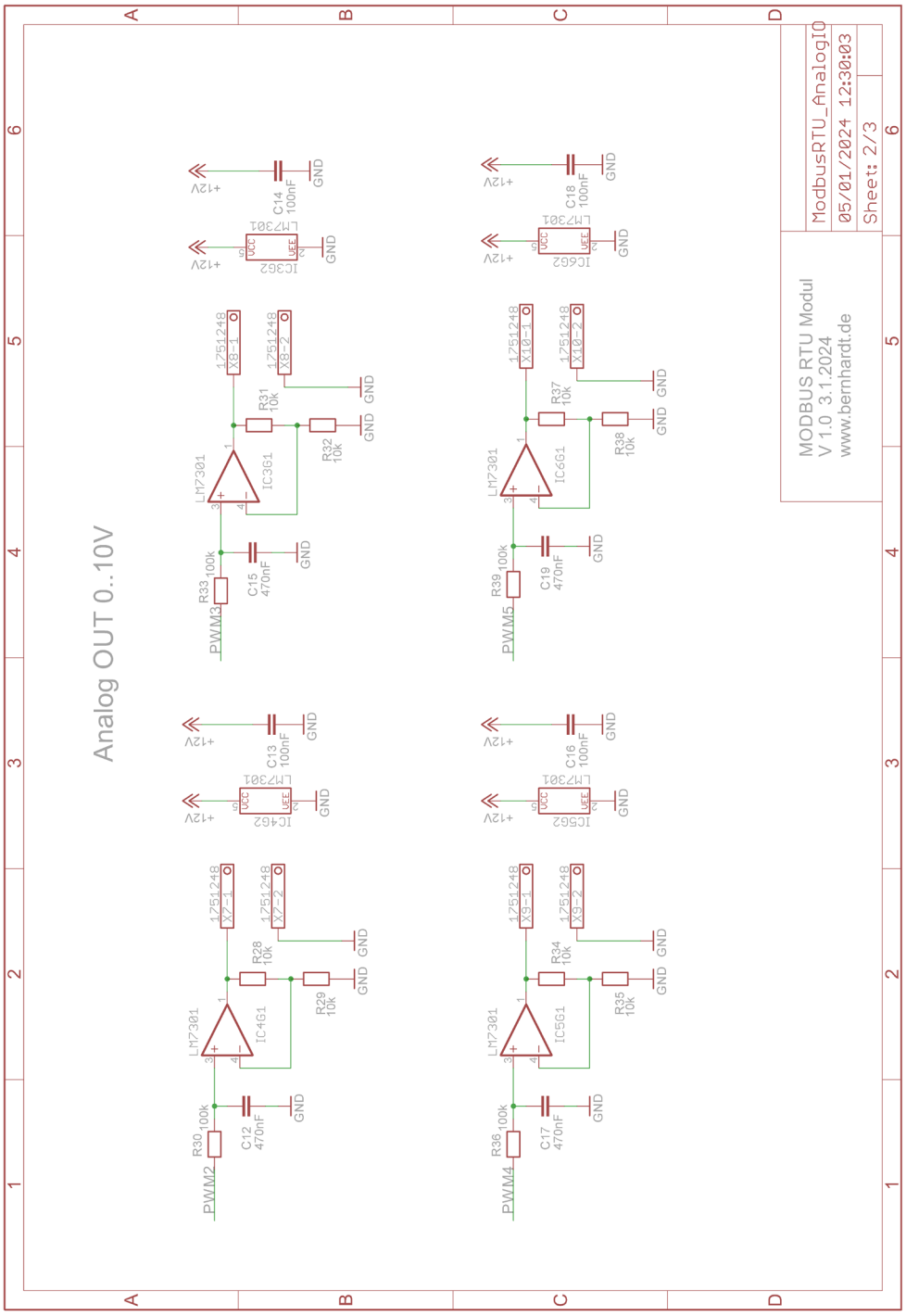
Arduino Nano Test Board Layout



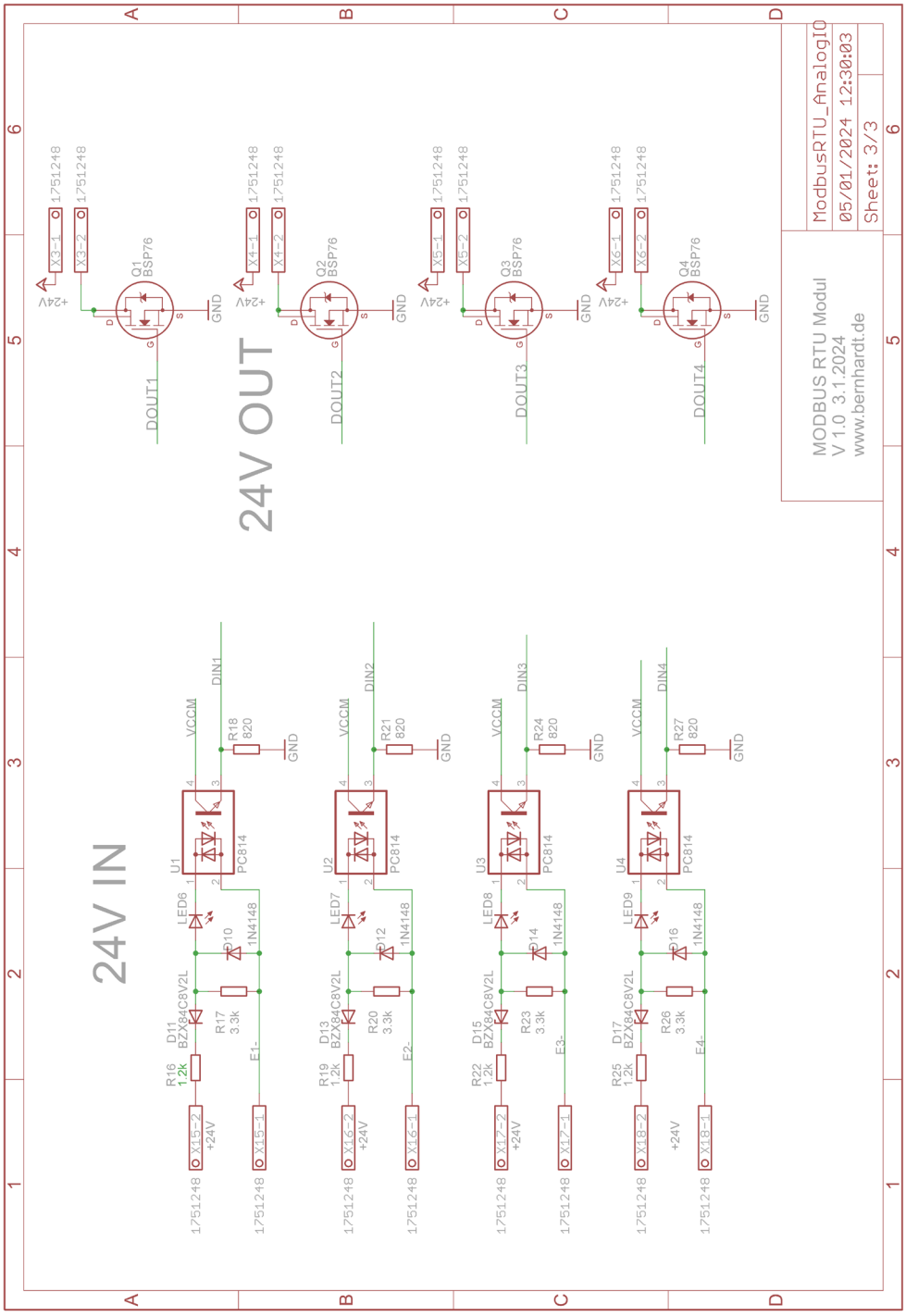
