Fachhochschule
Dortmund
University of Applied Sciences and Arts

Master Thesis

Design and Implementation of a Mixed-Signal Processing Chain for the Optical Determination of Rotation Angles

Embedded Systems Engineering
Department of Computer Science

First Examiner: Prof. Dr. Michael Karagounis
Second Examiner: Felix Schneider

Submission date: 06.10.2023
Submitted by: Ladan Alaee

ABSTRACT

The aim of this master thesis is the design and implementation of mixed-signal processing chain for the optical determination of rotation angles by means of four sensors implemented as photodiodes with integrated polarization filters and a high-precision CORDIC hardware design implemented on an FPGA in Verilog. Furthermore, a light source and a polarizer are integrated in the measurement setup which is configured using an QT application.

Table of Contents

1 Introduction....................19

2 POLDI principles....................21

2.1 Polarisation....................21

2.2 Measurement principle of the POLDI....................23

3 Signal processing concepts....................28

3.1 CORDIC algorithm....................28

3.1.1 Basic idea and concepts....................28

3.1.2 Generalization....................34

3.1.3 Application of the CORDIC algorithm for the calculation of functions 36

4 Light Source and Polarizer....................45

4.1 Hyperchromator....................45

4.2 Polarizer....................46

4.2.1 Double Glan-Taylor Calcite Polarizers (DGL10)....................47

4.2.2 Motorized Rotation Stage and Mount....................47

4.2.3 K-Cube™ Brushed DC Servo Motor controller (KDC101)....................48

5 Sensor Readout Electronics....................51

5.1 Power Supply and Reference Voltage Board....................52

5.1.1 Generated voltages and outputs....................52

5.1.2 Components of the Board....................53

5.2 Level shifter....................59

5.2.1 Level Shifter Board Structure....................60

5.2.2 Components of the Level Shifter....................62

5.3 Analog to Digital Converter Circuit Board (ADC) 64
5.3.1 Components of the ADC board 65
5.4 Poldi sensor board 78
5.5 Cmod A7-35T 79
6 Design of CORDIC blocks in Verilog 81
6.1 Requirements 81
6.2 Verilog design of the division calculation 83
6.2.1 “divv” submodule 83
6.2.2 Main module “divv_top” 88
6.2.3 Simulation 92
6.3 Verilog design of the Arcsine function 99
6.3.1 “Arcsin” Submodule 100
6.3.2 Main module “arcsin_top” 103
6.3.3 Simulation 105
7 Rotation Angle Processing Unit 111
7.1 “preprocessing” module 113
7.1.1 Functionality 113
7.1.2 Simulation of the Preprocessing module 119
7.2 “Processing” module 122
7.2.1 Functionality 122
7.2.2 Simulation 124
7.3 “Postprocessing” module 125
7.3.1 Functionality 125
7.3.2 Simulation 128

7.4 “sm_signalprocessing” module 130
7.4.1 Functionality 130
7.5 “signalprocessing” simulation 132
7.6 Summary 134
8 Integration of the “signalprocessing” module into the POLDI readout system 135
8.1 Poldi readout and control system 135
8.2 Integration of the Signal Processing Unit 137
8.2.1 “Datensendung” module 138
8.2.2 Design structure with integrated signal processing unit 142
8.2.3 Simulation 144
8.3 Summary 144
9 QT widget application for sensor readout control 145
9.1 Main Window 145
9.1.1 AD5686 field 145
9.1.2 Mode selection field 146
9.1.3 AD7980 field 148
9.1.4 AD5544 field 148
9.1.5 Measurement and Test field 149
9.2 Developed classes for the GUI and sensor readout control 159
9.2.1 The “communication_EvalBoard” class 159
9.2.2 The “MojoSerial” class 164
9.2.3 The “Angle_measurement” class 196
9.2.4 The “linearity_test” class 198

9.2.5 The “Angle_calculation” class 203
9.2.6 The “hyperchromator” class 209
9.2.7 The “kdc101” class 211
9.2.8 The “light_source_sweep” class 214
9.2.9 The “light_source_test” class 216
9.2.10 The “polarization_measurement” class 217
9.2.11 The “polarizer_sweep” class 218
9.2.12 The “Wavelength_polarizer_sweep” class 220
9.2.13 The “Wavelength_ sweep” class 226
10 Measurement results 230
10.1 Single Readout of sensor channels 230
10.2 Continuous Angle Measurement 231
10.3 “Linearity_Test” button 232
10.4 Test with Manual Polarizer Angle Setting 233
10.5 Sweep of Polarizer Angle 234
10.6 Angle Measurement 235
10.7 Light Source Test 237
10.8 Wavelength Sweep 238
10.9 Wavelength and Polarizer Angle Sweep 238
11 Conclusion & Outlook 240
Bibliography 241
Appendix A: Controlling the light source 244
1 Principle of low-level remote control 244
2 Communication Protocol 246

2.1 Binary command message structure..246

2.2 List of commands 247

2.2.1 Set Target Position 247

2.2.2 Update Position 248

2.2.3 Set Speed..248

2.2.4 Reset drive..248

2.2.5 Read current pos...249

2.2.6 Call function...249

Appendix B: Polarizer motor control..251

1 Communications Protocol 251

1.1 General message exchange rules..252

1.2 Conversion between position, velocity and acceleration values in standard physical units and their equivalent Thorlabs Software parameters.253

2 Motor Control Messages 254

2.1 MGMSG_MOD_IDENTIFY 255

2.2 MGMSG_MOD_SET_CHANENABLESTATE................................255

2.3 MGMSG_MOT_GET_POSCOUNTER...255

2.4 MGMSG_MOT_SET_VELPARAMS, MGMSG_MOT_REQ_VELPARAM, MGMSG_MOT_GET_VELPARAMS 256

2.5 MGMSG_MOT_MOVE_HOME 257

2.6 MGMSG_MOT_MOVE_HOMED 258

2.7 MGMSG_MOT_MOVE_RELATIVE..258

2.8 MGMSG_MOT_MOVE_COMPLETED 259

2.9 MGMSG_MOT_MOVE_ABSOLUTE 260

Appendix C: all command codes of the communication protocol 262

Table of Figures

Figure 2.1 The concept of angle measurement with a POLDI sensor [3] 21

Figure 2.2 principle of polarisation..22

Figure 2.3 the passage of light through two polarization filters 22

Figure 2.4 a) Signals without normalization and offset correction, b) After difference formation and offset correction of the current pairs, c) After normalization 24

Figure 2.5 Case distinction for angle calculation 27

Figure 3.1 Cartesian coordinate system showing a vector (x0, y0) rotated by the angle phi to the vector (xn,yn) 29

Figure 3.2 Concept of the rotating mode 36

Figure 3.3 Concept of vectoring calculation for arcsine....................................38

Figure 3.4 The rotations of the first two iterations 41

Figure 4.1 The Mountain Photonics Hyperchromator with the mounted Energetiq EQ-99X light source [11]..46

Figure 4.2 DGL10 Polarizer Mounted in an SM1L20 Lens Tube and PRM1 Rotation Mount [12]..48

Figure 4.3 K-Cube Brushed DC Servo Motor controller (KDC101) 50

Figure 5.1 Circuit diagram for the power supply and reference signals [14].....52

Figure 5.2 Extract of the circuit diagram with the wiring of the LDO voltage regulator ICs [14] 55

Figure 5.3 Symbol of the TSV914AIDR operational amplifier device [17] 55

Figure 5.4 Circuit diagram for generating the voltage RELNEGREF [14]........56

Figure 5.5 Circuit for generating loadable reference voltages [18]....................57

Figure 5.6 Circuit for generating the voltages VREFTIA and 1V_DGND [14] 58

Figure 5.7 Layout of the level shifter board 61

Figure 5.8 Level shifter circuit diagram in Altium Designer 61
Figure 5.9 Circuit for generating the level shifter levels [14] 62
Figure 5.10 Switching symbol of the multiplexer according to data sheet [20].63
Figure 5.11 Multiplexer's component in Altium Designer 63
Figure 5.12 Schematic of the Analog to Digital Converter board......................64
Figure 5.13 Layout of the ADC board 64
Figure 5.14 Circuitry of the ADC [21]. 66
Figure 5.15 Circuitry of the bandgap voltage reference [15]. 67
Figure 5.16 Schematic diagram of the MDAC with integrated feedback resistor [22] 68
Figure 5.17 Interconnection of the MDAC with the operational amplifier wired as a transimpedance amplifier [14] 69
Figure 5.18 Diagram for calculating the current to voltage conversion factor of the POLDI sensor readout circuit 71
Figure 5.19 Output voltage of the POLDI readout chain related to VREFTIA as a function of the DAC setting of channels...76
Figure 5.20 The POLDI sensor circuit board 79
Figure 6.1 Pipelining example 82
Figure 6.2 The concept for the realization of the division.................................83
Figure 6.3 Schematic of the division calculation for i=3...................................92
Figure 6.4 Behavior simulation of the division calculation for i=14.................94
Figure 6.5 Results of the time simulation of the division with i = 14 for a clock of 50 MHz 98
Figure 6.6 Concept for the realization of the CORDIC algorithm for the computation of the arcsine with FPGA...99
Figure 6.7 The schematic diagram of the arcsine calculation for i=3 106
Figure 6.8 Behavior simulation of the signals of the arcsine calculation for i=14 107
Figure 7.1 Block diagram “signalprocessing” module 111

Figure 7.2 Structure of the “signalprocessing” module 112
Figure 7.3 Structure of the “preprocessing” module 114
Figure 7.4 The state diagram of the “fsm_preprocessing” module 115
Figure 7.5 The state diagram of the “fsm_preprocessing” module 116
Figure 7.6 Simulation of the sampling process in the “preprocessing” module assuming 10 samples..119
Figure 7.7 Simulation of normalized results in the “preprocessing” module...120
Figure 7.8 Self checking simulation of normalized results in the “preprocessing” module..121
Figure 7.9 Structure of the “processing” module 122
Figure 7.10 State diagram of the “sm_processing” module 123
Figure 7.11 Simulation result in the “processing” module...............................124
Figure 7.12 Simulation results of all possible input signals in the “Processing” module..125
Figure 7.13 Structure of the “postprocessing” module.....................................126
Figure 7.14 State diagram of the “sm_postprocessing” module 127
Figure 7.15 State diagram of the “sm_postprocessing” module 127
Figure 7.16 Simulation result in the “postprocessing” module 129
Figure 7.17 Simulation results for different input values in the “postprocessing” module..130
Figure 7.18 State diagram 1 of the “sm_signalprocessing” module.................131
Figure 7.19 State diagram 2 of the “sm_signalprocessing” module.................131
Figure 7.20 Simulation in the “Processing” module 133
Figure 7.21 Simulation results for different input values in the “postprocessing” module in a self-checking testbench 133
Figure 8.1 Structure of the VHDL Design [9] 136
Figure 8.2 Structure of the “Datensendung” module..139
Figure 8.3 Hierachical state-diagram of the state-machine in the module “sm_Datensendung”...140

Figure 8.4 Hierarchical state “angles” of the “sm_Datensendung” module.....141
Figure 8.5 Structure of the integrated Design..143
Figure 8.6 Simulation of Integrated Design...144
Figure 9.1 Main window of the GUI 145
Figure 9.2 “Angle measurement” window 147
Figure 9.3 “Linearity_test” window 150
Figure 9.4 “Polarization_Test” window 151
Figure 9.5 “Polarization sweep Test” window 153
Figure 9.6 “Angle Calculation” window 154
Figure 9.7 “Light Source Test” window...155
Figure 9.8 “Wavelength_sweep” window 157
Figure 9.9 “Wavelength_Polarizer_Sweep” window 158
Figure 10.1 The display of readout voltages..230
Figure 10.2 Plot of calculated angles in continuous mode 231
Figure 10.3 Linearity test for gain sweep 232
Figure 10.4 Linearity test for offset sweep 232
Figure 10.5 Manual polarizer test 234
Figure 10.6 Polarizer sweep test 235
Figure 10.7 Angle Measurement for a polarizer angle sweep from 0° to 180° degrees with angle calculation in software 236
Figure 10.8 Angle Measurement for a polarizer angle sweep from 0° to 180° degrees with angle calculation in the FPGA hardware....................................236
Figure 10.9 Light Source communication and configuration to read intensity signals...237
Figure 10.10 Light source test with wavelength sweeps and polarizer at 90° .238
Figure 10.11 Sensor sensitivity as a function of the polarizer angle and for light with different wavelengths...239

Table of Tables

Table 3.1 Error due to the approximation of the scaling factor with the limit value 32
Table 3.2 Scaling factors for the generalized CORDIC [8, p. 18]. 35
Table 3.3 Calculation steps for Arcsine (0.9) with the CORDIC algorithm 39
Table 3.4 Calculation steps for Arcsine (0.99) with the CORDIC algorithm 40
Table 5.1 Measured values compared with the results of the formula. 77
Table 6.1 Calculation results for “x_in” = 14 and “y_in” = 1 94
Table 6.2 Theoretical calculations of 1/14 95
Table 6.3 Decimal calculation results after 15 iterations for different quotients 96
Table 6.4 Decimal arcsine calculation results with “Arg” = 0.5 108
Table 6.5 Theoretical calculations of arcsin(0.5) 109
Table 6.6 Calculation results after 15 iterations for different arguments 109

Table of Source Codes

Source code 6.1 Declaration of ports of the “divv” module 84
Source code 6.2 Conversion to internal signals 85
Source code 6.3 Process for performing an iteration step in the “divv” module87
Source code 6.4 Declaration of variables and ports for the “divv_top” module 89
Source code 6.5 Defining the types for the internal and output signals for the division calculation 90
Source code 6.6 Transformation of the input signal for y to 32-bit length 90
Source code 6.7 Generation and connection of the individual modules for the division calculation 91
Source code 6.8 Declaration of variables and ports for the “arcsin” module...100
Source code 6.9 Definition of the internal signals “arcsin” module 101
Source code 6.10 The process of performing an iteration step in the “arcsin” module..102
Source code 6.11 Declaration of variables and ports for the module “arcsin_top” 103
Source code 6.12 Generation and connection of the individual modules for the arcsine calculation..105
Source Code 9.1 Definition of variables within the header file of the “communication_EvalBoard” class 159
Source Code 9.2 Definition of the constructor and destructor of the “communication_EvalBoard” class 160
Source Code 9.3 Definition of the “sendDACdata(QString address, QString commandandchannel, QString data)” function within the “communication_EvalBoard” class 162
Source Code 9.4 Definition of the “sendADCdata(QString Address)” function within the “communication_EvalBoard” class 162

Source Code 9.5 Definition of the “startreadcont()” and “stopreadcont()” functions within the “communication_EvalBoard” class to start and stop the continuous readout of all ADCs respectively 163
Source Code 9.6 Definition of the “read_serial()” function within the “communication_EvalBoard” class for receiving data via USB 164
Source Code 9.7 Definition of the “getdata()” function within the “communication_EvalBoard” class to return the received data 164
Source Code 9.8 The constructor and destructor of the “MojoSerial” class 165
Source Code 9.9 Definition of the “decision()” function within the “MojoSerial” class 167
Source Code 9.10 Definition of the “ad7980()” function within the “MojoSerial” class 168
Source Code 9.11 Definition of the “writeinBrowser()” function within the “MojoSerial” class 169
Source Code 9.12 Definition of the “test_measurements()” function in the “MojoSerial” class to update different windows or user interface elements....172
Source Code 9.13 Definition of the “read_angle()” function within the “MojoSerial” class designed to process angle measurement data acquired from the FPGA...173
Source Code 9.14 Definition of variables in the “on_pushButton_clicked()” function of DAC “AD5686” field within the “MojoSerial” class 174
Source Code 9.15 Save the input of the command bits and convert to an integer value in the “on_pushButton_clicked()” function of DAC “AD5686”............174
Source Code 9.16 Form a string based on the selected channels in the “on_pushButton_clicked()” function of the DAC “AD5686” field 175
Source Code 9.17 Converting captured voltage input to a 16-bit hexadecimal string, transforming into the complete 16 data bits with proper zero-padding in the “on_pushButton_clicked()” function of the DAC “AD5686” field 176

Source Code 9.18 Validation of input conditions in the "on_pushButton_clicked()" function of the DAC "AD5686" field 178
Source Code 9.19 Define the mode of ADCs in the "on_Mode_set_clicked()" function within the "MojoSerial" class 179
Source Code 9.20 Configure of the DAC settings to initial values at zero in the "on_reset_clicked()" function within the "MojoSerial" class 180
Source Code 9.21 Instantiate an object of the "Angle_measurment" class in the "on_Angle_meaurment_clicked()" function..180
Source Code 9.22 Set flags for measurement control in the "on_Angle_meaurment_clicked()" function..180
Source Code 9.23 Definition of the "on_Angle_meaurment_clicked()" function within the "MojoSerial" class 181
Source Code 9.24 Definition of the "get_startcontdata()" function within the "MojoSerial" class 182
Source Code 9.25 Definition of the "get_stopcontdata()" function within the "MojoSerial" class 182
Source Code 9.26 Definition of the "angleform_exit()" function within the "MojoSerial" class 182
Source Code 9.27 Definition of the "on_Buttonlesen_clicked()" function within the "MojoSerial" class 185
Source Code 9.28 Definition of the "on_SliderU6A_valueChanged(int value)" function within the "MojoSerial" class to update the value of "BarU6A" to match the changed value of "SliderU6A" in response to the user's interaction 185
Source Code 9.29 Definition of the "on_SliderU6A_sliderReleased()" function within the "MojoSerial" class 186
Source Code 9.30 Definition of the "AD5544(int progress, QString Adresse, QString Kanal)" function within the "MojoSerial" class 187

Source Code 9.31 Definition of the “on_linearity_test_clicked()” function within the “MojoSerial” class 188
Source Code 9.32 Definition of the “ADC_setting()” function within the “MojoSerial” class 189
Source Code 9.33 Definition of the “on_polarization_measurement_clicked()” function within the “MojoSerial” class 190
Source Code 9.34 Definition of the “on_polarizer_sweep_clicked()” function within the “MojoSerial” class 191
Source Code 9.35 Definition of the “on_angle_calculation_clicked()” function within the “MojoSerial” class 193
Source Code 9.36 Definition of the “on_light_source_test_clicked()” function within the “MojoSerial” class 194
Source Code 9.37 Definition of the “on_WL_Sweep_clicked()” function within the “MojoSerial” class 195
Source Code 9.38 Definition of the “on_WL_ Polarizer _Sweep_clicked()” function within the “MojoSerial” class 196

List of abbreviations

DAC	Digital to Analog Convertor
ADC	Analog Digital Converter
VHDL	Very High-Speed Integrated Circuit Hardware Description Language
POLDI	Polarization Sensitive Photo Diodes
FPGA	Field Programmable Gate Array
TIA	Trans-Impedance Amplifier
SPI	Serial Peripheral Interface
MSB	Most Significant Bit
LSB	Least Significant Bit
LUT	Look Up Table
GUI	Graphical User Interface

1 Introduction

Due to the increasing digitalization of technical systems, there is a great need for sensors and actuators, which as an interface to the analog world enable a digital core usually implemented as a microprocessor, to perceive the environment and actively influence it. Angle sensors play a major role in many applications, for example in mechatronic systems since the detection of angles and speeds of rotation is of particular importance for the control of precise motion sequences. In mechatronic systems today, many sensors are not integrated into silicon chips, but rather the sensor type best suited to the application is typically selected and combined with separate evaluation electronics. Microelectronic integration of sensors offers important advantages, such as very compact design, low weight, the possibility to integrate control and evaluation electronics, as well as other functions (calibration, communication interfaces, etc.), and very low production costs.

The company advICo microelectronic GmbH is investigating an optical measuring principle called POLDI (Polarization Sensitive Photo Diodes) for the detection of angles and rotational movements, to circumvent the limitations of conventional sensor systems. This optical measurement method has the potential to detect very high rotation rates and allows a larger distance between the components of the sensor system. The integration of sensors and electronics on a chip in CMOS technology not only provides energy efficiency and miniaturization but also cost efficiency. As a preliminary work, TIA amplifiers circuits convert the photocurrents of the diodes into defined voltages and make them available for further signal processing [10]. In addition, discrete electronics is used to digitize the signals after amplification and offset correction. In this work, these digitized signals are read in an FPGA and then digitally processed to an absolute angle.

The objective of this master thesis is the development of the mixed-signal processing chain for the optical determination of the rotation angle by means of four POLDI sensors and a CORDIC hardware design implemented on an FPGA using Verilog. Furthermore, the thesis includes the integration of the light source and polarizer into the measurement setup which is configured using an QT application.

2 POLDI principles

POLDI (Polarization Sensitive Photo Diodes) sensors measure the polarization of an incident light beam using photodiodes and polarization filters integrated on a chip in standard CMOS technology. Several photodiodes are integrated on a CMOS chip, each covered by a linear polarization filter, as shown in Figure 1. The polarization filters are realized in one of the metallization planes of the CMOS process by parallel conductive tracks, which lie at four different angles 0°, 45°, 90°, and 135° to each other. When the POLDI sensor is irradiated with linearly polarized light, each diode delivers a photocurrent whose current intensity depends on the angle between the polarization plane of the incident light and the orientation of the polarization filter. [3]

Each photodiode supplies a different photocurrent, which flows into a separate Transimpedance Amplifier (TIA). The generated voltages are applied to separate ADCs and are made available to a Field-Programmable Gate Array (FPGA), which can calculate the angle of the incident light using digital signal processing algorithms.

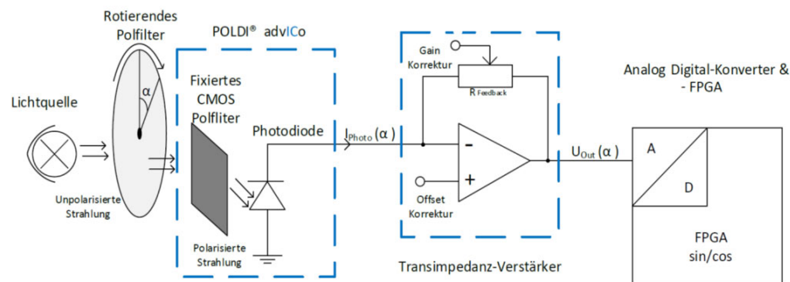


Figure 2.1 The concept of angle measurement with a POLDI sensor [3]

2.1 Polarisation

In the wave model, light is described as a wave that oscillates perpendicular to its direction of propagation, also known as a transverse wave. The polarization describes the specific direction of the oscillation. Natural light, such as sunlight,

is not polarized and oscillates randomly in all directions. To convert this light into polarized light requires a polarization filter. This polarization filter only allows light waves to pass that are parallel to the polarization axis of the filter, while all other light waves are absorbed by the filter. The light intensity decreases when passing the polarization filter. Figure 3.1 shows the principle of polarisation.

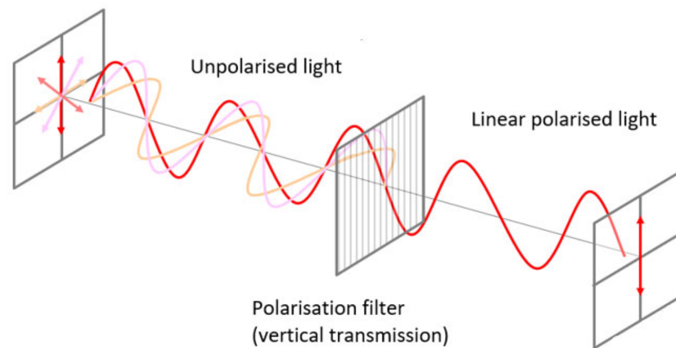


Figure 2.2 principle of polarisation

When using two polarizing filters, the orientation of their polarization planes to each other plays an important role in light transmission. If both filters have the same orientation this has no effect on the light intensity. If the polarizing filters are arranged perpendicular to each other, no light is transmitted. If the two polarization filters are aligned offset to each other, for example at 45°, the light can pass, but the polarization of the transmitted light corresponds to the direction of the second polarization filter. Figure 2.3 depicts the passage of light through two polarization filters.

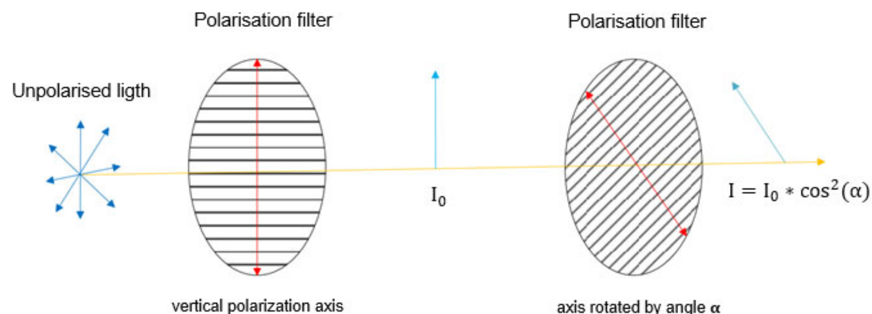


Figure 2.3 the passage of light through two polarization filters

According to which the transparency of two crossed polarizing filters depends on the relative angle φ of the filters to each other. To the intensity I of the light passing the two filters applies: $I \sim \cos^2(\varphi)$ which is called Malus law, where I_0 is the intensity of the fully polarized light before passing the second polarizing filter and α is the shift angle between the polarization planes of the polarizing filters.

2.2 Measurement principle of the POLDI

The measurement principle is based on Malus law. For the application as an angle sensor, linearly polarized light is required, which is generated by a polarizer in the beam path between light source and sensor and then falls on a photodiode with an integrated polarization filter. The current signal of the sensor is proportional to the intensity of the light, which passes the integrated polarization filter and contains an offset F_d, which is the dark current with additive noise. Malus law can be extended for the application to:

$$I_n = I_0 * \cos(\alpha + \alpha_n)^2 + F_d \tag{2.1}$$

here I_0 is the unknown intensity of the incident beam, α_n corresponds to the respective angles of 0°, 45°, 90°, or 135° of the integrated polarization filters of the sensors. By taking the difference between the 0° and 90° or 45° and 135° signal pairs and assuming that all diodes have the same dark current, the offset is eliminated.

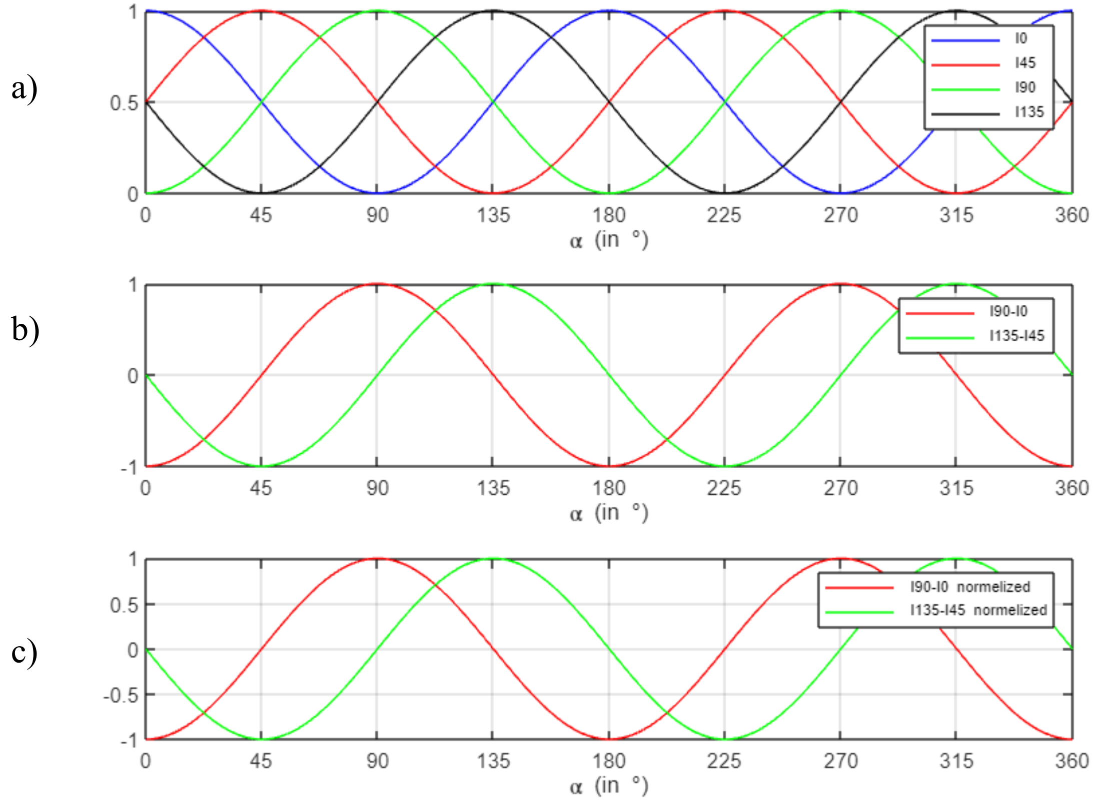


Figure 2.4 a) Signals without normalization and offset correction, b) After difference formation and offset correction of the current pairs, c) After normalization

To compensate for possible shifts on the Y-axis, the curves are normalized. The correction factors correspond to the minimum and maximum of the non-normalized differences.

$$I_{Norm} = \frac{2}{I_{Max} - I_{Min}} * \left(I_n - \frac{I_{Max} + I_{Min}}{2}\right) \tag{22}$$

Equation (1.2) is introduced into the difference pairs resulting in the following calculation steps:

$$\Delta_0 = I_{90°} - I_{0°} \tag{2.3}$$

$$\Delta_{45} = I_{135°} - I_{45°} \tag{2.4}$$

$$\Delta_0 = I_0 * \left(\cos^2\left(\alpha - \frac{\pi}{2}\right) - \cos^2(\alpha)\right) \tag{2.5}$$

$$\Delta_{45} = I_0 * \left(\cos^2\left(\alpha - \frac{3\pi}{4}\right) - \cos^2\left(\alpha - \frac{\pi}{4}\right)\right) \tag{2.6}$$

Equation (2.6) can be simplified as follows:

$$\frac{\Delta_{45}}{I_0} = \left(\cos^2\left(\alpha - \frac{3\pi}{4}\right) - \cos^2\left(\alpha - \frac{\pi}{4}\right)\right) \tag{2.7}$$

Using the addition theorem $\cos^2(x) = \frac{1}{2}(1 + \cos 2x)$ results in:

$$\frac{\Delta_{45}}{I_0} = \frac{1}{2}\left(1 + \cos\left(2\alpha - \frac{2 * 3\pi}{4}\right)\right) - \frac{1}{2}\left(1 + \cos\left(2\alpha - \frac{2\pi}{4}\right)\right) \tag{2.8}$$

Using the theorem $\cos x - \cos y = -2\sin\frac{x+y}{2} * \sin\frac{x-y}{2}$ follows:

$$\frac{2 * \Delta_{45}}{I_0} = -2 * \sin\left(\frac{2\alpha - \frac{2 * 3\pi}{4} + 2\alpha - \frac{2\pi}{4}}{2}\right) * \sin\left(\frac{-\frac{2 * 3\pi}{4} + \frac{2\pi}{4}}{2}\right) \tag{2.9}$$

$$\frac{-\Delta_{45}}{I_0} = \sin(2\alpha - \pi) * \sin\left(-\frac{\pi}{2}\right) \tag{2.10}$$

$$\frac{-\Delta_{45}}{I_0 * \sin\left(-\frac{\pi}{2}\right)} = \sin(2\alpha - \pi) \tag{2.11}$$

Applying the inverse function of the sin to the equation (1.11), we can solve the required angle α:

$$\text{arcsine}\left(\frac{-\Delta_{45}}{I_0 * \sin\left(\frac{-\pi}{2}\right)}\right) = 2\alpha - \pi \tag{2.12}$$

$$\alpha_{45} = \frac{1}{2}\left[\text{arcsine}\left(\frac{-\Delta_{45}}{I_0 * \sin\left(\frac{-\pi}{2}\right)}\right) + \pi\right] \tag{2.13}$$

Since $\sin\left(-\frac{\pi}{2}\right) = -1$ it follows for α_{45}:

$$\alpha_{45} = \frac{1}{2}\left[\text{arcsine}\left(\frac{\Delta_{45}}{I_0}\right) + \pi\right] \tag{2.14}$$

By a similar calculation, it follows for α_0:

$$\alpha_0 = \frac{1}{2}\left[\text{arcsine}\left(\frac{\Delta_0}{I_0}\right) + \frac{\pi}{2}\right] \tag{2.15}$$

As shown in Figure 2.4 (c), due to the restriction of the arcsine function, the range of α_0 is limited to values between 0° and 90°, and the range of α_{45} is limited to values between 45° and 135°. To cover the range of values between 0° and 180°, a case distinction is required as follows and is illustrated in Figure 3[7]:

- Range 0° to 45°: Both differences are negative.

$$\alpha_0 = \frac{1}{2}\left[\text{arcsine}\left(\frac{\Delta_0}{I_0}\right) + \frac{\pi}{2}\right] \tag{2.16}$$

$$\alpha_{45} = \frac{1}{2}\left[-\text{arcsine}\left(\frac{\Delta_{45}}{I_0}\right)\right] \tag{2.17}$$

- (2) Range 45° to 90°: Δ_0 is positive and Δ_{45} is negative.

$$\alpha_0 = \frac{1}{2}\left[\text{arcsine}\left(\frac{\Delta_0}{I_0}\right) + \frac{\pi}{2}\right] \tag{2.18}$$

$$\alpha_{45} = \frac{1}{2}\left[\text{arcsine}\left(\frac{\Delta_{45}}{I_0}\right) + \pi\right] \tag{2.19}$$

- (3) Range 90° to 135°: Both differences are positive.

$$\alpha_0 = \frac{1}{2}\left[-\arcsin\left(\frac{\Delta_0}{I_0}\right) - \frac{3\pi}{2}\right] \tag{2.20}$$

$$\alpha_{45} = \frac{1}{2}\left[\arcsin\left(\frac{\Delta_{45}}{I_0}\right) + \pi\right] \tag{2.21}$$

- (4) Range 135° to 180°: Δ_0 is negative and Δ_{45} is positive.

$$\alpha_0 = \frac{1}{2}\left[-\arcsin\left(\frac{\Delta_0}{I_0}\right) - \frac{3\pi}{2}\right] \tag{2.22}$$

$$\alpha_{45} = \frac{1}{2}\left[-\text{arcsine}\left(\frac{\Delta_{45}}{I_0}\right) - 2\pi\right] \tag{2.23}$$

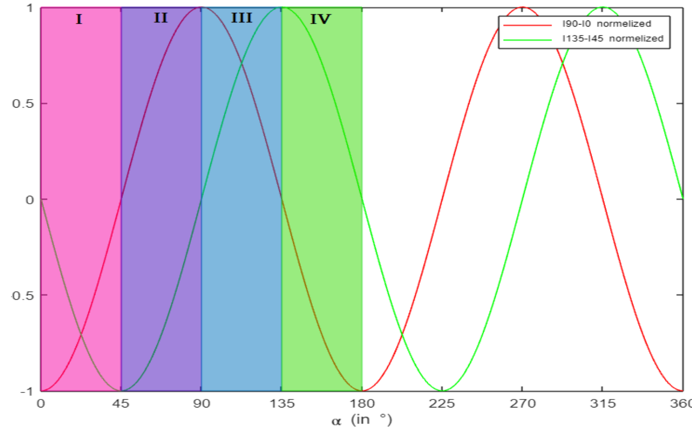


Figure 2.5 Case distinction for angle calculation

For the range from 180° to 360° the same case distinctions apply.

As can be seen from equations (2.16)- (2.23) the extraction of the rotation angle from the POLDI sensors requires the calculation of divisions and the arcsin function. The theoretical basis for the implementation of these mathematical expressions in hardware is based on the CORDIC algorithm which is described in the next chapter.

3 Signal processing concepts

As it became clear in the previous chapter, the calculation of the rotation angles requires the implementation of various mathematical operations, such as multiplication and division, and the evaluation of trigonometric inverse functions in the FPGA.

There are different realization possibilities. In this project, emphasis is given to a simple solution which requires only a small amount of hardware. Therefore, the CORDIC algorithm is used for the realization of the mathematical operations and functions mentioned above, since this algorithm allows an implementation based on addition/subtraction and shift operations.

In this chapter, the theoretical basics of the CORDIC algorithm and the selected method for computing the division and the arcsine function using the CORDIC algorithm in Verilog and the implementation on the FPGA are explained.

3.1 CORDIC algorithm

3.1.1 Basic idea and concepts

CORDIC is the abbreviation of COrdinate Rotation DIgital Computer and was introduced by Jack. E. Volder [1] as serial arithmetic or the calculation of trigonometric functions. In 1971 J.S. Walther [6] extended Volder's algorithm in such a way that the computation of all elementary functions became possible nowadays. Besides the serial architecture further implementation approaches exist like, for example, the iterative, the combinatorial as well as the pipelined CORDIC architecture. In the following the mathematical background of the classical CORDIC according to Volder is explained.

The idea behind the CORDIC algorithm is the rotation of a vector by multiplying it by a sequence of constant angles. For these multiplications, only factors are used which are a power of the number two. Thus, these multiplications can be

expressed in binary arithmetic with shifts and additions, which avoids the requirement of a general multiplier circuit.

The algorithm starts from a vector (x_0, y_0) which, as can be seen in Figure 5.1, rotates around the origin of a Cartesian coordinate system by the angle φ.

A source vector (x_0, y_0) can be described by polar coordinates in this Cartesian coordinate system as follows:

$$x_0 = r * \cos(\alpha) \tag{3.1}$$

$$y_0 = r * \sin(\alpha) \tag{3.2}$$

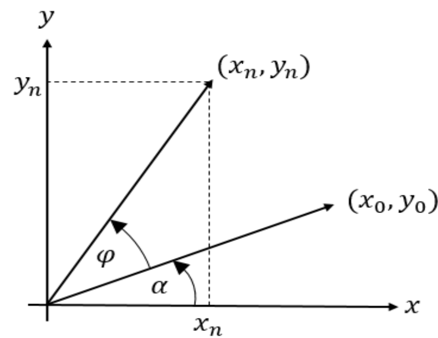


Figure 3.1 Cartesian coordinate system showing a vector (x0, y0) rotated by the angle phi to the vector (xn,yn)

For the desired vector (x_n, y_n) the following equations can be set up:

$$x_n = r * \cos(\alpha + \varphi) \tag{3.3}$$

$$y_n = r * \sin(\alpha + \varphi) \tag{3.4}$$

By applying the addition theorems for sine and cosine and substituting, the definition of the initial coordinates according to equations (3.1) and (3.2) follows:

$$x_n = r * \cos(\alpha) * \cos(\varphi) - r * \sin(\alpha) * \sin(\varphi) \tag{3.5}$$
$$= x_0 \cos(\varphi) - y_0 \sin(\varphi)$$

$$y_n = r * \sin(\alpha) * \cos(\varphi) + r * \cos(\alpha) * \sin(\varphi) \tag{3.6}$$

$$= y_0 \cos(\varphi) + x_0 \sin(\varphi)$$

By factoring out cos (φ), the equations can be written by means of a tangent.

$$x_n = \cos(\varphi)\ [x_0 - y_0 * \tan(\varphi)] \tag{3.7}$$

$$y_n = \cos(\varphi)\ [y_0 + x_0 * \tan(\varphi)] \tag{3.8}$$

The CORDIC algorithm only angles φ are chosen whose tangent can be represented by a power of two. Hence, the following relationship applies:

$$\tan(\varphi) = 2^{-i} \Leftrightarrow \varphi_i = \arctan(2^{-i}), i \in \mathbb{Z} \tag{3.9}$$

As a result, the multiplication by tan (φ) can be performed by a shift operation. To circumvent the restriction to discrete elementary angles introduced by the above relation rotation by an arbitrary angle is generated by a sequence of elementary rotations which reaches sufficient accuracy. Additionally, a sign variable $d_i = \pm1$ is introduced to allow the rotations in both directions. Each rotation step i is then calculated by the following equation set.

$$x_{i+1} = \cos(\varphi_i)\ \left[x_i - y_i\ * d_i\ *\ 2^{(-i)}\right] \tag{3.10}$$

$$y_{i+1} = \cos(\varphi_i)\ \left[y_i + x_i\ * d_i\ *\ 2^{(-i)}\right] \tag{3.11}$$

In general, for any angle the following equations apply.

$$\cos(\varphi) = \cos(-\varphi) \tag{3.12}$$

$$\varphi = \arctan(\tan(\varphi)) \tag{3.13}$$

Since the cosine is an even function, the sign does not play a role in the scaling. Thus, the cosine term in equations (3.10) and (3.11) can be replaced by the following expression:

$$\cos(\varphi_i) = \cos(\arctan(2^{(-i)})) = k_i \tag{3.14}$$

By substituting equation (3.14) in equations (3.10) and (3.11) it follows:

$$x_{i+1} = k_i\ [x_i - y_i\ * d_i\ *\ 2^{(-i)}] \tag{3.15}$$

$$y_{i+1} = k_i\ [y_i + x_i\ * d_i\ *\ 2^{(-i)}] \tag{3.16}$$

From this follows the general relationship describing the rotation to an arbitrary angle by a sequence i of elementary angles φ_k

$$x_{i+1} = \prod_{k=0}^{i} k_k * [x_k - y_k * d_k * 2^{(-k)}] \tag{3.17}$$

$$y_{i+1} = \prod_{k=0}^{i} k_k * [y_k + x_k * d_k * 2^{(-k)}] \tag{3.18}$$

The k factors are then grouped and treated seperately from the remaining expressions for example for the first N values of the K_k product the following expression results.

$$K_k = \prod_{i=0}^{N} k_i \tag{3.19}$$

K_i is the so-called scaling factor. According to [5], to avoid multiplications during the iterations, K_i can be neglected in each rotation step and only applied for scaling of the final result. This results in the simplified calculation step rule:

$$x_{i+1} = x_i - y_i * d_i * 2^{(-i)} \tag{3.20}$$

$$y_{i+1} = y_i + x_i * d_i * 2^{(-i)} \tag{3.21}$$

After N iteration steps the following intermediate result is obtained as an approximate solution:

$$x_N = \frac{1}{K_{N-1}} [x_0 * \cos(\varphi) - y_0 * \sin(\varphi)] \tag{3.22}$$

$$y_N = \frac{1}{K_{N-1}} [y_0 * \cos(\varphi) + x_0 * \sin(\varphi)] \tag{3.23}$$

With this approach, at the end of the calculation the results x_N and y_N only have to be multiplied once by K_{N-1} instead of calculating the factor k_i in each step. The scaling factor K_N can be calculated in advance based on the number of iteration steps. The K_N factor can be calculated by the following expression:

$$K_N = \prod_{i=0}^{N} \cos(\arctan(2^{-i})) = \prod_{i=0}^{N} \frac{1}{\sqrt{1+\tan^2(\arctan(2^{-i}))}}$$
$$= \prod_{i=0}^{N} \left(\frac{1}{\sqrt{1+2^{-(2i)}}} \right) \quad (3.24)$$

This value can be stored as constants in the system. For large values of N the scaling factor converges to the value of 0.6073 as the number of iterations goes to infinity [8, p.134]:

$$k = \lim_{N\to\infty} k_n \approx 0{,}60725294 \quad (3.25)$$

The numerical values of the scaling factor are approaching this limit rapidly for a low number of iterations N. Table 3.1 shows the first 10 values of the scaling factor and the relative deviation of the limit value according to equation (3.25) from these values.

Table 3.1 Error due to the approximation of the scaling factor with the limit value

N	K_n	Deviation K
1	0,632456	4.1504%
2	0,613572	1.0407%
3	0,608834	0.2605%
4	0,607648	0.0652%
5	0,607352	0.0164%
6	0,607278	0.0042%
7	0,607259	0.0011%
8	0,607254	0.0003%
9	0,607253	0.0001%
10	0,607253	0.0001%

It can be seen that even with a small number of iterations, the deviation is negligible. Thus, a calculation of the scaling factor can usually be omitted and instead the limit value for $N \to \infty$ can be used as a good approximation.

A total rotation about angle φ can be decomposed into a sequence of N elementary rotations with the corresponding direction d_i: [4]

$$\varphi = \sum_{i=0}^{n} d_i * \varphi_i \quad \text{where } \tan(\varphi_i) = 2^{-i} \quad \text{and} \quad d_i \epsilon \{-1, +1\} \tag{3.26}$$

The decision vector d_i with the signs of the individual rotations is different for each total rotation angle φ. The previous determination of the vector would result in an additional calculation and storage effort. To avoid this, a further auxiliary variable z_{i+1} is introduced to the two equations (3.15) and (3.16) which allows to keep track of the total rotation angle over several steps with the following equation:

$$z_{i+1} = z_i - d_i * \arctan(2^{-i}) \tag{3.27}$$

and is initialized with the total angle

$$z_0 = \varphi \tag{3.28}$$

The determination of the elementary decisions for d_i, i.e. add or subtract for the next iteration step is made on the basis of the sign of z_i. It applies:

$$d_i = \begin{cases} +1 & \text{when } z_i < 0 \\ -1 & \text{when } z_i \geq 0 \end{cases} \tag{3.29}$$

By this procedure implicitly after each step, the total rotation is compared with the desired value φ and the next step towards the desired rotation is determined.

Based on these considerations, the CORDIC algorithm can be summarized as follows. For the calculation of the rotation of a vector (x_0, y_0) around the angle φ the following calculation rules are used:

$$\begin{cases} x_{i+1} = x_i - y_i * d_i * 2^{-i} \\ y_{i+1} = y_i + x_i * d_i * 2^{-i} \\ z_{i+1} = z_i - d_i * \arctan(2^{-i}) \end{cases} \quad (3.30)$$

$$\text{where } d_i = \begin{cases} +1 & \text{when } z_i < 0 \\ -1 & \text{when } z_i \geq 0 \end{cases}$$

$$K_N = \prod_{i=0}^{N} \cos(\arctan(2^{-i})) = \prod_{i=0}^{N} \frac{1}{\sqrt{1 + \tan^2(\arctan(2^{-i}))}} = \prod_{i=0}^{N} \left(\frac{1}{\sqrt{1 + 2^{-(2i)}}}\right)$$

3.1.2 Generalization

The CORDIC algorithm can also be represented in matrix form:

$$\begin{bmatrix} x_{i+1} \\ y_{i+1} \\ z_{i+1} \end{bmatrix} = \begin{bmatrix} 1 & -m * d_i * \varphi_i & 0 & 0 \\ d_i\varphi_i & 1 & 0 & 0 \\ 0 & 0 & 1 & -d_i\varphi_i \end{bmatrix} \cdot \begin{bmatrix} x_i \\ y_i \\ z_i \\ 1 \end{bmatrix} \quad (3.31)$$

This approach was generalized by Walther [6], who defined specific values for m in the case of circular, linear, and hyperbolic transformations. This results in the following calculation rules for the iterations in the three possible cases:

- Circular (m = 1)

$$\begin{aligned} x_{i+1} &= x_i - y_i * d_i * 2^{-i} \\ y_{i+1} &= y_i + x_i * d_i * 2^{-i} \\ z_{i+1} &= z_i - d_i * \arctan(2^{-i}) \end{aligned} \quad (3.32)$$

- Hyperbolic (m = -1)

$$\begin{aligned} x_{i+1} &= x_i + y_i * d_i * 2^{-i} \\ y_{i+1} &= y_i + x_i * d_i * 2^{-i} \\ z_{i+1} &= z_i - d_i * \operatorname{arctanh}(2^{-i}) \end{aligned} \quad (3.33)$$

- Linear (m = 0)

$$\begin{aligned} x_{i+1} &= x_0 \\ y_{i+1} &= y_i + x_0 . d_i . 2^{-i} \\ z_{i+1} &= z_i - d_i . 2^{-i} \end{aligned} \tag{3.34}$$

The initial values and the determination of the sign depend on the function which is to be calculated and on the mode in which the CORDIC is used. Furthermore, it is to be noted that for the hyperbolic variant in equation (3.33) with i = 0 the undefined expression arctanh(1) results. Therefore, for this mode, the iteration starts at i =1, while for the other two modes it starts at i = 0. The resulting scaling factors for the various values of m and rotation angles φ_i of the individual iteration steps are summarized in Table 3.2.

Table 3.2 Scaling factors for the generalized CORDIC [8, p. 18].

mode	Rotation angle	Scaling factors	Scaling factor limit
m	φ_i	k_n	$k = \lim_{n\to\infty} k_n$
1	arctan (2^{-i})	$\prod_{i=0}^{n} \cos(\arctan(2^{-i})) = \prod_{i=0}^{n} (\frac{1}{\sqrt{1+2^{-2i}}})$	$\approx 0{,}607253$
0	2^{-i}	$\prod_{i=0}^{n} 1 = 1$	1
-1	arctanh (2^{-i})	$\prod_{i=0}^{n} \cosh(\arctan h(2^{-i})) = \prod_{i=0}^{n} (\frac{1}{\sqrt{1-2^{-2i}}})$	$\approx 1{,}207497$

Depending on the mode, the CORDIC algorithm can be used to calculate a whole range of functions. Linear mode (m=0) allows multiplication and division, circular mode (m=1) can perform trigonometric functions and inverse trigonometric functions for given arguments. Moreover, functions such as cosh(φ) and sinh(φ) can be calcualted in hyperbolic mode (m = -1) of the CORDIC algorithm.

In the following section the procedure for the calculation of functions, with the rotation mode and vectoring mode is explained.

3.1.3 Application of the CORDIC algorithm for the calculation of functions

The CORDIC rotator is normally operated in one of two modes: the rotation mode and the vectoring mode. In the rotation mode, the input vector is rotated by a defined angle φ (given as an argument), while in the vectoring mode the input vector is rotated until it is as close as possible to the x-axis while the angle required to make that rotation is recorded. Based on these considerations, the CORDIC algorithm is summarized as follows.

3.1.3.1 Rotating mode in circular mode

For this calculation, a suitable start vector is selected and rotated by the angle φ. After a sufficiently large number of iteration steps, the function values for φ can be derived from the approximate result. A simple calculation results for the unit vector $(x_0, y_0) = (1, 0)$ as the starting vector (see Figure 3.2). The starting value for z corresponds to the total rotation angle, i.e. $z_0 = \varphi$.

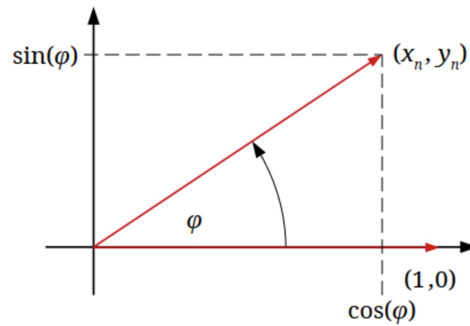


Figure 3.2 Concept of the rotating mode

From equation (3.22) and (3.23) it follows for the circular mode with this starting vector:

$$x_n = \frac{1}{k_{n-1}} . \cos(\varphi) \tag{3.35}$$

$$y_n = \frac{1}{k_{n-1}} . \sin(\varphi) \tag{3.36}$$

Thus, all elementary trigonometric functions of the rotation angle φ can be determined directly from the coordinates after n iterations:

$$\cos(\varphi) \approx k_{n-1} . x_n \tag{3.37}$$

$$\sin(\varphi) \approx k_{n-1} . y_n \tag{3.38}$$

$$\tan(\varphi) \approx \frac{y_n}{x_n} \tag{3.39}$$

$$\cot(\varphi) \approx \frac{x_n}{y_n} \tag{3.40}$$

3.1.3.2 Vectoring mode in circular mode

The vectoring mode is virtually an inversion of the rotation mode. In rotation mode, a start vector is rotated by a defined angle and the coordinates of the iteratively determined result vector are used to determine the trigonometric function for the angle. In contrast, with the vectoring mode, the start and target vectors are specified and the required angle of rotation is calculated. Therefore, the vectoring mode is suitable for calculating the different inverse trigonometric functions depending on the definition of the start and target vector. Since the arcsine function is used in this project, only this function is explained in the following.

3.1.3.2.1 Arcsine function

For the calculation of the arcsine, a start vector, which lies on the x-axis, is rotated until a given position is reached. The function works by seeking to minimize the y component of the residual vector at each rotation. The sign of the residual component is used to determine which direction to rotate next. When the angle accumulator is initialized with zero, it will contain the traversed angle at the end

of the iterations. The starting point for this consideration is that after the rotation by the angle φ the coordinates are given by:

$$\begin{aligned} x_n &= |r|.\cos(\varphi) \\ y_n &= |r|.\sin(\varphi) \end{aligned} \quad (3.41)$$

If a unit vector is rotated with |r| = 1, the y-coordinate after the final iteration corresponds to the sin of the rotation angle. It thus corresponds to the argument $\xi \in [-1\,;\,1]$, for which the arcsine is to be determined. Therefore, for the calculation, a unit vector is rotated from the x-axis until its y-coordinate corresponds to this argument (see Figure 3.3).

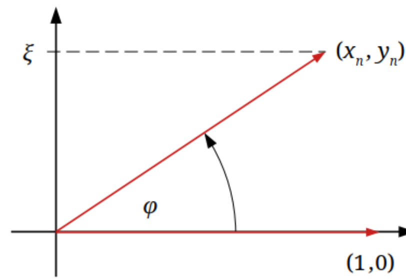


Figure 3.3 Concept of vectoring calculation for arcsine

This procedure is realized by continuing the rotation according to the deviation of y_i to this argument. Thus, the iteration rule is:

$$\begin{cases} x_{i+1} = x_i - y_i * d_i * 2^{-i} \\ y_{i+1} = y_i + x_i * d_i * 2^{-i} \\ z_{i+1} = z_i - d_i * \arctan(2^{-i}) \end{cases}$$

$$\text{where} \quad d_i = \begin{cases} -1 & \text{when} \quad \xi - y_i * K < 0 \\ 0 & \text{when} \quad \xi - y_i * K = 0 \\ +1 & \text{when} \quad \xi - y_i * K > 0 \end{cases} \quad (3.42)$$

$$x_0 = 1\,, \quad y_0 = z_0 = 0$$

At the end of the iteration we get:

$$y_n \approx \frac{\xi}{k_{n-1}} \qquad z_n \approx \varphi = \text{arcsine}\,(\xi) \tag{3.43}$$

The calculation can be simplified by using the initial value K_{n-1} for x_0. The following Table 3.3 illustrates the calculation of Arcsine (0,9) over 10 iteration steps. The individual values of the calculation can be taken from this Table. The exact value is Arcsine (0.9) = 64.158072°.

Table 3.3 Calculation steps for Arcsine (0.9) with the CORDIC algorithm

i	2^{-i}	Arctan(2^{-i})	k_i	x_i	y_i	d_i	z_i
0	1	0.785398	0.70717	0.607253	0	1	0°
1	0.5	0.463648	0.894427	0.607253	0.607253	1	45.0000°
2	0.25	0.244979	0.970143	0.303626	0.910879	-1	71.60135°
3	0.125	0.124355	0.992278	0.531346	0.834973	1	57.55799°
4	0.0625	0.062419	0.998053	0.426975	0.901391	-1	64.68662°
5	0.03125	0.031240	0.999512	0.483312	0.874705	1	61.10847°
6	0.015625	0.015624	0.999878	0.455977	0.889809	1	62.89929°
7	0.0078125	0.007812	0.999969	0.442074	0.896933	1	63.79492°
8	0.00390625	0.003906	0.999992	0.435067	0.900387	-1	64.24276°
9	0.001953125	0.001953	0.999998	0.438584	0.898688	1	64.01883°
10	0.0009765625	0.000977	1.000000	0.436828	0.898688	1	64.13079°

The calculation of the Arcsine as state above gives the correct result for arguments $-1 < \xi < 1$, but the error increases strongly for arguments larger than 0.98. According to [5], the error is due to the scaling of the vector to be rotated, since it becomes shorter than the reference vector (ξ) and thereby wrong decisions are made when determining the rotation direction (see Table 3.4).

Table 3.4 Calculation steps for Arcsine (0.99) with the CORDIC algorithm

i	2^{-i}	Arctan(2^{-i})	k_i	x_i	y_i	d_i	z_i
0	1	0.785398	0.70717	0.607253	0	1	0°
1	0.5	0.463648	0.894427	0.607253	0.607253	1	45.0000°
2	0.25	0.244979	0.970143	0.303626	0.910879	1	71.60135°
3	0.125	0.124355	0.992278	0.075907	0.986786	1	85.64471°
4	0.0625	0.062419	0.998053	-0.04744	0.996274	-1	92.77334°
5	0.03125	0.031240	0.999512	0.014826	0.999239	-1	89.19519°
6	0.015625	0.015624	0.999878	0.046052	0.998776	-1	87.40438°
7	0.0078125	0.007812	0.999969	0.061658	0.998057	-1	86.50875°
8	0.00390625	0.003906	0.999992	0.069455	0.997575	-1	86.06091°
9	0.001953125	0.001953	0.999998	0.073352	0.997304	-1	85.83698°
10	0.0009765625	0.000977	1.000000	0.0753	0.997303	-1	85.72502°

The exact value is Arcsine (0.99) = 81.89039°. The approximation of the CORDIC algorithm gives rise to a deviation 4-degree deviation with calculation in the CORDIC algorithm.

To make the calculations of the Arcsine accurate for all $-1 \leq \xi \leq 1$, the Double iterative CORDIC algorithm [4] can be used. Unlike the conventional CORDIC, the Double Rotation CORDIC algorithm scales the input argument and it gives the correct result over the complete range of values of the Arcsine. In the following section the procedure for the calculation of the Arcsine functions, with the Double Rotation CORDIC algorithm is explained.

3.1.3.2.2 Double Rotation CORDIC algorithm

The double Rotation algorithm for computing the Arcsine function is given by the following set of equations:

$$
\begin{cases}
\begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \begin{bmatrix} 1 & -d_i \,.\, 2^{-i} \\ d_i \,.\, 2^{-i} & 1 \end{bmatrix}^2 \begin{bmatrix} x_i \\ y_i \end{bmatrix} \\
z_{i+1} = z_i + 2.\, d_i \,.\, \arctan\left(2^{-i}\right) \\
t_{i+1} = t_i + t_i \,.\, 2^{-2i}
\end{cases} \tag{3.44}
$$

$$x_0 = 1\,, \quad y_0 = z_0 = 0,\ \ t_0 = \mathrm{abs}(t)$$

In this algorithm, d_i is chosen such that $d_i = \mathrm{sign}(x_i)$ when $(t_i - y_i) \geq 0$ else $d_i = -\mathrm{sign}(x_i)$.

By doing this modification, the vector (x_i, y_i) will always rotate in the first and the second quadrant and y_i and z_i will not get negative values. According to Formula (3.45), the initial vector (x_0, y_0) should be a unit vector on the x-axis. As is shown in Figure 3.4, in the first iteration, it is sure that $d_0 = 1$ and (x_0, y_0) will rotate by an angle of $2^*\tan^{-1}(2^{-0}) = 90°$ counterclockwise, so (x_1,y_1) is a vector on the y-axis. While in the second iteration, $d_1 = -1$ and (x_1, y_1) will rotate by an angle of $2^*\ \tan^{-1}(2^{-1})$ clockwise, since the destination angle is in the range of [-90°, 90°]. According to the above analysis, the first two 90° rotations always happen in the same way and as a result (x_2, y_2) will be always the same vector independent of the input argument.

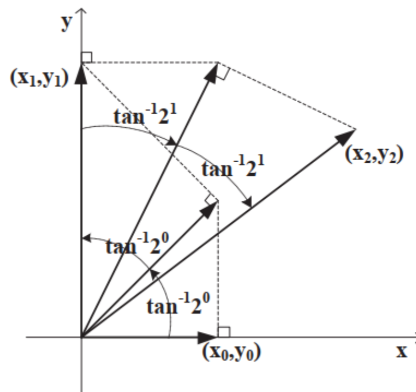


Figure 3.4 The rotations of the first two iterations

Unlike the conventional CORDIC, the Double CORDIC does not scale the input vector but the input argument. This algorithm delivers the correct result over the complete range of values of the Arcsine.

The main advantage of the double rotation approach is that due to the two subsequent rotations the square root of the scaling factor vanishes and becomes $1/\cos^2(\tan^{-1} 2^{-i}) = 1/(1+2^{-2i})$. A multiplication by the denominator of this term reduces to an addition and a shift. Another advantage is that the convergence domain of the algorithm becomes larger. It gives a correct result for $\varphi \in [-2\sum_{n=0}^{\infty} \tan^{-1}2^{-n}, +2\sum_{n=0}^{\infty} \tan^{-1}2^{-n}] \approx [-3.48657, +3.48657]$; therefore, we can compute $\sin^{-1} t$ for any $t \in [-1, 1]$.

3.1.3.3 Function calculations in linear mode

The CORDIC algorithm in linear mode is able to calculate multiplications in rotation mode and divisions in vectoring mode. One benefit of the linear mode is the simple scaling factor $k_n = 1$. As a result, no scaling corrections are required and no errors arise due to an insufficient scaling factor approximation.

The rotation calculation in linear mode, starts from the start vector $(x_n, 0)$ on the x-axis and results into the following vector at the end of the iteration [8, p.13]:

$$\begin{cases} x_n = x_0 \\ y_n = y_0 + x_0 * z_0 = x_0 * z_0 \\ z_n = z_0 - \sum_{i=0}^{n} d_i * 2^{-i} \approx 0 \end{cases} \tag{3.45}$$

Here z_0 is not freely selectable, since the sum of the partial displacements is limited. It applies:

$$\lim_{n\to\infty} \sum_{i=0}^{n} 2^{-i} = 2 \tag{3.46}$$

Even in the case that all partial rotations have the same direction, i.e. all d_i have the same value, the total displacement cannot be greater than two. The approximation of z_n to zero according to equation (3.45) is therefore only possible if the condition

$$|z_0| < \sum_{i=0}^{\infty} 2^{-i} = 2 \tag{3.47}$$

is satisfied [8, p. 13].

The vectoring calculation applys a rotation of the start vector (x_0, y_0) towards the direction (x_n, 0). After n steps the following result is obtained in linear mode:

$$x_n = x_0 \tag{3.48}$$

$$z_n = \sum_{i=0}^{n} d_i \cdot 2^{-i} \approx \frac{y_0}{x_0} \tag{3.49}$$

In this case, the application of the CORDIC algorithm results into the calculation of the quotient of the components of the inital vector (x_0, y_0). The constraint according to equation (3.46) leads to the following condition for a correct calculation result:

$$\left|\frac{y_0}{x_0}\right| < 2 \tag{3.50}$$

The general iteration rule for vectoring calculation in linear mode is:

$$\begin{cases} x_{i+1} = x_i = x_0 \\ y_{i+1} = y_i + x_i \cdot d_i \cdot 2^{-i} \\ z_{i+1} = z_i - d_i \cdot 2^{-i} \end{cases}$$

$$\text{where } d_i = \begin{cases} +1 & \text{when } y_i < 0 \\ 0 & \text{when } y_i = 0 \\ -1 & \text{when } y_i > 0 \end{cases} \tag{3.51}$$

$$x_0 > 0\,, \quad z_0 = 0$$

The advantages of the CORDIC divider are low latency and less hardware complexity compared to the other fast dividers. Accuracy is also a concern where high precision is needed.

4 Light Source and Polarizer

In the following chapter, an overview of components and devices of tools used in the measurement setup of this project is given.

4.1 Hyperchromator

The Hyperchromator is a high throughput monochromator designed for the EnergetiqEQ-99X light source. Due to its extremely high radiance, the EQ-99X is especially well suited for generating monochromatic light in the wavelength range 220 nm–2200 nm (UV/VIS/NIR). Bandwidths of 1 nm to 10 nm are possible. The light is collected directly from the plasma of the lamp with an aperture of up to f/1.5 without using an additional entrance slit. This makes this tunable light source very efficient. The output side has been designed with a very flexible opto-mechanical interface. This allows for a multitude of illumination or light coupling options using standard catalog components, rendering the integration of the Hyperchromator into your setup hassle free and straight-forward. Possible configurations include fiber coupling, collimated or free-beam output. The wavelength is selected via USB/RS-232 interface using low-level command sequences or high-level implementation from a PC and an intuitive GUI. The low-level control commands are based on the protocol of the motor controller. Thereby binary command sequences are transmitted, which are assembled according to a certain scheme. With some lines of code, it is possible to generate the necessary sequences for variable commands, e.g., to approach different wavelengths, also dynamically. But wavelength calibration has to be handled, i.e., by reading in a calibration file, interpolating wavelengths into motor positions and also by setting the order filters. In the high-level implementation, only the wavelength is passed while the calibration is processed automatically [11]. In Figure 4.1, the Mountain Photonics Hyperchromator is depicted with the Energetiq EQ-99X light source mounted on it.

In this work, the monochromatic light source is connected to an optical fiber and a collimator. The light is decoupled from the light source via an optical fiber. At the end of the fiber there is a collimator which is used to produce a parallel beam of light allowing for better focus, accuracy, and efficiency. The control of the light source is integrated into a self-written software framework based on Qt which allows for automated measurements to simplify recurring measurements at e.g. different wavelengths that will be explained in the 10.1.5.7 on GUI later.

Figure 4.1 The Mountain Photonics Hyperchromator with the mounted Energetiq EQ-99X light source [11]

4.2 Polarizer

To polarize the light emitted from the light source during automated measurements with controllable polarization angle, a Double Glan-Taylor Calcite Polarizers and a motorized precision rotation stage bundled with a DC Servo Motor Driver (KPRM1E/M) are used which allows the rotation angle to be defined via the software interface. The units of measurement used for the specifications or dimensions of the rotation stage such as length, diameter, or any other relevant parameters are provided in the metric system (e.g., millimeters, centimeters). The DGL10 polarizer can be mounted to a KPRM1E/M rotation

mount by using a SM1L20 lens tube. These devices are explained in more detail in the following.

4.2.1 Double Glan-Taylor Calcite Polarizers (DGL10)

The Double Glan Taylor Polarizers offer a solution for high-quality polarization with extinction ratios greater than 100 000:1 for laser beams in the 0.35-2.3 µm wavelength range. These triple prism, air-spaced polarizers are made from the highest optical grade of calcite. The triple prism design of the polarizer gives it a larger field of view (FOV) than standard Glan-Taylor polarizers. Unpolarized light enters the polarizer and is split at the two internal interfaces between crystals. The ordinary rays are reflected at each interface, causing them to be scattered and partially absorbed by the polarizer housing. The extraordinary rays pass straight through the polarizer, providing a polarized output. Unlike standard Glan-Taylor designs, this double Glan-Taylor prism also has a symmetric FOV. This polarizer is only 33 mm long and features a large clear aperture that measures 9 mm x 9 mm. [12]

4.2.2 Motorized Rotation Stage and Mount

The KPRM1E/M is a small, compact, DC servo motorized 360° rotation mount and stage that accepts optics with a diameter of 1 inch (Ø1") optics and SM1-threaded components. The device serves as a companion for achieving smooth, continuous motion which can be automated through a software interface. It establishes a connection to a computer via USB, allowing control through the transmission of serial commands in software over the USB connection. The user can measure the angular displacement by using the Vernier dial in conjunction with the graduation marks that are marked on the rotating plate in 1° increments. This rotation stage/mount is also equipped with a home limit switch to facilitate automated rotation to the precise 0° position, allowing absolute angular positioning thereafter. The limit switch is designed to allow continuous rotation of the stage over multiple 360° cycles with rotational velocity of 25 degree per

second. The central aperture of the KPRM1E/M rotation mount has a standard SM1 internal thread, for compatibility with a range of optics [13]. Figure 4.2 shows the Mounted DGL10 Polarizer in a SM1L20 Lens Tube and PRM1 Rotation Mount. The polarized light which passes the crystal should fall on a sensor chip which in this project is placed on a PCB.

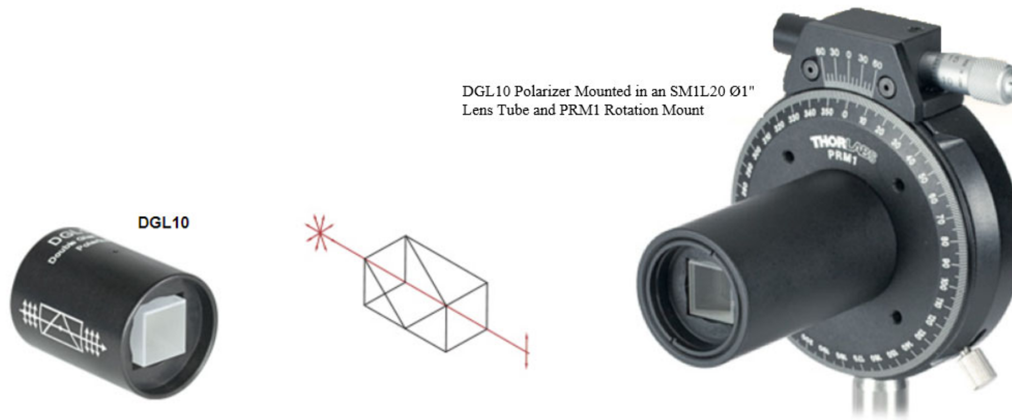


Figure 4.2 DGL10 Polarizer Mounted in an SM1L20 Lens Tube and PRM1 Rotation Mount [12]

4.2.3 K-Cube™ Brushed DC Servo Motor controller (KDC101)

The K-Cube™ Brushed DC Servo Motor Controller is a specific type of motor controller designed to operate and control the movement of brushed DC servo motors offered by Thorlabs, a company specializing in photonics and optomechanical systems. It can be used to drive a motorized rotation stage/ mount (KPRM1E/M).

The controller is designed to provide precise control over the position, velocity, and acceleration of a brushed DC servo motor. It accepts various input signals, such as analog voltage or digital commands, to regulate the motor's operation. It offers features such as closed-loop feedback, PID control algorithms, and built-in motion profiles for smooth and accurate manual and automatic motor control. Manually adjusting motor positions is done using the top panel controls (velocity control wheel). With its compact size and user-friendly interface, the K-Cube

Brushed DC Servo Motor Controller offers convenient integration into experimental setups or industrial applications that require precise motor control. It offers a fully featured motion control capability including velocity profile settings, limit switch handling and “on the fly” changes in motor speed and direction. It can be controlled remotely through computer interfaces, enabling automation and compatibility with software-based control systems.
Each unit contains a front-located power switch that, when turned off, saves all user-adjustable settings.

The KDC101 provides a USB or an RS-232 interface to communicate with the host PC. The electrical interface within the Thorlabs controllers uses a Future Technology Devices International (FTDI) FT232BM USB peripheral chip to communicate with the host PC. This USB interfacing chip provides a serial port interface to the embedded system (i.e., Thorlabs controller) and USB interface to the host control PC.

USB connectivity provides easy 'Plug-and-Play' PC-controlled operation with a legacy APT (Advanced Positioning Technology) software package which allows the user to quickly set up complex move sequences with advanced controls made possible via the ActiveX programming environment. For example, all relevant operating parameters are set automatically by the software for Thorlabs stage and actuator products. Furthermore, this programming library is compatible with many development tools such as LabView, Visual Basic, Visual C++, C++ Builder, LabWindows/CVI, Matlab and Delphi.
The overall communications protocol is independent of the transport layer (for example, serial communications could also be used to carry commands from the host to the controller. Before any PC USB communication can be established with a Thorlabs controller, the client program is required to set up the necessary FTDI chip serial port settings used to communicate to the Thorlabs controller embedded system. The low-level communications protocol and commands used between the

host PC and controller units within the Thorlabs motion control helps to write applications to interface to the Thorlabs range of controllers without the constraints of using a particular operating system or hardware platform. Communications parameters are fixed at:

• 115200 bits/sec

• 8 data bits, 1 stop bit

• No parity

• RTS/CTS Handshake

Figure 4.3 K-Cube Brushed DC Servo Motor controller (KDC101)

5 Sensor Readout Electronics

As a preliminary work, design and development of boards with readout circuits for the Poldi sensor manufactured by AdvICo Microelectronics was carried out. The Poldi sensor provides a voltage signal which is proportional to the photocurrent of the integrated diodes. To digitalize these analog output voltages, analog-to-digital converters are utilized. Due to the diode reverse currents, a constant offset current can be superimposed on the signal-dependent photocurrent. In addition, the feedback resistors of the transimpedance amplifiers integrated in the sensor may vary, resulting in different output voltages for the same photocurrent values. To correct these effects, Multiplying Digital to Analog Converters (MDAC) were employed to influence various parameters of the circuit. The DAC (AD5544) is based on a resistor network and has the task of converting the input voltage into an output current. The conversion factor is determined by the DAC setting. Another channel of the DAC is used to generate a negative output current of variable magnitude, which can be subtracted from the signal-dependent current.

The operational amplifier (AD8656) is connected as a transimpedance amplifier, whereby for the feedback, a resistor integrated in the Digital to Analog converter is used. The transimpedance amplifier converts the output current of the DAC back into a voltage. For the further processing of the measurement signal in the FPGA. The voltages are digitized with an Analog to Digital converter (AD7980). The digitized data is transferred to the FPGA via a serial interface and used for performing angle calculations.

Since the FPGA is operated with a lower supply voltage than the rest of the electronics, the control signals are raised to 5V via a level shifter. A description of electronic parts is provided below.

5.1 Power Supply and Reference Voltage Board

This board is responsible for providing the voltage supply and generating reference voltages which are required by the Poldi readout circuits. These circuit parts were redesigned with Altium Designer and produced as a stand-alone PCB.

Figure 5.1 Circuit diagram for the power supply and reference signals [14]

5.1.1 Generated voltages and outputs

The following voltages and output signals are generated on the board designed for the power supply [14]:

- "AGND" → Ground (terminal block 2 - pin 1)
- "+5V" → The first voltage regulator converts the input voltage into 5V. (Terminal block 2 - Pin 2)

- “+2V5OUT” → The IC2 converts the 5V voltage to 2.5V.(Terminal block 2 - Pin 3)
- “3V3ANA” → The IC3 converts the 5V to 3.3V. (Terminal block 2 - Pin4)
- “RELVTIA” → is 2.5V higher than “VREFTIA”. (Terminal block 1 - pin 1)
- “1V_DGND” → Ground (terminal block 1 -Pin2).
- “VREFTIA” → Reference voltage of the POLDI sensor, which is 1V and used as the Ground voltage of many circuits of the readout chain. (Terminal block 1 - pin 3)
- “RELNEGREF” → Reference voltage which is 0.375V above the global ground voltage, but 0.625V below VREFTIA (-0,625V). This voltage is generated by the AD780 component. (Terminal Block 1 - Pin 4)
- “1V_LOCAL” → Reference voltage of 1V generated by the AD510. (Terminal block 1 - pin 5)

5.1.2 Components of the Board

The components of the board are as following:

5.1.2.1 Shunt Voltage Reference

A Shunt Voltage Reference has a similar function to a Zener diode but generates a much more precise output voltage. The Shunt Voltage Reference is needed for the generation of the reference voltage of 1V, which is called “1V_LOCAL” in the original circuit diagram. Specifically, this project uses the ADR510 device, which provides 1.0V. To ensure a consistent flow of current through the shunt, a series resistor RBIAS is employed. This resistor allows a controlled current ranging from 100µA to 10mA. To prevent loading of the shunt output, an operational amplifier with high input resistance is connected to the reference

signal "1V_LOCAL". This configuration ensures minimal input current into the operational amplifier, maintaining the integrity of the shunt output voltage. ICs linked to the reference potential "1V_LOCAL" operate with a supply current under 1mA. By selecting a 1kOhm series resistor (RBIAS), a total load current (IL) of 4mA is achieved, falling within the specified range. [14]

5.1.2.2 Voltage Regulator

Voltage regulators are used to stabilize electrical DC voltages and to compensate for fluctuations in the input voltage and the load current. Three voltage regulators are employed to generate the output voltages of 2.5V, 3.3V and 5V.

Each regulator has a total of five pins ("IN", "GND", "OUT", "BYP", "EN"). The pin "IN" corresponds to the input voltage. A 1uF capacitor is connected to this pin for all three regulators, with its second electrode connected to GND. This capacitor serves as an energy reserve when the load current increases, the voltage regulator needs time to react due to its limited bandwidth. This delay causes voltage drops or rises, depending on load changes. To address this, a capacitor of 2.2µF is connected to the controller output pin "OUT", helping to reduce voltage spikes by charging and discharging during load fluctuations.

For the last controller with the instance name IC3 no bypass was used, which allows the device to regulate the 3.3V output faster. "BYP" pin is used to reduce the output noise, an external 470pF capacitor is connected to the reference bypass.

The "EN" pin is used to activate or deactivate the controller. If the input voltage is above 2V, the controller is activated. Below 2V it is automatically switched off internally. The "EN" is connected to "IN", therefore the controller is always activated as soon as the input voltage is above 2V [16]. Figure 5.1 shows the circuit diagram of voltage regulators.

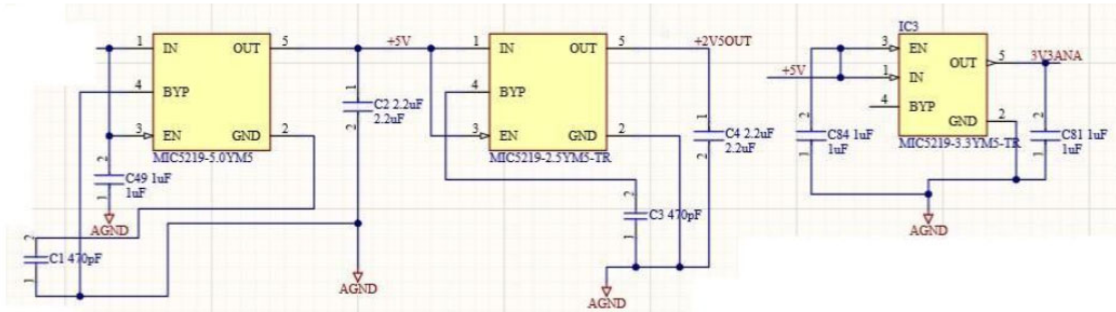


Figure 5.2 Extract of the circuit diagram with the wiring of the LDO voltage regulator ICs [14]

5.1.2.3 Operational Amplifier TSV914IDR

The TSV914AIDR component is a quad-channel operational amplifier (opamp), specifically designed for general-purpose applications. It only takes up very small input currents and is therefore very well suited for buffering analog voltages.

Since the readout system of the POLDI sensor requires three operational amplifiers (OPs) and the TSV914AIDR component comprises four operational amplifiers one additional operational amplifier remains unused in this setup.; one is dedicated to the shunt, while two are employed for generating the "RELNEGREF" voltage. [17]

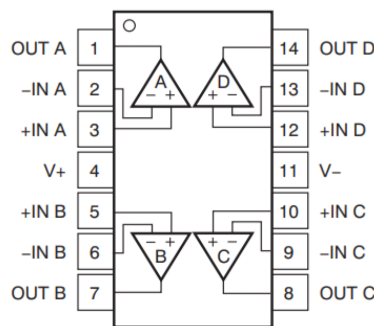


Figure 5.3 Symbol of the TSV914AIDR operational amplifier device [17]

The operational amplifier "A" in figure 5.3 is wired as an inverting amplifier. The output RELNEGREF (-0.625V) is fed back to the inverting input with resistor R1 of 1kOhm. The voltage RELVTIA, which is supplied by the AD780 voltage

regulator, is applied to resistor R62 of 4.02kOhm, which relates to its other terminal to the inverting input of the operational amplifier. The non-inverting input +IN is connected to the voltage VREFTIA.

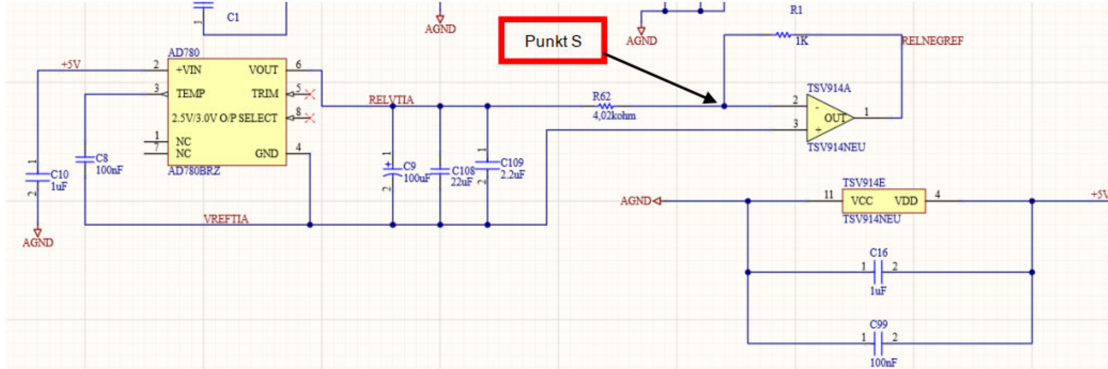


Figure 5.4 Circuit diagram for generating the voltage RELNEGREF [14]

Due to this inverting circuit, the output voltage becomes negative compared to VREFTIA when the input voltage is positive. Due to the negative feedback, the OP regulates the voltage at the inverting input to the same voltage as at the non-inverting input VREFTIA and the voltage difference of these two inputs is zero. Therefore, “node S” in figure 5.3 is always close to VREFTIA, i.e. 1V. [9] The calculation for the voltage gain A results in:

$$A = -\frac{R1}{R62} = -\frac{1kOhm}{4{,}02kOhm} \approx -0{,}25 \qquad (4.1)$$

$$RELVTIA = \frac{RELNEGREF}{A} = \frac{1kOhm}{4{,}02kOhm} \approx 2{,}5 \qquad (4.2)$$

The operational amplifier “B” was combined with the ADR510ARTZ to ensure load-independent and precise generation of the VREFTIA and 1V_DGND voltages.