Arduino Mega PLC Board Circuit Page 1



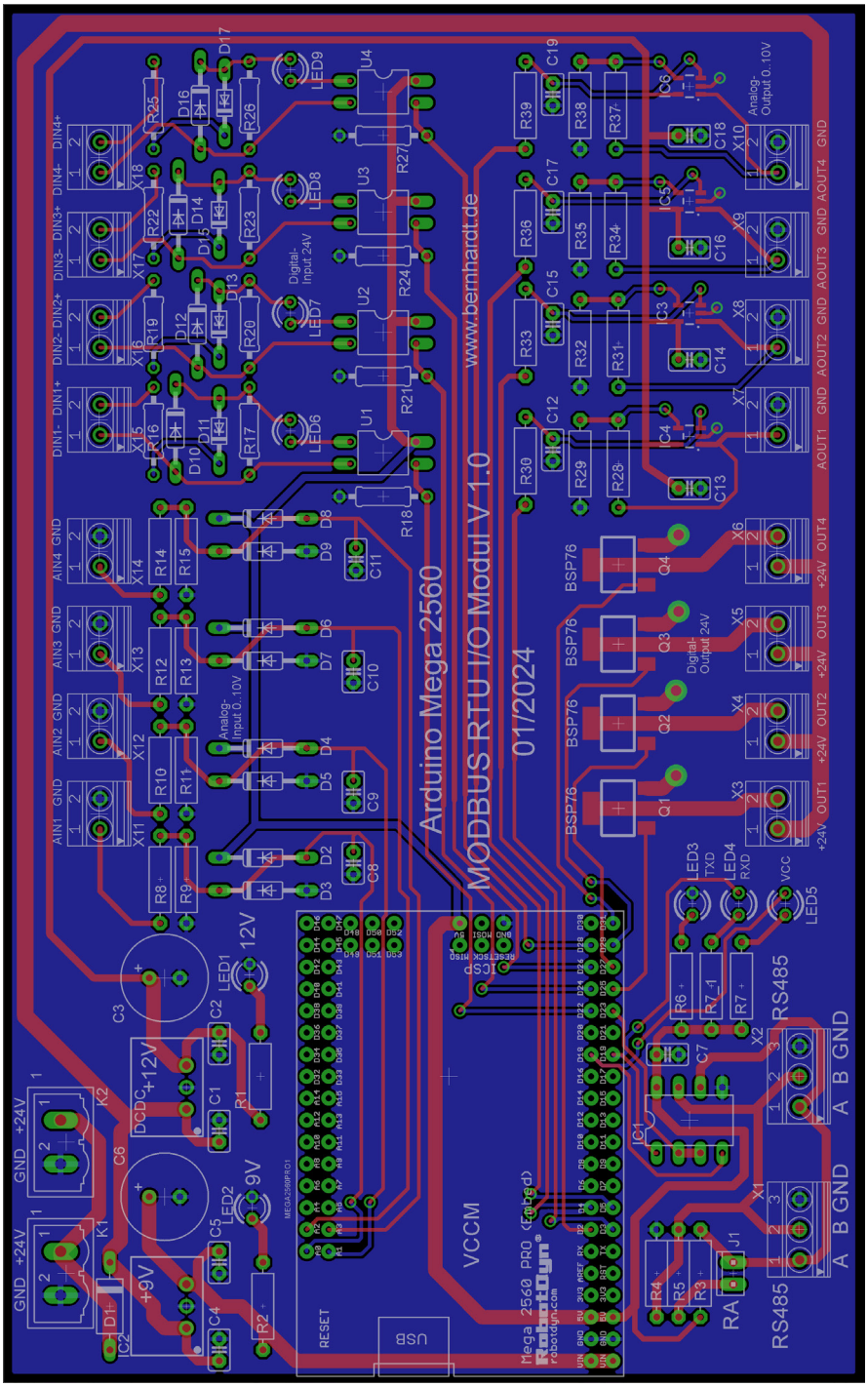
Arduino Mega PLC Board Circuit Page 2