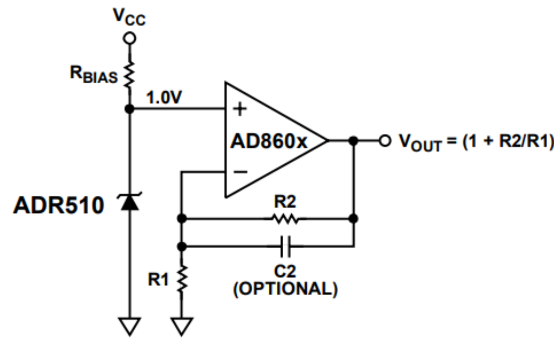


Figure 5.5 Circuit for generating loadable reference voltages [18].

The voltage at the operational amplifier's output is influenced by the external circuitry, primarily the choice of resistors. In instances where no resistors are employed, an amplification factor of 1 is configured. This configuration effectively transforms the operational amplifier circuit into a voltage buffer.

The voltage between the ADR510, which acts like a Zener diode, and the resistor is 1 Volt. If the current ratios were to change, this would lead to a reduction of the reference voltage below 1V. By using the operational amplifier, which acts as an impedance converter, the output voltage of 1V remains stable as long as a connected load does not require more current than the operational amplifier can supply. Therefore, the current is supplied by the operational amplifier and not by the circuit in front of it. The operational amplifier "C" is wired as operational amplifier "B" and has the same function for the output voltage "1V_LOCAL". As operational amplifier "C", 1V_DGND is provided at the output.

The operational amplifier "D" has no function, and the inputs are only connected to ground.

The positive 5V supply voltage is connected to VDD, the ground potential to VCC. In addition, two capacitors are connected in parallel between VCC and GND to filter unwanted interference at high frequencies. Each line has a resistor and an inductance. When a rapid load change occurs, this rapid load change is also passed through the resistor and inductor, causing the voltage at VDD to drop.

These fluctuations in the supply voltage are called ripples. These ripples are absorbed by the blocking capacitance, in which the current change is partially fed from the capacitance and not via the supply line, so that the voltage drops at the resistors and the inductance are smaller and the ripples are smoothed out. Figure 5.6 illustrates Circuit for generating the voltages VREFTIA and 1V_DGND.

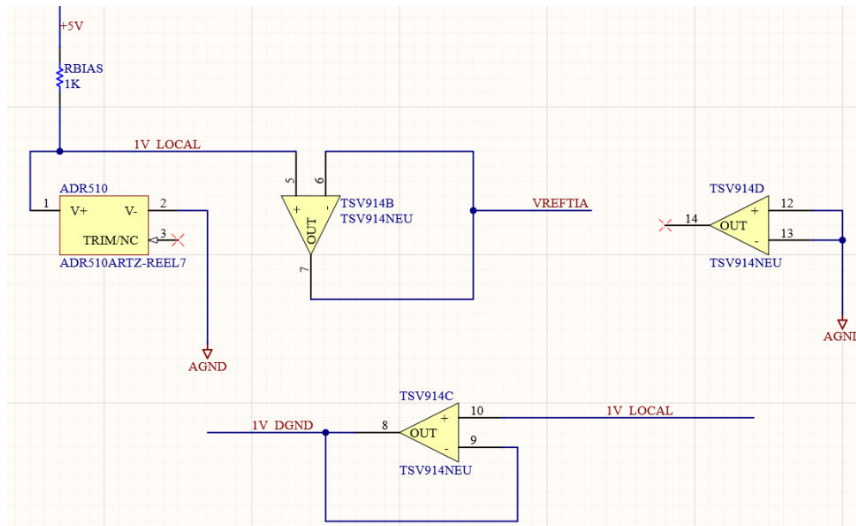


Figure 5.6 Circuit for generating the voltages VREFTIA and 1V_DGND [14]

5.1.2.4 Voltage Reference AD780BRZ

The AD780 is a bandgap voltage reference circuit that generates an output voltage between 2.5V and 3V. in this project the generated bandgap reference voltage is 2.5V and requires an input or supply voltage between 4V and 36V. The device is a good choice for improving the performance of high-resolution ADCs and DACs, as well as for all general-purpose precision reference applications [15]. The component is used to generate the “RELNEGREF” and “RELVTIA” voltages.

The voltage “RELVTIA” is 2.5V higher than the voltage “VREFTIA” due to the connection of the regulator's GND to “VREFTIA”. The operational amplifier is configured as an inverting amplifier with a gain of -0.625.

The voltage of -0.625V is referenced to the local ground voltage “VREFTIA” which is at 1V. The absolute value of the voltage “RELNEGREF” referred to the global ground, is calculated as 1V - 0.625V, resulting in 0.375V. Utilizing a 4.02K resistor introduces a slight deviation for “RELNEGREF” e.g, -0.622V.

5.2 Level shifter

A level shifter is an electronic circuit used to translate signals between different voltage levels, ensuring seamless communication between various components within a system. It finds widespread use in electronics, facilitating compatibility and accurate signal transmission between integrated circuits or components operating at varying voltage levels, like converting signals between a 3.3V and 5V IC.

The AD5544 is used for digital to analog conversion, serving as a multiplying DAC (MDAC). The chip possesses both analog and digital ground connections. Maintaining proximity between these ground voltages is crucial to prevent undesirable current flow. For instance, if digital ground is 0V and analog ground is 1V, undesired current circulates. Balancing the ground involves raising the digital ground also to 1V. When driven by an FPGA, the low and high levels of 0V and 3.3V correspond to -1V and 2.3V respectively, relative to the AD5544's digital ground at 1V. This voltage difference can cause current flow which can damage the chip.

The AD5544 operates at 5V and requires a low level of 0V and a high level of 5V at its digital input pins, However, a high level of 2.4V is also sufficient. The FPGA levels are between 0V and 3.3V. If only the ground is set to 1V, the high output level of the FPGA is only 2.3V and cannot be recognized as high by AD5544. Therefore, A multiplexer (NLASB3157) is employed to adjust the voltage levels conveniently. It has a digital select input that can be used to determine the output, where 0 sets the output signal to 1V while sets the output

signal to 5V with respect to global ground. The select input of the multiplexer is connected directly to the FPGA.

The datasheet specifies that the select input's high level for the NLASB3157 must be a minimum of 0.7VCC, while the low level should not exceed 0.3VCC. With VCC set at 5V, this implies a high level of 3.5V, surpassing the FPGA's capability. To address this, a VCC supply of 4V is chosen for the NLASB3157, ensuring the FPGA's 2.8V high level is sufficient to trigger the multiplexer. However, this decision limits the maximum switched voltage to 4V with an FPGA high signal while the FPGA's low 0V level is adjusted to 1V. This configuration results in a voltage swing of 3V which meets the requirement of the AD5544 component of 2.4V voltage swing.

5.2.1 Level Shifter Board Structure

The design of the level shifter circuit board involves arranging four ICs with their corresponding terminal blocks on each side. On the top side, there are three terminal blocks with eight pins and one terminal block with four pins, all of which are connected to a total of 13 ICs. Among these, twelve are multiplexers (NLASB3157) responsible for switching analog voltages. Multiplexers enable the sequential switching of multiple signals to a single input. As a result, a multiplexer is allocated to each input on the terminal block. One specific component, IC5, is the TSV911AIDCKR operational amplifier, utilized here to generate a steady 4V voltage output.

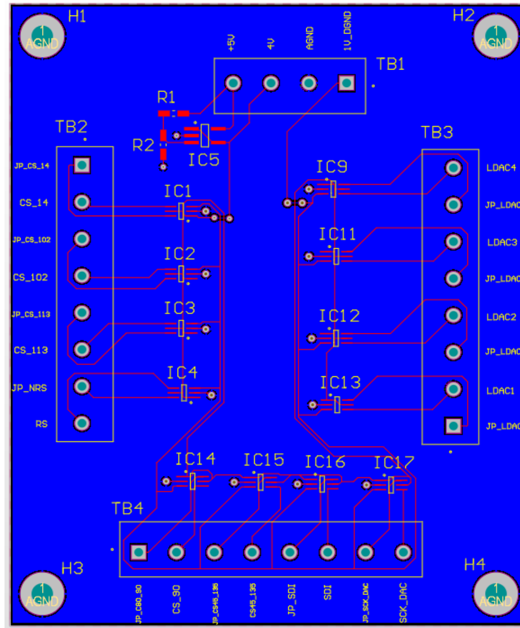


Figure 5.7 Layout of the level shifter board

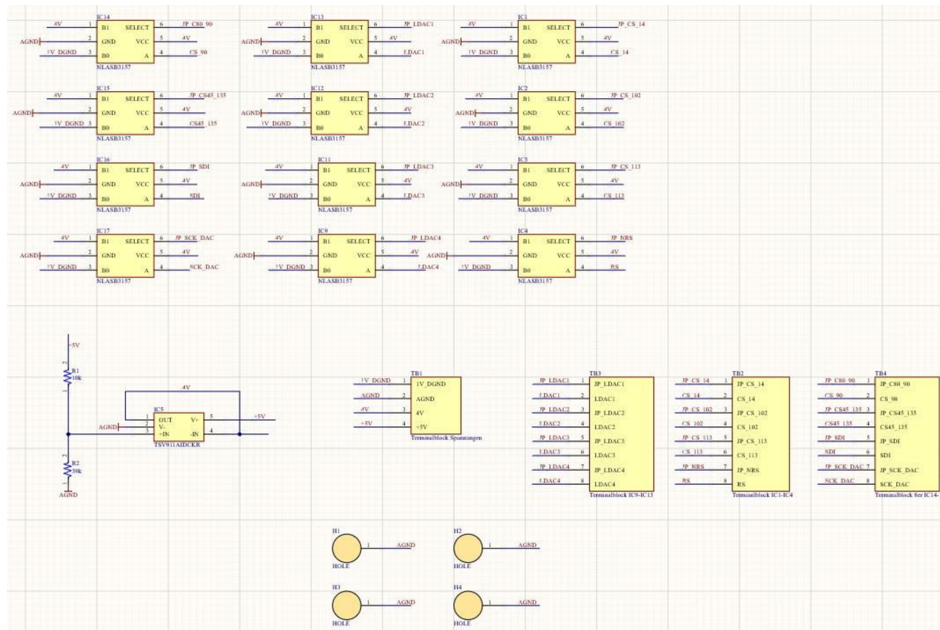


Figure 5.8 Level shifter circuit diagram in Altium Designer

5.2.2 Components of the Level Shifter

5.2.2.1 Operational Amplifier (TSV911AIDCKR)

This component serves as a rail-to-rail operational amplifier, functioning within a 5V supply voltage. To achieve a four-volt output, a voltage divider divides the initial five volts. The operational amplifier operates as a buffer with a gain of one, ensuring the voltage of the divider remains stable even under load. To prevent any undue load on the voltage divider, a high impedance input operational amplifier is employed.

The TSV911 operational amplifier is chosen with an input current as low as 1-10pA [19].

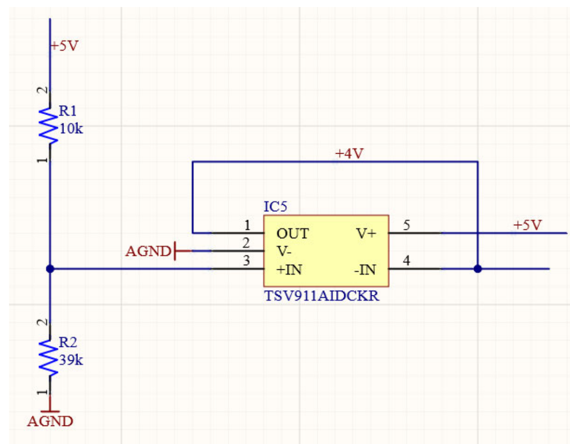


Figure 5.9 Circuit for generating the level shifter levels [14]

5.2.2.2 Multiplexer NLASB3157DFT2G

The NLASB3157 is an analog multiplexer/demultiplexer integrated circuit (IC). It is designed to switch analog signals between different inputs and outputs. It allows selection and routing of analog signals between inputs and outputs, with bidirectional capabilities. Figure 5.10 illustrates switching symbol of the multiplexer according to the data sheet [20].

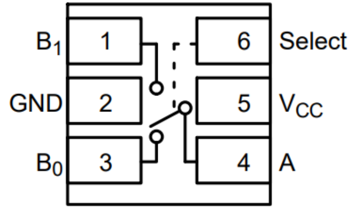


Figure 5.10 Switching symbol of the multiplexer according to data sheet [20].

As an example, we consider instance IC17 from the circuit diagram (see Figure 5.11). The FPGA output “JP_SCK_DAC” is connected to the input “SELECT”. The “SELECT” signal determines which input channel is switched through. When the input signal SELECT is 0, then B0 is connected to A; when it's 1, B1 is connected to A. This connection also works bidirectionally, but here A is used as the output. As described above, when the input signal is 0V, there is a voltage of 1V at output A, and at 1V input, there's a 4V output. This principle applies to all twelve “NLASB3157DFT2G” components.

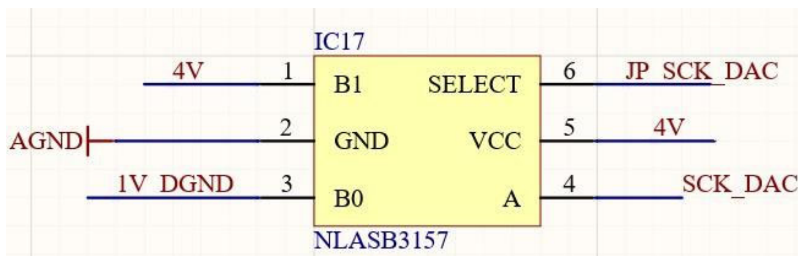


Figure 5.11 Multiplexer's component in Altium Designer

5.3 Analog to Digital Converter Circuit Board (ADC)

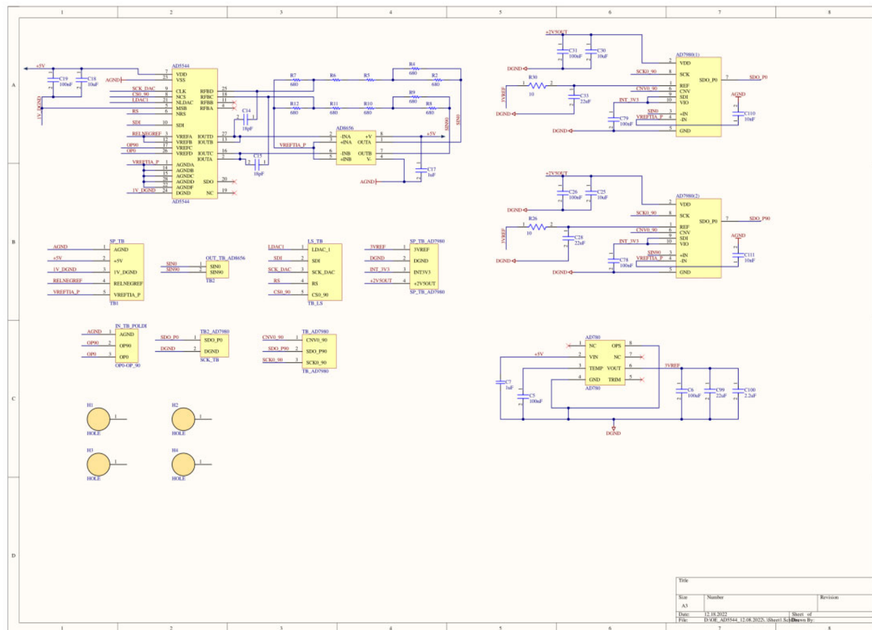


Figure 5.12 Schematic of the Analog to Digital Converter board

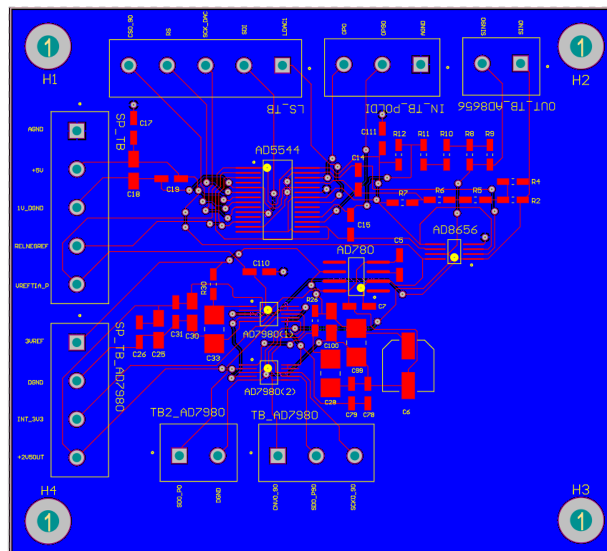
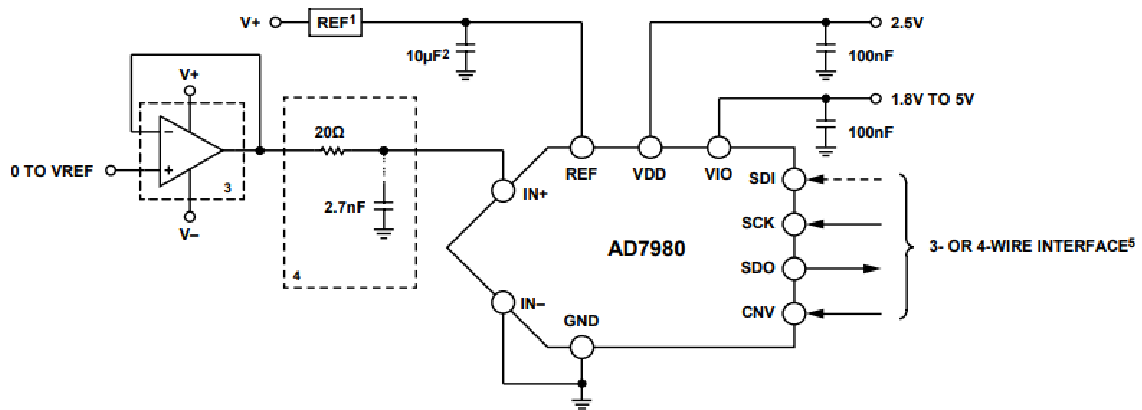


Figure 5.13 Layout of the ADC board

5.3.1 Components of the ADC board

5.3.1.1 AD7980

The AD7980 is a 16-bit Analog to Digital converter (ADC) which is supplied with 2.5V from a positive operating voltage VDD. It contains a 16-bit sampling ADC and a serial interface port. Furthermore, it features an additional digital input/output interface named “VIO” which requires a voltage within the range of 1.8V to 5V; in this case, 3.3V is applied. The CNV input serves multiple functions. Primarily, it converts and selects the interface mode, whether chain or CS mode. In CS mode, the SDO pin is triggered when the CNV signal is low. In contrast, when operating in chain mode, it samples an analog input, labled as IN+, on the rising edge of the CNV signal. This sampling process occurs within a voltage range of 0 V to REF relative to DGND, with the corresponding complementary input denoted as IN-. The reference voltage, REF, is applied externally and can be a voltage between 2.4V and 5.1V relative to DGND. In this project, a voltage of 3V is applied to REF. The SDI pin handles serial data input, similarly, influencing the interface configuration alongside CNV. If SDI is low during a rising CNV edge, chain mode activates. In this state, SDI functions as data input, channeling conversion results from multiple ADCs to one SDO line. Conversely, if SDI is high at the rising CNV edge, the CS mode is engaged, activating serial output signals through SDI or CNV. For the reference input voltage, a voltage between 2.4V and 5.1V relative to DGND is supplied to the REF input pin. The conversion result is output serially via the SDO output, synchronized with the data clock input SCK pin [21]. To protect the circuit from interference, an RC element is connected in series with the REF pin. This consists of a resistor and a capacitor [14]. Figure 5.14 shows the circuitry of the ADC suggested in the data sheet [21].



1SEE THE VOLTAGE REFERENCE INPUT SECTION FOR REFERENCE SELECTION.
2CREF IS USUALLY A 10µF CERAMIC CAPACITOR (X5R).
3SEE THE DRIVER AMPLIFIER CHOICE SECTION.
4RECOMMENDED FILTER CONFIGURATION. SEE THE ANALOG INPUTS SECTION.
5SEE THE DIGITAL INTERFACE FOR THE MOST CONVENIENT INTERFACE MODE.

Figure 5.14 Circuitry of the ADC [21].

The following equation can be used to calculate the difference between the voltages applied to the “V+” and “V-” pins.

$$V_{IN} = V_{REF} * \frac{D}{2^N} \tag{5.1}$$

V_{IN} stands for the difference between the voltages applied to the “V+” and “V-” pins. V_{REF} stands for the reference voltage, which in this project is 3 V. N is the resolution of the ADC in bits, which in this case is 16, and D is the decimal equivalent of the 16 bits obtained with one readout. [21]

5.3.1.2 AD780

The AD780 is a Band Gap Reference Voltage, which generates a temperature stable reference voltage of 2.5 or 3V from an input voltage between 4 and 36 Volts. In this project an input voltage of 5V is applied and a reference voltage of 3V is generated. The reference voltage of 3V is connected to the REF pin of the AD7980 device via the RC element. The noise can be extremely reduced by using two external capacitors. A load capacitor between output and ground and a compensation capacitor C2. C2 should be between the TEMP pin and ground. As outlined in the datasheet, optimal stabilization is achieved with 100nF for C2 and

100uF for C1. Furthermore, a combination of 22uF and 2.2uF capacitors is utilized to further enhance voltage stability. The temperature output pin allows the AD780 to be configured as a dependable temperature transducer while providing a stable output reference [15].

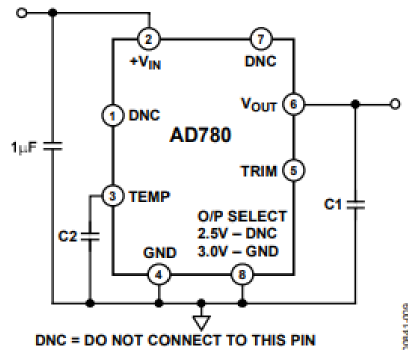


Figure 5.15 Circuitry of the bandgap voltage reference [15].

5.3.1.3 AD5544

The AD5544 is a multiplying Digital to Analog converter (MDAC) with an SPI control interface operated by an FPGA device that communicates with a PC. The DAC AD5544 has four different channels and works with a supply voltage of 2.7 V to 5.5 V. It outputs currents with a 16-bit resolution. Figure 4.16 shows the Schematic diagram of the MDAC device with integrated feedback resistor.

In combination with a transimpedance amplifier, the correction of the amplification and the offset is made possible. The input voltages are “VREFA” to “VREFD”, where VREF corresponds to the reference voltage at the input of the MDAC, R_{FB} corresponds to the integrated feedback resistor of the transimpedance amplifier of 5kΩ DAC. Via a data word (16 bits DAC value) the magnitude of the output current can be controlled. A MDAC is a circuit that effectively multiplies a digital input value with an input voltage. In this configuration, when the digital value is set to zero, the output current “IOUT” is also zero. Similarly, when the digital value is set to its maximum, the resulting

current at “IOUT” reaches its maximum, corresponding to the reference voltage “VREF”.

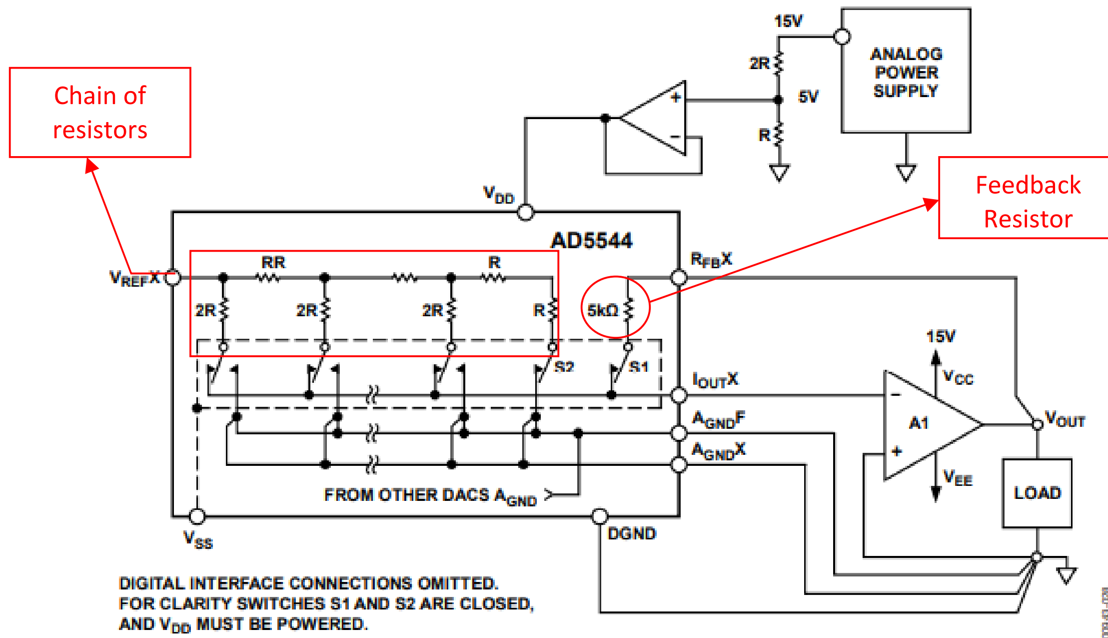


Figure 5.16 Schematic diagram of the MDAC with integrated feedback resistor [22]

Figure 5.16 shows interconnection of the MDAC with the operational amplifier wired as a transimpedance amplifier [14]. Analog voltages “VREFC” and “VREFD” originate from the POLDI sensor, while the reference voltage “RELNEGREF” is present at both “VREFA” and “VREFB” due to their interconnected pins.

The MDAC's output currents, namely “IOUTA” and “IOUTC” are connected and summed. This addition of currents occurs because these outputs function as current sources. Consequently, the currents from channel A (with “RELNEGREF”) and channel C (with “OP90”) are combined. A similar summation principle applies to “IOUTB” and “IOUTD”combining the current from channel B, to which the voltage "RELNEGREF" is applied and the current from channel D, to which the voltage of the POLDI sensor “OP0” is applied.

Each output of the AD5544 MDAC is linked to an AD8656 operational amplifier, tasked with converting the MDAC's current output into a corresponding voltage. The operational amplifier is configured as a transimpedance amplifier, with its feedback resistance determining the current-to-voltage conversion factor.

In the specific case, additional resistors are used at the operational amplifier's output and feedback path to further increase the conversion factor. The resistors required for this are not available on the market with the specific values. Therefore, the combination of multiple resistors in series and parallel is used to achieve the desired values. It's essential to address the fact that a portion of the feedback resistor (5kOhm) resides within the MDAC and connects via the pins RFBC or RFBD respectively. Due to error propagation, achieving high accuracy demands the use of very precise resistors.

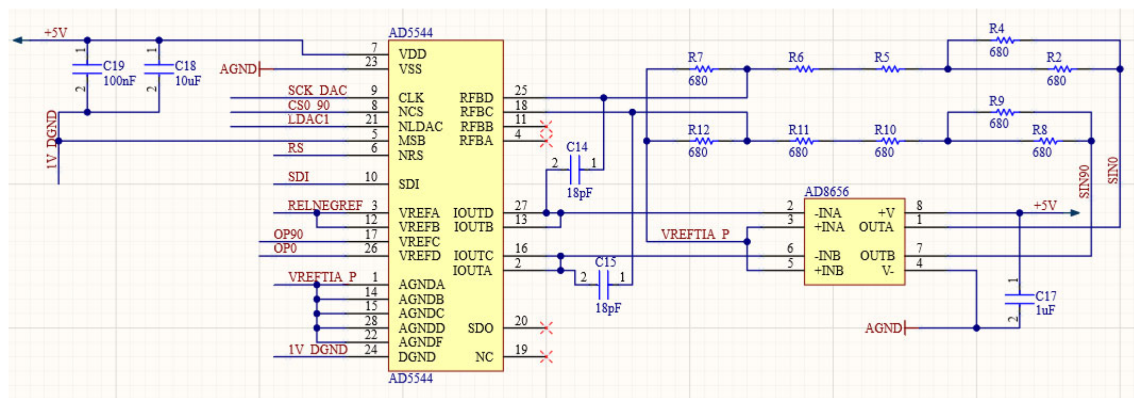


Figure 5.17 Interconnection of the MDAC with the operational amplifier wired as a transimpedance amplifier [14]

The input voltage is converted into output currents in the IC via the chain of switched resistors RDAC1 and RDAC2 (see figure 5.17). The controlling digital value determines which switches are closed or opened, resulting in a weighting of the input voltage. Each channel of the MDAC has an internal feedback resistor RFB, which is 5kOhm and can be connected externally. In principle, the adjustable resistors lie between the input voltage VREF from Figure 5.16 and the output pin IOUT.

This configuration involves linking two channels, such as Channel "B" and "D". One channel receives the signal voltage, while the other channel receives a reference voltage. Channel D is connected to the output voltage OP0 of the Poldi sensor. By examining the digits, one can determine the specific sensor channel of the Poldi sensor under consideration. The OP0 output of the Poldi sensor reflects the sensor signal after passing through a polarization filter aligned at 0 degrees. The OP90 output corresponds to the Poldi sensor output after passing through a polarization filter aligned at 90 degrees. In Figure 5.18, Channel D is labeled as UOP0, and Channel B is labeled as URELNEGREF. VREFTIA represents the local ground, though it is shifted by 1V with respect to the global ground. Within the Altium schematic, a parallel connection of two 340-ohm resistors, R2 and R4, is paired in series with two 680-ohm resistors, R5 and R6, resulting in the following calculated value:

$$\mathrm{R5} + \mathrm{R6} = 680\ \mathrm{Ohm} + 680\ \mathrm{Ohm} = 1360\ \mathrm{Ohm} \tag{5.2}$$

The parallel connection of R2 and R4 results in:

$$\frac{\mathrm{R2R4}}{\mathrm{R2} + \mathrm{R4}} = \frac{680\ \mathrm{Ohm} * 680\ \mathrm{Ohm}}{680\ \mathrm{Ohm} + 680\ \mathrm{Ohm}} = 340\ \mathrm{Ohm} \tag{5.3}$$

When connected in series, their combination results in a cumulative resistance of 1700 ohms. These resistors are consolidated and labeled as R3 in Figure 5.18.

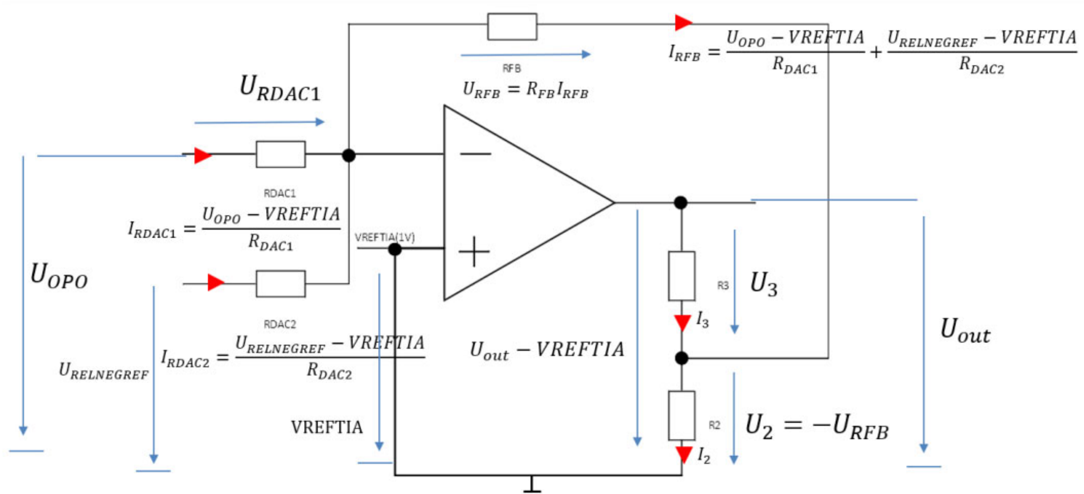


Figure 5.18 Diagram for calculating the current to voltage conversion factor of the POLDI sensor readout circuit

The sum of currents IOUTB and IOUTD is fed into the node at the inverted input -IN of the AD8656 operational amplifier. This operational amplifier's output is connected to the internal feedback resistor RFB of the 5kOhm DAC through the 1700Ohm resistor R3. Additionally, R3 is connected to the +INA and +INB inputs through an additional 680Ohm resistor, R2. Capacitor C14, with a capacitance of 18pF, is utilized to connect "IOUTB" and "IOUTD" to "RFBD". This capacitor serves as a filter for mitigating disturbances at higher frequencies.

When the output of an operational amplifier circuit is fed back to the negative input, a virtual short circuit is effectively established. This concept of a virtual short circuit implies that due to this feedback, the inverting input and the non-inverting input settle at an identical potential or voltage level.

Since the positive input +IN is linked to VREFTIA, VREFTIA is also present at the negative input -IN. Furthermore, UOP0 is connected to the negative input -IN via RDAC1. To calculate the voltage across the resistor "RDAC1" employing the loop rule, yields the following outcome:

$$-U_{OP0} + U_{RDAC1} + V_{REFTIA} = 0 \tag{5.4}$$

$$U_{RDAC1} = U_{OP0} - V_{REFTIA} \tag{5.5}$$

The voltage at the second resistor, where “RELNEGREF” is applied, is calculated using the same computational principle as follows:

$$-U_{RELNEGREF} + U_{RDAC2} + V_{REFTIA} = 0 \tag{5.6}$$

$$U_{RDAC2} = U_{RELNEGREF} - V_{REFTIA} \tag{5.7}$$

Now, to determine the currents “I_{RDAC1}” and “I_{RDAC2}” the respective voltage is divided by the corresponding resistor:

$$I_{RDAC1} = \frac{U_{OP0} - V_{REFTIA}}{R_{DAC1}} \tag{5.8}$$

$$I_{RDAC2} = \frac{U_{RELNEGREF} - V_{REFTIA}}{R_{DAC2}} \tag{5.9}$$

These two currents converge at the node and thus result in a total current.

The current then flows through the feedback resistor “RFB”. The feedback current “I_{RFB}” corresponds to the sum of the two currents:

$$I_{RDAC2} + I_{RDAC2} - I_{RFB} = 0 \tag{5.10}$$

$$I_{RFB} = I_{RDAC2} + I_{RDAC2} \tag{5.11}$$

Across “RFB” a voltage drop “URFB” occurs, which can be calculated using Ohm's law with the assistance of “RFB” and “I_{RFB}”:

$$U_{RFB} = R_{FB} I_{RFB} \tag{5.12}$$

Next, one can determine “U3” by the mesh rule:

$$-U_3 + U_{out} - VREFTIA + U_{RFB} = 0 \tag{5.13}$$

$$U_3 = U_{out} - VREFTIA + U_{RFB} \tag{5.14}$$

With “U_3”, the current “I3” can be calculated via Ohm's law:

$$I_3 = \frac{U_3}{R_3} = \frac{U_{out} - VREFTIA + U_{RFB}}{R_3} \tag{5.15}$$

The voltage U_2 can be calculated in a similar way. VREFTIA is present at the non-inverting input -IN because of the virtual short circuit. The application of the mesh rule results in:

$$-V_{REFTIA}+U_2+U_{RFB}+V_{REFTIA}=0 \quad (5.16)$$

$$U_2+U_{RFB}=0 \quad (5.17)$$

Dissolving according to U2 results in:

$$U_2=-U_{RFB} \quad (5.18)$$

The current I_2 is then given by Ohm's law as:

$$I_2=\frac{-U_{RFB}}{R_2} \quad (5.19)$$

The node rule can be used to relate the currents I_2, I_3 and I_{RFB}. The node rule states that the sum of all currents is zero. Therefore, the result is:

$$I_3+I_{RFB}-I_2=0 \quad (5.20)$$

Now the formulas are used for each current:

$$\frac{U_{out}-V_{REFTIA}+U_{RFB}}{R_3}+I_{RFB}+\frac{U_{RFB}}{R_2}=0 \quad (5.21)$$

$U_{RFB}=RFB*I_{RFB}$ is inserted into the equation and I_{RFB} is excluded. Then the result is:

$$\frac{U_{out}-V_{REFTIA}+R_{FB}I_{RFB}}{R_3}+I_{RFB}+\frac{R_{FB}I_{RFB}}{R_2}=0 \quad (5.22)$$

$$\left(\frac{R_{FB}}{R_3}+1+\frac{R_{FB}}{R_2}\right)I_{RFB}+\frac{U_{out}-V_{REFTIA}}{R_3}=0 \quad (5.23)$$

Since the purpose of this calculation is to calculate the output voltage of the circuit U_{OUT}, this formula is resolved to “U_{OUT}”:

$$\frac{U_{out}-V_{REFTIA}}{R_3}=-\left(\frac{R_{FB}}{R_3}+1+\frac{R_{FB}}{R_2}\right)I_{RFB} \quad (5.24)$$

Multiplying both sides of the equation by R3 gives:

$$U_{out} - V_{REFTIA} = -\left(R_{FB} + R_3 + \frac{R_3 R_{FB}}{R_2}\right) I_{RFB} \tag{5.25}$$

Factoring out -RFB on the right-hand side of the equation gives:

$$U_{out} - V_{REFTIA} = -R_{FB}\left(1 + \frac{R_3}{R_{FB}} + \frac{R_3}{R_2}\right) I_{RFB} \tag{5.26}$$

Term 1 in equation (5.25) is expanded to the fraction R2/R2. This fraction is combined with R3/R2 to give:

$$U_{out} - V_{REFTIA} = -R_{FB}\left(\frac{R_3 + R_2}{R_2} + \frac{R_3}{R_{FB}}\right) I_{RFB} \tag{5.27}$$

I_{RFB} from equation (5.11) is substituted into equation (5.27) together with equations (5.8) and (5.9) to give:

$$U_{out} - V_{REFTIA} = -R_{FB}\left(\frac{R_3+R_2}{R_2} + \frac{R_3}{R_{FB}}\right)\left(\frac{U_{OP0}-V_{REFTIA}}{R_{DAC1}} + \frac{U_{in2}-V_{REFTIA}}{R_{DAC2}}\right) \tag{5.28}$$

Factoring out R_{DAC1} and R_{DAC2} gives:

$$U_{out} - V_{REFTIA} = -\frac{R_{FB}}{R_{DAC1}}\left(\frac{R_3 + R_2}{R_2} + \frac{R_3}{R_{FB}}\right)(U_{OP0} - V_{REFTIA}) - \frac{R_{FB}}{R_{DAC2}}\left(\frac{R_3 + R_2}{R_2} + \frac{R_3}{R_{FB}}\right) * (U_{RELNEGREF} - V_{REFTIA}) \tag{5.29}$$

For R_{DAC1} applies:

$$R_{DAC1} = \frac{R_{FB} * 65536}{N_{DAC1}} \tag{5.30}$$

For R_{DAC2} applies:

$$R_{DAC2} = \frac{R_{FB} * 65536}{N_{DAC2}} \tag{5.31}$$

$N_{DAC1,2}$ are the selected DAC settings between 1 and 65536. Now the equations for R_{DAC1} and R_{DAC2} are inserted in formula (4.28).

$$U_{out} - V_{REFTIA} = -\frac{N_{DAC1}}{65536}\left(\frac{R_3+R_2}{R_2} + \frac{R_3}{R_{FB}}\right)(U_{OP0} - V_{REFTIA}) \qquad (5.32)$$
$$- \frac{N_{DAC2}}{65536}\left(\frac{R_3+R_2}{R_2} + \frac{R_3}{R_{FB}}\right)(U_{RELNEGREF} - V_{REFTIA})$$

By inserting the resistor values used and the values for the voltages V_{REFTIA} and $U_{RELNEGREF}$, the output voltage can be calculated. Inserting the following values R2=680Ohm; R3=1700Ohm; RFB=5kOhm; $U_{RELNEGREF}$=0.375V and V_{REFTIA}=1V into equation (5.32) gives equation (5.33):

$$U_{out} - 1V = -\frac{N_{DAC1}}{65536} * 3.84 * (U_{OP0} - 1V) + \frac{N_{DAC2}}{65536} * 2.4V \qquad (5.33)$$

Formula (5.33) is converted to U_{OUT}, and this results in two separate terms. One term is multiplied by U_{OP0} and $U_{RELNEGREF}$ and once by V_{REFTIA}. At the end, +1 is added for V_{REFTIA}.

$$U_{out} = -\frac{N_{DAC1}}{65536} * 3.84 * U_{OP0} - \frac{N_{DAC2}}{65536} * 1.44V \qquad (5.34)$$
$$+ 1V\left[3.84 * \left(\frac{N_{DAC1}}{65536} + \frac{N_{DAC2}}{65536}\right) + 1\right]$$

5.3.1.3.1 DAC settings

Figure 5.19 illustrates the relationship between output voltage measurements with respect to VREFTIA, with NDAC2 configured as a function of NDAC1. The red curve represents NDAC2/65536=0.8, while the blue curve corresponds to NDAC2/65536=0.3.

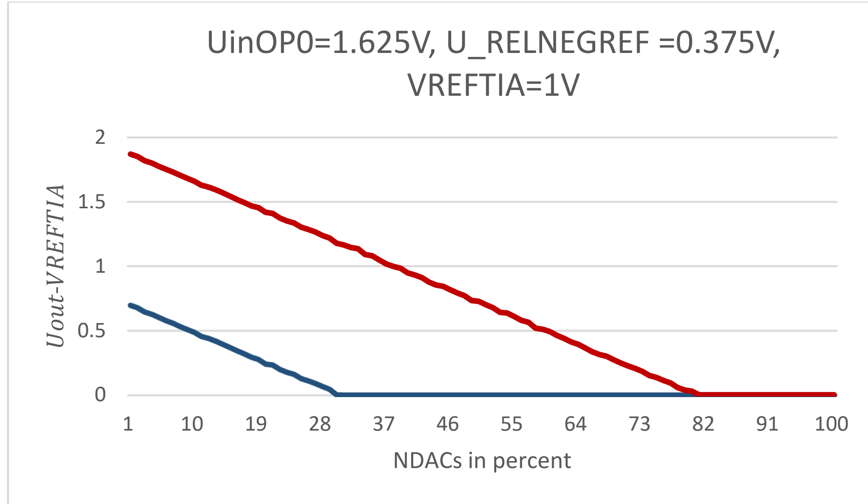


Figure 5.19 Output voltage of the POLDI readout chain related to VREFTIA as a function of the DAC setting of channels

The variation of NDAC1 is depicted along the x-axis in this graph, revealing a noteworthy pattern: as NDAC1 is raised, the corresponding output voltage progressively diminishes. The parameters UOP0=1.625V, URELNEGREF=0.375V, and VREFTIA=1V are selected constant. The Y-axis displays the output voltage with respect to VREFTIA, while the X-axis represents the percentage value of NDAC1 relative to the value 65536. In other words, a value of 10 signifies 10% of 65536. Employing a program to manage the MDAC facilitates the configuration of NDAC values spanning from 1 to 65536. The disparity between DAC1 and DAC2 values bears direct influence on the output voltage relative to VREFTIA; a greater difference between these values corresponds to a higher output voltage against VREFTIA. Notably, Figure 4.19 illustrates a linear reduction in output voltage as NDAC1 approaches the value of NDAC2. Once NDAC1 equals NDAC2 or surpasses it, the outcome is an output voltage of 0V.

$$U_{out} - V_{REFTIA} = -\frac{N_{DAC1}}{65536}\left(\frac{R_3+R_2}{R_2} + \frac{R_3}{R_{FB}}\right)(U_{OP0} - V_{REFTIA}) - \frac{N_{DAC2}}{65536}\left(\frac{R_3+R_2}{R_2} + \frac{R_3}{R_{FB}}\right)(U_{RELNEGREF} - V_{REFTIA}) \quad (5.35)$$

$$U_{out} - V_{REFTIA} = -\frac{N_{DAC1}}{65536} * 2.4V + \frac{N_{DAC2}}{65536} * 2.4V \qquad (5.36)$$

In equation (5.35), the value within the blue highlight is negative, while the one within the red highlight is positive. UOP0 – VREFTIA yields a positive number, and when multiplied by -NDAC1, it results in a negative value within the blue-marked range. Conversely, in the red region, (URELNEGREF - VREFTIA) equates to a negative value, given that RELNEGREF has a value of -0.625V. When multiplied by -NDAC2, it becomes a positive number. The negative value reduces the output voltage. As the percentage increases, the negative value gains more prominence, thereby offsetting a larger portion of the positive term. Consequently, a voltage drop occurs, as depicted in Figure 5.19.

When the NDAC1 setting is increased, the RDAC1 resistance becomes smaller, this can be seen in formulas (5.35) and (5.36). As NDAC1 value is increased, the output voltage decreases (see Figure 5.19).

Formula (5.36) is used to compare whether the measured values correspond to the expected values. (Rounded to two decimal places).

Table 5.1 Measured values compared with the results of the formula.

NDAC1 setting	NDAC2 setting	Measured value	Result of the formula Uout-VREFTIA	Deviation
10%	30%	0.49V	0.48V	0.01V
20%	30%	0.24V	0.24V	0v
50%	30%	0V	(-0.48V)	(-0.48V)
10%	80%	1.66V	1.66V	0V
20%	80%	1.42V	1.44 V	0.02V
50%	80%	0.73V	0.72V	0.01V
80%	80%	0.03V	0V	0.03V

The red curve was taken to illustrate the calculation. NDAC2 is set to 80%. NDAC1 is set at 50% for example. This results in this formula:

$$U_{out} - 1V = -0.5 * 3.84 * (1.625V - 1V) - 0.8 * 3.84 * (0.375V - 1V) = 0.72 \quad (5.37)$$

The comparison of the measured values with the values from formula (5.37) results in small deviations in the two-digit Millivolt range.

If NDAC1 is greater than NDAC2, the result of the formula is a negative number. However, in the measurements, the lowest number is zero because the ADC's input voltage range is limited to zero volts.

5.4 Poldi sensor board

POLDI is an optical sensor manufactured by AdvICo Microelectronics. This optical sensor measures the polarization angle of linear polarized light. The incident light is measured by four photodiodes. These photodiodes are integrated on a CMOS chip, each covered by a linear polarization filter. The polarization filters are realized in one of the metallization planes of the CMOS process by parallel conductive tracks, which lie at four different angles (0°, 45°, 90°, and 135°). When the POLDI sensor is irradiated with linearly polarized light, each diode delivers a photocurrent whose current intensity depends on the angle between the polarization plane of the incident light and the orientation of the polarization filter. [3]

The Poldi sensor features 40 pins for integration with other boards. to facilitate seamless and efficient pin connections with other boards. The design of the POLDI sensor circuit board involves connecting the POLDI sensor to the board, a header pin and a terminal block with 12 pins terminal block connects a total of eight readout channels. Four channels are provided for the readout of the diodes, which are covered with polarization filters with different orientations (0°, 45°, 90°, 135°). The remaining four channels are available for the readout of the

diodes without polarization filters, which can be used to determine the intensity of the incident light. Furthermore, the necessary power supply is supplied via its dedicated pin. Figure 5.20 shows the POLDI sensor circuit board.

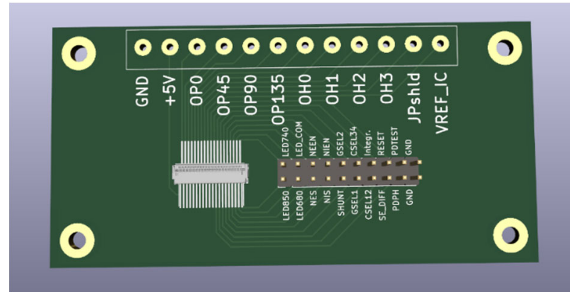


Figure 5.20 The POLDI sensor circuit board

In this work, only the four channels which are covered with polarization filters are used (OP0, OP45, OP90, OP135).

5.5 Cmod A7-35T

The Cmod A7 is a compact, 48-pin DIP form factor board centered around a Xilinx Artix-7 FPGA. This board boasts essential components, including a USB-JTAG programming circuit, USB-UART bridge, clock source, and basic I/O devices. It presents 44 Digital FPGA I/O signals and two FPGA analog inputs, making it seamless for users to incorporate programmable logic designs into a solderless breadboard setup.

FPGA configuration data is stored as bitstreams with a .bit file extension. These files are transferred from a PC to the FPGA using the onboard USB-JTAG system via a micro-USB port. Using Xilinx's Vivado software, bitstreams can be generated from VHDL or Verilog source files. Bitstreams reside in the FPGA's volatile memory cells, defining its logical functions and circuit connections. This configuration persists until overwritten by a new bitstream or upon removal of board power.

The Cmod A7 features an FTDI FT2232HQ USB-UART bridge connected to a Micro-USB port. This bridge enables communication between the board and PC applications using standard Windows COM port commands. Additionally, the FT2232HQ manages the Digilent USB-JTAG setup. By integrating USB-UART and USB-JTAG functionalities into a single device, the Cmod A7 is programmable, communicative via UART, and powered through a single Micro-USB cable, streamlining its usage [22].

In this project the Cmod A7-35T board acts as an interface between the POLDI sensor and FPGA that extracts the angle information by digitally processing the sensor signals. The four channel signals which are covered with polarization filters and digitalized with the hardware described before are connected to the digital I/O pins.

6 Design of CORDIC blocks in Verilog

In this chapter, the two Verilog designs for the calculation of quotients and the arcsine function are explained. Starting from the Verilog source code the two calculation modules are explained in detail. In addition, simulations in each case verify the functional correctness of the design.

6.1 Requirements

The following requirements were considered in the design of the two processing blocks.

- No use of multipliers:

Compared to additions or subtractions, multiplications require a high computational effort. Here the CORDIC algorithm is used among other things because this effort is avoided.

- No use of floating-point arithmetic

The use of floating-point signals requires complex floating-point arithmetic leading to increased hardware effort. To avoid this, a fixed-point number format is used instead. For this purpose, the numbers are multiplied by a power of 2 before processing which corresponds to a simple digit shift on the binary level to the left to reserve bit positions for the coding of values less than one. This can be realized in a simple way by a shift register. The factor to be used results from the number of iteration steps in the CORDIC algorithm. In the last, n-th step, 2^{-n} is added or subtracted.

- Program flexibility:

The source code should allow the flexible modification of design parameters e.g. the number of iterations. The computational accuracy of the algorithm and the time required for its execution is scaled over the number of iterations. Moreover,

since accuracy also depends on the nature of the computational task, it should be possible to adapt the number of iterations and the bid width of crucial signals to the respective requirements.

- Pipelining:

The calculation is to be implemented as a pipeline process. This is a concept that is used in processors (see Figure 6.1). Instead of calculating different values one after the other, the individual steps of the operation are executed in parallel. This means that as soon as the first step of the first value has been executed, the first iteration for the calculation of the next value is started.

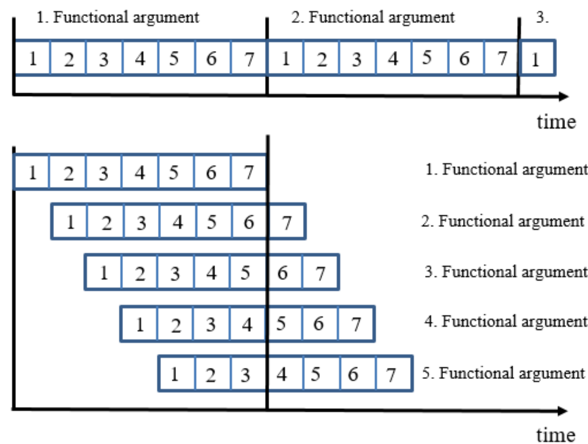


Figure 6.1 Pipelining example

In this case, a total operation is the calculation of a function value. Its individual step is a single iteration according to the CORDIC algorithm. The latency, i.e. the time required for the calculation of the function value, does not change in this case because it is specified by the cycle time and the number of iteration steps. However, the number of function evaluations per time unit can be increased by parallel execution. Since as many function evaluations are executed in parallel as iteration steps, only the clock frequency of the FPGA limits the maximum rate of input and output values.

6.2 Verilog design of the division calculation

The block diagram for the realization of the division operation is shown in Figure 6.2. The basis for the calculation is the vectoring mode of the CORDIC algorithm in linear mode, which was presented in chapter 3.1.3.3

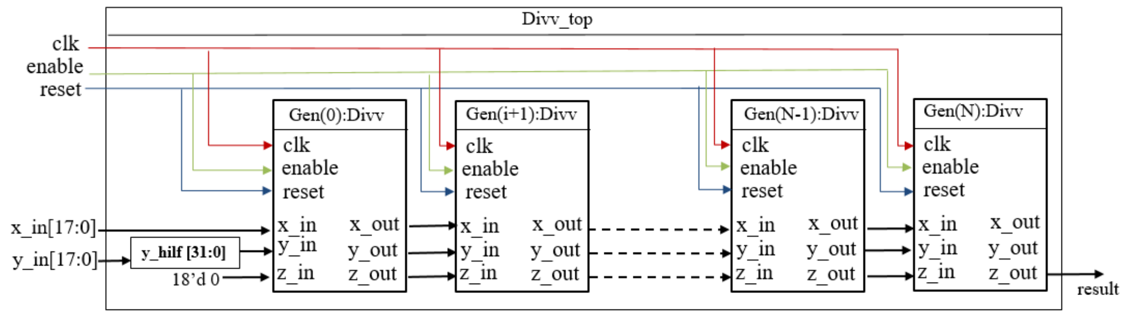


Figure 6.2 The concept for the realization of the division

The calculation of the individual iteration steps is performed in each case by a separate module. Each of these modules receive the calculation results (x_{i-1}, y_{i-1}, z_{i-1}) of the previous step or module as input and provides the results (x_i, y_i, z_i) of the respective iteration step as an output of the corresponding iteration step.

In Verilog, this module concept is represented by a hierarchical structure. The top level is the "Divv_top" module, which controls the overall iteration process. It consists of the modules for the individual iteration steps. Due to their similarity, only one module (named "divv") needs to be designed, which is instantiated repeatedly. This reduces the implementation effort and also guarantees the simple adjustment of the number of iterations. In the present case, the calculation of the quotient is performed with up to 15 iterations. The modules for this calculation are explained in the following two sections.

6.2.1 "divv" submodule

The implementation of the module starts with its declaration and the definition of the inputs and outputs (see source code 6.1). The module receives the generic

integer parameter k1 from the main module, which is needed to perform the calculations.

```
14 module divv #(parameter k1 =1)
15 (
16    input   wire                     clk,
17    input   wire                     reset,
18    input   wire                     enable,
19    input   wire  signed   [17:0]   x_in,
20    input   wire  signed   [31:0]   y_in,
21    input   wire  signed   [17:0]   z_in,
22    output  reg            [17:0]   x_out,
23    output  wire           [31:0]   y_out,
24    output  wire           [17:0]   z_out
25 );
```

Source code 6.1 Declaration of ports of the "divv" module

The inputs and outputs of the module are defined in lines 16 to 24. declaring the port name, the port type (in or out) and the type of the port's signal. The first three inputs, (lines 16 to 18 in source code 6.1) are signals, which are used to control the module and have the following function:

- **"clk":** The system clock is provided at this input.
- **"reset"**: Via this input the module receives an asynchronous active-low reset. This signal originates from the "divv_top" module and is passed on to all instances of the module "divv", so that all modules are reset at the same time.
- **"enable":** This input is used to activate the module. This also originates from the module "divv_top", i.e. all modules of the type "divv" are simultaneously enabled (enable = 1) or disabled (enable = 0).

The other three inputs and the three outputs (lines 19 to 24 in source code 5.1) are used to transmit numeric values. For this purpose, signals with the bit length of the respective numeric value are assigned. The length is identical for the input and respective output. The declared lengths of 18 or 32 bits are explained in the following sections.

- **“x_in”, “x_out”:** The input and output signal of the ports x_in and x_out size are identical because the algorithm does not change this signal. Therefore, the signal length corresponds to the length of the initial value, which is set to 16 bits. Although the signal and its size does not change, it is used in every iteration step, which is why all “divv” modules get this signal as an input port.

- **“y_in”, “y_out”:** The value of y is reduced to zero by the process iterations. Since no floating-point numbers are to be used, the required accuracy is achieved by multiplying the 18-bit starting value by a factor of 2^{14} in the module “divv_top”. Thus, y is a fixed-point number with 14 places after the decimal point.

- **“z_in”, “z_out”:** In the division algorithm z represents the sum of the displacements (see equation (4.48)). The individual displacements have a size of 2^{-i} and have a length of up to 14 bits in the selected fixed-point format. Considering that the maximum value which can be processed is two which can be coded by two bits and an additional bit for the sign, the length of this signal is 18 bits.

```
30 reg   signed [31:0] y1;
31 reg   signed [17:0] z1;
32 wire signed [31:0] x_in1, y_in1 ;
33 wire signed [17:0] z_in1 ;
34
35 assign x_in1 = ((x_in) * (2 ** 14));
36 assign y_in1 = (y_in);
37 assign z_in1 = (z_in);
```

Source code 6.2 Conversion to internal signals

The signal is defined by type and length when the module is created (see source code 6.2). The input signal “x_in” is multiplied by 2^{14} to convert it into the same fixed-point number format as the other two signals y_in and z_in. Additionally,

the corresponding input signal is assigned into an internal signal (see line 35 in source code 6.2).

In addition to the input signals, the module-internal signals “y1” and “z1” are defined, which record the calculation results from which the output signals are generated.

For the processing of the signals, i.e. the execution of the individual iteration steps of the CORDIC algorithm, a process is implemented. The design for this is shown in source code 6.3. A process is executed only when it is active. The activity of the process is determined, as already mentioned, by the two signals “clk” and “reset”. These are passed to the process in the so-called sensitivity list (see line 39). If one of these inputs changes, the process is triggered and the values of the signals are processed accordingly.

First, it is checked whether the calculation is to be reseted, the signal “reset” has the logical value 0, and the corresponding assignments are made (lines 43 to 45). In this case, the internal signals y1 and z1 are set to zero. The output signal for x should never assume the value 0 since it serves as divisor for a quotient calculation in the further course. Therefore, value 1 is assigned here.

The actual calculation is activated via the clock signal “clk”, which periodically changes between the values 0 and 1. The calculation is triggered on a rising edge of the clock signal. This is evaluated in line 47 by the condition “else”. The execution of the calculation is also linked to the condition that the module should be active, i.e. the signal “enable” has the logical value 1 (line 49). Otherwise, the following calculations are skipped, i.e. the module effectively makes no changes to the signals.

```
39 always @( posedge clk  or negedge reset)
40 begin
41     if ( reset == 1'b0 )
42         begin
43          y1 <= 18'd0;
44          z1 <= 18'd0;
45          x_out <= {{17{1'b0}},{1'b1}};
46         end
47     else
48       begin
49        if ( enable== 1'b1 )begin
50          if ( y_in1 > 0 ) begin
51             if ( x_in1 > 0 )begin
52              y1 <= ( y_in1 - ( x_in1 / k1 ) );
53              z1 <= z_in1 + ((2**14)/k1);
54             end
55 
56             else if ( x_in1 < 0 )begin
57              y1 <= ( y_in1 + ( x_in1 / k1 ) );
58              z1 <= ( z_in1 - ( ( 2 ** 14 ) / k1 ) );
59             end
60          end
61 
62          else if(y_in1 < 0 )begin
63             if ( x_in1 < 0 )begin
64               y1 <= ( y_in1 - ( x_in1 / k1 ) );
65               z1 <= ( z_in1 + ( ( 2 ** 14 ) / k1 ) );
66             end
67 
68              else if ( x_in1 > 0 )begin
69                y1 <= ( y_in1 + ( x_in1 / k1 ) );
70                z1 <= ( z_in1 - ( ( 2 ** 14 ) / k1 ) );
71              end
72          end
73 
74          else
75            begin
76             y1 <=  y_in1;
77             z1 <= z_in1 ;
78           end
79           x_out <= x_in;
80         end
81      end
82   end
83 
84 assign y_out = y1;
85 assign z_out = z1;
86 
87 endmodule
```

Source code 6.3 Process for performing an iteration step in the "divv" module

Lines 50 to 82 correspond to the actual calculation. According to the iteration rule in equation (3.51), the sign of "y_in1" determines the decision factor and thus the calculation of "y1" and "z1", which results in a distinction in three cases. In deviation from the iteration rule, arbitrary signs are allowed for the x-value "x_in1". For negative x-values the sign of the changes of the y and z values must be changed, otherwise, the desired shift of y in direction 0 will not be realized.

This results in two subcases each and a total of four cases for performing the calculation. According to the iteration rule, the the y-value is changed $x_i.2^{-i}$ and the z-value is changed by 2^{-i} in the i-th step. Taking into account the shift by 2^{14}, the following value results for x_in1. 2^{14-i} in the i-th module. Since the i-th module number is not available as a variable, a parameter k1 with the value 2^i is passed to the module instance by the main module. With the help of the parameter k1 the new values of y1 and z1 can be calculated by "x_in1/k1" or "2**14/k1" resepectively.

After completion of the calculation, the output signals are generated. The x-value is not changed during the process so that the input variable can be passed on (see line 79). Since the input signal "x_in" already has the correct format, it is immediately assigned to the output "x_out". The assignment is made within the process because otherwise the signal is fed through without any storage element. This means that this assignment is not clocked with the edge of the clock. As a result, the clock-controlled pipeline would have inconsistent data in the pipeline stages. Further assignments can be done outside the process because the used internal signals "y1" and "z1" are already clocked (see lines 84 and 85).

6.2.2 Main module "divv_top"

The implementation of the main "divv_top" module starts, with the declaration of the signals and the inputs and outputs of the module (see source code 6.4). With the parameter "Drehung" in line 13 the total number of iteration steps is defined. Since the iteration counting starts with 0, a number of 15 iterations is defined by specifying a value of 14. This quantity is used in the following to create a corresponding number of modules of the type "divv". By changing the assigned value, the number of iterations can be easily adjusted, where 15 is the maximum number of iterations that can be performed. A larger number of iterations could be specified, but effectively only 15 iteration steps are executed. As already described for the "divv" module, regardless of the number of iterations

the signals are implemented as integers representing a fixed-point number with 14 decimal places. Iterations beyond the number 15 would result in values that are less than 1 in the defined format and are therefore considered to be 0. As a result, the associated modules “divv” would not introduce any modification to the input signals. Thus, a value of more than 14 for parameter “Drehung” does not lead to an error, but also not to a more accurate result. In case a higher precision is required also the fixedpoint number scheme has to be modified by choosing a higher bit-width and adding additional decimal places.

```
13 module divv_top #(parameter drehung = 14)
14 (
15    // Port Declarations
16    input   wire            clk,
17    input   wire            enable,
18    input   wire            reset,
19    input   wire    [17:0]  x_in,
20    input   wire    [17:0]  y_in,
21    output  wire    [17:0]  result,
22    output  wire    [17:0]  x_out,
23    output  wire    [31:0]  y_out
24 );
```

Source code 6.4 Declaration of variables and ports for the “divv_top” module

The signals in the module definition from line 16 to 23 in source code 6.4 correspond in their format and meaning to the port signals of the “divv” module and have already been described in detail in chapter 5.2.1. The main difference is that these are the external ports of the top level design via which the input signals for the overall calculation and its results are transmitted. The signal “result” contains the result for z after the given number of iterations and thus the desired approximate value for the quotient “y_in” by “x_in”. The outputs “x_out” and “y_out” are not required. They are implemented during Verilog design so that the correctness of the calculation can be checked and the causes of any errors can be better located.

Furthermore, the internal signals that connect the individual modules of the type “divv” are to be defined. Since the number of iterations should be easily adaptable and a generic signal designation is costly, these signals are summarized in arrays,

whose size can be varied. When generating the modules, the required signal can be assigned by specifying the index from the array.

The implemented arrays are lists with signals of the same bit length of 18 bit and 32 bits respectively (see source code 6.5).

```
25 // Internal Declarations
26 wire [17:0]x_outl [0:drehung];
27 wire [31:0]y_outl [0:drehung];
28 wire [17:0]z_out [0:drehung];
29 wire [31:0]y_in_hilf ;
```

Source code 6.5 Defining the types for the internal and output signals for the division calculation

Furthermore, the input signal "y_in" has to be transformed to the already described format of a fixed-point number with 14 decimal places. This is achieved by appending 14 bits 0. For this the auxiliary signal "y_in_hilf" is introduced. Its first 16 bits (from 31 to 14) are the same as "y_in". The other 14 elements (from 13 to 0) are set to zero (see source code 6.6).

```
31 assign y_in_hilf[31:14] = y_in  ;
32 assign y_in_hilf[13:0] = { 14'd0 };
```

Source code 6.6 Transformation of the input signal for y to 32-bit length

The "y_in_hilf" signal generated in this way serves as input signal for the first "divv" module. For the input signal "x_in" such a transformation is not necessary, because the shift of 14 digits necessary for the calculation is done within the module.

The main task of the "divv_top" module is to generate the sub-modules of type "divv" and to establish the connections between the modules using the corresponding port assignments. The implementation is shown in source code 6.7.

```
36 for ( i = 0 ;  i <= drehung  ; i = ( i + 1 ) )
37  begin : gen1
38   if (i==0)
39     begin : beginn
40     divv  #( .k1(2 ** i))  normierung(.clk(clk),.reset(reset),.enable(enable),
41                                   .x_in(x_in),.y_in(y_in_hilf),.z_in({ 18'd0 }),
42                                   .x_out(x_outl[i]), .y_out(y_outl[i]) ,.z_out(z_out[i]) );
43
44     end
45   else if(i>=1)
46     begin: berechnung
47     divv #( .k1(2 ** i))  div(.clk(clk),.reset(reset),.enable(enable),.x_in(x_outl[ i - 1]),
48                              .y_in(y_outl [ i - 1] ) ,.z_in(z_out[ i - 1 ]),.x_out(x_outl[i]),
49                              .y_out(y_outl[i]) ,.z_out(z_out[ i ]) );
50
51     end
52  end
53
54 assign result = z_out[ drehung ];
55 assign x_out = x_outl[ drehung ];
56 assign y_out = y_outl[ drehung ];
57
58 endmodule
```

Source code 6.7 Generation and connection of the individual modules for the division calculation

The modules are generated and the individual entries for variables and ports are assigned according to the declaration in source code 6.1. Here a distinction must be made between the first and all further modules since the input signals from "divv_top" are to be transferred to the first module. During the initialization of the module, parameters and ports are assigned according to the sequence in source code 6.4 (lines 16 to 24). The inputs "clk", "reset", "enable" and "x_in" of "divv_top" are taken directly from the main module. Instead of "y_in" the signal "y_in_hilf" is used, as already explained. The iteration starts with z_in =0, so the "divv_top" module has no input signal. Therefore, this input sets to 0 in the first module. The outputs of the first module form the first element (i=0) of each result array.

The modules following the first module are similarly defined for arbitrary positive values of i. As with the first module, the control signals "clk", "reset" and "enable" are taken from the main module. The only difference is the connection of the inputs for the signals x, y and z. They are the output signals from the previous module, which can be read chosen from the array with the

output signals. Finally in line 54 to 56 the output signals of the last “divv” module are passed to the outputs of “divv_top” module.

6.2.3 Simulation

In this section, the individual signals are simulated for the design using Vivado to verify the correctness of the implementation. The first step is to check the design of the circuit by generating a schematic depending on the given number of iterations. Figure 6.3 shows an exemplary schematic of a design with four iterations (i = 3).

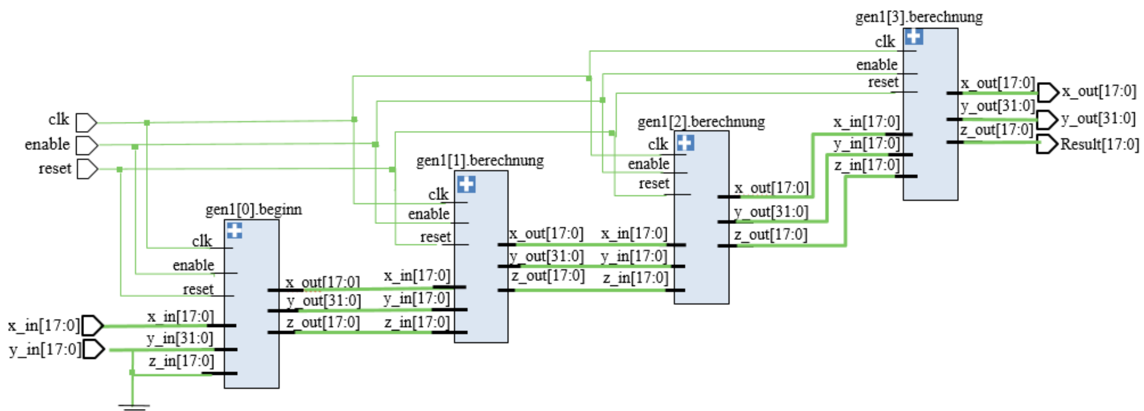


Figure 6.3 Schematic of the division calculation for i=3

It can be seen that the correct number of modules is created, with the assignments of the instances “begin” on the left and “berechnung” for all the further instances (source code 6.7) made correctly. The first module receives the external signals “x_in” and “y_in” as input. The initial value for z is always zero. The corresponding assignment for the input “z_in” of the first module is implemented correctly by connecting the input to the ground. The outputs of the previous module for x, y, and z are connected to the corresponding inputs. The output signals of the last module, which the final results of the calculation are connected to the outputs “x_out”, “y_out” and “Result” respectively. In the upper part of Figure 6.3 the three control signals “clk”, “enable” and “reset” are passed as

planned directly from the input of the top level design to the corresponding inputs of the “divv” modules.

After the correctness of the circuit has been checked, the individual signals are simulated. As an example, consider the calculation with 15 iterations ($i = 14$) in Figure 6.4. The right part of the figure shows the time history of the individual signals. The numeric values are represented in the format of a fixed-point number with 14 digits.

The top three signals are the control signals. While “reset” and “enable” assume the constant values 0 and 1 respectively, the clock signal “clk” changes periodically between 0 and 1. As can be seen from the time scale above, a total period of 20 ns is defined. However, the simulation primarily serves to check the signals and does not reflect the actual time sequences for the individual calculations. An additional post-synthesis timing simulation is used to quantify the actual time sequences and, if necessary, to adjust the period duration of the clocking signal.

The test signals “x_in” and “y_in” are generated on the basis of numerical series with arbitrary numbers. With user-defined numbers, whereby the signal changes are carried out at the time of a falling edge (change from 1 to 0) of the clocking signal. By comparing the clocking signal with the underlying signals, it can be seen that the calculations in the submodules and the changes at their output signals are performed at the rising edge of the clock signal respectively. Furthermore, the pipeline mode of operation can be seen, 14 calculations are executed in parallel at a time, which are started with a time delay.

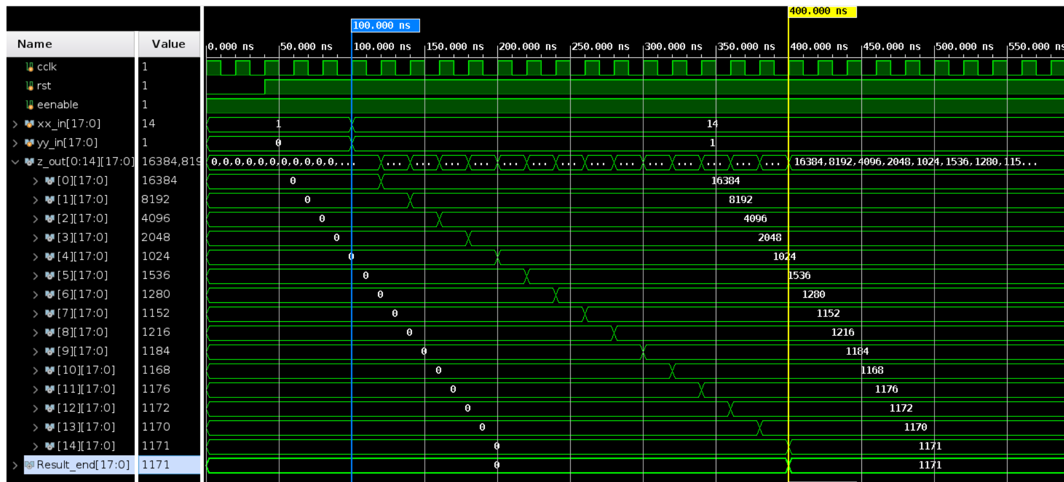


Figure 6.4 Behavior simulation of the division calculation for i=14

The results of the calculation for the quotient 1/14 which is obtained by choosing the input signals "x_in" = 14 and "y_in" = 1 are listed in Table 6.1. The relative deviation to the exact result 1/14 ≈ 0.071429 is listed.

Table 6.1 Calculation results for "x_in" = 14 and "y_in" = 1

Module Nr.i	$Z_out(i)=z_i.2^{14}$	z_i	ERROR
0	16384	1,000000	1300,00%
1	8192	0,500000	600,00%
2	4096	0,250000	250,00%
3	2048	0,125000	75,00%
4	1024	0,062500	12,50%
5	1536	0,093750	31,30%
6	1280	0,078125	9,40%
7	1152	0,070312	1,50%
8	1216	0,074219	4,00%
9	1184	0,072265	1,20%
10	1168	0,071289	0,17%
11	1176	0,071777	0,50%
12	1172	0,071533	0,50%
13	1170	0,071411	0,01%
14	1171	0,071472	0,08%

After 15 iterations, a relative deviation of 0.08% is achieved. The results in Table 6.1 prove that the individual iteration steps are basically execution of the individual iteration steps. Numerically, these values are the same as theoretical calculations with 16-bit fixed-point numbers and shown in table 6.2. Apart from the sign bit and the pre-decimal place, this format has 14 binary decimal places. Therefore, all fractions smaller than 1.2^{-14} are deleted without replacement and they are rounded down.

Table 6.2 Theoretical calculations of 1/14

i	x_i	y_i	z_i	d_i	2^{-i}	$Z_i.2^{14}$
0	14	1	0.000000	-1	1	0
1	14	-13	1.000000	1	0.5	16384
2	14	-6	0.500000	1	0.25	8192
3	14	-2.5	0.250000	1	0.125	4096
4	14	-0.75	0.125000	1	0.0625	2048
5	14	0.125	0.062500	-1	0.03125	1024
6	14	-0.3125	0.093750	1	0.015625	1536
7	14	-0.09375	0.078125	1	0.0078125	1280
8	14	0.015625	0.070312	-1	0.00390625	1152
9	14	-0.03906	0.074219	1	0.001953125	1216
10	14	-0.01172	0.072265	1	0.0009765625	1184
11	14	0.001953	0.071289	-1	0.00048828125	1168
12	14	-0.00488	0.071777	1	0.000244140625	1176
13	14	-0.00146	0.071533	1	0.0001220703125	1172
14	14	0.000244	0.071411	-1	6.103515625E-05	1170

The last line contains the result for a calculation with 15 iterations. As the variables have only 14 binary decimal places, calculations with a number of iterations $i \geq 15$ lead to the same result as with a number of iterations of $i = 14$, in this case: 1170.2^{-14}.

In the following, the calculation for different values and signs of the input signals is investigated. The results of these simulations are summarized in Table 6.3.

Table 6.3 Decimal calculation results after 15 iterations for different quotients

Nr.	x_in	y_in	z_{exact}	z_out(i)= $Z_{14}.2^{14}$	Z_{14}	Error
1	14	1	0,071428	1171	0,071472	0,08%
2	-14	-1	0,071428	1171	0,071472	0,08%
3	14	-1	-0,071428	-1171	-0,071472	-0,08%
4	-14	1	-0,071428	-1171	-0,071472	-0,08%
5	2	3	1,5	22575	1,4999389	-0,0041%
6	10	19	1,9	31129	1,8999633	-0.0019%
7	1000	1	0,001	17	0,0010375	3,7597%
8	9000	1	0,0001111	1	0,0000610	-45,0684%
9	10000	-1	0,0001	-1	-0,0000610	-38,4844%
10	4	7	1,75	28672	1,75	0,0000%
11	1	2	2	32767	1,9999389	-
12	-1	9	-9	-32767	-1,9999389	-

The first four lines contain the simulated results for the possible inputs for the calculation of 1/14 or -1/14. It can be seen that regardless of the sign of the quotient, the same result is achieved in terms of amount. This proves that input signals are independent of the sign for the computation and the algorithm has been implemented correctly.

The two following lines 5 and 6 show that the calculations also work for the case of "y_in" greater than "x_in" i.e. for quotients greater than 1. Furthermore, it can be seen that the relative error tends to decrease with increasing magnitude of the quotient. This is because the algorithm works with a limited accuracy of 14 binary decimal places with 15 iteration steps. Thus, the amount of the absolute error can be estimated upwards with $e_{max} = 2^{-14} \approx 6.1035.10^{-5}$. Consequently, the relative error related to the value of the quotient decreases with the size of the quotient.

Accordingly, the calculation of very small quotients leads to large errors. Line 7 shows the example of the calculation of 1/1000. If quotients are calculated that are smaller than 2^{-13}, as in lines 8 and 9, the calculation reaches the limit of its resolving power and the result is always 2^{-14}, which means a very large error. Quotients smaller than 2^{-14} cannot be calculated due to the restriction of the bit length for the input variables.

A special case is the calculation of quotients that correspond to an integer power of 1/2. If this power is less than or equal to 14, the iteration leads to the exact result. Line 10 shows this with the example of the calculation of 4/7.

Furthermore, as already explained in chapter 3.1.3.3, the application of the CORDIC algorithm is limited to quotients with magnitudes of less than 2. Lines 11 and 12 show the calculation results for invalid entries. Depending on the sign, the calculation increases or decreases continuously until the maximum amount of

$$|z_{14}|_{\max} = \sum_{i=0}^{14} 2^{-i} = \sum_{i=0}^{14} (2^{-1})^{i} = \frac{1-2^{-15}}{1-2^{-1}} = \frac{2^{15}-1}{2^{14}} = \frac{32767}{16384} \approx 1{,}9999389 \qquad (6.1)$$

is reached. This value corresponds to the largest quotient that can be calculated with 15 iteration steps. For any input whose magnitude is larger than this limit, this result is always calculated.

The simulations prove that the design represents the CORDIC algorithm correctly and with no errors. The problems of the low accuracy for small quotients and the upper limits quotients can be solved by proper preprocessing of the input signals. By a simple digit shift in the binary range, more favorable or permissible values can be achieved for the input signals. One possible approach to this is to equalize the number of binary digits of the two input values by multiplying the smaller value by 2^k. The result can be shifted back to the actual value by multiplication with or division with 2^k. For example, instead of 9/1 the calculation for 9/8 would be carried out and the result would be multiplied by 8, which can be achieved by

shifting three binary digits to the left. Instead of the quotient 1/1000, 512/1000 would be calculated in this approach and the result would be shifted to the right by nine digits. However, these operations cannot be implemented for output signal with 16 bits length. Because this limits the binary format of the calculation result to one place before the decimal point and 14 places after the decimal point. Therefore, non-zero numbers can only represent the value range $2^{-14} \leq z \leq 2 - 2^{-14}$.

In addition, a further simulation is performed to check the correct function of the design at the timing level. A section of the simulation result is shown in Figure 6.5.

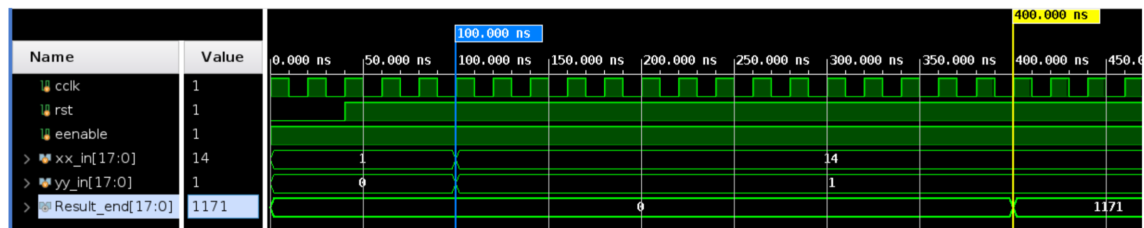


Figure 6.5 Results of the time simulation of the division with i = 14 for a clock of 50 MHz

The calculation starts at a positive edge with the time coordinate 100 ns (blue line). The result of the calculation can be seen by the lowest signal, which is shown in Figure 6.5, when the signals changes from 0 to 1171 at a time coordinate of 400ns (yellow line). With the ideal operation without time delay, the calculation would require 14 clock cycles of 20 ns, i.e. 280 ns. Taking into account the initial time, the ideal termination time of the simulation is 380 ns. With the FPGA used, clock frequencies of up to 100 MHz are possible; The simulation shows that for frequencies of more than 50 MHz or clock times of less than 20 ns, the calculation works correctly.

The simulations show that the design works as desired. Considering the explained limitations resulting from the CORDIC algorithm and the maximum signal

length, the calculation results are correct. Furthermore, the correct function can be guaranted up to a clock frequency of 100 MHz.

6.3 Verilog design of the Arcsine function

The concept for implementing the calculation of the arcsine using the CORDIC algorithm is similar to the approach for the division calculation (see Figure 6.6). Two module types are created for the calculation. The main module “arcsin_top” controls the calculation and creates the submodules of the type “arcsin”, which implement a single iteration of the algorithm.