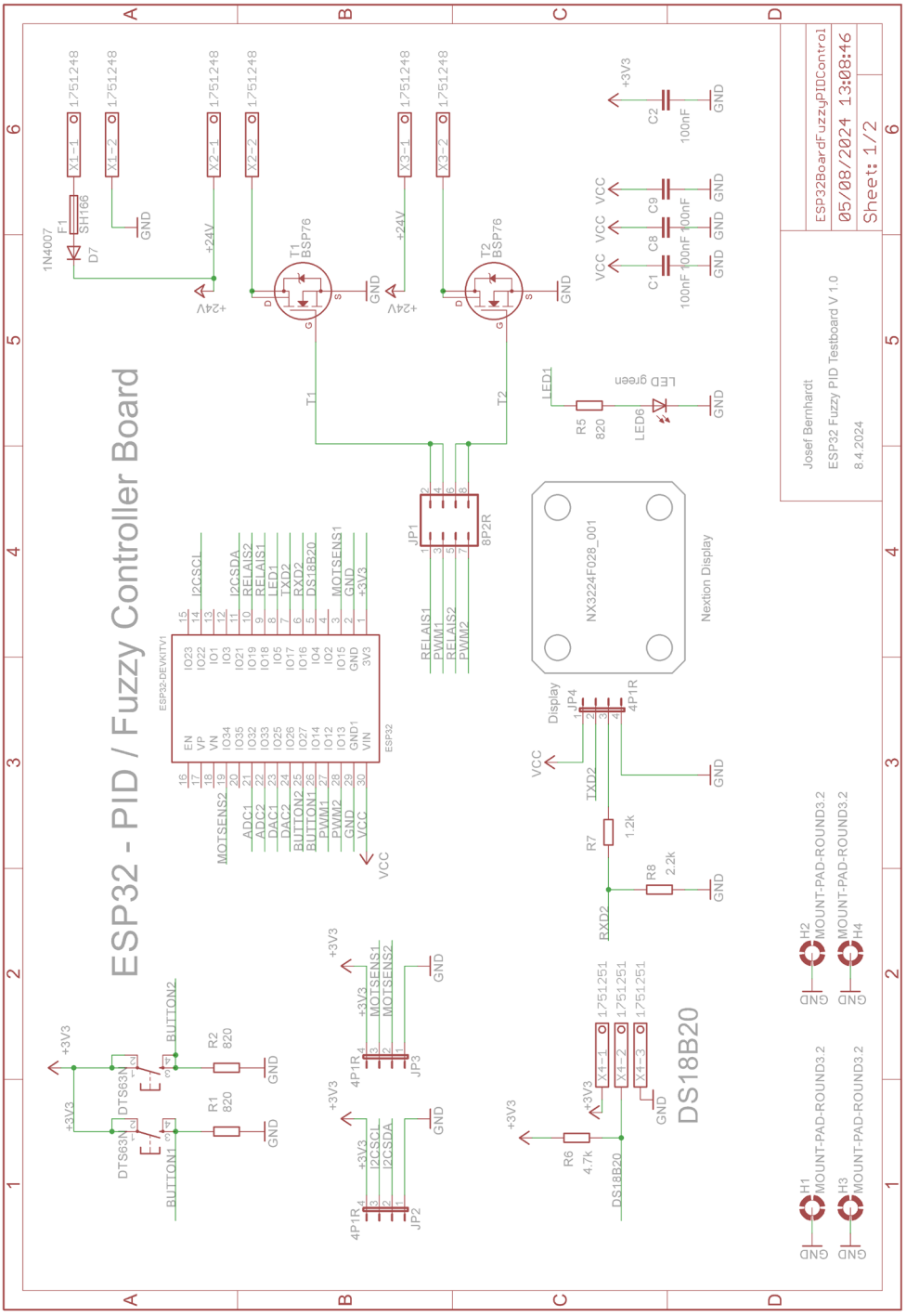
Arduino Mega PLC Board Circuit Page 3



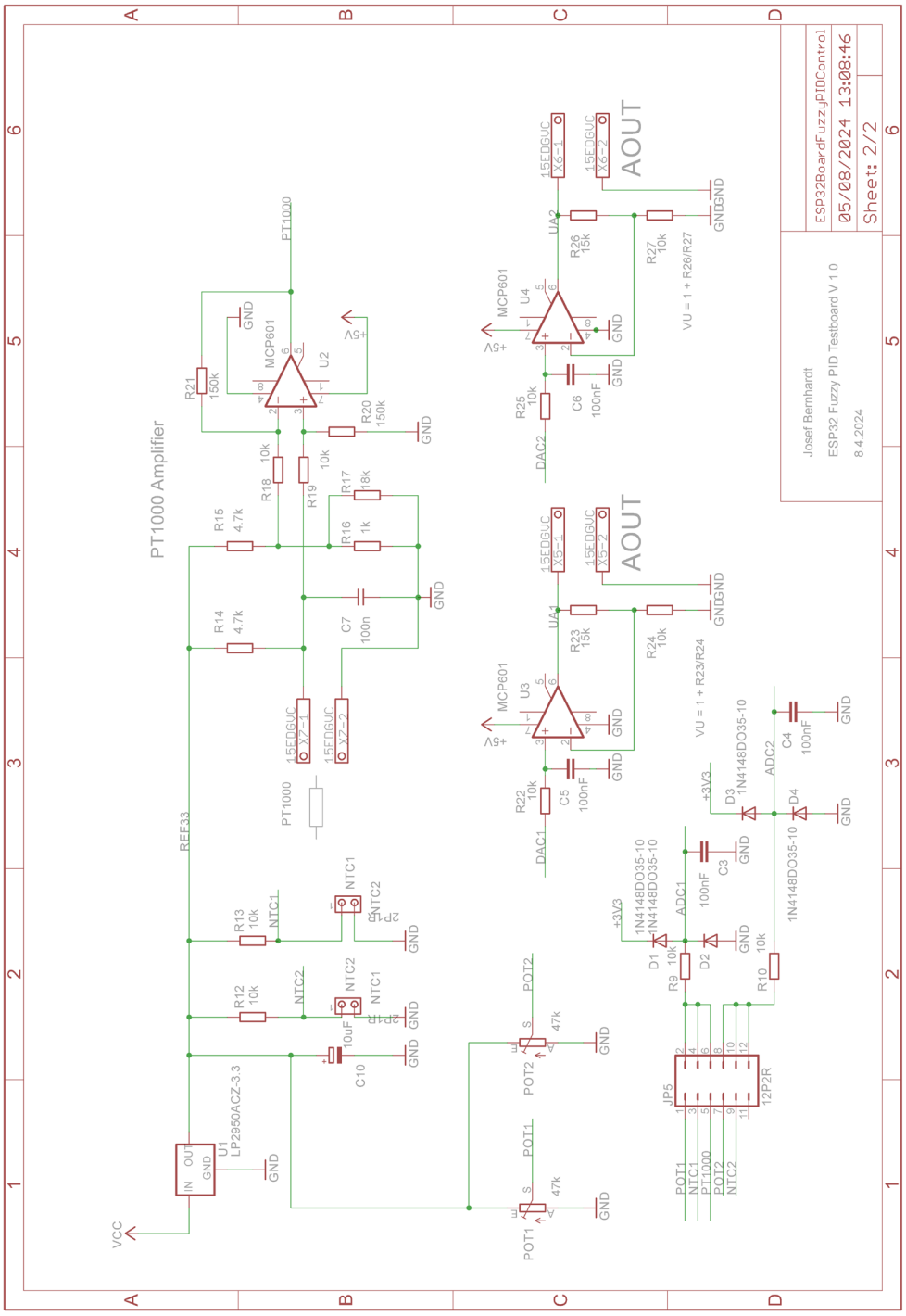