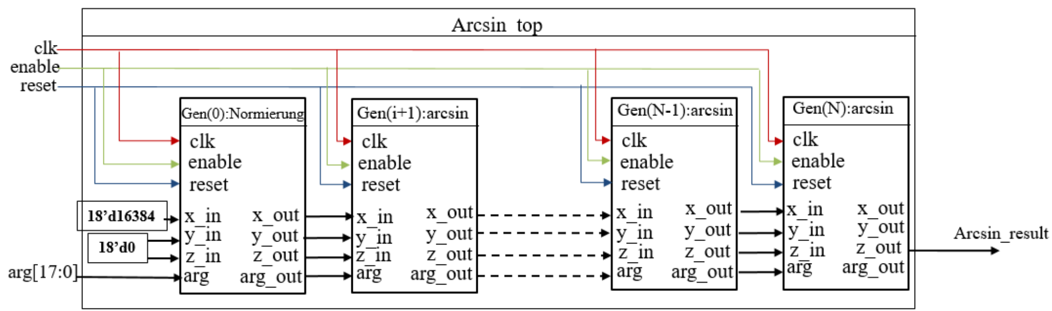


Figure 6.6 Concept for the realization of the CORDIC algorithm for the computation of the arcsine with FPGA

The essential difference to the division block is located at the input signals. For the arcsine calculation, the signal “arg” is used additionally. This represents the argument for which the calculation must be executed. In all iterations the argument is only required for the determination of the decision factors and is therefore forwarded unchanged from submodule to submodule. In contrast, in the division block, the quotient to be calculated is determined by the input values for x (denominator) and y (denominator). Here, input signals for x, y and z are not required, because the initial values for these variables are given by the iteration rule (see chapter 3.1.3.2.1).

6.3.1 "Arcsin" Submodule

The module starts with the declaration of the inputs and outputs signals (see source code 6.8). The module uses three parameters (line 13) which are chosen according to the iteration number i by the main module. Parameter "k1" is the divisor of value 2^i, which is needed for the calculation of the output signal "y_out". This parameter is similar to the parameter "k1", which was introduced in the division calculation. The second parameter "kp2" is the divisor of value 2^{2i}. The third parameter takes the value $\arctan(2^{-i})$ and represents the step size for the change of z in the i-th iteration step.

```
13 module arcsin#(parameter integer k1=1,parameter integer kp2=1, parameter integer arct=0)
14                (
15                 input wire clk,
16                 input wire rst,
17                 input wire enable,
18                 input wire signed [17:0] x_in,
19                 input wire signed [17:0] y_in,
20                 input wire signed [17:0] z_in,
21                 input wire signed [17:0] ARG,
22                 output wire signed [17:0] Arg_out,
23                 output wire signed [17:0] x_out,
24                 output wire signed [17:0] y_out,
25                 output wire signed [17:0] z_out
26                );
```

Source code 6.8 Declaration of variables and ports for the "arcsin" module

The input and output signals on lines 15 to 25, except for the additional signals "ARG" and "Arg_out", are similar in format and meaning to those of the module "divv" (see source code 6.1) and have already been explained there.

The special feature of this module is that signals with the same bit width of 18 bits are used for all numerical variables. This format corresponds in each case a signed fixed-point number with 14 decimal places. The actual limits that occur are the following:

- Since the sine function maps to the value range [-1, 1], permissible inputs can only be in this range, i.e. $|ARG| \leq 1$.

- When calculating z, an amount of arctan(2^{-i}) is added or subtracted in each step. Thus, the maximum value that can be calculated is limited to

$$|z| \leq \sum_{i=0}^{14} \arctan\left(2^{-i}\right) \approx 1{,}78 \tag{6.2}$$

- The values for x and y, starting from the initial values $1/_{k^2} = 1$ and 0 respectively strive to the following limits:

$$\lim_{i\to\infty} x_i = \sqrt{1-\arg^2} \quad \text{and} \quad \lim_{i\to\infty} y_i = \arg \tag{6.3}$$

Therefore, the values in the iteration are always smaller than the maximum value of the argument, which is the value 1.

The module calculation is done with internal signals which represent the initial values (“x0”, “y0”, “z0”, “t0”) and the calculation results (“x1”, “y1”, “z1”, “t1”). Here the same number format is used for all external signals which are signed fixed point numbers with 18-bit length (see source code 6.9).

```
28 wire signed [17:0] x0 ,y0 ,t0 ,z0 ;
29 reg signed [17:0]  x1;
30 reg signed [17:0]  y1;
31 reg signed [17:0]  z1;
32 reg signed [17:0]  t1;
33
34 assign x0 = x_in;
35 assign y0 = y_in;
36 assign z0 = z_in;
37 assign t0 = ARG;
```

Source code 6.9 Definition of the internal signals “arcsin” module

In source code 6.9, first the internal signals are declared with signed format and a length of 18 bits. Since the external signals use the same number format, the external input signals can be directly assigned to the corresponding internal signals (see source code 6.9).

The design which implements the iteration steps of the CORDIC algorithm in the submodule is shown in source code 6.10. The “always” block has a sensitivity containing of the control signals “clk” and “rst”. The operations that are performed event as a result of the change of one of these signals can be seen in the source code 6.10: If the signal “rst” has the logical value 0, the calculation is reset by setting all output signals to 0 (see lines 43 to 46). Otherwise, under the condition that “enable” is high, on a rising edge of the clock signal “clk” the calculation is triggered (see line 51).

```
39 always@(posedge clk or negedge rst )
40 begin
41   if (rst== 1'b0 )
42    begin
43     x1 <= 18'd0;
44     y1 <= 18'd0;
45     z1 <= 18'd0;
46     t1 <= 18'd0;
47    end
48   else
49    begin
50     if (enable == 1'b1)
51      begin
52       if ((t0 - y0) < 0)
53              begin
54               x1 <= x0 -(x0/kp2) + (y0*2/k1) ;
55               y1 <= y0 - (y0/kp2)  - (x0*2/k1) ;
56               z1 <= z0 - arct  ;
57               t1 <= t0+(t0/kp2);
58              end
59       else if ((t0 - y0) > 0)
60             begin
61              x1 <= x0 -(x0/kp2) - (y0*2/k1) ;
62              y1 <= y0 - (y0/kp2)  + (x0*2/k1) ;
63              z1 <= z0 + arct  ;
64              t1 <= t0+(t0/kp2);
65             end
66       else
67       begin
68         x1 <= x0;
69         y1 <= y0;
70         z1 <= z0;
71         t1 <= t0 ;
72       end
73    end
74  end
75 end
76   assign x_out = x1;
77   assign y_out = y1;
78   assign z_out = z1;
79   assign Arg_out = t1;
80 endmodule
```

Source code 6.10 The process of performing an iteration step in the “arcsin” module

The actual calculation, which is described in lines 52 to 75, corresponds to the iteration rule in equation 3.45. The three calculation variants, which result from the three possible expressions of the decision factor, are chosen by the evaluation of if-conditions. For the calculation of “x1”, “y1”, “z1” and “t1” the step sizes 2^{-i}, 2^{-2i} and $\arctan(2^{-i})$ are required. The iteration number “i” is introduced by means of parameters. The parameters k1, kp2 and arct which are generated by the “arcsin_top” module and passed to the submodule during instantiation. In the last part of the process, after completion of the calculations, the calculation results are assigned to the corresponding output variables (see lines 76 to 79).

6.3.2 Main module “arcsin_top”

The definition of the variables as well as the inputs and outputs of the main module for the calculation of the arcsine is done with the help of the shown design in source code 6.11.

```
17 module arcsin_top  #(parameter N = 14)( clk, rst, enable, Arg, Arcsin_result);
18 input wire clk;
19 input wire rst;
20 input wire  enable;
21 input wire signed [15:0] Arg;
22 output wire signed [15:0] Arcsin_result;
```

Source code 6.11 Declaration of variables and ports for the module “arcsin_top”

The parameter “N” defines the number of iteration steps. The assignment of the value 14 corresponds to the execution of 15 iterations of the CORDIC algorithm. As already explained in the context of the division calculation and proven by simulation, any number of iterations can be specified, but values of $N > 14$ do not lead to a more exact result because of the limitation of the number format to 14 binary decimal places, but only extend the calculation time. The ports “clk”, “rst” and “enable” are control signals. The signal “Arg” corresponds to the value for which the arcsine is to be calculated. “Arcsin_result” is the calculation result after N+1 iterations. No external signals are required for the signals x, y, z and t. Because they are only auxiliary use and are passed on as internal signals from

submodule to submodule. For this purpose, as already explained for the main module of the division, several arrays are generated, which contain all calculation results of the respective size (see source code 6.10, lines 76 to 79). As seen in line 79 of the source code 6.10, the internal signal “Arg_out” is defined to scale the input argument. Thus, it is required to pass the scaled argument from submodule to submodule. The component $i \in \{0,1,...,N\}$ of “x_out”, “y_out”, “z_out” and “Arg_out” contains the corresponding value of the output of submodule i.

The key task of the main module is the generation and connection of the submodules, which perform the individual calculation steps. The corresponding code is shown in source code 6.12. As submodules require 18bits to calculate the arcsine and the input argument has a bit width of 16 bits, the number of bits have to be extended with respect to the most significant bit (line 30). $2*\arctan(2^{-i})$ are calculated converted into the fixed-point binary format which is used for the calculations and introduces as constants with the parameter initial_values in line 31. The “generate” statement generates the individual submodules. A distinction must be made between the first module with i = 0 (lines 37 to 44) and the other modules (lines 45 to 52). In both cases, the parameters “k1”, “kp2” and “arct” are defined first, which contain the displacements for the quantities x, y, z and t during the execution of the CORDIC algorithm. By using them as parameters, with constant values no additional hardware for their calculation is created.

```
30 assign Arg_hilf = {{2{Arg[15]}},Arg};
31 parameter  initial_values  = { 16'd25736,16'd15193,16'd8027,16'd4075,16'd2045,16'd1024,
   16'd512,16'd256,16'd128,16'd64,16'd32,16'd16,16'd8,16'd4,16'd2};
32 generate
33 genvar i;
34 for(i = 0; i <= N; i=i+1)
35   begin: gen
36
37    if(i == 0)
38     begin :beginn
39        arcsin #(.k1(2**i),.kp2(2**(2*i)), .arct(initial_values[((N-i)*16)+:16]))
40               asin( .clk(clk), .rst(rst), .enable(enable), .x_in(18'd16384),
41                     .y_in(18'd0), .z_in(18'd0), .ARG(Arg_hilf),
42                     .Arg_out(arg_out_1[i]), .x_out(x_out_1[i]),
43                     .y_out(y_out_1[i]), .z_out(z_out_1[i]) );
44     end
45    else if (i>=1)
46      begin : D
47       arcsin #(.k1(2**i),.kp2(2**(2*i)), .arct(initial_values[((N-i)*16)+:16]))
48              asin (.clk(clk), .rst(rst), .enable(enable), .x_in(x_out_1[i-1]),
49                    .y_in(y_out_1[i-1]), .z_in(z_out_1[i-1]), .ARG(arg_out_1[i-1]),
50                    .Arg_out(arg_out_1[i]), .x_out(x_out_1[i]),
51                    .y_out(y_out_1[i]), .z_out(z_out_1[i]) );
52     end
53   end
54 endgenerate
55 assign Arcsin_result =z_out_1[N]  ;
56 endmodule
```

Source code 6.12 Generation and connection of the individual modules for the arcsine calculation

The main difference between the two submodule types consists in the assignment of inputs and outputs, which is done according to the sequence in the module definition (see source code 6.8). (lines 40 - 43), the input “x_in” of the first module receives the value of the 1 as a decimal number, i.e.16384, while the inputs “y_in” and “z_in” are set to zero. Furthermore, the input “Arg” is assigned the input signal “Arg_hilf” of the main module. For the following modules, these inputs are connected to the outputs of the previous module (lines 48 - 51). Finally, the calculation result of the last module is assigned to the corresponding output “Arcsin_result” of the main module.

6.3.3 Simulation

The Verilog design for the calculation of the arcsine is checked in the same way as the design for the division calculation using Vivado based on several simulations. First, the circuit is checked, then the calculation results are analyzed and finally the time behavior is defined. As an example, the schematic for the

circuit with 2 iteration steps (i = 3) is considered. The schematic generated by the Schematic function is shown in Figure 6.7.

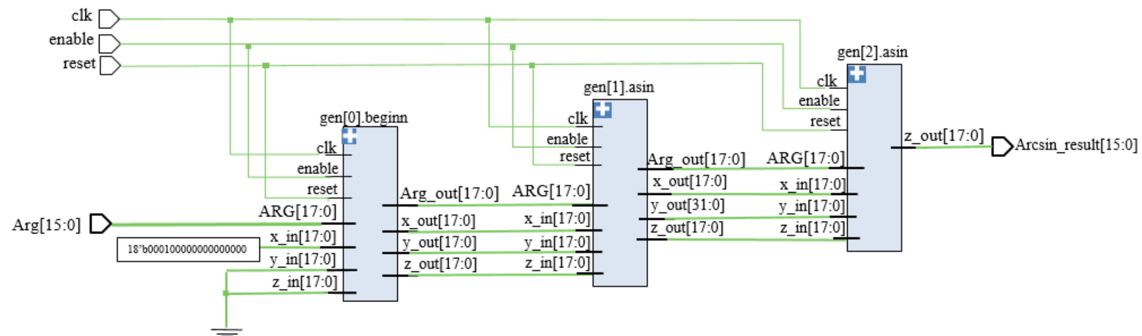


Figure 6.7 The schematic diagram of the arcsine calculation for i=3

The schematic proves the implementation is basically correct. It is clear that three submodules have been created. The three external control signals “clk”, “enable” and “reset” are connected to the corresponding inputs of all submodules. The signal “Arg” with the argument to be evaluated is only connected to the first module. Furthermore, the first module receives a signal with a constant input (“x_in”), where the exact value is shown in the circuit diagram as a binary number. The two remaining inputs (“y_in” and “z_in”) are connected to the ground, implicating an assignment with 0. It can also be seen how the four internal signals are each connected from the submodule outputs to the corresponding inputs of the subsequent module. An exception is only the last module, where only one output is required, namely the one for the calculation result.

In the next step, the signal characteristics and the calculation results of the individual modules are analyzed. The result for the simulation with 15 iterations (i=14) is shown in Figure 6.8. The process is similar to division calculation. The table at the left margin shows the individual signal designations and the values at the position of the yellow marker. The marker is located at the end of the first iteration run.

The three top signals are the control signals, which can change between the values 0 and 1. At first, the “reset” signal is set to “0”, whereby all output signals and get the value “0”. Since “enable” permanently has the value “1”, it is possible to start the calculation after the change of the “reset” signal from “0” to “1”. The values to be calculated are assigned with the signal “Arg”. At first, this signal has the value “0”. Afterward, a sequence of values is assigned, whereby the change to the next value takes place in each case updating the input signals with the falling edge of the “clk” signal.

It can be seen that after the value assignment, the signals between the modules, which all initially have the value 0, successively receive the calculation results of the first argument. Subsequently, calculation results are forwarded to the following iteration step. This means that the pipeline works as desired.

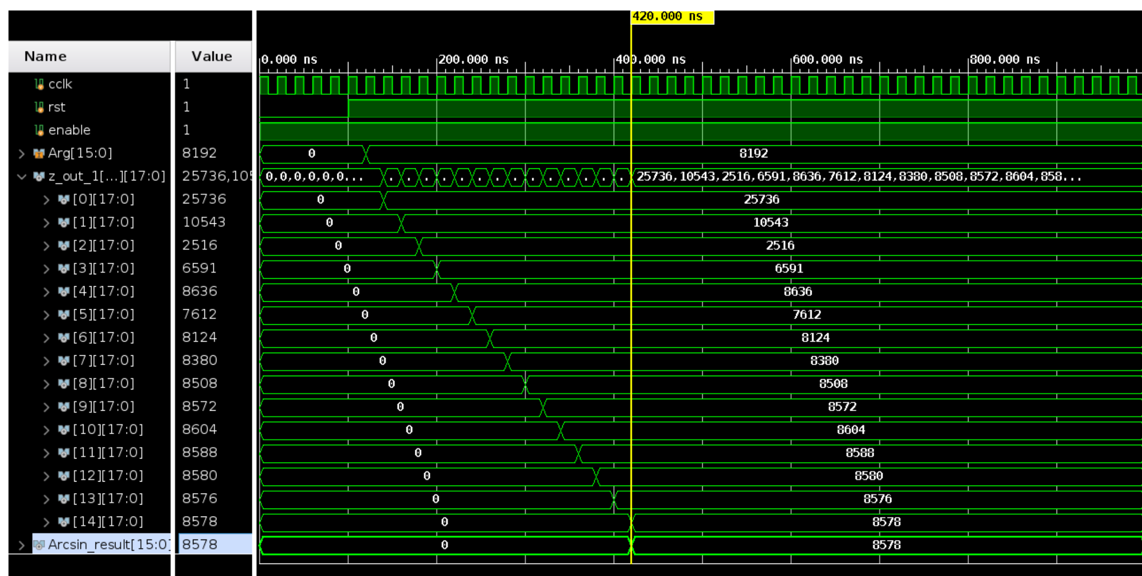


Figure 6.8 Behavior simulation of the signals of the arcsine calculation for i=14

In the following, the calculation process for an argument of $0.5 = 8192.2^{-14}$ is considered in detail. The results of the individual submodules are summarized in Table 6.4. Additionally, the relative deviation to the exact result of $\pi/6 = 30°$ is given.

Table 6.4 Decimal arcsine calculation results with “Arg” = 0.5

Nr.	z_out(i)= $Z_i.2^{14}$	Z_i		Error
0	25736	1.570801	90.000°	200%
1	10543	0.643494	36.870°	-22.90%
2	2516	0.153544	8.797°	-70.68%
3	6591	0.402254	23.047°	-23.18%
4	8636	0.527091	30.200°	0.67%
5	7612	0.464612	26.620°	-11.27%
6	8124	0.495859	28.411°	-5.30%
7	8380	0.511484	29.306°	-2.31%
8	8508	0.519296	29.754°	-0.82%
9	8572	0.523203	29.978	-0.08%
10	8604	0.525156	30.089°	0.30%
11	8588	0.524179	30.033°	0.11%
12	8580	0.523691	30.005°	0.02%
13	8576	0.523447	29.991°	-0.03%
14	8578	0.523569	29.998°	-0.01%

After 15 iterations, a relative deviation of 0.01% is achieved. The calculation results shown correspond to the theoretical calculation, which is shown in table 6.4. In this theoretical calculation, the decimal parts of the individual terms in the calculation formulas that exceed 14 binary decimal were deleted with rounding. Considering this restriction, which results from the selected number format, the individual results match exactly. Thus, the calculation is implemented correctly.

To evaluate the correctness of the design for any arguments and the accuracy of the algorithm, further simulations are performed for different arguments with i=14. The simulation results are summarized in Table 6.5. Additionally, the exact result, which is calculated, as well as the relative deviation of the iteration result from these values is listed.

Table 6.5 Theoretical calculations of arcsin(0.5)

i	$x_i.2^{14}$	$y_i.2^{14}$	$z_i.2^{14}$	$t_i.2^{14}$	2^{-i}	$2.\text{Arctan}(2^{-i})$	d_i
Initial values	16384	0	0	8192	-	-	1
0	0	32768	25736	16384	1	25736	-1
1	32768	24576	10543	20480	0.5	15193	-1
2	43008	6656	2516	21760	0.25	8027	1
3	40672	17304	6591	22100	0.125	4075	1
4	38350	22320	8636	22186	0.0625	2045	-1
5	39708	19902	7612	22208	0.03125	1024	1
6	39076	21138	8124	22213	0.015625	512	1
7	38743	21747	8380	22215	0.0078125	256	1
8	38573	22049	8508	22215	0.00390625	128	1
9	38487	22200	8572	22215	0.001953125	64	1
10	38443	22275	8604	22215	0.0009765625	32	-1
11	38465	22238	8588	22215	0.00048828125	16	-1
12	38476	22219	8580	22215	0.000244140625	8	-1
13	38481	22209	8576	22215	0.0001220703125	4	1
14	38479	22214	8578	22215	6.103515625E-05	2	1

Table 6.6 Calculation results after 15 iterations for different arguments

Nr.	Arg	$\text{Arg}.2^{14}$	Arcsin_result $= z_{14}.2^{14}$	z_{14}	z_{exact}	Error
1	0.25	4096	4138	14.471°	14.478°	-0.05%
2	-0.25	-4096	-4138	-14.471°	-14.478°	-0.05%
3	0.75	12288	13892	48.581°	48.590°	-0.02%
4	-0.75	-12288	-13892	-48.581°	-48.590°	-0.02%
5	0.01	164	164	0.574°	0.573°	0.17%
6	0.001	16	16	0.056°	0.057°	-1.75%
7	2^{-13}	2	0	0°	0.007°	-100%
8	1	16384	25736	90.000°	90.000°	0%
9	-1	-16384	-25736	-90.000°	-90.000°	0%

Simulations 1 to 4 show the results for different values with different signs. The results are independent of the sign of arguments. With the same amount of argument, an identical result is achieved. This is not only true for the results shown after 15 iterations, but also for each individual step of the iteration. In general, good accuracy is achieved for all arguments.

For very small arguments, as in simulations 5, 6 and 7, the result is usually equal to the argument. In view of the fact that for very small angles the approximation relation applies

$$\sin(\varphi) = \varphi - \frac{1}{6}\varphi^3 + \frac{1}{120}\varphi^5 + \cdots + \frac{(-1)^{2n-1}}{(2n-1)!}\varphi^{2n-1} + \cdots \approx \varphi \quad (6.4)$$

In simulations 8 and 9, the arcsine values for the maximum permissible arguments are determined. Here, the simulation leads to the correct result of $\pi/2 = 90°$.

In general, it can be seen that the arcsin function can be calculated correctly and with high accuracy for all values by means of the double rotation CORDIC algorithm. Furthermore, a time simulation was carried out. This leads to the result that the arcsine calculation can be realized error-free with a clock frequency up to 100 MHz.

7 Rotation Angle Processing Unit

This chapter presents the hardware architecture which is used to compute rotation angles by means of arcsine and division arithmetic implemented as CORDIC blocks.

Figure 7.1 shows the module for the entire signal processing with its inputs and outputs.

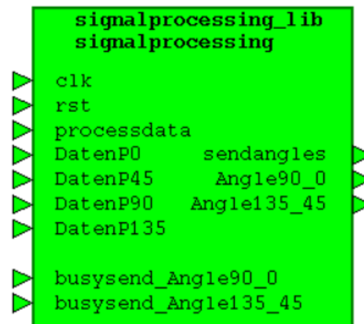


Figure 7.1 Block diagram “signalprocessing” module

The input signals “DatenP0” to “DatenP135” as well as “processdata” of the “signalprocessing” module are connected to the ports of the AD7980 interface component. The AD7980 interface component takes over access to all ADCs of the readout system via an SPI interface in 3-WIRE mode. These ADCs are used to convert the voltage signals of the readout system into digital values. The input signals “busysend_Angle90_0”, and “busysend_Angle135_45” as well as the output signals “Angle90_0”, “Angle135_45”, “sendangles”, are used for communicating and sending calculated angles to the PC via the “Datensendung” module.

The exact function of the AD7980 interface and “Datensendung” modules will be discussed in more detail in the next chapter where integration of the “signalprocessing” module into the VHDL design is explained [9].

Figure 7.2 shows the structure of the "signalprocessing" module. The module consists of five components, where "cnt_signalprocessing" corresponds to a counter, "sm_signalprocessing" to a state machine, "Preprocessing" to normalize the input signals and preprocessed for the angle calculation by the "Processing" module. The "Processing" module takes over the calculation of the required trigonometric function. The result of the calculation is then prepared in the "Postprocessing" module.

Explanations of the individual input and output signals of the modules can be found in the chapters of the respective modules. Since all modules have the input signals "clk" and "rst", these are explained once for simplification:

- **"clk"** is the system clock and has a frequency of 50MHz.
- **"rst"** is an asynchronous active low reset signal which ensures that after implementation of the Verilog design on the FPGA all modules are reset.

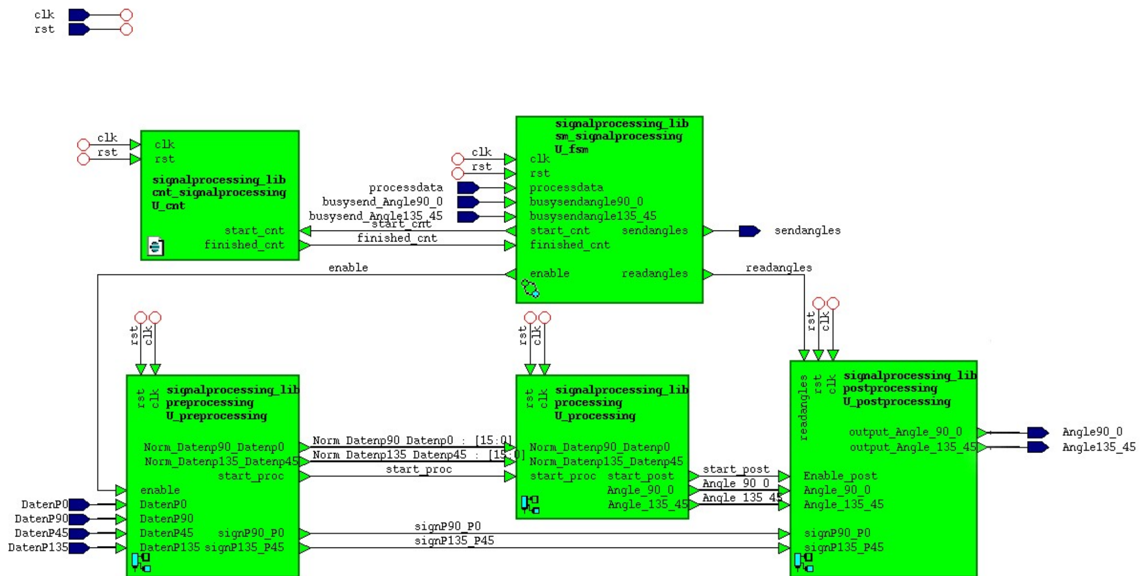


Figure 7.2 Structure of the "signalprocessing" module

The function of each of the "signalprocessing" modules will be discussed in more detail in the following.

7.1 "preprocessing" module

7.1.1 Functionality

The "preprocessing" module reads data from the "AD7980" module and prepares the ADC values for the calculation of the angles. This includes the determination of gain and offset correction values for the POLDI sensor signals which are required to normalize the signals. The processing is done in several steps based on the method described in chapter 3. For this purpose, first, the module gathers some samples. During sampling, the input signals of all four channels are read out and difference signals are formed. For each difference signal formed in this way, the maximum and minimum values are determined to calculate the offset correction, as well as the gain factor coefficients. When the sampling process is completed, the normalized values are calculated for the new input signals. First, the difference between the signals of the new input signals is calculated and their value is compared with the maximum and minimum values of the initial sampling process. If these values are larger or smaller than the maximum/minimum values, they are considered as the new maximum and minimum values respectively, and then used to normalize the signals by a gain factor and an offset correction term. After that, the magnitude and the signs of the signals are separated. The normalized signals are sent to the "processing" module to calculate the arcsine function while the sign values are stored for 15 clock cycles and sent to the "postprocessing" module after the arcsine Cordic block has finished its calculation.

Figure 7.3 shows the structure of the "preprocessing" module. The module consists of four modules, where "cnt_preprocessing" corresponds to a counter, "calc_preprocessing" is sequential logic which is used to find the maximum and minimum of the POLDI sensor signals, "divvend_preprocessing" module is an instance of the "divv_top" module explained in [24] is used to execute the gain

normalization, while the "fsm_preprocessing" module corresponds to a state machine.

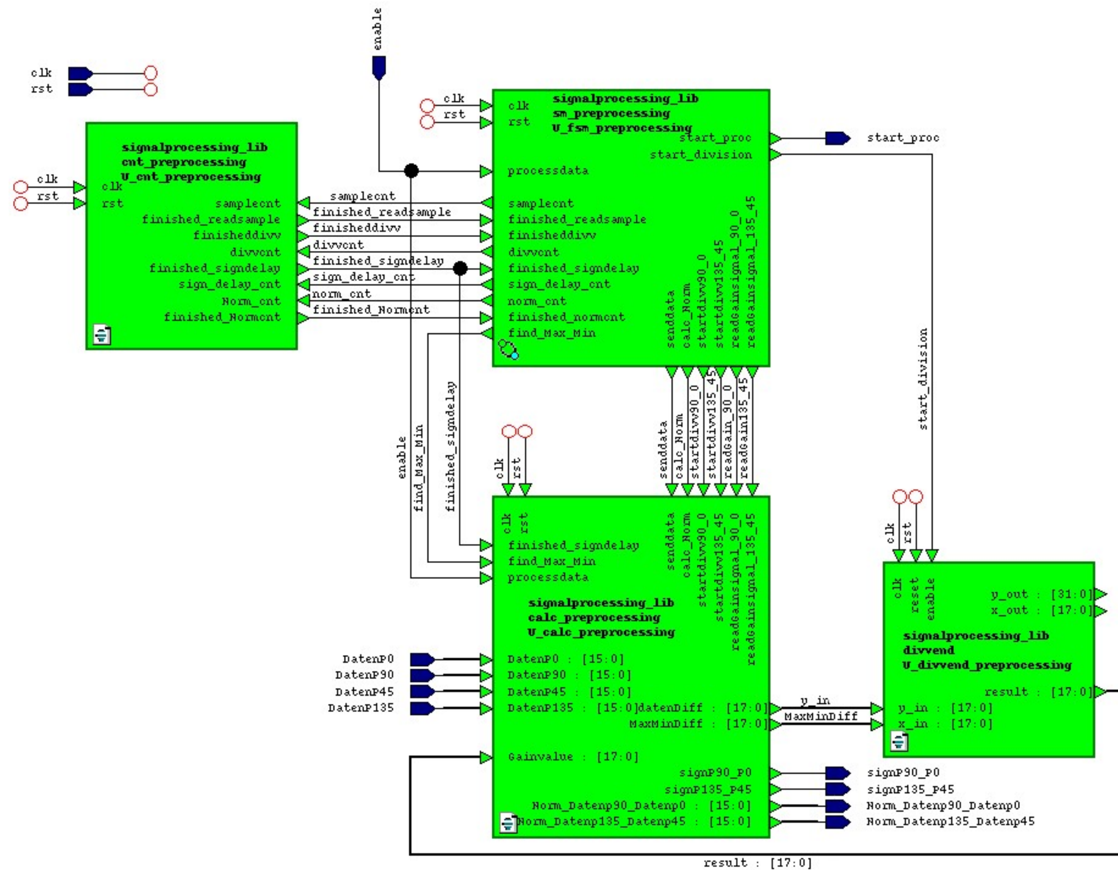


Figure 7.3 Structure of the "preprocessing" module

The "preprocessing" component has the following input signals:

"Enable" indicates the availability of new POLDI sensor signals.

"DatenP0", "DatenP45", "DatenP90", "DatenP135" are 16-bit unsigned vectors and contain the POLDI sensor data to be normalized.

The "preprocessing" component has the following output signals:

"start_proc" enables the "processing" module whenever normalized data has been prepared for processing.

“signp90_p0”, “signp135_p45” forwarded the signs of calculated differences of the Poldi sensor signal channels P90/P0 and P135/P45 respectively. for further processing in the “postprocessing” component.

Finaly **“Norm_Datenp90_Datenp0”** and **“Norm_Datenp135_Datenp45”** are the normalized differences of the Poldi sensor signal channels P90/P0 and P135/P45 respectively.

This module contains two state machines that are operated concurrently. Figure 7.4 and Figure 7.5 show the state diagram of the “fsm_preprocessing” modules. The state machine in Figure 7.4 is used to control the normalization process of the POLDI signals. While the state machine in Figure 7.5 is used for synchronizing the sign signals and the calculated angles in the “postprocessing” module.

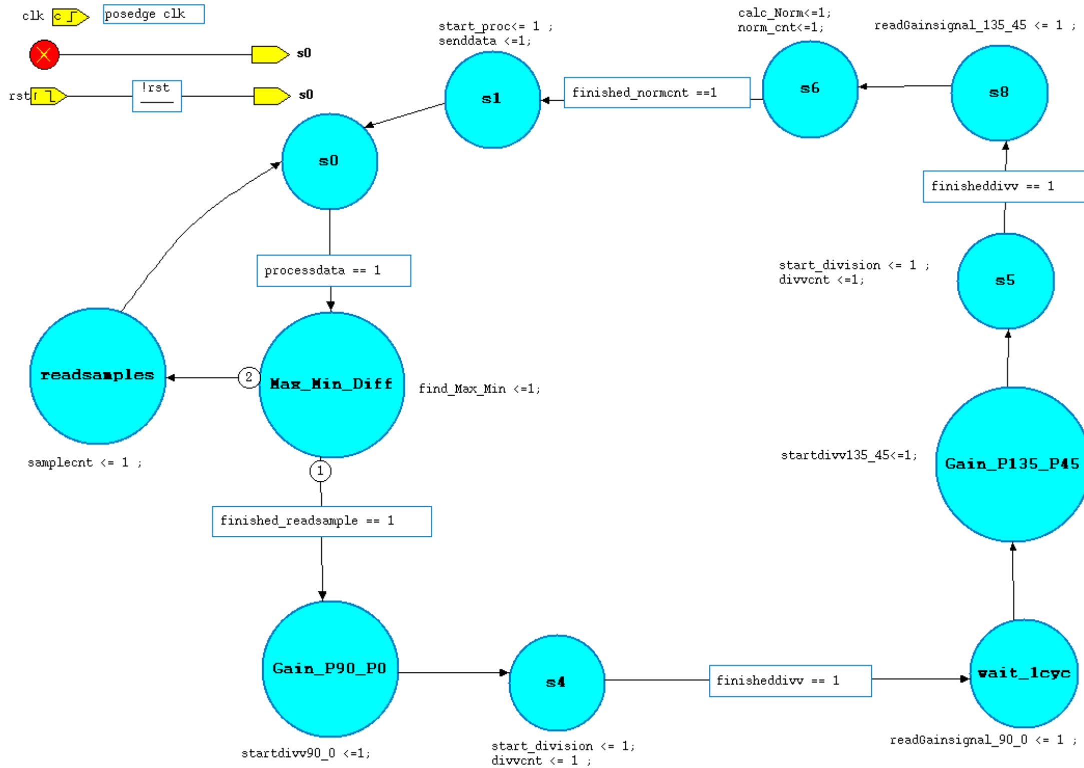


Figure 7.4 The state diagram of the “fsm_preprocessing” module

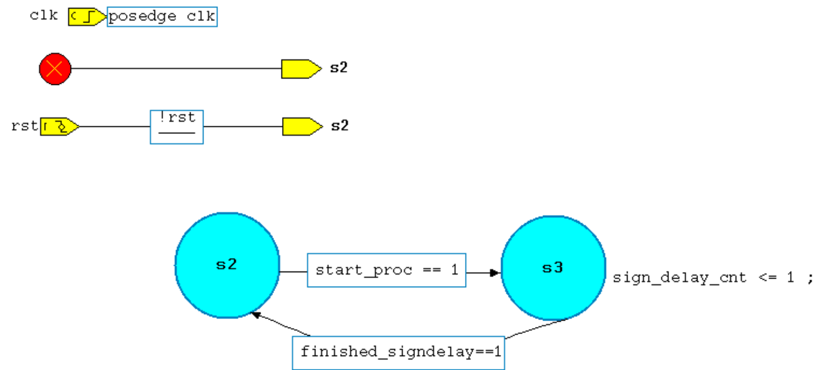


Figure 7.5 The state diagram of the “fsm_preprocessing” module

In the state machine Figure 7.4, the states “s0”, “Max_Min_Diff” and “readsamples” are used for sampling. When the signal “processdata” is equal to one, the state machine changes from the initial state “s0” to the “Max_Min_Diff” state. In this state, the signal “find_Max_Min” is set to one to find the maximum and minimum values. These values are formed by reading out the input signals of all four ADC channels and by calculating the differences “DatenP90-DatenP0” and “DatenP135- DatenP45”, which determine the gain factor coefficients. As long as the signal “finished_readsample” is not equal to one, the state machine changes to the “readsamples” state with the next rising clock edge, and the signal “samplecnt” is set to equal one, Thus the “cnt_preprocessing” module counts the number of samples. With the next rising clock edge, the state-machine changes again to the state “s0” and waits for the arrival of a new sample.

If the signal “finished_readsample” is equal to one, then the process of collecting the samples is completed, and the process for calculating the gain value for the normalization factor is started by a transition to the next state “Gain_P90_P0”. Since divisions of fixed-point numbers are very hardware-intensive, gain coefficients are calculated one after the other. In this state, the signal “startdivv90_0” is set to one to enable the “calc_preprocessing” module and to forward the difference between the maximum and the minimum of the

“DatenP90-DatenP0” sampled sensor signals to the CORDIC division block. Since the “divvend_preprocessing” module takes 15 clock cycles to calculate the gain factor, it is required to count clock cycles. Thus, in the “s4” state “start_division” and “divvcnt” signals are set to one. The “start_division” signal is used to enable the “divvend_preprocessing” module calculates the gain factor and the “divvcnt” signal enables the “cnt_preprocessing” module to count the number of the rising clock edges. In turn the counter sets the signal “finisheddivv” equal to one with the sixteenth rising clock. If this is the case, the state machine switches with the next rising clock edge to the state “wait_1cyc” in which the signal “readGainsignal_90_0” is set to one to forward the calculated result to the next processing entity.

The gain coefficient for the input signals “DatenP90” and “DatenP0” is calculated with the following equation in the “divvend_preprocessing” module.

$$\text{Gain_value90_0} = \frac{2}{\text{MAX_daten90_0} - \text{MIN_daten90_0}} \tag{7.1}$$

The “Gain_value135_45” is calculated with the same equation by using the “MAX_daten135_45” and “MIN_daten135_45” signals instead. Since the input signal values are in 16-bit and are interpreted as fixed-point format, the “divvend_preprocessing” module is also considered a fixed-point number. The divisor is defined by the maximum magnitude of the difference signal and is transmitted as an 18-bit data word.

For the offset correction of the P90_P0 difference signal, the following equation applies:

$$\text{Offset90_0} = \frac{\text{MAX_daten90_0} + \text{MIN_daten90_0}}{2} \tag{7.2}$$

which can be realized in hardware by addition and a shift operation.

After calculating the gain value for the difference signal DatenP90-DatenP0, the state changes to the "Gain_P135_P45" state to calculate the gain value for the difference signal DatenP135-DatenP45. In sequence, the explained transition condition, as well as the operation of the explained states, are repeated another time in the states, "Gain_P135_P45", "s5" and "s8", to calculate the gain factor of the DatenP135-DatenP45 difference signal, whereby in the states the according signals are set, which are related to the DatenP135-DatenP45 difference signal. In these states the signal "startdivv135_45" is set to one to enable the "calc_preprocessing" module and to forward the difference between the maximum and the minimum of the "DatenP135-DatenP45" sensor signals to the Cordic division block. The "start_division" signal enables the "divvend_preprocessing" module to calculate the gain factor and the "divvcnt" signal enables the "cnt_preprocessing" module to count the number of the rising clock edge. The signal "finisheddivv" indicates again that the calculation of the gain factor is completed. With the next rising clock edge, the state-machine changes to the state "s8" in which the signal "readGainsignal_135_45" is set to one and the calculated result is taken over by the "calc_preprocessing" module. With the next rising clock edge, the state-machine switches to the "s6" state and the "calc_Norm" and "norm_cnt" signals are set to one. The "calc_Norm" signal enables the "calc_preprocessing" module to normalize the input signals by applying the calculated offset and gain factors. Since the process of normalization takes three clock cycles, it is required to wait until the result is ready. For this purpose, a counter is activated by "norm_cnt". If the signal "finished_normcnt" is set to one, the three clock cycles are completed and the normalized result is ready. Then the state-machine changes to the "s1" state and in this state the signals "start_proc" and "senddata" are set to one. The "start_proc" signal enables the "processing" module and the normalized values of the input signals are forwarded to the "processing" module by the "senddata" signal.

As the calculation of the arcsine function in the “processing” module takes time, a synchronization between the calculated angles and the corresponding signs in the “postprocessing” module is required and implemented by the state-machine shown in Figure 7.5.

The initial state of this state-machine is “s2”. The “start_proc” signal is set to one, when the calculated normalized values are ready and as a consequence the state-machine changes to state “s3”. In this state, the state-machine waits until the signal “finished_signdelay” is set to one, to ensure that the signs are sent to the “postprocessing” module after 15 clock cycles, which corresponds to the amount of clocks cycles that are required to calculate the arcsine function in the “processing” module.

Each new sample requires 1520 ns for readout over the ADC interface. This gives 76 clock cycles. In addition, the “preprocessing” module requires 42 additional clock cycles.

7.1.2 Simulation of the Preprocessing module

In the following simulations, only signals of the “preprocessing” module that are relevant for the operation of the sequence are shown.

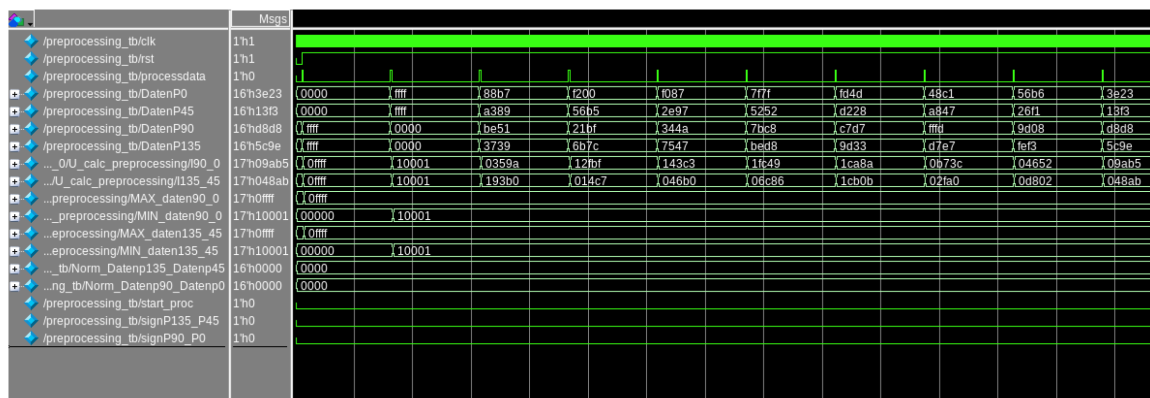


Figure 7.6 Simulation of the sampling process in the “preprocessing” module assuming 10 samples

Figure 7.6 shows a simulation of the sampling process assuming 10 samples in the “preprocessing” module. It can be seen that when the signal “rst” is set to zero all signals have the value of zero. While this signal is set the to a value of one, if the “proceessdata” signal is at a high level, the four input signals “DatenP0”, “DatenP45”, “DatenP90” and “DatenP135 are read out and the difference signals are formed to find the maximum and minimum values. During the sampling, the signals “MAX_daten90_0”, “MIN_daten90_0”, “MAX_daten135_45” and “MIN_daten135_45” store the maximum and minimum values, while the output signals receive a value of zero.

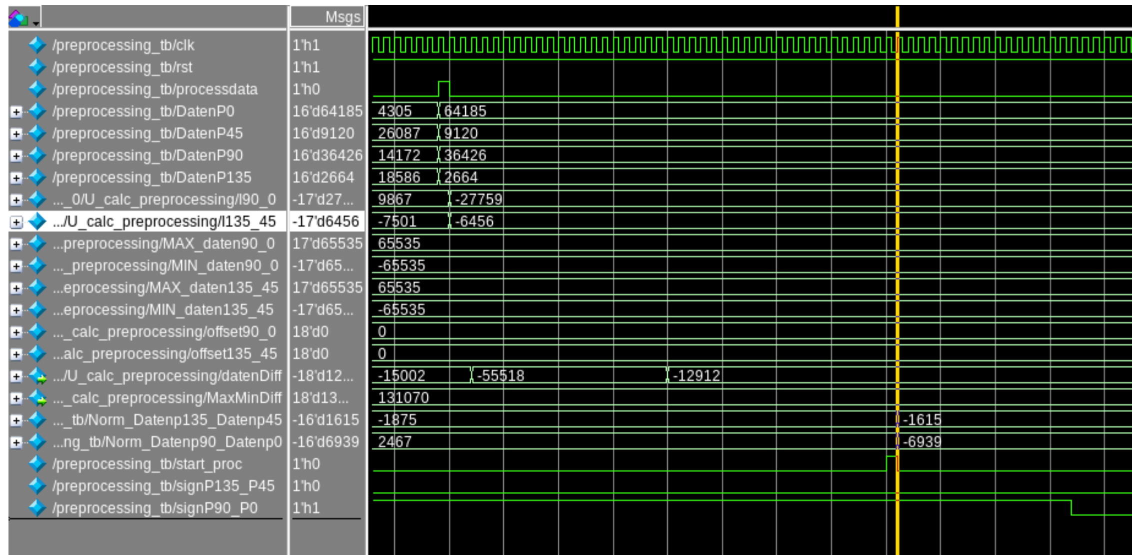


Figure 7.7 Simulation of normalized results in the “preprocessing” module

Figure 7.7 shows a simulation of the normalized values. When the “processdata” signal reaches a high level, the signals “DatenP0”, “DatenP90”, “DatenP45” and “DatenP135” are read out and stored. The difference between the signals is calculated and their value is compared with the maximum and minimum values obtained from the sampling. If each of the calculated difference values is larger or less than the maximum or minimum values, it is considered as the new maximum or minimum, respectively. Then the offset values (equation 7.2) are calculated using the maximum and minimum values and stored in the

“offset90_0” and “offset135_45” signals. The difference between the maximum and minimum values related to the appropriate sets are then saved in the “MaxMinDiff” signal for use as the denominator of the normalization factor. Furthermore, the difference between the input signals from the offset is stored in the “datenDiff” signal as the nominator of the division. Then these signals are sent to the “divvend_preprocessing” module for calculating the gain factor. It can be seen that values are processed one after the other to calculate the gain coefficients.

After completing the gain coefficients calculations, the “start_proc” signal is set to a high level, and with the edge of the next rising clock edge, the normalized values of the input signals are assigned to the output signals “Norm_Datenp90_Datenp0” and “Norm_Datenp135_Datenp45” and the “start_proc” signal takes a low level again. Furthermore, 16 clock cycles later the signals “signP90_P0" and “signP135_P45” are assigned their final values.

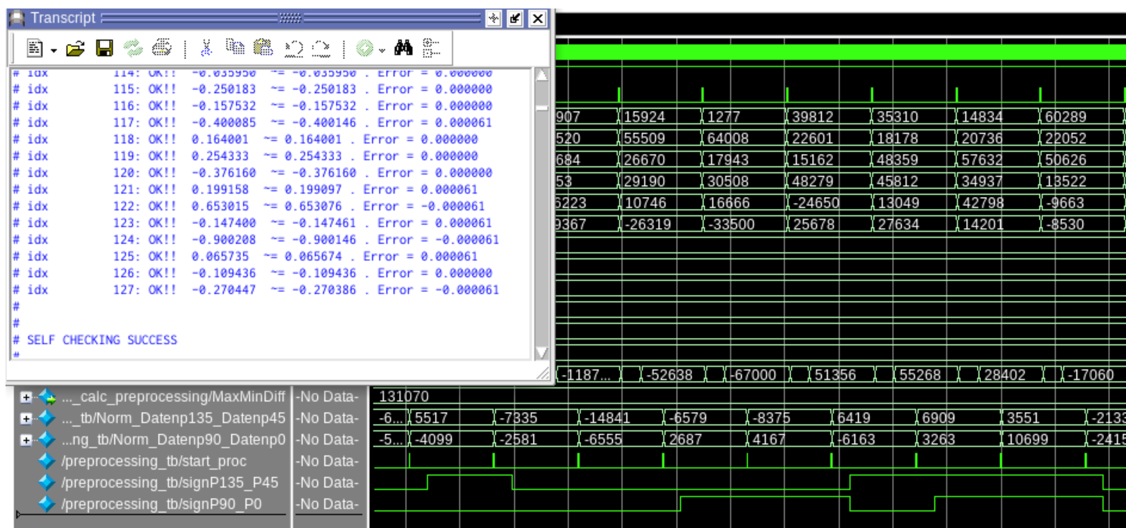


Figure 7.8 Self checking simulation of normalized results in the “preprocessing” module

A self-checking simulation is executed. The simulation results are shown in Figure 7.8. In general, it can be seen that the normalization process is calculated the correct results with high accuracy for all values.

7.2 “Processing” module

The “processing” module contains two “Arcsine” CORDIC modules to calculate the arcsine function of the normalized values, which are generated from the “preprocessing” module as described in chapter 7.1.1.

7.2.1 Functionality

The “processing” module calculates angles for normalized values using the Arcsine function.

Figure 7.9 shows the structure of the “processing” module. The module consists of four modules, where “cnt_processing” corresponds to a counter, the “arcsin_end_90_0” and “arcsin_end_135_45” modules are instances of the “arcsin_top” module and it performs the calculations of the arcsine function. In addition, the “sm_processing” module corresponds to a state machine.

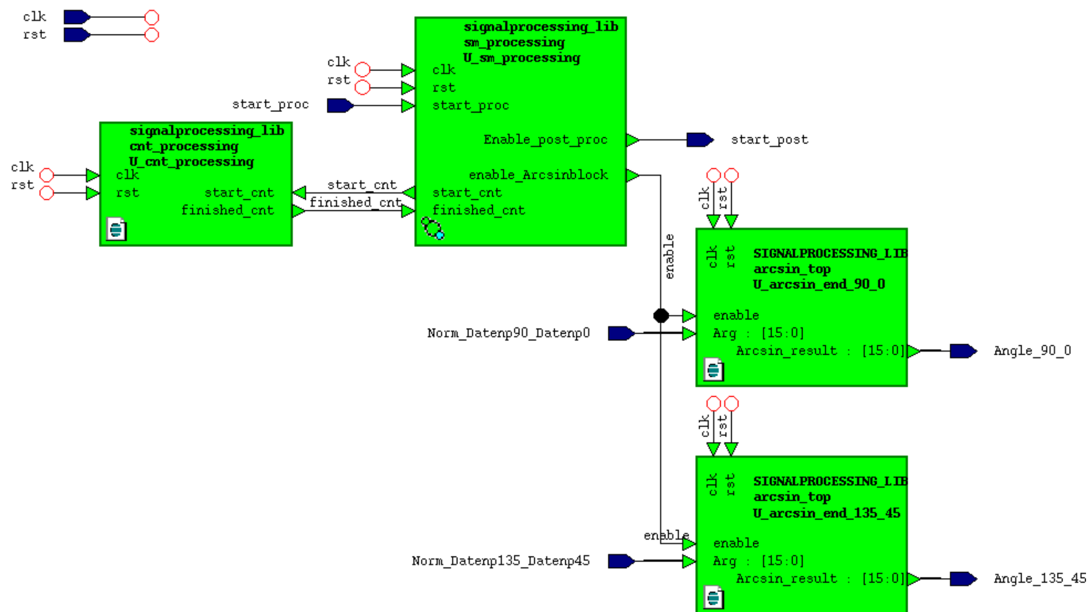


Figure 7.9 Structure of the “processing” module

The “processing” module has the following input signals:

“start_proc” enables the “processing” module to start the calculation of angles.

“Norm_Datenp135_Datenp45”, and “Norm_Datenp90_Datenp0” are 16 bits signed signals and contain the normalized data as an input argument of the arcsine function.

The “preprocessing” component has the following output signals:

“start_post” is used to enable the “postprocessing” module.

“Angle_90_0”, and “Angle_135_45” are 16 bits signed signals which hold the angle calculation results and are forwarded for further processing to the “postprocessing” module.

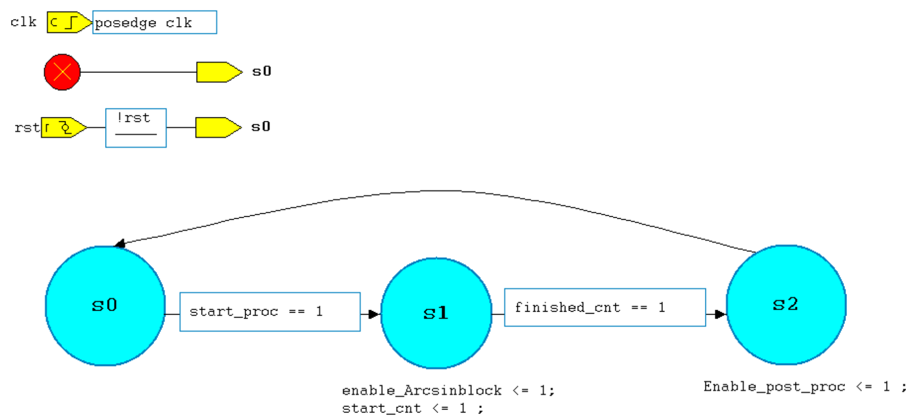


Figure 7.10 State diagram of the “sm_processing” module

Figure 7.10 shows the state diagram of the “sm_processing” module. This state machine is initialized to the state “s0” after reset. If the “start_proc” signal is equal to one, the state machine changes from state “s0” to state “s1”. In this case, the “enable_Arcsinblock” and “start_cnt” signals are set to one. The “enable_Arcsinblock” signal activates the “arcsin_end” modules, and the “start_cnt” signal enables the “cnt_processing” module to count the number of rising clock edges.

The “arcsin_end” module takes 15 clock cycles to perform the calculations. When the counter reaches 15, the signal “finished_cnt” is equal to one, and with the next rising clock edge, the state-machine changes to the state “s2”. In this state, the

"Enable_post_proc" signal takes a high level to enable the "postprocessing" module.

Finally, with the next rising clock edge, the state-machine switches to the state "s0" and waits until the "start_proc" signal is activated.

7.2.2 Simulation

In the following simulations, only signals of the "processing" module are shown.

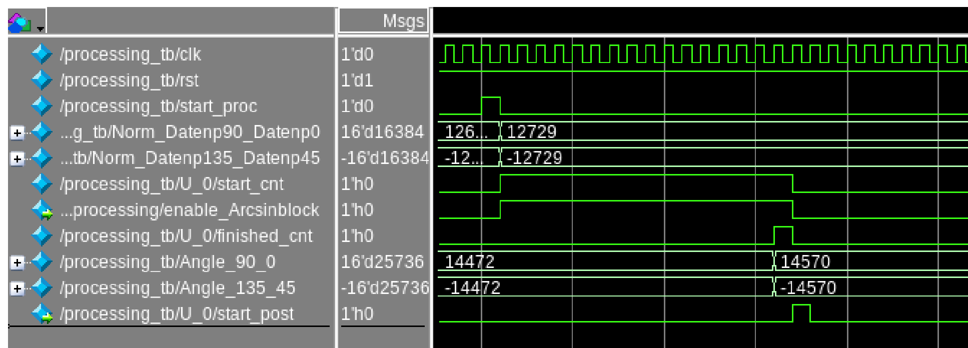


Figure 7.11 Simulation result in the "processing" module

Figure 7.11 shows a simulation of the "processing" module. It can be seen that when the "start_proc" signal is at a high level, with the next rising clock edge the input signals "Norm_Datenp90_Datenp0" and "Norm_Datenp135_Datenp45" are read-in as arguments of the arcsine function. Furthermore, the "start_cnt" and "enable_Arcsinblock" signals are set to activate the "arcsin_end" modules and the "cnt_processing" module. These signals are activated for 15 clock cycles until the calculation of the Arcsine function is completed. When the counter reaches a value of 15, the signal "finished_cnt" is set to one and the calculation results are assigned to the output signals. With the next rising clock edge, the "start_post" signal is set to one and enables the "Postprocessing" module for further processing.

Figure 7.12 shows self-checking simulation results of all possible input signals in the "Processing" module.

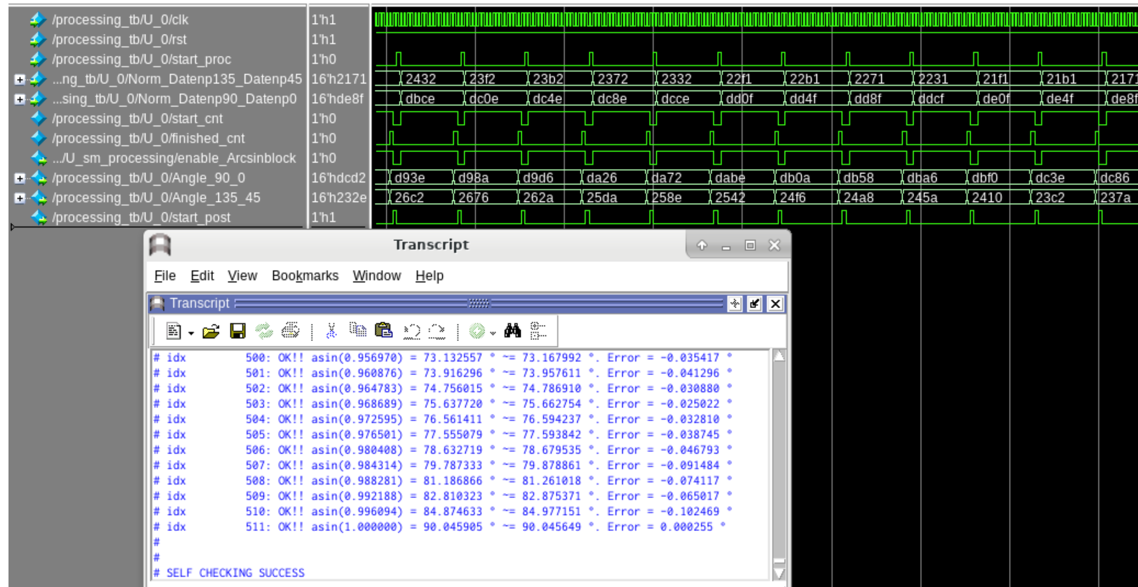


Figure 7.12 Simulation results of all possible input signals in the “Processing” module

In general, the simulations show that the design works as desired with high accuracy for all values, while the maximum deviation between the calculated results and the expected values is less than 0.1 degree.

7.3 “Postprocessing” module

The “postprocessing” module performs the angle mapping and implements the case discrimination for the angle calculation described in chapter 3. The “postprocessing” module receives the angles of the “processing” module and the corresponding signs from the “preprocessing” module. With these signs, the module checks to which range the angles have to be mapped.

7.3.1 Functionality

The “postprocessing” module generates the final results for the rotation angle extracted from the POLDI sensor signals. Figure 7.13 shows the structure of the “postprocessing” module. The module consists of three submodules, where “cnt_postprocessing” corresponds to a counter, the “sr_post_processing” module calculates the angles and the “sm_postprocessing” module corresponds to a state machine.

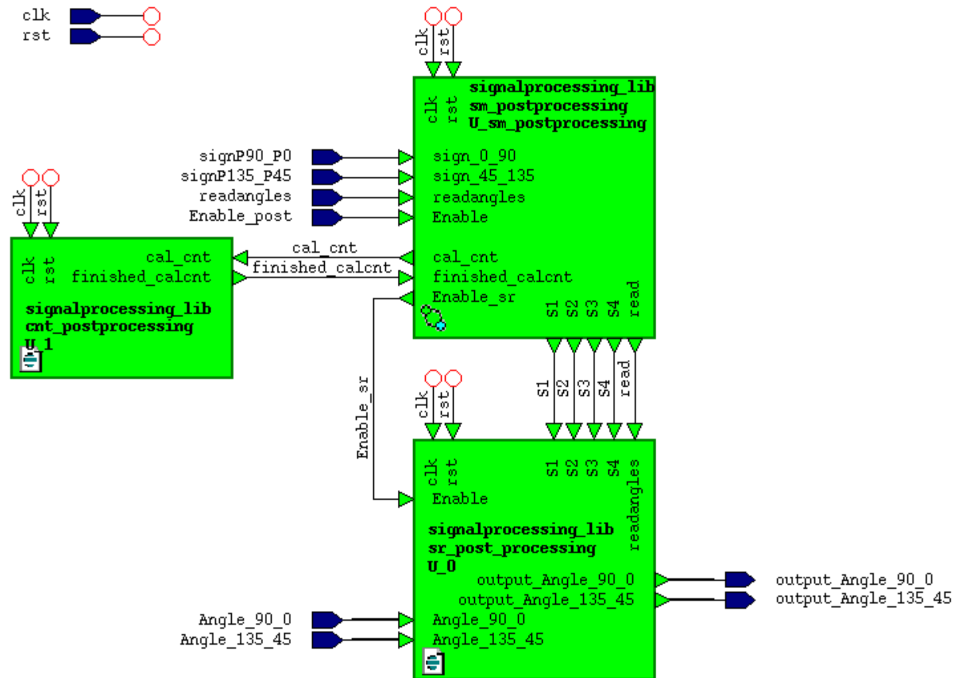


Figure 7.13 Structure of the "postprocessing" module

The "postprocessing" module has the following input signals:

"Enable_post" enables angle range mapping in the postprocessing module.

"readangles" indicates that the sign signals are ready to be read, and also enables the "sr_post_processing" module to read-in the input angles.

"signP90_P0" and "signP135_P45" are sign signals calculated by the "preprocessing" module.

"Angle_90_0" and "Angle_135_45" are 16 bits signed signals and contain the angle results of the "processing" module.

The "postprocessing" component has the following output signals:

"output_Angle_90_0" and "output_Angle_135_45" are 16 bits unsigned signals and contain the calculated rotation angles.

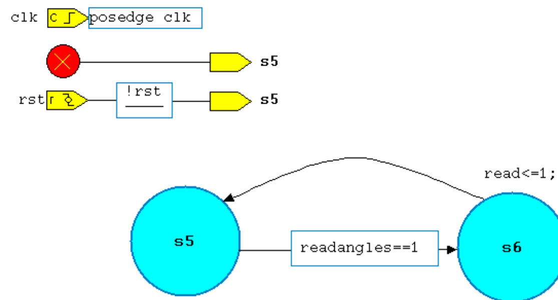


Figure 7.14 State diagram of the “sm_postprocessing” module

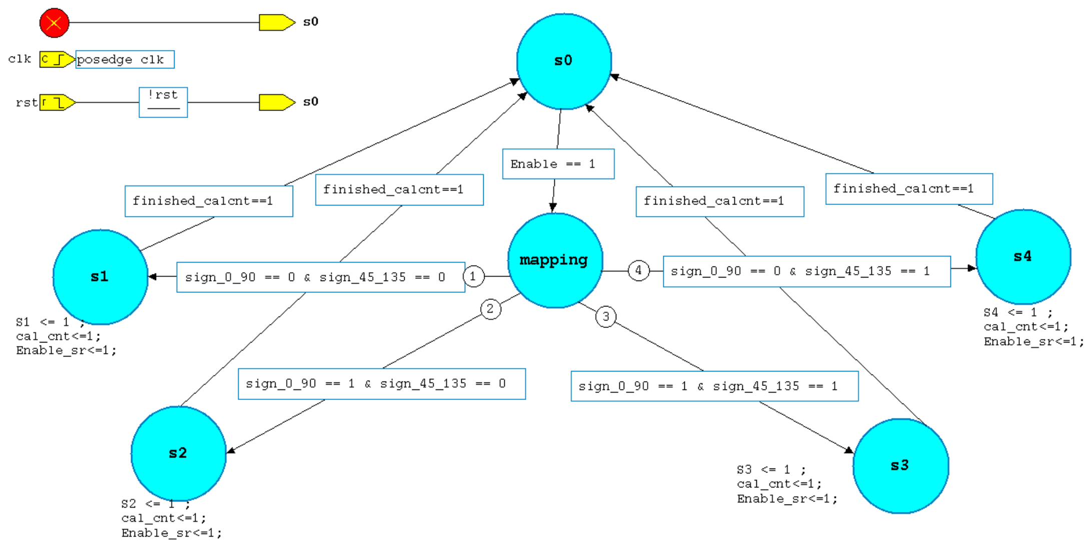


Figure 7.15 State diagram of the “sm_postprocessing” module

This module contains two state machines that are executed concurrently. Figure 7.14 and Figure 7.15 show the state diagrams of the “sm_postprocessing” module.

The state machine in Figure 7.14 is used to activate the “sr_post_processing” module for the read-out of the output angles. These angles to the “Datensendung”, which is explained in the next chapter for transmission to the PC. Meanwhile the state machine in Figure 7.15 controls the angle mapping of the input signals.

The state-machine shown in Figure 7.14 is initialized in the state “s5” after reset. When the signal “readangles” is equal to one, the state machine changes from the

state “s5” to the state “s6”. In this state, the signal “read” is set to one to read-out the output angle values. The state-machine changes to the state “s5” and waits for the “readangles” signal to reactivate.

Figure 7.15 shows the state diagram of the “sm_postprocessing” module for the angle range mapping. This state-machine is initialized in the state “s0” after reset. If the “Enable” signal is equal to one, the state machine changes to the “mapping” state. In this state the signals “signP90_P0” and “signP135_P45” are investigated. As explained in chapter 2 the angle can be located in four different ranges (I, II, III, IV) These ranges are represented in the state-machine by four states (s1, s2, s3, s4). If both sign signals have a value of zero, the next state is “s1” and the signal “S1” are set to one. If the signal “signP90_P0” is one and the signal “signP135_P45” is zero, the state-machine changes to state “s2” and the signal “S2” is driven to high level. In the case that both sign signals have a value of one, the next state is “s3” and the signal “S3” is activated, and when the signal “signP90_P0” is zero and the signal “signP135_P45” is one, the next state is “s4” and the signal “S4” is set to one.

In all four states, the “Enable_sr” and “cal_cnt” signals are set to one. The “Enable_sr” signal activates the “sr_post_processing” module to calculate the angles according to the chosen range and the “cal_cnt” signal enables the “cnt_postprocessing” module to count the rising clock edges. As the calculation of angles takes 3 clock cycles, the signal “finished_calcnt” is set to one when the counter reaches a value of 3, and with the next rising clock edge, the state changes to the state “s0”.

7.3.2 Simulation

In the following simulations, only relevant signals of the “postprocessing” are shown.

Figure 7.16 shows the simulation result of the "postprocessing" module by setting the input signals "Angle_90_0" = 14836 and "Angle_135_45" = -714 and investigating the result of the rotation angle calculation for all possible combinations of the sign signals.

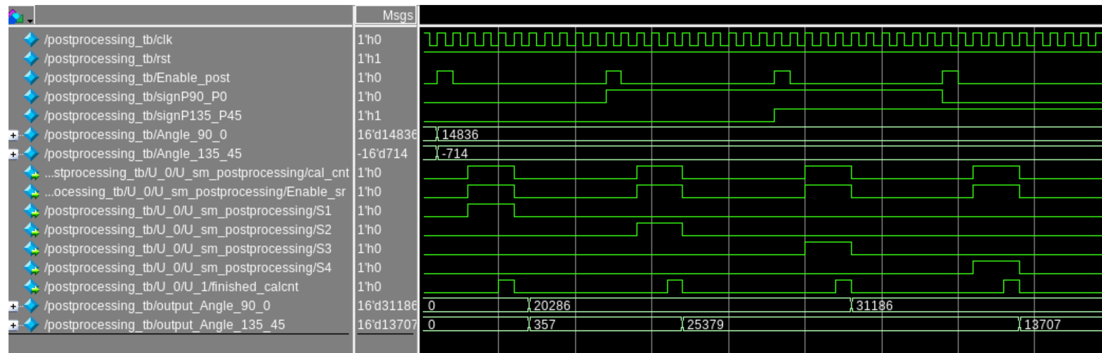


Figure 7.16 Simulation result in the "postprocessing" module

When the "Enable_post" signal is at a high level, the input signals "Angle_90_0" and "Angle_135_45" are read in. With the next rising clock edge, the "signP90_P0" and "signP135_P45" signals are read-in to determine the correct range of the angles. In the first iteration both sign signals are set to zero. Therefore, the next state is "s1" and the signals "S1", "Enable_sr" and "cal_cnt" are set to one and the angles are calculated in the next step. As the angle mapping process takes 3 clock cycles, the "cal_cnt" signal is activated for 3 clock cycle until the calculations is completed. Then the "finished_calcnt" signal is set to one and in the next rising clock edge the calculated results are assigned to the output signals "output_Angle_90_0" and "output_Angle_135_45". This process is repeated for the other three combinations of the sign signals, and in turn instead of the "S1" signal, signals, "S2", "S3" and "S4" are set to high respectively.

The individual results of the simulation show that the calculation is performed correctly. In addition, to evaluate the correctness of the design for different input angles and the accuracy of the algorithm, further simulations are performed a self-

checking testbench for all possible input values of the "postprocessing" module. The simulation results are summarized in Figure 7.17.

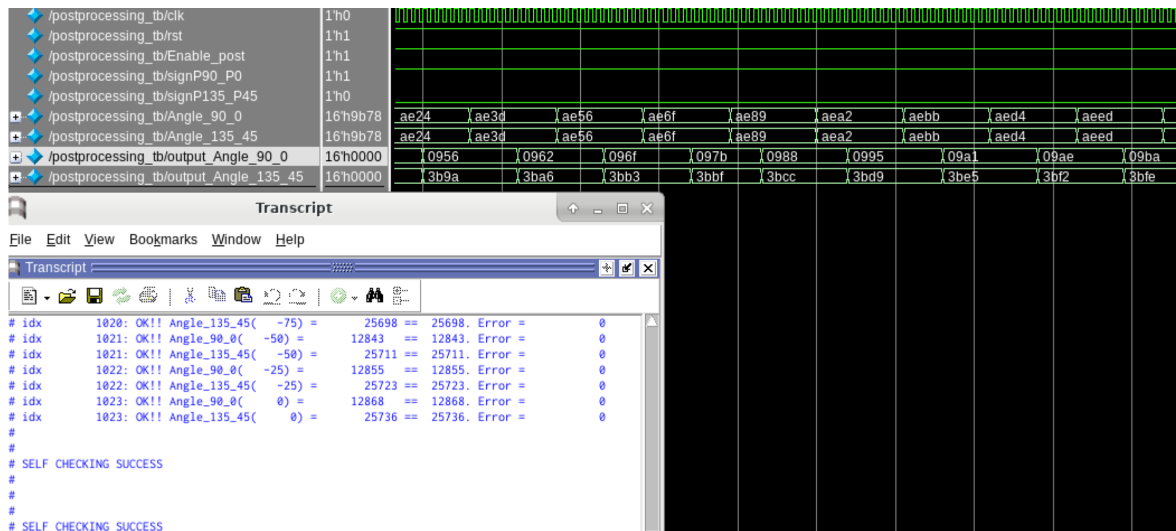


Figure 7.17 Simulation results for different input values in the "postprocessing" module

As can be seen from Figure 7.17 no deviation of the expected results is detected.

In general, it is observed that the "Postprocessing" module works with high accuracy for all values, and the result is achieved without error for both output angles. This module takes 6 clock cycles to prepare the final rotation angle results in total.

7.4 "sm_signalprocessing" module

7.4.1 Functionality

This module controls the "signalprocessing" module and manages the communication between the "signalprocessing" module and the "Datensendung" and "AD7980" modules.

It contains two state machines that are executed concurrently. Figure 7.18 and Figure 7.19 show the state diagrams of the "sm_signalprocessing" module.

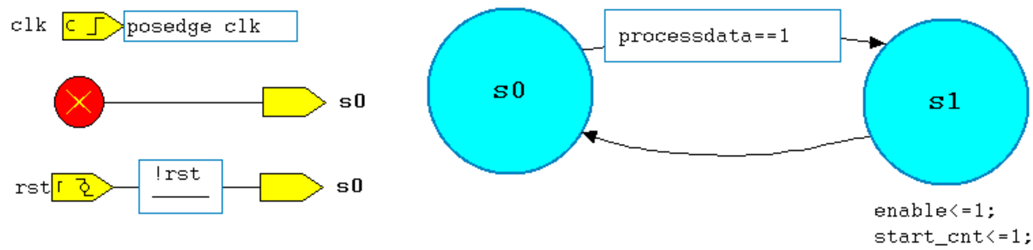


Figure 7.18 State diagram 1 of the “sm_signalprocessing” module

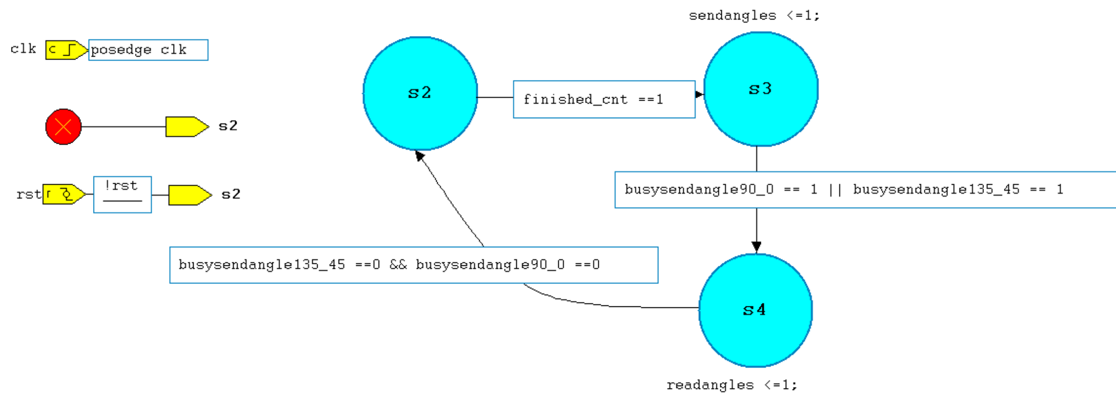


Figure 7.19 State diagram 2 of the “sm_signalprocessing” module

The state-machine in Figure 7.18 is used in the “preprocessing” module to control the angle calculation process. This state machine is initialized in the state “s0” after reset. When the signal “processdata” is set to one, the state-machine switches to the state “s1”. In this state, the “enable” and “start_cnt” signals are set to one. The “enable” signal triggers the “preprocessing” module to start the process of angle calculation, while the “start_cnt” signal enables the “cnt_signalprocessing” module to count the number of rising clock edges.

Figure 7.19 shows the hierarchical state diagram 2 of the “sm_signalprocessing” module. This module controls the angle handover process from the “signalprocessing” to the “Datensendung” modules. When the “finished_cnt” signal is equal to one, the angle calculation process is completed and the angle values are ready to be transmitted to the PC via the “Datensendung” module.

"s2" is the initial state. If the signal "finished_cnt" is equal to one, the state-machine changes to the state "s3". In this state, the "sendangles" signal is set to one to inform the "Datensendung" module that the angles have been calculated and are ready to be sent to the PC. In this state the state-mchine waits until the "busysendangle90_0" or the "busysendangle135_45" signal is high. In this case, the state switches to the state "s4" and the "readangle" signal is set to one. The "readangle" signal is connected to the "postprocessing" module, which notifies the module to assert the angle values at the output ports at this time. The "busysendangle90_0" and "busysendangle135_45" signal whether the component is busy sending the angle data. After the corresponding angle has been sent, it is set to zero again. If both signals "busysendangle90_0" and "busysendangle135_45" have a value of zero, the state-machine changes with the next rising clock edge, to the state "s2".

The "sm_signalprocessing" module has the following input signals:

"processdata" indicates availability of new signals.

"busysendangle90_0" and **"busysendangle135_45"** indicate whether the module is busy sending angle data.

The "sm_signalprocessing" module has the following output signals:

"sendangles" informs the "Datensendung" module that the angle data has been calculated and is ready to be sent to the PC.

7.5 "signalprocessing" simulation

In the following simulations, only relevant signals of the "signalprocessing" module are shown.

Figure 7.20 shows a simulation of the "signalprocessing" module for the input signals "DatenP0" =1324, "DatenP45" =65535, "DatenP90" =53417 an

"DatenP135" = 32403. This simulation is considered with maximum and minimum values of 65535 and -65535, respectively.

Figure 7.20 Simulation in the "Processing" module

It can be seen that when the "processdata" signal is at a high level, the input signals "DatenP0", "DatenP45", "DatenP90" and "DatenP135" are read-in and the "signalprocessing" module starts to calculate the angles. The individual calculated result of each module can be seen in this simulation. In addition, the calculation is performed correctly, and results are as expected.

A self-checking simulation for investigating the results for all possible input values has been executed. The simulation results are shown in Figure 7.21.

Figure 7.21 Simulation results for different input values in the "postprocessing" module in a self-checking testbench

In general, the simulations show that the design calculates the correct results with high accuracy for all values, while the maximum deviation between the calculated results and the expected values is less than 0.1 degree.

7.6 Summary

In this chapter, the architecture and the Verilog source code for the angle calculation are introduced. The ADC interface provides values of the sampled POLDI signals to the “preprocessing” module every 1520 ns. At the beginning of the operation, the “preprocessing” module first determines the required gain and offset correction values, and then calculates the normalized values. These values are sent to the “Processing” module, which calculates the required arcsine values from the “arcsin_top” module and forwards them to the “Postprocessing” module. The “Postprocessing” module determines the correct angle region and calculates the rotation angles.

Calculating the output angles in the “signalprocessing” module requires a total amount of 64 clock cycles. The “Preprocessing” module requires 42 clock cycles, the calculation in the “Processing” module takes 16 clock cycles in the “Processing” module, while the “Postprocessing” module requires 6 clock cycles to calculate the expected values.

8 Integration of the “signalprocessing” module into the POLDI readout system

When the POLDI sensor is irradiated with linearly polarized light, each diode supplies a different photocurrent whose current intensity depends on the angle between the polarization plane of the incident light and the orientation of the polarization filter. Therefore, each photocurrent flows into a separate Trans-Impedance Amplifier (TIA) and different voltages are applied to separate ADCs. These digitized signals are made available to the digital signal processing module called “signalprocessing” which is implemented on an FPGA and can extract the angle information out of the incident light sent to the PC using the “Datensendung” module. For this purpose, the “signalprocessing” module, which is described in Chapter 7, is integrated into the POLDI sensor readout and control system which has been implemented as a VHDL design [9].

This chapter presents the integration of the “signalprocessing” module in the POLDI sensor readout and control system.

8.1 Poldi readout and control system

The design consists of eight interconnected main modules. Each module of the design contains an asynchronous reset asserted by the signal “rst” and a clock signal called “clk”. The “clk” signal corresponds to the system clock of 50 MHz and the “rst” signal ensures that the modules are reset and that all state-machines return to their initial states.

Figure 8.1 shows the interconnection of the individual modules of the POLDI sensor readout and control system.

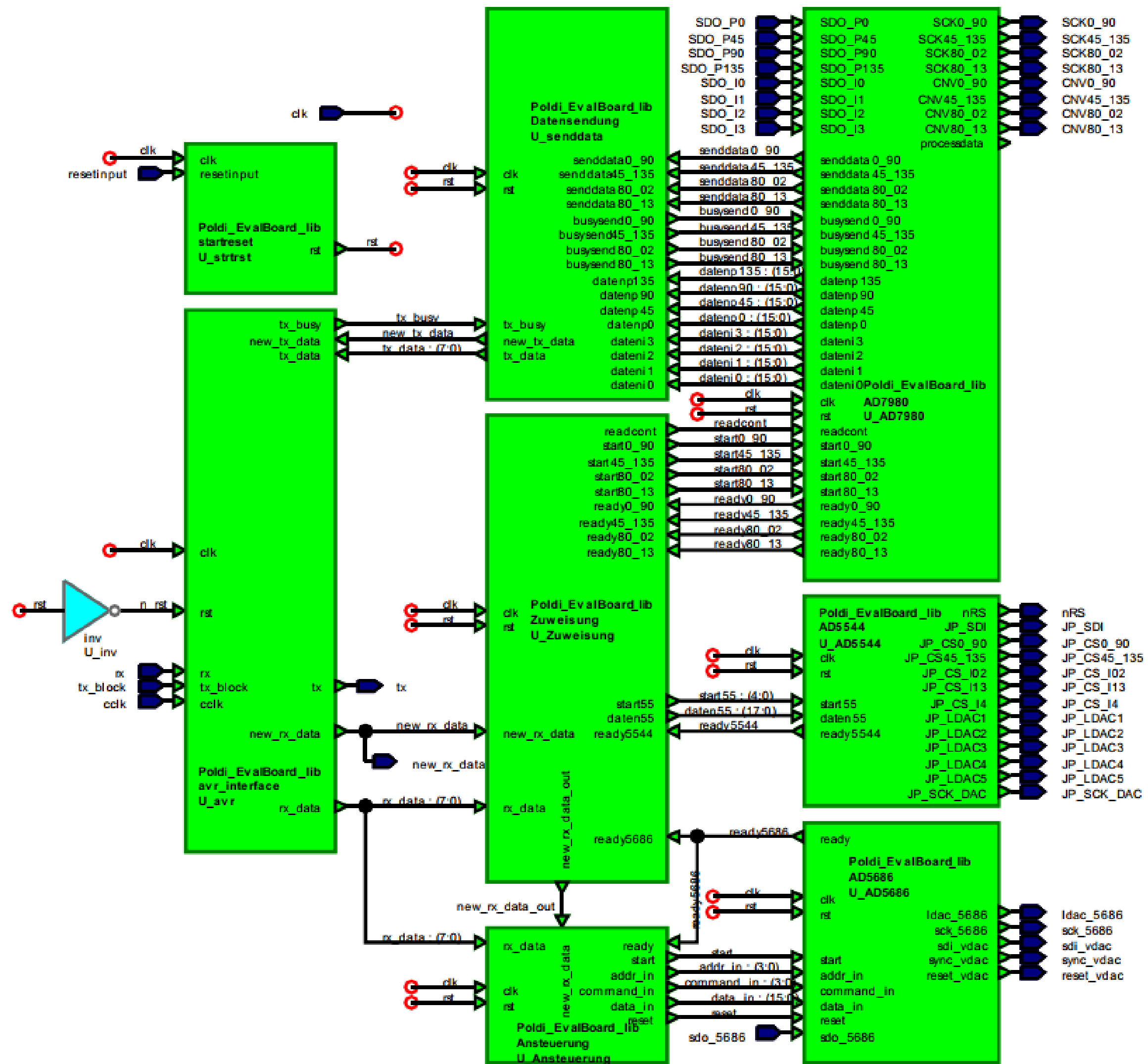


Figure 8.1 Structure of the VHDL Design [9]

The ports of the right three components are connected to pins of the DACs and ADCs available on the Cmod A7 board described in chapter 5.5. These ports are given the same name as the nets connected to the DACs and ADCs on the Cmod A7 board.