Arduino Mega PLC Board Layout



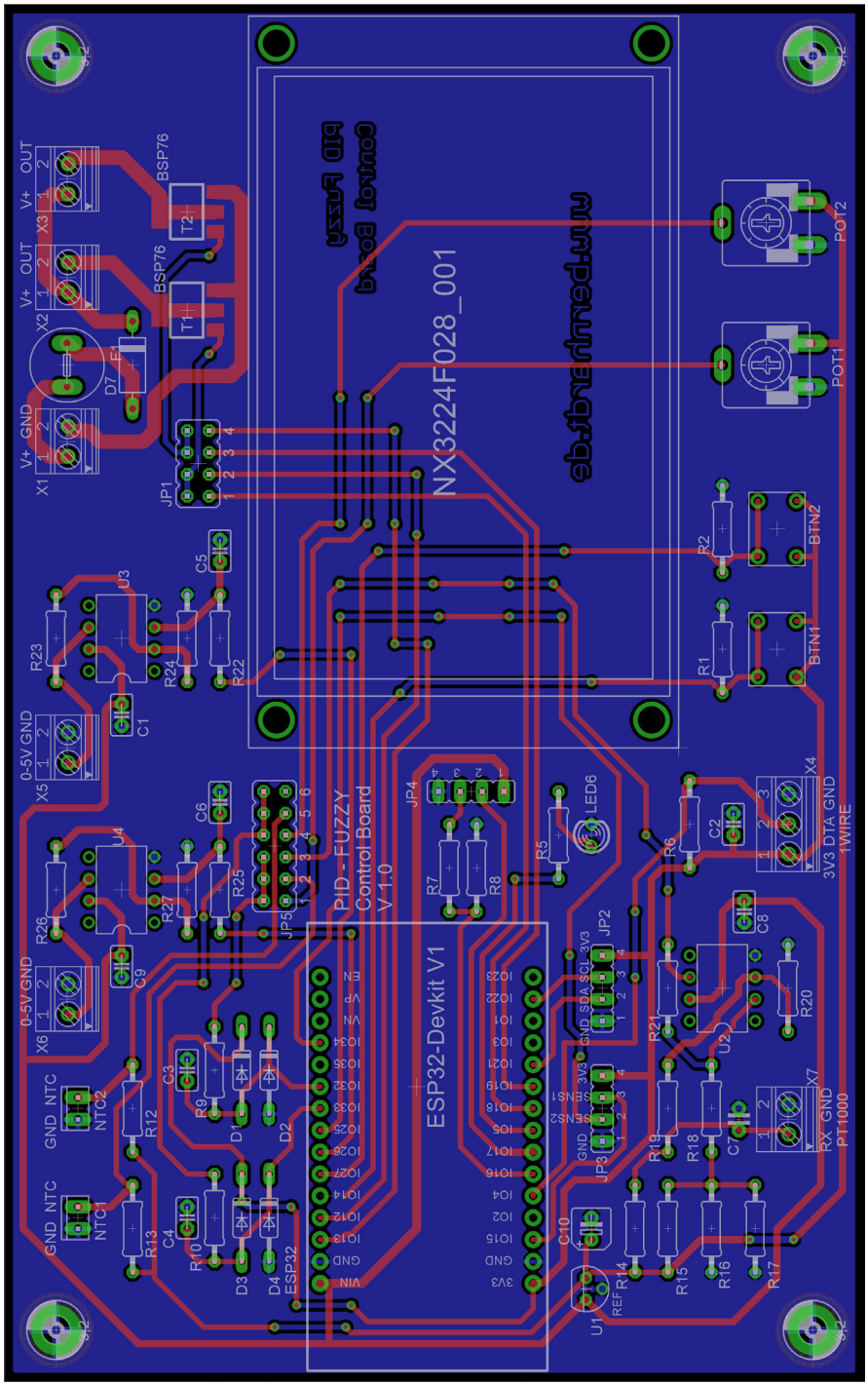
ESP32BoardFuzzyPIDControl Circuit Page 1



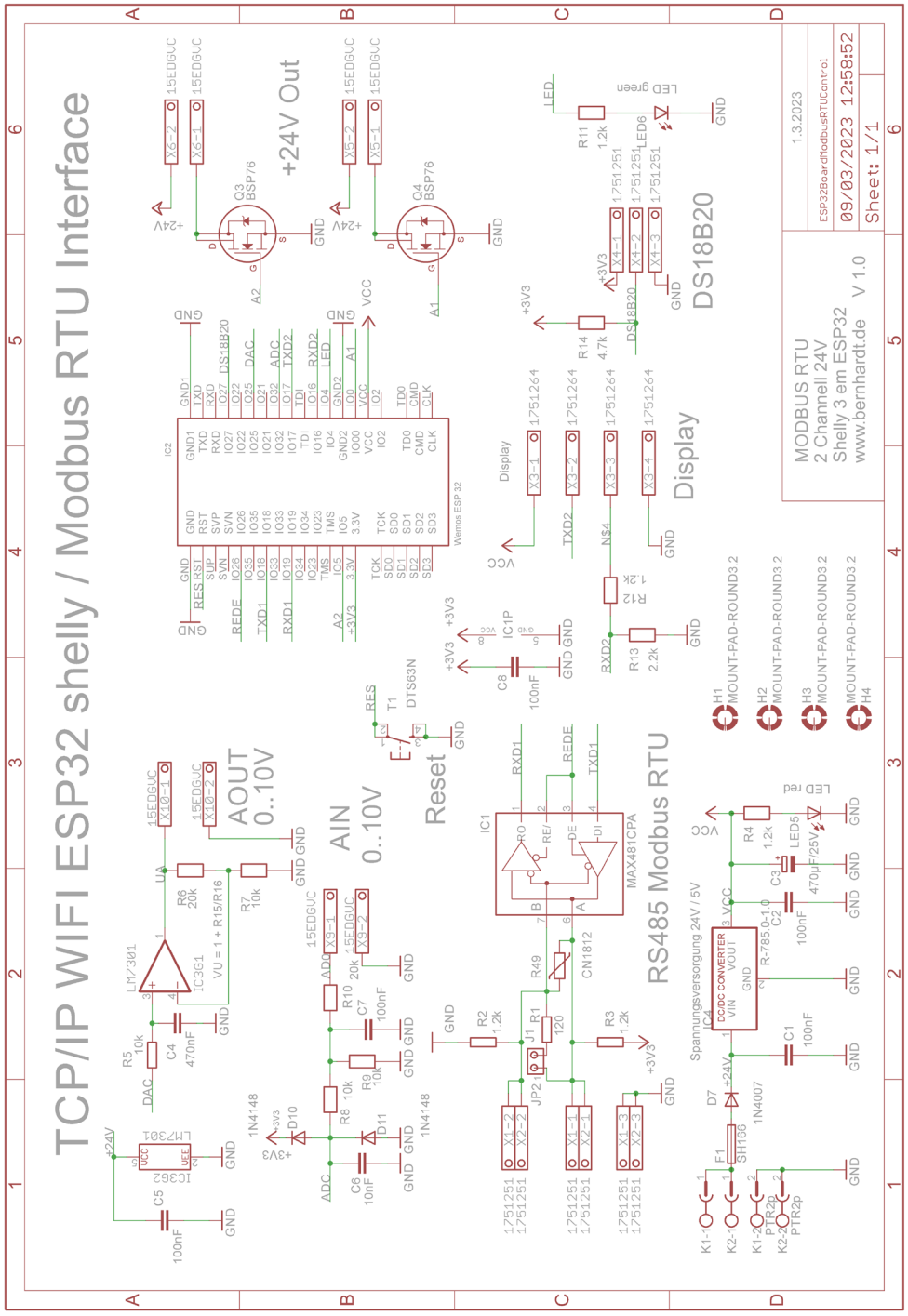
ESP32BoardFuzzyPIDControl Circuit Page 2