The main function of the individual modules is briefly explained below. The operation of these modules is explained in detail in [9].

- **“startreset”** ensures that the modules are reset after a configuration of the FPGA.

- **“avr_interface”** is provided by the manufacturer of the Mojoboard of the company “Embedded Micro” and manages the communication between the microcontroller and the FPGA via an UART interface.
- **“Zuweisung”** implements a communication protocol and provides and receives data from and to the “avr_interface”. It establishes synchronization between the application software and the FPGA logic and forwards control signals and data to the components responsible for communication with the DACs and ADCs on the boards described in chapter 5.
- **“Ansteuerung”** and **“AD5686”** are responsible for the control of the DAC of type “AD5686”.
- **“AD5544”** controls the five separate “AD5544” DACs.
- **“AD7980”** controls the data readout of eight different ADCs and stores the read data. For this purpose, it contains two modes. In one mode, a single readout is performed and the readout data is then sent to the PC using the “Datensendung” module, while in the alternative mode a continuous readout of the ADCs is performed.
- **“Datensendung”** with this module the data read out from the “AD7980” ADCs can be send to the PC using the “Datensendung” and “avr_interface” modules.

8.2 Integration of the Signal Processing Unit

The angle calculation is based on the digital signals digitized by the Analog-to-Digital Converters (ADCs). The “AD7980” controls the conversion process of all ADCs and reads-in the digitized data. This module is operated in the continuous readout mode for the signal processing in the FPGA and the extracted rotation angles are also transmitted continuously to the PC. The data read in from the

ADCs are assigned to the output signals “datenp0”, “datenp45”, “datenp90” and “datenp135”. After a conversion and readout cycle is completed, the “processdata” signal is set to one to indicate that the system is ready for further signal procssing and in addition the next conversion and readout cycle of the ADCs is started immediately.

The output signals “datenp0”, “datenp45”, “datenp90” and “datenp135” as well as “processdata” of the “AD7980” module are connected to the “DatenP0”, “DatenP45”, “DatenP90”, “DatenP135” and “processdata” ports of the “signalprocessing” module respectively. The signal processing is performed in the “signalprocessing” module, and the angles are calculated. To send the calculated angles to the PC, a connection between the “signalprocessing” and “Datensendung” modules is necessary. For this purpose, the “Datensendung” module must be modified. In the following, the “Datensendung” module and the applied modifcations are described.

8.2.1 “Datensendung” module

8.2.1.1 Functionality

With this module, data can be prepared for transmission data can be prepared for transmission to the PC via the module “avr_interface”. The calculated angles have to be fed into this module.

Figure 8.2 shows the structure of the “Datensendung” module. This module contains the “sm_Datensendung” module, which corresponds to a state machine, and the “ff_Datensendung” module.

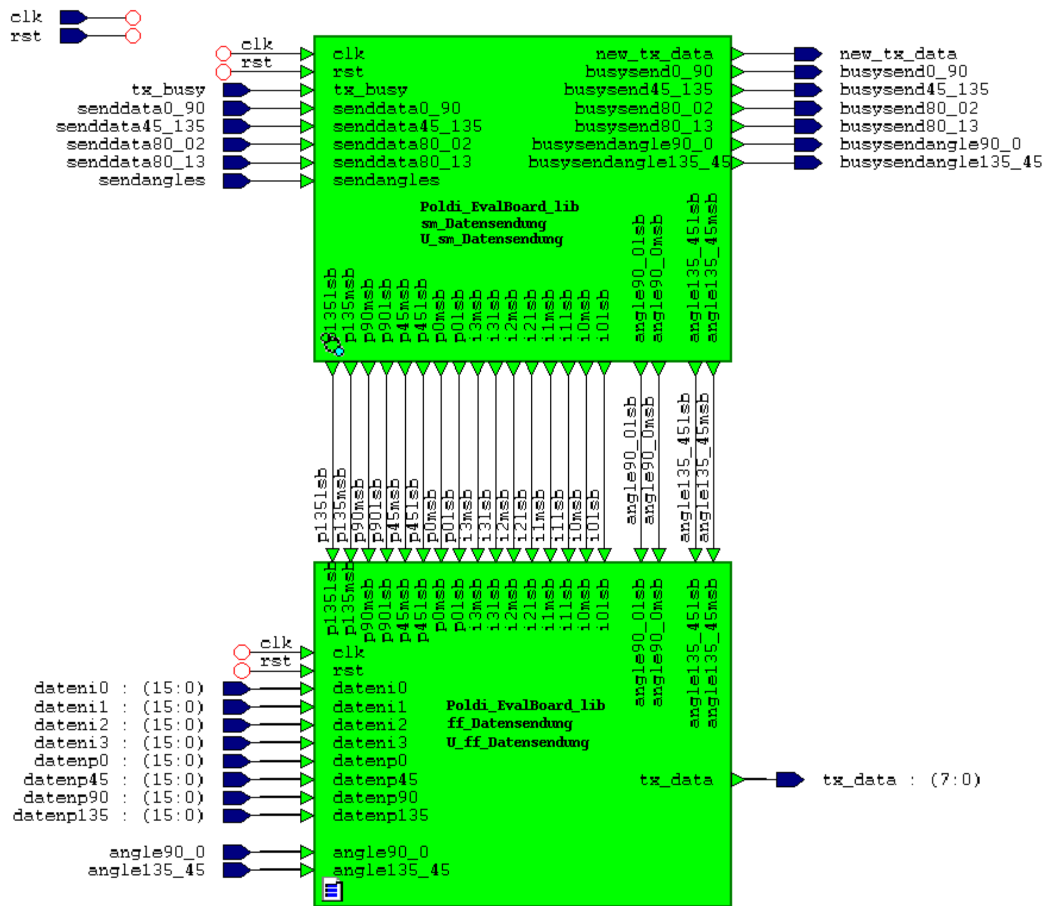


Figure 8.2 Structure of the “Datensendung” module

The “Datensendung” module has the following input signals:

- **“sendangles”** indicates whether data is to be sent to the PC from the “signalprocessing” module. The “sendangles” signal should be set to zero when both signals “busysendangle90_0”, “busysendangle135_45” are high.

- **“angle90_0”** and **“angle135_45”** contain the data of the calculated angles, which are provided by the “postprocessing” module.

Output signals of the “Datensendung” module:

- **“busysend_Angle90_0”** and **“busysend_Angle135_45”** indicate whether the module is busy sending angle data provided form the “signalprocessing” module.

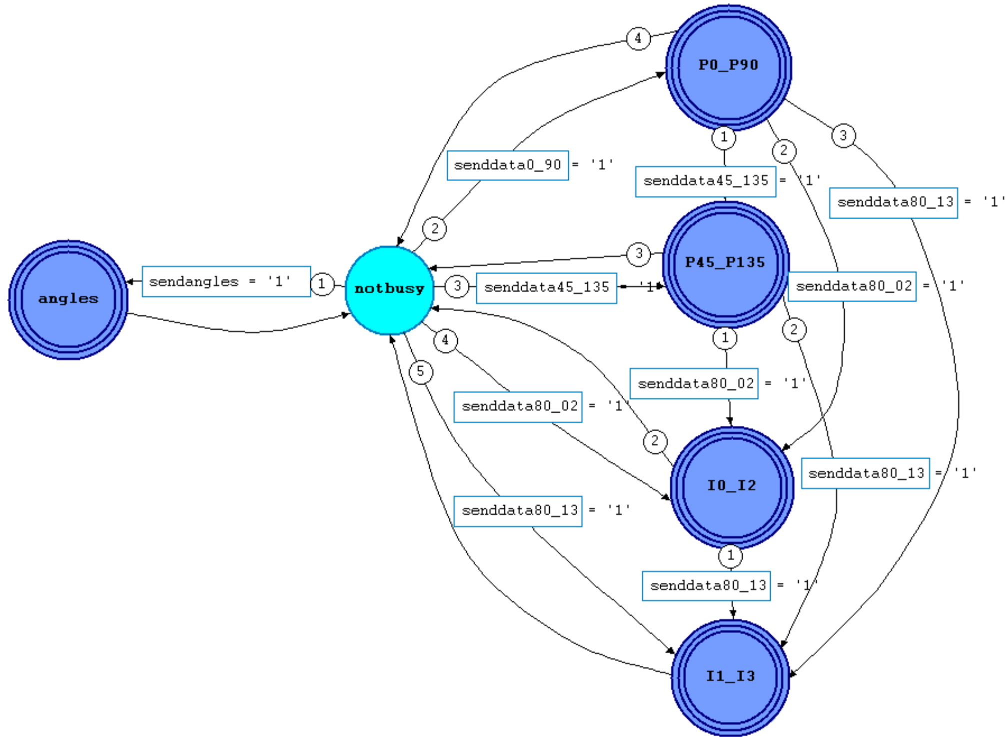


Figure 8.3 Hierachical state-diagram of the state-machine in the module “sm_Datensendung”

Figure 8.3 shows the state-machine of the “sm_Datensendung” module. Each of the states control the data transmission to the PCs. The states “P0_P90”, P45_P135”, “I0_I2” and “I1_I3”, control the data tranmissiom readout from the ADCs to the PC, while the “angles” state controls the transmission of the angle by “signalprocessing” module to the PC.

From the initial state “notbusy” the state is changed to “angles” with the next rising clock edge, if the signal “sendangles” equals one. If the signal is not equal to one, the signals “senddata” corresponding to the ADCs are checked, which is explained in detail in [9].

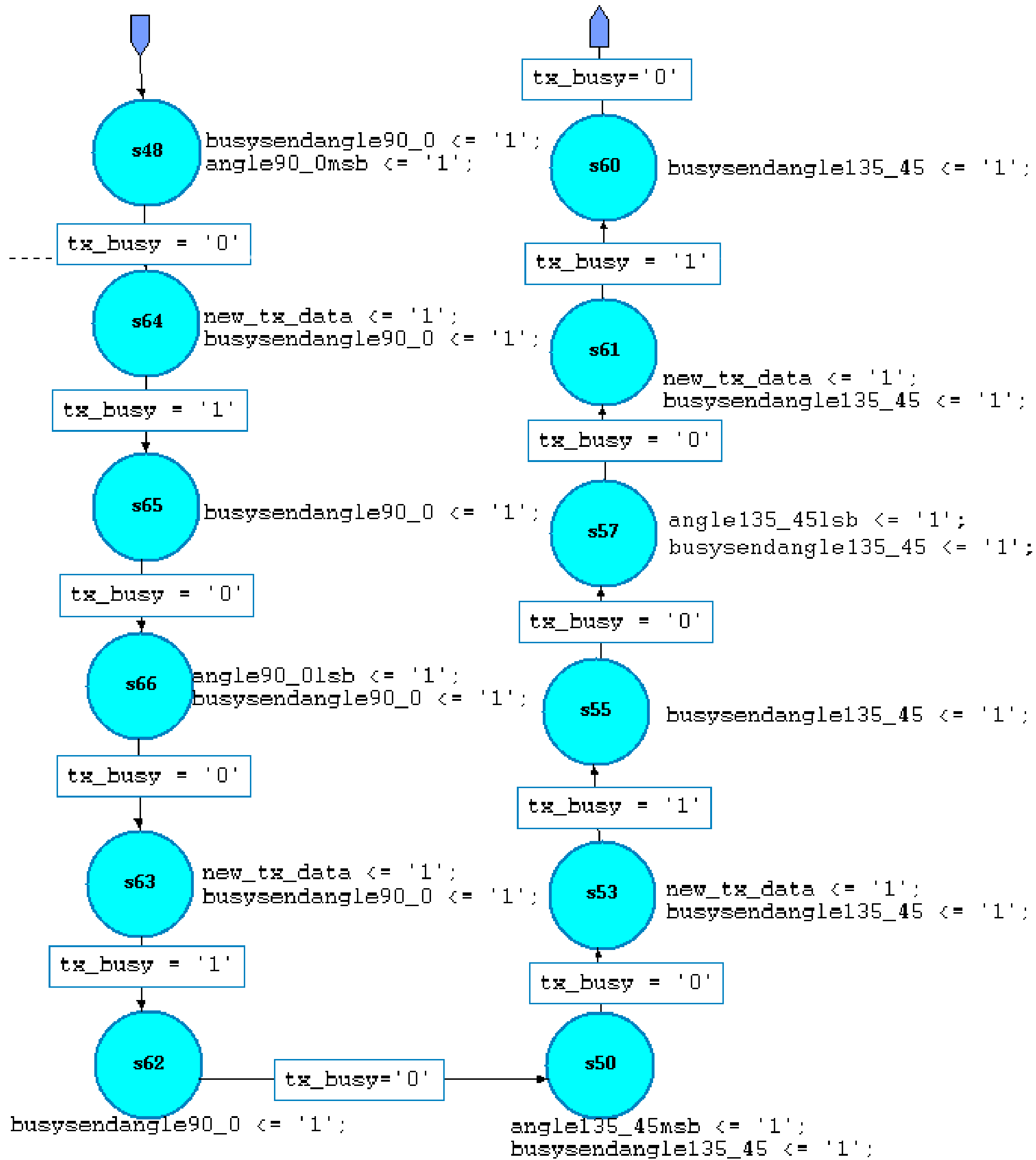


Figure 8.4 Hierarchical state “angles” of the “sm_Datensendung” module

Figure 8.4 shows the hierarchical state “angles” of the “sm_Datensendung” module. In the first state “s48”, the signals “angle90_0msb” and “busysendangle0_90” are set to one. Thus, the module “ff_Datensendung” assigns the eight MSBs of the vector “angle90_0” to the vector “tx_data” with the next rising clock edge.

When the module “avr_interface” is ready to send data over the UART interface, the state machine changes from state “s48” to state “s64” with the next rising clock edge, in which the signals “new_tx_data” and “busysendangle90_0” are set to one. Thereby “tx_data” contains the eight MSBs of the vector “angle90_0”. While the “avr_interface” module is busy sending the data, the state machine changes to the state “s65” with the next rising clock edge. From this state, if the “avr_interface” module is ready to send data, the state machine changes to the state “s66” with the next rising clock edge. In this state, the signals “angle90_0lsb” and “busysendangle90_0” are set to one. Thus, the module “ff_Datensendung” the eight LSBs of the vector “angle90_0” to the vector “tx_data” with the next rising clock edge.

The explained three state conditions are repeated two more times in the following states, whereby in the states “s50” and “s57” the signals “busysendangle135_45” and “angle135_45” are set, which ensures that the vector “tx_data” receives the appropriate data to be sent. The successor states of the state “s60"” ensures that first the eight MSBs, then the eight LSBs of the vectors” angle90_0”, then the eight MSBs, and finally the eight LSBs of the vector “angle135_45” are sent to the PC.

8.2.2 Design structure with integrated signal processing unit

Figure 8.5 shows the interconnection of the individual modules of the POLDI readout and control system after integration of the signal processing unit. It can be seen that the “signalprocessing” module is included on the design top level and is connected to the “AD7980” and “Datensendung” modules.

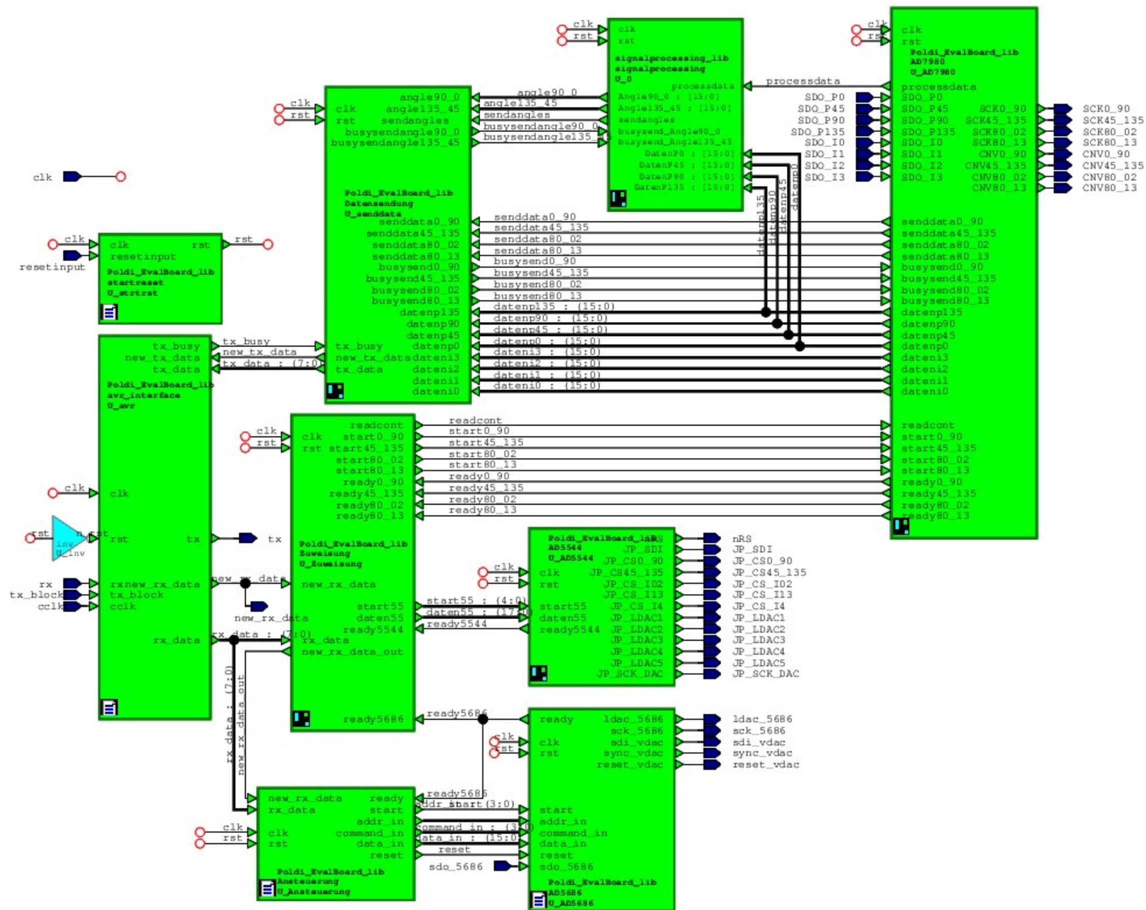


Figure 8.5 Structure of the integrated Design

After integration of the "signalprocessing" module into the POLDI readout and control system, the FPGA available on the Mojo V3 FPGA development board which has been used in a previous work [9], is not sufficient for the design implementation because of restricted logic resources. For this reason, the Cmod A7 board from Digilent with 48-pin and a Xilinx Artix 7 FPGA is used instead. The Cmod A7 board includes a single 12 MHz clock input. However, the input clock can be fed to a Mixed-Mode Clock Manager (MMCM) to generate clocks of various frequencies. The MMCM is configured via the Clocking Wizard IP core such that it generates the 50 MHz clocks required for the design.

8.2.3 Simulation

In the following simulations (Figure 8.6), only relevant signals of the Poldi readout and control system with integrated signal processing unit are shown.

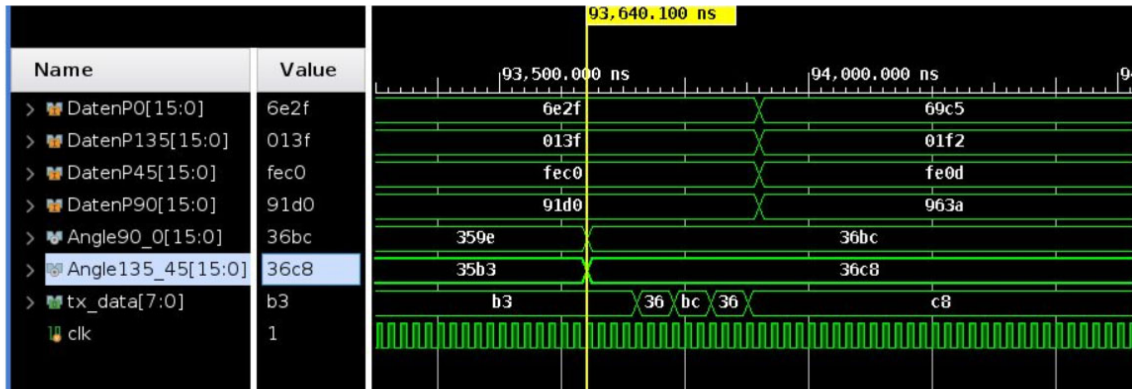


Figure 8.6 Simulation of Integrated Design

It can be seen that the angle signals calculated angles in the “signalprocessing” module are sent to the “avr_interface” module using the “tx_data” vector. First, the eight MSB then the eight LSBs of the vector “Angle90_0”, then the eight MSBs, and finally the eight LSBs of the vector “Angle135_45” are sent to the PC. The simulations show that the design works as desired.

8.3 Summary

In this chapter, the “signalprocessing” module is integrated into the POLDI sensor readout and control system. When the POLDI sensor is irradiated with linearly polarized light, each diode supplies a different photocurrent. Each photocurrent flows into a separate Transimpedance Amplifier (TIA) and different voltages are applied to separate ADCs. The ADC interface provides values of the sampled POLDI signals to the “signalprocessing” module, which calculates the data angles of the incident light and sends them to the PC using the “Datensendung” module.

9 QT widget application for sensor readout control

As a preliminary work, a graphical user interface (GUI) was designed for controlling different digital-to-analog converters (DACs) by means of the QT application. In addition, by selecting different analog-to-digital converters (ADCs), their voltages are displayed in the GUI [9]. In this project, this GUI is extended to control and adjust the polarizer and light source, as well as to read and display the intensity signals and calculated angles.

9.1 Main Window

The main window consists of five fields, each separated by lines. Figure 9.1 illustrates the Main window of the GUI.

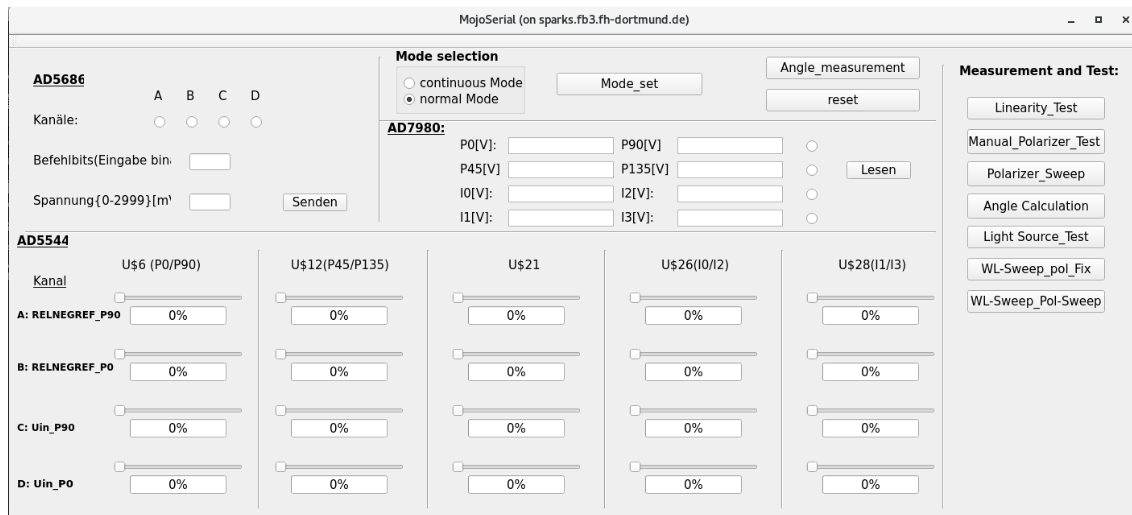


Figure 9.1 Main window of the GUI

9.1.1 AD5686 field

The field located at the top left is designated for controlling the “AD5686” type DAC. For the selection of the channels of the “AD5686” DAC, four “Radio Buttons” have been employed. Labels with the names of the DAC channels (A to D) were positioned above each individual button. Additionally, a label with the text “Kanäle” has been placed to the left of the buttons to indicate that these buttons should be used for channel selection. By deactivating the checkmarks in the “auto-exclusive” boxes in the properties of the four radio buttons, it is possible

to activate multiple buttons simultaneously. This allows the user to control several channels concurrently within the GUI.

Two input fields have been added to facilitate the entry of command bits and voltage. These fields are appropriately labeled next to them, with the text "Command Bits (Input binary):" providing clarity to the user regarding the entry of command bits in binary format, and "Voltage {0-2999} [mV]" indicating that the voltage input is measured in Millivolts and can range between 0 and 2999 mV.

To accommodate the four command bits and the requirement for the input voltage to be in Millivolts with a maximum length of four characters, the maximum allowable character length for these fields has been limited to four.

Additionally, a "Push Button" labeled as "Senden" has been created to enable the transmission of the entered data.

Upon pressing the "Senden" button, if no DAC channel is chosen or if there's incorrect input in either of the two input fields, a new window emerges, displaying an error message.

9.1.2 Mode selection field

This field is located at the center-top of the GUI and is designed for controlling the mode of the "AD7980" type ADC.

As mentioned previously, there are two modes for reading out the ADCs: normal mode and continuous mode. In the normal mode, a single readout of the selected ADCs is performed, and the readout data is then transmitted to the PC. Conversely, the continuous mode is utilized for signal processing in the FPGA and extracting rotation angles. It's important to note that all ADCs are read out in this mode.

Before reading out the ADC data, it's necessary to set it to either normal mode or continuous mode. For this purpose, two Radio Buttons for selection have been positioned, labeled with the corresponding mode names. Additionally, a Push Button labeled “Mode_Set” has been included to transmit the chosen mode, which then configures the ADCs to the specified mode.

A Push Button labeled as “reset” has been placed to allow for the reset of all configurations within the GUI. Furthermore, the “Angle Measurement” button is employed to perform angle readouts in continuous mode. A new Qt Designer form class is incorporated into the project for of this button. Figure 9.2 depicts the window that opens upon pressing the “Angle Measurement” button.

Within this window, two buttons can be found: one to initiate continuous signal readings and the other to halt the reading process. Accompanying these buttons are two LCD Number widgets, designed to display the calculated angles within the FPGA. Furthermore, a QwtPlot has been integrated to visualize the acquired angles in real-time plot.

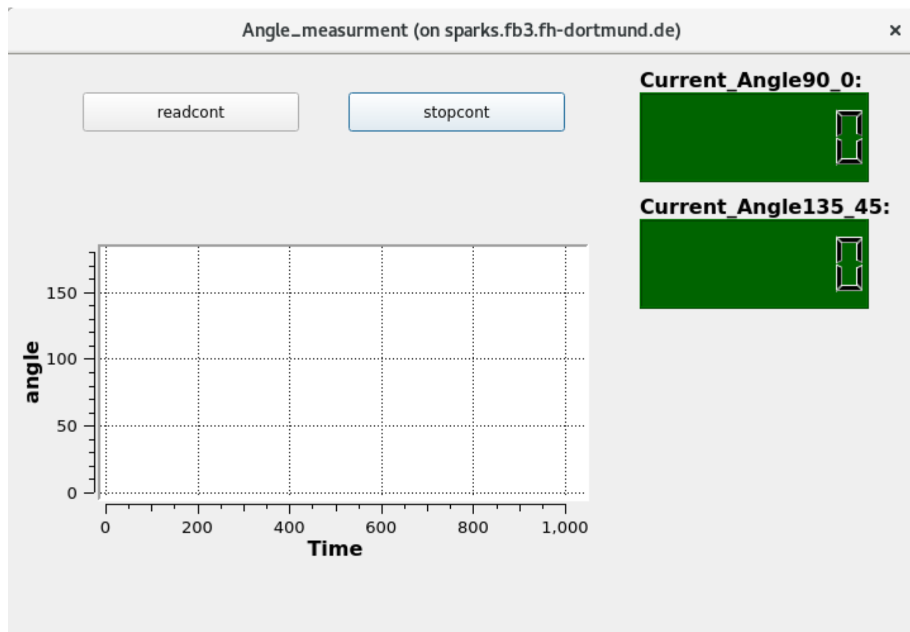


Figure 9.2 “Angle measurement” window

9.1.3 AD7980 field

This section serves the purpose of selecting ADCs to perform voltage readings and exhibit the applied voltage values. Considering that two, four, six, or eight ADCs can be simultaneously read, each row is equipped with two "Text Browser" fields to present the voltages for two ADCs, accompanied by a corresponding "Radio Button" for selection.

Adjacent to the voltage fields for the ADCs, labels indicating the ADC names along with "[V]" have been included. This arrangement ensures clarity in associating each button with its corresponding ADC, while the displayed voltage is expressed in volts. The ADC names are derived from the pins of the ADCs from which data is obtained.

Positioned to the right of this section is a Push Button labeled "Lesen" designed to initiate the reading of the chosen ADCs. Subsequently, the voltage values corresponding to the ADC readings are presented within their respective fields.

Notably, the activation of multiple buttons simultaneously is feasible. Additionally, it's necessary to set the normal mode in the "Mode selection" field before reading the voltages. This entails selecting the "normal mode" initially and pressing the "Mode_Set" button, which are the essential settings for normal readings sent to the ADCs to establish the mode.

If the "Lesen" button is pressed without selecting the mode in the "Mode_selection" field or without selecting an ADC, a corresponding window with an error message is triggered.

9.1.4 AD5544 field

The lower section within the main window of the GUI serves as the interface for configuring the five "AD5544" type DACs. This area is divided into five distinct columns, each demarcated by lines. Within each column, four sliders have been

positioned, accompanied by an underlying progress bar that visually illustrates slider adjustments.

To the left of this arrangement, a label bearing the text “Channel:” has been placed, followed by labels displaying the names of the DAC channels. At the column's uppermost part, a label is positioned, featuring the name of the respective “AD5544” DAC.

Following the DAC names, which directly impact the voltages of the ADCs, additional labels have been included, identifying the corresponding ADCs. These labels are enclosed in brackets adjacent to the DAC designations. This design provides clear insight into the connection between the sliders and the specific channels of each DAC, ultimately influencing the voltage of corresponding ADCs.

9.1.5 Measurement and Test field

This section is positioned on the right side of the main window of the GUI and is designated for the configuration, control, measurement, and testing of the polarizer and light source. This field comprises seven buttons which trigger the opening of new windows corresponding to their designated tasks. To accomplish this, a new Qt Designer form class is incorporated into the project for each of these buttons. Further details about the functionality of these buttons are elaborated upon in the subsequent sections.

9.1.5.1 “Linearity_Test” button

The “Linearity Sweep” button is utilized to present the linearity of the readout voltage measurements. This is achieved by adjusting the Gain and Offset values within the DAC settings. Furthermore, it visually demonstrates the interdependence of gain and offset. Figure 9.3 shows the “Linearity_Test” window.

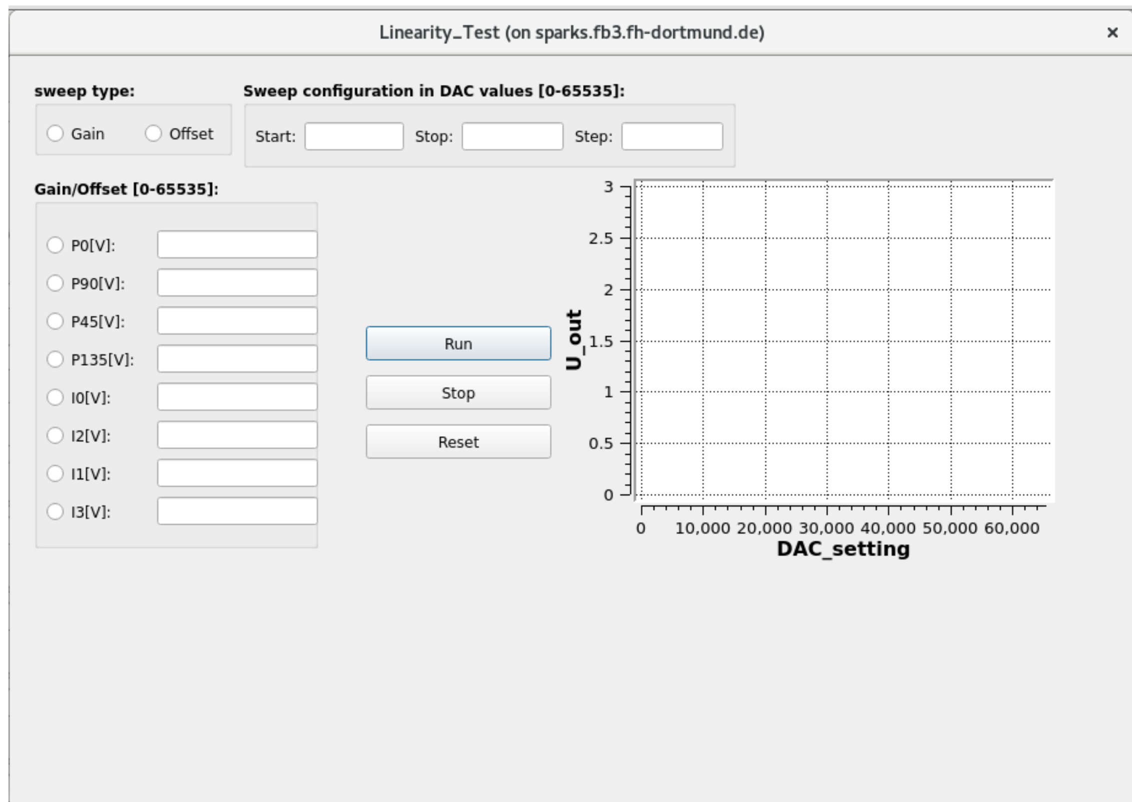


Figure 9.3 “Linearity_test” window

This window comprises three sections. The first section, named “Sweep Type,” incorporates two radio buttons labeled “Gain” and “Offset”. The selection of one of these options, such as “Gain,” permits the measurement of Gain as a relative function of Offset, or vice versa.

The second section, titled “Sweep Configuration in DAC Values [0-65535]” features three input fields designated as “Start”, “Stop” and “Steps”. These input fields serve the purpose of defining the parameters for the sweep process, including the start and stop values along with the increment steps.

The third section, denoted as “Gain/Offset [0-65535],” is employed to specify the DAC for measurement, with a range from 0 to 65535. Adjacent to each DAC name is an input field where the value for the selected channel in the “Sweep Type” section can be input. The allowable range for this input is from 0 to 65535.

Additionally, the window features three buttons: “Run” which initiates the measurement process execution; “Stop” which halts the ongoing measurement process; and “Reset” which restores the DAC configurations to their default settings. An integrated QwtPlot is also present to visually represent the ongoing measurement process.

9.1.5.2 “Manual-Polarizer_Test” button

The “Manual_Polarizer_Test” button is used for manual establishment, configuration, and voltage readout of the Polarizer. Pressing this button will open a window titled “Polarization Test”. Figure 9.4 displays the “Polarization_Test” window.

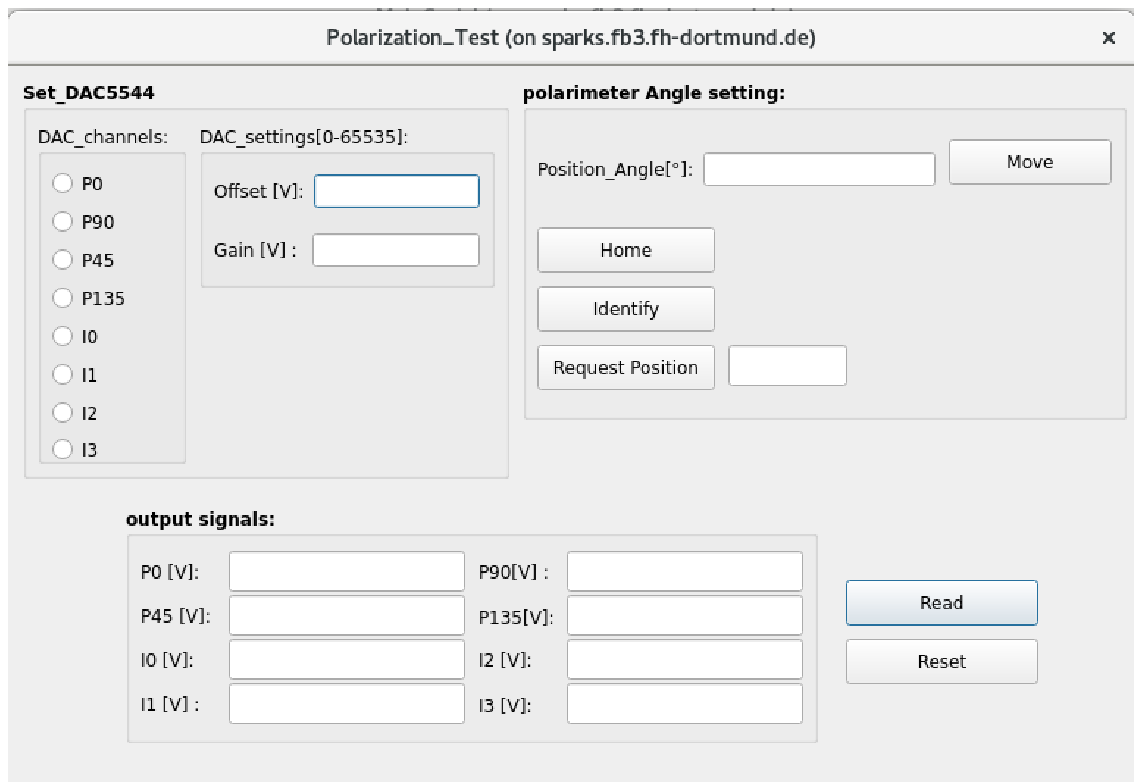


Figure 9.4 “Polarization_Test” window

The “Polarization_Test” window comprises three sections that facilitate the selection and configuration of DAC settings for the “AD5544” the setup of the polarimeter, and the visualization of the readout output signals.

Within the “Set_DAC5544” section, two subsections are present: “DAC Channels,” enabling the selection of a specific channel, and “DAC Settings [0-65535]” used to configure the Gain and Offset values for the chosen channel. Input values within the range of 0 to 65535 can be applied to establish these settings.

The “Polarimeter Angle Setting” section serves the purpose of configuring the polarimeter to a defined angle. This segment encompasses an input field for entering the desired angle, a button labeled “Move” to direct the polarimeter to the specified angle, a “Home” button to reposition the polarimeter to zero degree, which is considered the home position. Furthermore, there is an “Identify” button to activate the front panel LEDs of the polarizer for identification, and a "Request Position" button. Upon pressing the “Request Position” button, the current position of the polarimeter will be displayed in a nearby textbox.

The “Read” button performs voltage readings and displays the applied voltage values in the respective fields within the output signals section. When the "Read" button is pressed and either the “Set_DAC5544” section or the “Position_Angle” in the "Polarimeter Angle Setting" section is not configured, a window displaying a warning message will appear. This message will indicate the specific fields that need to be set in order to proceed with the read process.

9.1.5.3 “Polarizer_Sweep” button

Pressing the “Polarizer_Sweep” button triggers a sweep of polarization angles. During this process, intensity data is collected and subsequently plotted in relation to the angle sweep of polarization. Figure 9.5 displays the “Polarization Sweep Test” window.

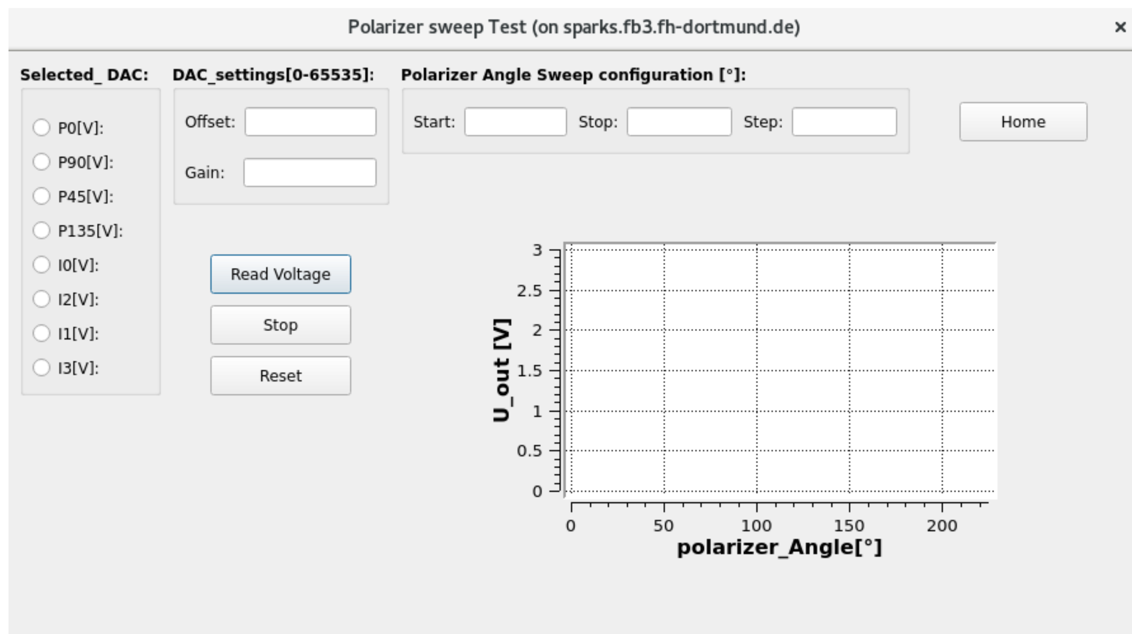


Figure 9.5 “Polarization sweep Test” window

Within the “selected_DAC” field the desired DAC is selected, while the Gain and Offset can be configured within the “DAC_setting” field with values ranging between 0 and 65535.