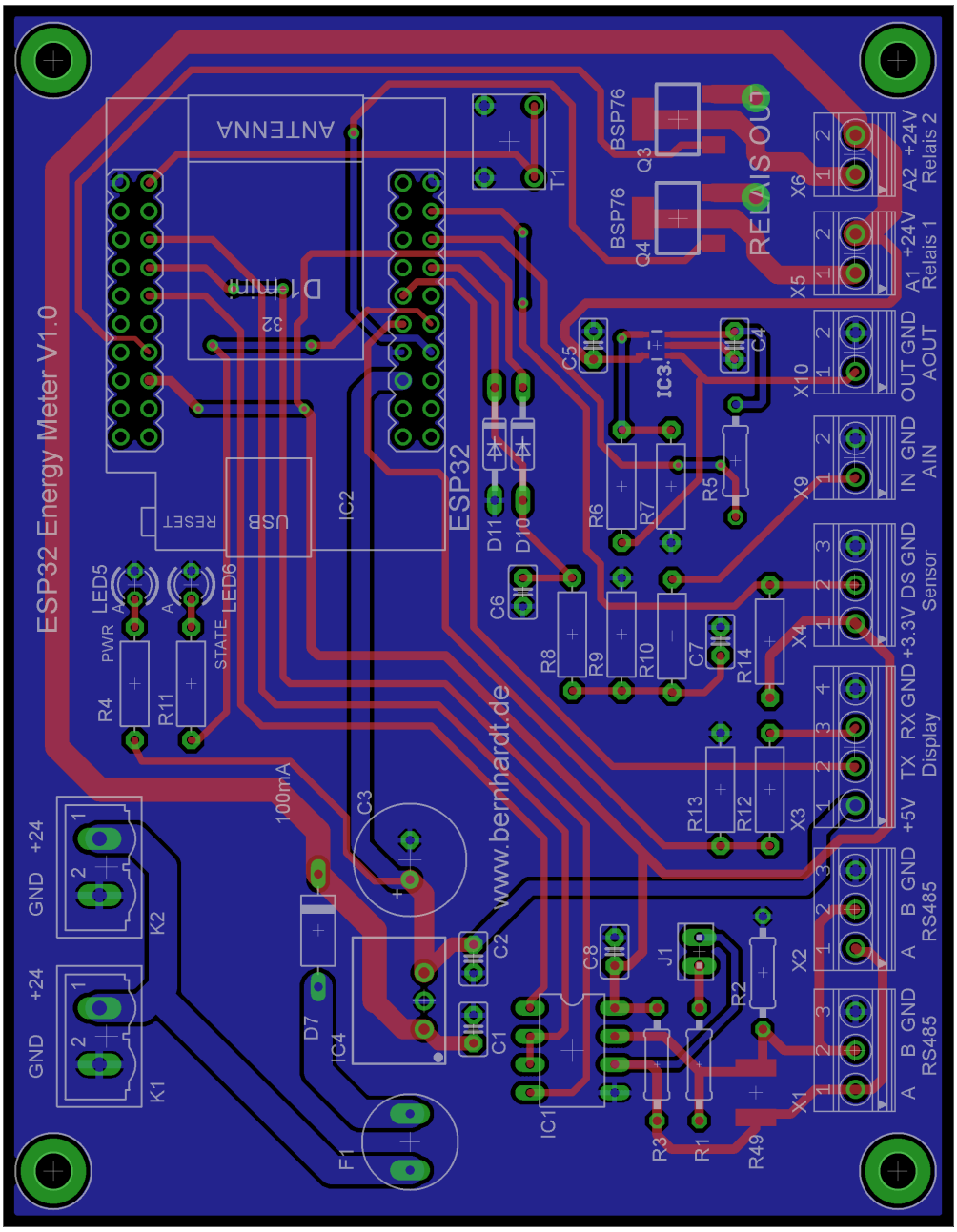
ESP32BoardFuzzyPIDControl Layout



ESP32BoardModbusRTUControl Circuit Page 1



ESP32BoardModbusRTUControl Layout



Arduino Nano Test Board Part List

Qty	Value	Device	Package	Parts
1	Arduino Nano Modul	ARDUINO_NANO	ARDUINO_NANO	ARDUINO_NANO1
5	DTS63	DTS63N	DT63N	S1, S2, S3, S4, S5
8	LED	LED3MM	LED3MM	LED1, LED2, LED3, LED4, LED6, LED7, LED8, LED9
5	100nF	C RM2.5	RM 2.5	C1, C2, C3, C4, C5
5	47k	Poti	CA9V	R13, R14, R15, R16, R17
13	820	R	R 0207/10	R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R18

Arduino Mega PLC Board Part List

Qty	Value	Device	Package	Parts
16	1751248	CON RM 3.5 2p	1751248	X3, X4, X5, X6, X7, X8, X9, X10, X11, X12, X13, X14, X15, X16, X17, X18
2	1751251	CON RM 3.5 3p	1751251	X1, X2
12	1N4148	1N4148DO35-10	DO35-10	D2, D3, D4, D5, D6, D7, D8, D9, D10, D12, D14, D16
4	BZX84C8V2L	BZV10	DO34Z7	D11, D13, D15, D17
9	100nF	C2.5/2	C2.5-2	C1, C2, C4, C5, C7, C13, C14, C16, C18
4	10nF	C2.5/2	C2.5-2	C8, C9, C10, C11
4	470nF	C2.5/2	C2.5-2	C12, C15, C17, C19
2	100µF	Elko	E3,5-10	C3, C6
1	1N4007	DIODE D-10	D-10	D1
4		LED3MM	LED3MM	LED6, LED7, LED8, LED9
5		LED3MM	LED3MM	LED1, LED2, LED3, LED4, LED5
1	MAX481CPA	MAX481CPA	DIL08	IC1
4	BSP76	MOSFET_IRLL024	MOSFET SOT-223	Q1, Q2, Q3, Q4
1		MTA02-100	10X02MTA	J1
4	LM7301	LM7301	SOT23-5	IC3, IC4, IC5, IC6
4	PC814	PC814	DIL04	U1, U2, U3, U4
4	3.3k	R 0207/10	R 0207/10	R17, R20, R23, R26
4	820	R 0207/10	R 0207/10	R18, R21, R24, R27
1	R-7812-0.5	R-78C3.3-1.0	SIP3	DCDC
1	R-789.0-0.5	R-78C3.3-1.0	SIP3	IC2
5	1.2k	RESEU-10	R-10	R16, R19, R22, R25 ,R3, R4, R6, R7, R7_1
4	100k	RESEU-10	R-10	R30, R33, R36, R39
17	10k	RESEU-10	R-10	R1, R8, R9, R10, R11, R12, R13, R14, R15, R28, R29, R31, R32, R34, R35, R37, R38
1	120	RESEU-10	R-10	R5
1	6.8k	RESEU-10	R-10	R2
2		W232-2	W232-2	K1, K2
1	MEGA2560PRO-EMBED	MEGA2560PRO-EMBED	MEGA2560PRO-EMBED	MEGA2560PRO1

ESP32BoardFuzzyPIDControl Part List

Qty	Value	Device	Package	Parts
3	15EDGVC	1751248	1751248	X5, X6, X7
3	1751248	1751248	1751248	X1, X2, X3
1	1751251	1751251	1751251	X4
4	1N4148DO35-10	1N4148DO35-10	DO35-10	D1, D2, D3, D4
9	100nF	C2.5/2	C2.5-2	C1, C2, C3, C4, C5, C6, C7,C8, C9
1	10uF	CPOL-EUB45181A	B45181A	C10
1	1N4007	DIODE-D-10	D-10	D7
2	DTS63N	DTS63N	DT63N	BTN1, BTN2
1	ESP32	ESP32DEVKITV1	ESP32-DEVKITV1	ESP32
3	4P1R	JP4E	JP4	JP2, JP3, JP4
1	8P2R	JP4Q	JP4Q	JP1
1	12P2R	JP6Q	JP6Q	JP5
1	LED Red	LED3MM	LED3MM	LED6
1	LP2950ACZ-3.3	LP2950ACZ-3.3	TO92	U1
2	BSP76	MOSFET_IRLL024	MOSFET_SOT-223	T1, T2
2	2P1R	MTA02-100	10X02MTA	NTC1, NTC2
1	NX3224F028_001	NX3224F028_001	NX3224F028_001	U$1
1	1.2k	R 0207/10	0207/10	R7
10	10k	R 0207/10	0207/10	R9, R10, R12, R13, R18, R19, R22, R24, R25, R27
2	15k	R 0207/10	0207/10	R23, R26
3	820	R 0207/10	0207/10	R1, R2, R5
1	18k	R 0207/10	0207/10	R17
1	1k	R 0207/10	0207/10	R16
1	2.2k	R 0207/10	0207/10	R8
2	150k	R 0207/10	0207/10	R20, R21
3	4.7k	R 0207/10	0207/10	R6, R14, R15
3	MCP601	MCP 601	DIL08	U2, U3, U4
1	SH166	TR5 Fuse	TR5	F1
2	47k	Poti CA9V	ACP TECHNOLOGIES CA9V	POT1, POT2

ESP32BoardModbusRTUControl Part List

Qty	Value	Device	Package	Parts
4	15EDGVC	1751248	1751248	X5, X6, X9, X10
3	1751251	1751251	1751251	X1, X2, X4
1	1751264	1751264	1751264	X3
2	1N4148	1N4148DO35-10	DO35-10	D10, D11
5	100nF	C2.5/2	C2.5-2	C1, C2, C5, C7, C8
1	10nF	C2.5/2	C2.5-2	C6
1	470nF	C2.5/2	C2.5-2	C4
1	470µF/25V	Elko E3.5-10	E3,5-10	C3
1	1N4007	DIODE-D-10	D-10	D7
1	DTS63N	DTS63N	DT63N	T1
1	LED green	LED3MM	LED3MM	LED6
1	LED red	LED3MM	LED3MM	LED5
1	MAX481CPA	MAX481CPA	DIL08	IC1
2	BSP76	MOSFET_IRLL024	MOSFET_SOT-223	Q3, Q4
4	MOUNT-PAD-ROUND3.2	MOUNT-PAD-ROUND3.2	3,2-PAD	H1, H2, H3, H4
1	JP2	MTA02-100	10X02MTA	J1
1	LM7301	LM7301	SOT23-5	IC3
5	1.2k	R 0207/10	R-10	R2, R3, R4, R11, R12
4	10k	R 0207/10	R-10	R5, R7, R8, R9
1	2.2k	R 0207/10	R-10	R13
2	20k	R 0207/10	R-10	R6, R10
1	4.7k	R 0207/10	R-10	R14
1	120	R 0207/10	0207/10	R1
1	R-785.0-1.0	R-78C3.3-1.0	SIP3	IC4
1	SH166	Fuse TR5	TR5	F1
1	CN1812	VARISTORCN1812	CT/CN1812	R49
2	PTR2p	W232-2	W232-2	K1, K2
1	Mini D1 ESP32	WEMOS_D1_32	WEMOS-D1-MINI-32	IC2

Appendix B - Bibliography

Books (in German)
Thomas Tilli: Automatisierung mit Fuzzy Logic

Ulrich Schulte: Einführung in Fuzzy Logic Fortschritt durch Unschärfe

Adolf Grauel: Fuzzy-Logic Einführung in die Grundlagen mit Anwendungen

Peter Beater: Physical Computing – Automatisieren mit dem Arduino

Gerd Schulz: Regelungstechnik Mehrgrößenregelung – Digitale Regelungstechnik – Fuzzy-Regelung

Papers
Lotfi a. Zadeh: Fuzzy Logic = Computing with Words

B. Wamister: L12 Fuzzy Logic

Jim Sibigtroth: A Graphical Introduction to Fuzzy Logic

Jim Sibigtroth: Fuzzy Logic for Embedded Microcontrollers

TI Timothy A. Adcock: (SPRA028) What is Fuzzy Logic? An Overview of the Latest Control Methodology

TI Andreas Dannenberg: (SLAA235) Fuzzy Logic Motor Control with MSP430

Dr. Varodom Toochinda: Digital PID Controllers

Michael Hubo/ Matthias Ortmann: Einführung in die Fuzzy_Logik

Appenix C • Weblinks

Arduino web page
https://www.arduino.cc/en/software

Nextion Display Webpage
https://nextion.tech/

Download page of the Codeblocks IDE
https://www.codeblocks.org/

Library for Modbus RTU Master
https://github.com/reaper7/SDM_Energy_Meter

FELJC Fuzzy-Control
https://staff.ltam.lu/feljc/school/asser_t3/Fuzzyregler.pdf

Jim Sibigtroth Fuzzy Logic for Embedded Microcontrollers
https://circuitcellar.com/wp-content/uploads/2015/02/Sibigtroth-CC-1995-03-Issue-056.pdf

A Graphical Introduction to Fuzzy Logic
https://shrubbery.net/~heas/willem/PDF/Motorola/fuzzy.html

Ball and Beam example
https://my.hidrive.com/share/njy0u0ljzu

Crane Controller
https://www.codeproject.com/Articles/23387/Introduction-to-C-and-Fuzzy-Logic

Control Notes from Jacques Smuts, Ph.D., P.E.
https://blog.opticontrols.com/

Fuzzy Logic in C Dr. Dobbs Journal
https://www.chebucto.ns.ca/Science/AIMET/archive/ddj/fuzzy_logic_in_C/

Fuzzy Logic in C Dr. Dobbs Journal Update
http://www.chebucto.ns.ca/Science/AIMET/archive/ddj/fuzzy_logic_in_C_-_an_update/

Fuzzy and PID online
https://www.rentanadviser.com/pid-fuzzy-logic/pid-fuzzy-logic.aspx

Wikipedia Fuzzy Control
https://en.wikipedia.org/wiki/Fuzzy_control_system

Book and basics on neural networks
http://www.dkriesel.com/science/neural_networks

Index

1-Wire® 66
3D representation 210

A

ADC characteristic curve 35
ADC Error 35
aluminium block 98, 135
angular velocity 149
anti-lock braking systems 156
anti-windup 131
applyDamping() 177
Arduino 10
Arduino Nano 11
Arduino Nano Test board 13
Arduino UNO 10
artificial intelligence 279
Åström 127
ATmega328P 11
attenuation function 234
automotive industry 71, 124
average 37

B

backprop 279
backPropagation() 299
backpropagation algorithm 279, 316
backward propagation 282
Bell Memberships 174
bias value 274
Biological neuron 273
Bluetooth connectivity 327
Board Manager 22
BSP76 87
BTS7960 89
buttons 28
button states 52

C

calculated output 315
Calibration 67, 260
calibrator 34, 60
Callendar van Dusen 58
C-functions 168
characteristic curve 170
charge controller 326
Chromel-Alumel 61
circuit 21
classification 282
Clock Frequency 12
CodeBlocks 40
coefficients 85
cold junction compensation 64
color recognition 296
connection status 51
control deviation 186
control law 251
ControllerDisplay 24
controller equation 124
control signal values 123
control strategies 290
control value 177
conveyor belts 220
CounterSeconds 28
CSV file 150
Cubic splines 81

D

damping factor 233, 264
debounce period 31
decision-making processes 155
degrees of membership 229
determining the weights 288
development environment 21, 23
difference equation 131
direction of rotation 221
disturbances 196
DS18B20 65

E

EEPROM 12
efficiency gains 156
Elementa 63
Elmer Sperry 124
energy consumption 321, 332
energy meters 19
equation of motion 146
Error feedback 280
ESP32 13

ESP32BoardModbusRTUControl 19
ESP32 Fuzzy PID Board 18
ESP32 Fuzzy PID Controller Board 17
ESP32 Fuzzy PID Test board 16
ESP32 WebSocket server 47
Euler method 95
Excel 277

F

feedforward networks 272, 276
feed-in power 330
FILTER_SIZE 262
Flash Memory 12
flow temperature 214
Flow temperatureHeating 211
Force sensors 70
forward pass 314
Frank Rosenblatt 272
fuzzification 155
fuzzy controllers 42
fuzzy Gaussian controller 289
Fuzzy Leftopen 158
Fuzzy Triangle 158

G

Gauss Memberships 172
GCC 40
Geoffrey Hinton 279
gradients 279
graphic elements 23
gravity method 228

H

Hägglund 127
Hall sensors 221
H-bridge 89
heat capacity 97
heating system 97
heat output 99
hidden layers 276
high-current PWM motor drives 90
HTML code 47
http request 323
HTTP server instance 51
HX711 72
hysteresis 122

I

I²C interface 303
Incremental encoder 69
inference phase 155
Infineon 89
initArrayNTC() 263
input neurons 318
input patterns 313
integral 143
integrate expert 155
intercept 73
interpolation 35, 60
interval 37
Inverted pendulum 147
IRAM_ATTR 225

J

John Hopfield 272
JSON 19, 50

L

layouts 21
learning rate factor 299
linear interpolation 261
linear regression 75
Linear regression 73
loop() 145
Lotfi Zadeh 155

M

machine learning 279
Magnetic field change 224
Marvin Minsky 272
MAX function 209
mean value 229
measured values 39
measuring amplifier 59
membership 157
micro-inverter 320
micro-SD 24
MIN function 209
Modbus 21
Modbus TCP 15
MotorPositionControlFuzzy() 245
MOTSENS1 225
moving average 233

MPPT technology 326
MQTT 19

N

network calculation 304
network performance 307
neuron 274
Nextion 23
Nextion display 248
Nicholas Minorsky 124
NTC1 sensor 262
NTC thermistors 54
NTC voltage divider 77

O

off-grid systems 326
on/off relay 127
open-source platform 10
optocoupler 88
oscillation method 126
oscillation period 126
output layer 292
overvoltage protection 323

P

pattern recognition 282
PeakAndPeriod() 133
Peltier elements 186
Perceptrons 272
Phase shift 224
pH measuring amplifier 68
pH sensors 67
PID controller equation 125
PLC 15
position algorithm 132
position control 70, 132
Power regulator 91
program examples 15
PT2 controlled 93
PT2 system 95
PT100 57
PT1000 57
Pulse heating control 99
PWM channels 113

R

random numbers 312
read_btn1() 28
real-time applications 43
real-time control 50
real-time monitoring 333
RELAIS1 28
relay method 127
relay states 52
relay status 332
responsive design 46
REST 19
RGBC sensor 303
Robustness 252
Ronald J. Williams 279
Rotary encoder 68
RP2040 14
RS485 20
RTD 57
Rule evaluation 159

S

SDA 294
seconds counter 106
SerialPortReader 42
SerialPortTerminal.exe 86
Seymour Papert 272
Shelly 20
Shelly 3EM 321
Shelly Plus 1PM 323
Sigmoid 206
sigmoid activation function 276
singletons 159
sliders 48
slidingModeControllerEXP 253
sliding surface 250
sliding surface equation 269
slope 73
solar power 320
Soyo Control 332
Speed control 70
speed of the motor 221
Spline interpolation 77
SRAM 12
state machine 32
Strain gauges 71

subtractor 59
system response 96

T
target position 241
TCS74725 294
temperature controller 186, 194
temperature control process 293
temperature curve 196
temperature profile 183
temperature sensors 54
Tensilica 14
textbox 27
TFT file 24
Thermocouple amplifier 63
Thermocouples 61
thermoelectric effect 61
Thyristor 91
ticker 26
time measurement 30
timer 24
timer callback 30
timerCallback() 105
timer function 26
Tracking performance 251
training process 273
trapezoidal 168
Two-point controller 119

V
Victron Connect 328
voltage divider 55

W
WebSocket clients 50
WebSocket connection 48
WebSockets 43
WebSocket server 43
Windows PV Monitor 329

X
XML 19
XOR problem 277
XOR table 278

Y
Yann LeCun 279

Z
Ziegler-Nichols 126

elektor books

Control Engineering with Fuzzy Logic

Practical Applications and Projects with Arduino, ESP32, and RP2040

Immerse yourself in the fascinating world of control engineering with Arduino and ESP32! This book offers you a practical introduction to classic and modern control methods, including PID controllers, fuzzy logic, and sliding-mode controllers.

In the first part, you will learn the basics of the popular Arduino controllers, such as the Arduino UNO and the ESP32, as well as the integration of sensors for temperature and pH measurement (NTC, PT100, PT1000, and pH sensor). You will learn how to use these sensors in various projects and how to visualize data on a Nextion TFT display.

The course continues with an introduction to actuators such as MOSFET switches, H-bridges, and solid-state relays, which are used to control motors and actuators. You will learn to analyze and model controlled systems, including PT1 and PT2 control.

The book focuses on the implementation of fuzzy and PID controllers for controlling temperature and DC motors. Both the Arduino UNO and the ESP32 are used. The sliding-mode controller is also introduced.

In the second-to-last chapter, you will explore the basics of neural networks and learn how machine learning can be used on an Arduino.

In the last chapter, there is a practical example of a fuzzy controller for feeding electricity into the household grid.

This book is the perfect choice for engineers, students, and electronics engineers who want to expand their projects with innovative control techniques.



Josef Bernhardt discovered his passion for electronics at a young age. He took his first steps in programming in the 1980s with the Apple II and the IBM PC. With over 30 years of experience in electronics development at the University of Regensburg, Josef Bernhardt is an expert in his field. There, he worked in both electronics and software development. Through his involvement in his own SMD production, he successfully realizes electronics projects for customers. His passion for electronics and programming, as well as his constant interest in new technologies, have accompanied him since his youth and have characterized his entire life.

Elektor International Media
www.elektor.com

ISBN 978-3-89576-660-2
9 783895 766602

elektor
design > share > earn