The “Polarizer Angle Sweep Configuration [°]” field is utilized to establish the start, stop, and steps for the polarizer sweep in degree. To reposition the polarizer to the zero-degree orientation, the “Home” button is employed.

The “Read Voltage” button is used to initiate the measurement process using the specified settings. In case it becomes necessary, the “Stop” button can be employed to cease ongoing readings, while the “Reset” button facilitates the complete configuration reset.

The plot provides a representation of the readout voltages concerning the polarizer angle sweep.

9.1.5.4 “Angle Calculation” button

Pressing the “Angle Calculation” button initiates the opening of an “Angle Calculation” window. This window enables the sweeping of polarization angles

and reads out both calculated angles in continuous mode. The results of these calculations are then displayed. This window is depicted in Figure 9.6.

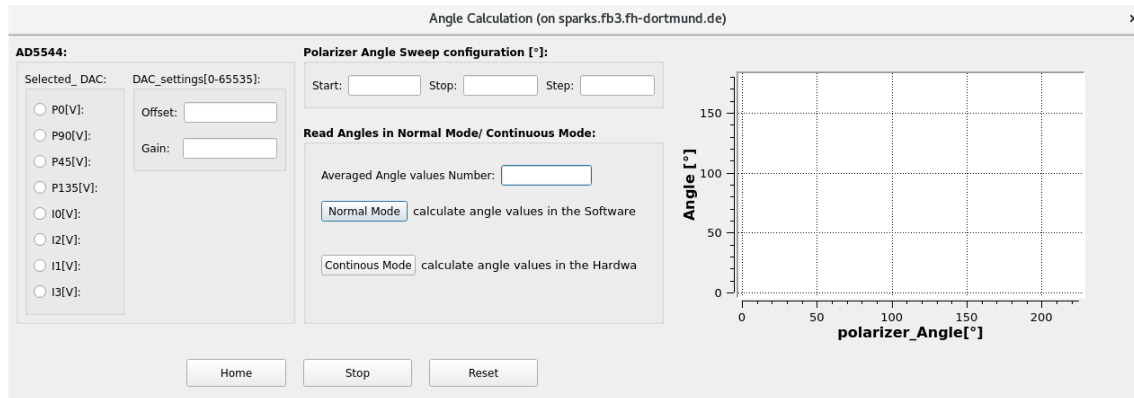


Figure 9.6 “Angle Calculation” window

This window includes the “AD5544” field, serving to choose the appropriate DAC and set its Gain and Offset, and the “Polarizer Angle Sweep Configuration [°]” field to establish the start, stop, and steps for the polarizer sweep in degree, the same functionality found in the “Polarization Sweep Test”.

Moreover, a field titled “Read Angles in Normal Mode/Continuous Mode” is present. The “Normal Mode" button facilitates angle calculations through software, while opting for the “Continuous Mode” button enables the FPGA-based angle calculation.

The input labeled “Averaged Angle Values Number” is employed when calculating angle values through software by selecting the “Normal Mode” button. The given input value defines the number of sampling processes used for averaging per polarizer angle step.

The window also features a plot, graphically depicting the readout angles across the polarizer angle sweep. In addition, this window offers the option to halt and reset the execution process. The “Stop” button interrupts ongoing readings, while the “Reset” button enables a full configuration reset. Additionally, a “Home”

button is available to reset the polarimeter to the zero-degree position, serving as the designated home position.

9.1.5.5 "Light Source Test" button

The "Light Source Test" button is used for connecting and configuring the light source, including the selection of filters and wavelengths. Upon pressing this button, the "Light Source Test" window is triggered to open. Figure 9.7 illustrates the "Light Source Test" window.

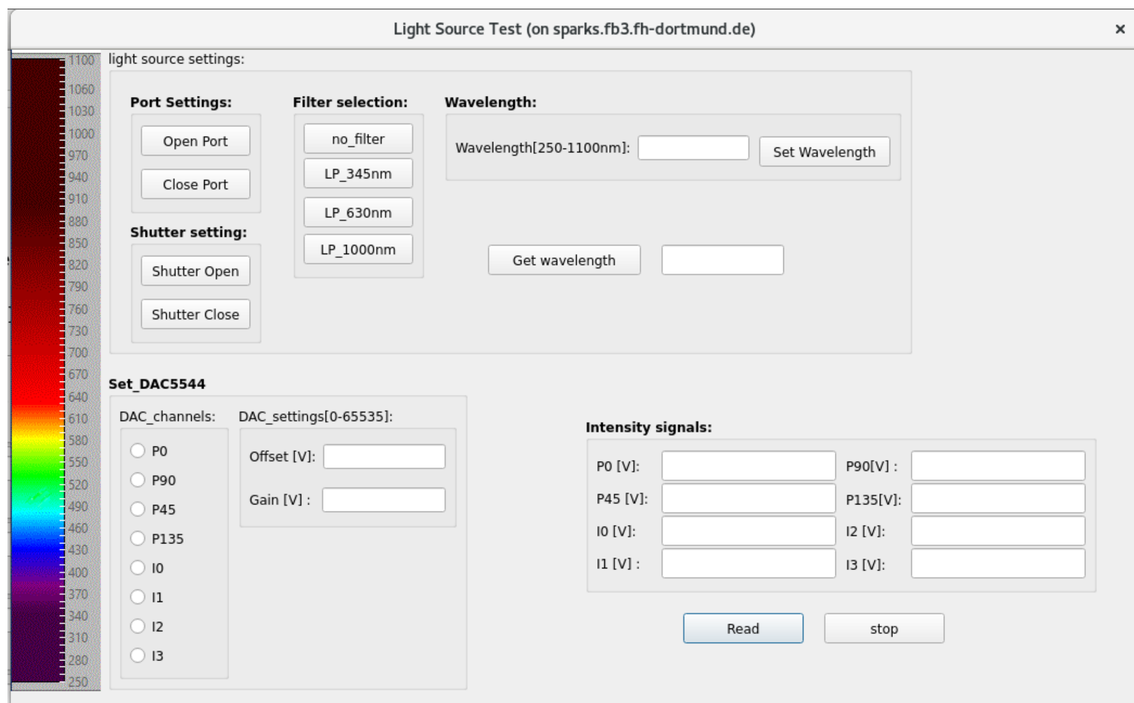


Figure 9.7 "Light Source Test" window

This window is divided into distinct sections. A section for light source configurations which is designated for configuring the light source. The "DAC Setting" field is similar to the functionality described in the previous section (9.1.5.4). Furthermore, it encompasses fields for "Port settings", "shutter setting", "Filter selection" and "Wavelength".

The “Port Settings” option serves to establish a serial communication link between the PC and the light source. This section comprises two buttons: one for initiating the opening of the serial communication and the other for its closure.

Within the “Shutter Setting” field, two buttons are presented one for opening the shutter and the other for closing it.

In the “Filter Selection” section, buttons corresponding to specific filters are available. By selecting any of these buttons, the corresponding command is transmitted to the light source to configure it according to the chosen filter. Additionally, the “Wavelength” field permits the selection of a desired wavelength. The entered wavelength must fall within the range of 250nm to 1100nm. Utilizing the “Set Wavelength” button applies the selected wavelength to the light source, while the “Get Wavelength” button allows checking the currently set wavelength in the light source.

The “Read” button initiates readings and displays the intensity signal values within the “Intensity Signals” section. Furthermore, a “Stop” button is provided to cease ongoing readings.

This comprehensive setup facilitates light source configuration, DAC setting, and the display of readout intensity signals.

9.1.5.6 “WL_Sweep_pol_Fix’ button

The “WL_Sweep_pol_Fix” button is employed to capture intensity signals during a wavelength sweep while keeping the polarizer at a specific angle. Clicking this button initiates the opening of the “Wavelength Sweep” window, as depicted in Figure 9.8.

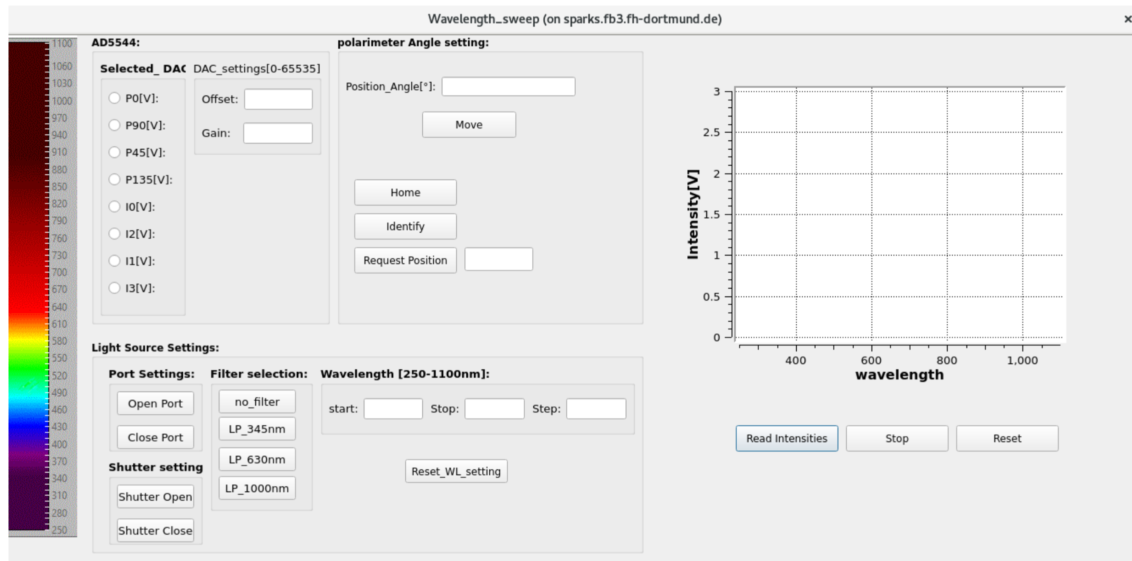


Figure 9.8 “Wavelength_sweep” window

Within this window, the configuration of Gain and Offset for the chosen DAC is facilitated within the “AD5544” field, following the explanations in the previous sections.

In the “Polarimeter Angle Setting” field, the ability to set the polarizer at a specific angle is provided. This can be achieved by entering the desired angle value into the “Position_Angle[°]’ field and subsequently using the ‘Move Polarizer” button to adjust the polarizer accordingly.

The “Home,” “Identify,” and “Request Position” buttons function as described in section 9.1.5.2. The “Port Settings”, “Shutter Setting” and “Filter Selection” fields within the “Light Source Settings” section retain the same functionality outlined in the “Light Source Test” window (refer to Section 9.1.5.5). Additionally, within this field, there is a “Wavelength [250-1100nm]” subsection, comprising three subfields: “Start”, “Stop”, and “Step”. These settings facilitate a wavelength sweep from the defined starting point to the designated stopping point, with specified wavelength steps. The “Reset_WL_setting” button is provided to reset the configuration of the wavelength settings.

Pressing the “Read Intensities” button triggers the presentation of intensity signal readouts across the wavelength sweep through a plotted graph.

Moreover, this window provides the capability to stop and reset the execution process. The “Stop” button interrupts ongoing readings, while the “Reset” button permits a reset of the configuration.

Notice that all configurations across all fields must be completed. Failing to do so and subsequently pressing the “Read Intensities” button will result in the appearance of a warning window, indicating the necessary configurations that need to be addressed.

9.1.5.7 “WL_Sweep_pol_Sweep” button

The “WL_Sweep_pol_Sweep” button is utilized to read out intensity signals while conducting both a wavelength and a polarizer angle sweep.

Clicking this button triggers the opening of the “Wavelength_Polarizer_Sweep” window, as illustrated in figure 9.9.

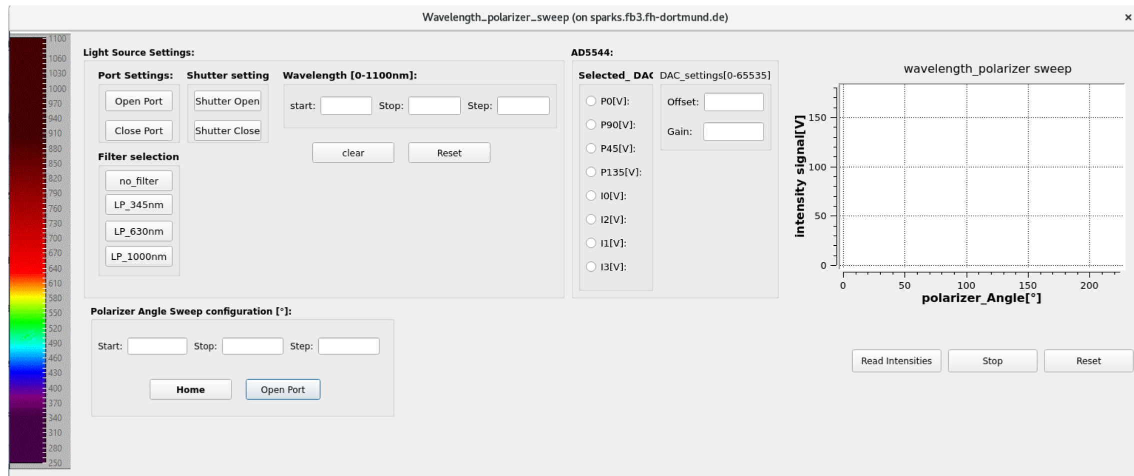


Figure 9.9 “Wavelength_Polarizer_Sweep” window

This window offers identical functionality to the “Wavelength Sweep” window described in the preceding section. The sole distinction lies in the approach: rather than employing a fixed angle for the polarizer, the polarizer will sweep between

the designated start and stop positions in predefined steps as given input in the “Polarizer Angle Sweep configuration [°]” field. Throughout this procedure, the readout intensity signals will be plotted based on the polarizer's angle position.

the “Read Intensities” button starts the process of reading and presentation of intensity signal readouts across the polarization angle sweep through a plotted graph. The “Stop” button is provided to stop ongoing readings and the “Reset” button enables a configuration reset.

9.2 Developed classes for the GUI and sensor readout control

The functionality of the GUI and control software is achieved through the development of classes. The subsequent sections detail these classes.

9.2.1 The “communication_EvalBoard” class

This class is responsible for sending data to the FPGA and for receiving the data sent by the FPGA. The following variables have been defined in the header file of this class:

```
QString      USB_Port;            //Port name
QSerialPort serial;
QString    Byte1;                 // String for synchronization
QByteArray  DatenDAC;             // QByteArray DatenDAC for transmission
QByteArray  DatenADC;             //QByteArray DatenADC for transmission
QByteArray  readarray;            //QByteArray readarray for the received data
bool ok2;                         //Boolean value for converting values
bool read_cont;                   //Boolean value for read continuous mode
bool stop_readcont;               //Boolean value for read stop continuous mode
```

Source Code 9.1 Definition of variables within the header file of the “communication_EvalBoard” class

The constructor and destructor of this class have been inherited from a previous project [9]. The constructor facilitates automated port detection, interface configuration, and the automatic invocation of the “read_serial()” function upon

data reception. Within the constructor, the synchronization algorithm has been accommodated by appending eight 'ones' to the “Byte1” string. On the other hand, the destructor ensures the termination of the connection to the port. The subsequent lines present the source code for both the constructor and the destructor.

```
// constructor

communication_EvalBoard::communication_EvalBoard()
{
  //automatic determination of the port
  foreach(const QSerialPortInfo &serialPortInfo, QSerialPortInfo::availablePorts()){
    if((serialPortInfo.vendorIdentifier()==1027) && (serialPortInfo.productIdentifier()==24592)){
        USB_Port = serialPortInfo.portName();
        qDebug() << USB_Port << "\n";
        qDebug() << serialPortInfo.vendorIdentifier() << "\n";
        qDebug() << serialPortInfo.productIdentifier() << "\n";
    }
}
//Configuration of the interface:
  serial.setPortName(USB_Port);
  serial.setBaudRate(QSerialPort::Baud115200);
  serial.setDataBits(QSerialPort::Data8);
  serial.setParity(QSerialPort::NoParity);
  serial.setStopBits(QSerialPort::OneStop);
  serial.setFlowControl(QSerialPort::NoFlowControl);
  serial.open(QIODevice::ReadWrite);//start Port

   //readyRead-Signal embedded:
  QObject::connect(&serial, SIGNAL(readyRead()), this, SLOT(read_serial()));
  Byte1 ="11111111";   //QString Byte 1 mit acht Einsen füllen
  read_cont=false;
  stop_readcont=false;

}

//Destructor
communication_EvalBoard::~communication_EvalBoard()
{
  //close Port
  serial.close();
}
```

Source Code 9.2 Definition of the constructor and destructor of the “communication_EvalBoard” class

For sending DAC data to the FPGA the function "sendDACdata(QString address, QString commandandchannel, QString data)" has been defined. Upon calling this function, it initiates by transmitting four bytes, all set to '1', to ensure synchronization. Subsequently, it transmits the data received as an argument in the function call. Here's a breakdown of the function's usage:

- **address**: This string corresponds to the command code of the communication protocol used to designate the desired DAC.
- **commandandchannel**: This string has a length of eight characters. In the case of data intended for an "AD5686" type DAC, it should encompass the eight most significant bits (MSBs) for that DAC. However, if the data pertains to an "AD5544" type DAC, the first six characters must be zeroes, while the remaining two characters should accommodate the first two bits required for that DAC.
- **data**: This string is four characters long and should hold the hexadecimal representation of the data bits meant to be written to the chosen DAC.

This function uses a QByteArray, which is sent to the FPGA. One field of this array is one byte. The first four fields are filled with bytes full of ones. The fifth field is filled with the string "address", the sixth with the string "commandandchannel" and the last two with the string "data". Then the QByteArray is sent to the FPGA. The following lines show the source code of the described function:

```
void communication_EvalBoard::sendDACdata(QString address, QString commandandchannel, QString data) {
  //Filling the QByteArray

  DatenDAC[0]=Byte1.toInt(&ok2,2);              //Element 1 = Integer value of the byte with eight ones
  DatenDAC[1]=Byte1.toInt(&ok2,2);              //Element 2 = Integer value of the byte with eight ones
  DatenDAC[2]=Byte1.toInt(&ok2,2);              //Element 3 = Integer value of the byte with eight ones
  DatenDAC[3]=Byte1.toInt(&ok2,2);              //Element 4 = Integer value of the byte with eight ones
  DatenDAC[4]=address.toInt(&ok2,2);            //Element 5 = Integer value from address
  DatenDAC[5]=commandandchannel.toInt(&ok2,2);      //Element 6 = Integer value of CommandandChannel
  DatenDAC[6]=data.left(2).toInt(&ok2,16);        //Element 7 = Integer value of the two left characters of Data
  DatenDAC[7]=data.right(2).toInt(&ok2,16);        //Element 8 = Integer value of the two right characters of
Data
```

```
  serial.write(QByteArray::fromRawData(DatenDAC,8));  //QByteArray DataDAC Send
}
```

Source Code 9.3 Definition of the “sendDACdata(QString address, QString commandandchannel, QString data)” function within the “communication_EvalBoard” class

For reading the ADCs, a more streamlined process is employed where only one additional byte needs to be sent in conjunction with synchronization bytes. To facilitate this task, a distinct function named “sendADCdata(QString address)” has been established. This function undertakes the transmission of synchronization bytes, as well as the data contained within the “address” string, which is provided to the function during its invocation. It's crucial that the “address” string corresponds to a command code within the communication protocol, serving to initiate the ADC readout process.

The function incorporates a QByte array that is dispatched to the FPGA. The initial four fields of this array are filled with synchronization bytes, maintaining consistency with the synchronization approach.

The fifth field is filled with the string “Address”. Subsequently, the QByteArray containing this data is dispatched to the FPGA. The subsequent lines present the source code for the described function:

```
void communication_EvalBoard::sendADCdata(QString Adresse) {
  //Filling the QByteArray
  DatenADC[0]=Byte1.toInt(&ok2,2);        //Element 1 = Integer value of the byte with eight ones
  DatenADC[1]=Byte1.toInt(&ok2,2);        //Element 2 = Integer value of the byte with eight ones
  DatenADC[2]=Byte1.toInt(&ok2,2);        //Element 3 = Integer value of the byte with eight ones
  DatenADC[3]=Byte1.toInt(&ok2,2);        //Element 4 = IInteger value of the byte with eight ones
  DatenADC[4]=Adresse.toInt(&ok2,2);      //Element 5 = Integer value of the two right characters of datenhex
  serial.write(QByteArray::fromRawData(DatenADC,5));  //QByteArray DatenADC Send
}
```

Source Code 9.4 Definition of the “sendADCdata(QString Address)” function within the “communication_EvalBoard” class

For the start and stop of the continuous readout of all ADCs, functions have been created which use the function “sendADCdata(QString address)” function, employing a command code that triggers the start or stop of continuous readout

after synchronization. Furthermore, two variables are used to control the state of continuous ADC readout whether the continuous ADC readout is active or not. The source code for these functions is exemplified below:

```
void communication_EvalBoard::startreadcont()   {
   read_cont=true;
   stop_readcont=false;
   sendADCdata("10000111");
   qDebug() << "ADCaddress:"<<"10000111";
}

void communication_EvalBoard::stopreadcont()
{
   read_cont=false;
   stop_readcont=true;
   sendADCdata("00000111");
   qDebug() << "ADCaddress:"<< "00000111";
}
```

Source Code 9.5 Definition of the "startreadcont()" and "stopreadcont()" functions within the "communication_EvalBoard" class to start and stop the continuous readout of all ADCs respectively

The function "read_serial()" is triggered when data is received. The data received is stored within the QByteArray named "readarray". Upon complete reception of the data, the "connectsignal()" signal is activated. This signal has been defined within the header file of this class. The source code of the function "read_serial()" is in the following lines:

```
void communication_EvalBoard::read_serial() {

   if(stop_readcont){
      serial.clear();
      stop_readcont=false;
   }
   if(read_cont ){                   //read serial port in continous mode
       readarray = serial.read(4);      //QByteArray fill readarray with the received data
       connectsignal();
   }
   else{
       if(!stop_readcont){
          readarray = serial.readAll();      //QByteArray fill readarray with the received data
          connectsignal();
       }

   }
```

```
}
```

Source Code 9.6 Definition of the "read_serial()" function within the "communication_EvalBoard" class for receiving data via USB

The function "getdata()" retrieves and returns the QByteArray "readarray", which holds the most recently received data. The source code for this function could resembles the following structure:

```
QByteArray communication_EvalBoard::getdata() {
  return readarray;
}
```

Source Code 9.7 Definition of the "getdata()" function within the "communication_EvalBoard" class to return the received data

9.2.2 The "MojoSerial" class

The "MojoSerial" class serves a variety of functions within the context of the application. It acts as an intermediary to process user input destined for the DACs in the GUI. Subsequently, this processed data is made available for utilization by various functions of the "communication_EvalBoard" class. The primary objective is to facilitate the transmission of the input from the GUI to the selected DACs within the FPGA.

Additionally, the "MojoSerial" class plays a pivotal role in the communication process with the FPGA. This encompasses both the transmission of data to the FPGA through the "communication_EvalBoard" class and the reception of data from the FPGA. This data, acquired from the FPGA via the "communication_EvalBoard" class, is subsequently utilized for calculations to determine voltage levels corresponding to specific ADCs. The outcome of these calculations are then displayed within the GUI.

Furthermore, the "MojoSerial" class encompasses functionalities related to communication with external components like the polarizer and light source. This

entails configuration, measurement, and plotting of intensity signals and calculated angles, etc. In essence, it orchestrates the interaction with these external components, ensuring proper configuration, measurement, and visualization of the intensity signals.

The constructor and destructor of this class is automatically generated by the Qt Creator as part of creating a “Qt Widgets application”. In the class's constructor, a connection is established between the “connectsignal()” signal, which is defined within the “communication_EvalBoard” class and set after data reception, and the “decision()” function within this class. This linkage ensures that the “decision()” function is always invoked after data has been received. Furthermore, the constructor facilitates access to functions of the “communication_EvalBoard” class using the identifier “communication”.

```
MojoSerial::MojoSerial(QWidget *parent) :
   QMainWindow(parent),
   ui(new Ui::MojoSerial)
{

 ui->setupUi(this);

   Kommunikation = new communication_EvalBoard();
   connect(Kommunikation, SIGNAL(connectsignal()), this, SLOT(decision()));

   msgbox=new QMessageBox(this);

}

MojoSerial::~MojoSerial()
{
   delete ui;
}
```

Source Code 9.8 The constructor and destructor of the “MojoSerial” class

9.2.2.1 “decision()” function

The “decision()” function serves as a central control point, determining the actions to be undertaken based on the flags and conditions present following data reception. The type of operations executed by the function depends on the interaction between various flags and fulfilled conditions.

The variable “readarray” is assigned the data obtained from the “getdata()” function of the “Kommunikation” object of the class “communication_EvalBoard which returns the received data by storing it in a QByteArray. The function uses a series of if and else-if queries to determine the appropriate course of action based on various flags and conditions. If the “lesen” flag is true, the function ad7980() is called to perform operations related to the “AD7980” field. The second condition checks various combinations of flags such as “linearity_test_flag”, “polarizer_sweep_flag”, “polarizer_settle”, “polarization_flag”, “lightsource_flag”, “angle_calculation”, “set_normalmode”, “WL_sweep_flag”, and “WL_Pol_flag”. If any of these conditions are met, the function “test_measurements()” is called.

The last query checks the “flag_measure” and “angle_calculation”, “polarizer_settle” and “set_contmode” flags. If these conditions are met, a timer is used to execute the “read_angle()” function after a delay of 4000 milliseconds (4 seconds) using the “timer_readangle” QTimer object which utilizes the singleShot method. The singleShot method allows to set up a one-time timer event after a specified delay. This causes the “read_angle” function to be called after some preparation time.

```
void MojoSerial::decision(){
   readarray = Kommunikation->getdata();

   if (lesen){
      ad7980();
   }
   else if (linearity_test_flag or (polarizer_sweep_flag and polarizer_settle) or polarization_flag or
lightsource_flag  or
         (angle_calculation and polarizer_settle and set_normalmode) or WL_sweep_flag or (WL_Pol_flag and
polarizer_settle))
     {
      test_measurements();
   }

   else if (flag_measure or (angle_calculation and polarizer_settle and set_contmode)){
    // read_angle();
      timer_readangle->singleShot(4000,this,SLOT(read_angle())) ;
   }
```

```
}
```

Source Code 9.9 Definition of the “decision()” function within the “MojoSerial” class

9.2.2.2 “ad7980()” function

Since the FPGA sends the data of the ADCs in a fixed order, they can be assigned to the individual ADC using if queries in the “ad7980()” function. The sequencing of data transmission from the FPGA to the PC is outlined in Appendix A. For a more comprehensive understanding, refer to reference [9].

The process of assigning received data is illustrated here with an example involving two ADCs. The explanation focuses on the “P45P135” button. If the button's state is equal to one indicating it was selected, it signifies that the Read” button was clicked and the FPGA received instructions to read data from these specific ADCs and transmit the acquired data back to the PC. Hence, the data from these two ADCs has been successfully received. The subsequent process involves checking the activation state of the “P0P90” button. If it is active, the fifth and sixth bytes of the QByte array “readarray” correspond to the ADC “P45” data, and the seventh and eighth bytes relate to the ADC “P135” data. These bytes hold the data for "P135" since the first four bytes encompass the data for the “P0’ and “P90” ADCs.

However, when the “P0P90” button is not activated, the first two bytes contain the data of the “P45” ADC, and the third and fourth bytes contain the data of “P135” ADC. This data, originally in binary form, is converted into hexadecimal values and are represented as strings. Following this conversion, the function “writeinBrowser(QString data1str, QString data3str, QString text fields)” is invoked, passing the generated strings along with a description that the data is originating from the “P45” and “P135” ADCs. The following lines show the source code of the function “ad7980()”:

```
void MojoSerial::ad7980() {

  if (P0P90=="1") {
    QString p0str = readarray.left(2).toHex();
    QString p90str = readarray.mid(2,2).toHex();
    writeinBrowser(p0str, p90str, "textP0P90");
  }

  if (P45P135=="1") {
    QString p45str;
    QString p135str;
    if (P0P90=="1") {
     p45str = readarray.mid(4,2).toHex();
     p135str = readarray.mid(6,2).toHex();
    } else {
     p45str = readarray.left(2).toHex();
     p135str = readarray.mid(2,2).toHex();
    }
    writeinBrowser(p45str,p135str, "textP45P135");
  }

  if (I0I2=="1") {
    QString i0str;
    QString i2str;
    if (P0P90=="1" and P45P135=="1"){
     i0str = readarray.mid(8,2).toHex();
     i2str = readarray.mid(10,2).toHex();
    } else if (P0P90=="1" or P45P135=="1") {
     i0str = readarray.mid(4,2).toHex();
     i2str = readarray.mid(6,2).toHex();
    } else {
     i0str = readarray.left(2).toHex();
     i2str = readarray.mid(2,2).toHex();
    }
    writeinBrowser(i0str, i2str, "textI0I2");
  }

  if (I1I3=="1") {
   QByteArray i1i3array;
   i1i3array = readarray.right(4);
   QString i1str = i1i3array.left(2).toHex();
   QString i3str = i1i3array.right(2).toHex();
   writeinBrowser(i1str, i3str, "textI1I3");
   }
}
```

Source Code 9.10 Definition of the “ad7980()” function within the “MojoSerial” class

9.2.2.3 “writeinBrowser(QString daten1str, QString daten3str, QString textfelder)” function

The function “writeinBrowser(QString daten1str, QString daten3str, QString textfelder)” serves the purpose of accepting hexadecimal data from two ADCs, along with information about which ADCs the data originate from. The function

includes converting this data into voltage values corresponding to the actual voltage levels at both ADC inputs. The computed voltage values are transformed into strings. These string representations of these voltages are then placed into the designated text fields in the GUI, ensuring that the voltages applied to the selected ADCs are displayed.

```
void MojoSerial::writeinBrowser(QString daten1str, QString daten2str, QString textfelder) {

  int daten1 = daten1str.toInt(&ok, 16);
  float spannung1 = ((daten1 * 3.0000) / 65536.0000);
  QString spannung1str = QString::number(spannung1, 'f', 6);

  int daten2 = daten2str.toInt(&ok, 16);
  float spannung2 = ((daten2 * 3.0000) / 65536.0000);
  QString spannung2str = QString::number(spannung2, 'f', 6);

  if(lesen){

    if (textfelder == "textP0P90") {
      ui -> textP0 -> setText(spannung1str);
      ui -> textP90 -> setText(spannung2str);

    } else if (textfelder == "textP45P135") {
      ui -> textP45 -> setText(spannung1str);
      ui -> textP135 -> setText(spannung2str);

    } else if (textfelder == "textI0I2") {
      ui -> textI0 -> setText(spannung1str);
      ui -> textI2 -> setText(spannung2str);

    } else if (textfelder == "textI1I3") {
      ui -> textI1 -> setText(spannung1str);
      ui -> textI3 -> setText(spannung2str);
    }
  }
```

Source Code 9.11 Definition of the “writeinBrowser()” function within the “MojoSerial” class

9.2.2.4 “test_measurements()”function

An additional function which is used within the “decision()” function in the “Mojoserial” class is named “test_measurements()”. This function takes the received data, calculates voltage values, and updates different windows or user interface elements based on the combination of received flags, ensuring that the appropriate windows are updated with the relevant information.

The function extracts sections of the received data array “readarray” and converts them to hexadecimal strings (p0str, p90str, etc.). It then converts these hexadecimal strings to integer values (data0, data90, etc.) using base 16 (hexadecimal) and calculates the corresponding voltage values in a range of 0 to 3.0000 Volts using the formula: (data * 3.0000) / 65536.0000.

It converts the calculated voltage values back to strings with up to 6 decimal places (spannung_0str, spannung_90str, etc.). Depending on the combination of flags (“linearity_test_flag”, “polarizer_sweep_flag”, “polarizer_settle”, “polarization_flag, angle_calculation”, “set_normalmode”, “lightsource_flag”, “WL_sweep_flag”, “WL_Pol_flag”), it updates different windows or user interface elements with the calculated voltage values.

If “linearity_test_flag” is true and “polarizer_sweep_flag” is false, it updates the “linearity_window” which corresponds to the linearity measurement. If neither “linearity_test_flag” nor “polarizer_sweep_flag” is true, it checks if “polarizer_sweep_flag” and “polarizer_settle” are true and updates the “polarizer_test_window” which corresponds to the polarization measurement. If the “polarization_flag” is true, it updates the “polarization_window”. If neither the “linearity_test_flag” nor the “polarizer_sweep_flag” are true, and the “angle_calculation”, the “polarizer_settle”, and the “set_normalmode” are true, it calculates angles and updates the “angle_calculation_window”. If the “lightsource_flag” is true, it updates the “lightsource_window”. If the “WL_sweep_flag” is true, it updates the “WL_sweep_window”. If both the “WL_Pol_flag” and the “polarizer_settle” are true, it updates the “WL_Polarizer_window”. After all the updates are done, it sets the “polarizer_settle” flag to false. This flag is employed within the processes that involve polarizer sweeping. The source code of this function is as follows:

```
void MojoSerial::test_measurements()
{
   QString p0str = readarray.left(2).toHex();                   //p0str mit dem Hexadezimalwert der zwei linken
Zeichen von readarray füllen
   QString p90str = readarray.mid(2,2).toHex();                 //p90str mit dem Hexadezimalwert der zwei rechten
Zeichen von readarray füllen
   QString p45str = readarray.mid(4,2).toHex();
   QString p135str = readarray.mid(6,2).toHex();
   QString i0str= readarray.mid(8,2).toHex();
   QString i2str=readarray.mid(10,2).toHex();
   QByteArray i1i3array;
   i1i3array = readarray.right(4);
   QString i1str = i1i3array.left(2).toHex();
   QString i3str = i1i3array.right(2).toHex();

  // qDebug() << "p0str:"<<p0str;
  // qDebug() << "p90str:"<<p90str;

   int data0 = p0str .toInt(&ok,16);
   float spannung_0 = ((data0 * 3.0000) / 65536.0000);
   int data90 = p90str .toInt(&ok,16);
   float spannung_90 = ((data90 * 3.0000) / 65536.0000);
   int data135 = p135str .toInt(&ok,16);
   float spannung_135 = ((data135 * 3.0000) / 65536.0000);
   int data45 = p45str .toInt(&ok,16);
   float spannung_45 = ((data45 * 3.0000) / 65536.0000);
   int datai0 = i0str .toInt(&ok,16);
   float spannung_i0 = ((datai0 * 3.0000) / 65536.0000);
   int datai1 = i1str .toInt(&ok,16);
   float spannung_i1 = ((datai1 * 3.0000) / 65536.0000);
   int datai2 = i2str .toInt(&ok,16);
   float spannung_i2 = ((datai2 * 3.0000) / 65536.0000);
   int datai3 = i3str .toInt(&ok,16);
   float spannung_i3 = ((datai3 * 3.0000) / 65536.0000);

   QString spannung_0str = QString::number(spannung_0, 'f', 6);
   QString spannung_90str = QString::number(spannung_90, 'f', 6);
   QString spannung_45str = QString::number(spannung_45, 'f', 6);
   QString spannung_135str = QString::number(spannung_135, 'f', 6);
   QString spannung_i0str = QString::number(spannung_i0, 'f', 6);
   QString spannung_i1str = QString::number(spannung_i1, 'f', 6);
   QString spannung_i2str = QString::number(spannung_i2, 'f', 6);
   QString spannung_i3str = QString::number(spannung_i3, 'f', 6);

   if(linearity_test_flag and  !polarizer_sweep_flag)
      linearity_window ->update_values(spannung_0 ,
spannung_90,spannung_45,spannung_135,spannung_i0,spannung_i1,spannung_i2,spannung_i3);
   else if(!linearity_test_flag and  (polarizer_sweep_flag and polarizer_settle) )
      polarizer_test_window-> update_values(spannung_0 ,
spannung_90,spannung_45,spannung_135,spannung_i0,spannung_i1,spannung_i2,spannung_i3);
   else if(polarization_flag)
         polarization_window ->update_values(spannung_0str ,
spannung_90str,spannung_45str,spannung_135str,spannung_i0str,spannung_i1str,spannung_i2str,spannung_i3
str);
   else if(!linearity_test_flag and  !(polarizer_sweep_flag and polarizer_settle) and ( angle_calculation and
polarizer_settle and set_normalmode))
```

```
        angle_calculation_window-> calculate_angles(spannung_0 ,
spannung_90,spannung_45,spannung_135);
   else if(lightsource_flag ){
        lightsource_window  -> update_values(spannung_0str ,
spannung_90str,spannung_45str,spannung_135str,spannung_i0str,spannung_i1str,spannung_i2str,spannung_i3
str);

   }
   else if(WL_sweep_flag){
      WL_sweep_window -> update_values(spannung_0 ,
spannung_90,spannung_45,spannung_135,spannung_i0,spannung_i1,spannung_i2,spannung_i3);
   }

   else if(WL_Pol_flag and polarizer_settle){
      WL_Polarizer_window -> update_values(spannung_0 ,
spannung_90,spannung_45,spannung_135,spannung_i0,spannung_i1,spannung_i2,spannung_i3);
    }

   polarizer_settle=false;

}
```

Source Code 9.12 Definition of the “test_measurements()” function in the “MojoSerial” class to update different windows or user interface elements

9.2.2.5 “read_angle()” function

This function is employed within the “decision()” function in the “Mojoserial” class. It is responsible for processing angle measurement data obtained from the FPGA. Initially, it extracts the first 2 bytes (4 characters) of the “readarray” as a first angle measurement data (angle90_0 value) and the next 2 bytes (4 characters) of the readarray starting from the 3rd byte as a second angle measurement data (angle135_45) value and converts them to a hexadecimal string and assigns it to the “angle90_0str” and “angle135_45str” variables. Then these hexadecimal strings are converted to integer values. The if-else queries are used to differentiate the behavior of the function based on the mode of operation, ensuring that the angle measurement data is processed appropriately for each mode.

If the “flag_measure” is true, the obtained values correspond to the “angle_measurment” form and by using the “angle_measurment” object, the “calculate_measurement” function is called with angle90_0 and angle135_45

values. Additionally, the "angle90_0str" value is written to a file named "angle90_0.txt" and the "angle135_45str" value is written to a file named "angle135_45.txt" for recording purposes.

The else-if query checks if the flags "angle_calculation", "polarizer_settle", and "set_contmode" are all true. If these conditions are met, it indicates that the data is related to "Angle_Calculation" form in continuous mode. Then the calculated integer values of "angle90_0" and "angle135_45" are converted to degree values. These converted angle values are then sent to the "Angle_Calculation" form by calling the "realtimeplot()" function. This function is responsible for updating a real-time plot and displaying angle values on the GUI. If the "polarizer_settle" flag is set to false, it indicates the end of a specific phase of settling.

```
void MojoSerial::read_angle() {
  angle90_0str = readarray.left(2).toHex();
  angle135_45str = readarray.mid(2,2).toHex();

  int angle90_0=angle90_0str .toInt(&ok,16);
  int  angle135_45=angle135_45str .toInt(&ok,16);

  if(flag_measure) {
    angle_measurment ->calculate_measurement(angle90_0 , angle135_45);

    QFile file3("/user/lalaee/Desktop/measurements/angle90_0.txt");
    if(file3.open((QIODevice::WriteOnly)| QIODevice::Append |QIODevice::Text))
    {
      QTextStream stream(&file3);
      stream <<angle90_0str <<Qt::endl;
      file3.close();
    }
  }
  else if(angle_calculation and polarizer_settle and set_contmode){
    float angle90=((float(angle90_0))/16384.0) *(180.0/3.14); //convert radian value of angle90 to degree
    float angle135= ((float(angle135_45))/16384.0)*(180.0/3.14);//convert radian value of angle135 to
degree

    qDebug() << "angle90_0:" << angle90;
    qDebug() << "angle135_45:" << angle135;
    angle_calculation_window-> realtimeplot(angle90, angle135);
  }
  polarizer_settle=false;

}
```

Source Code 9.13 Definition of the "read_angle()" function within the "MojoSerial" class designed to process angle measurement data acquired from the FPGA

9.2.2.6 Functions for the "AD5686" field of the GUI

In the form file of the main window, the action "clicked()" was linked to the "Send" button. This function defines the behavior when the button is clicked. Initially, the following variables are defined:

```
QString commandstr;              //String for the entered command bits
QString D, C, B, A;              //Strings for channels D, C, B, A
QString DCBA;                    //Strings to summarize D, C, B, A
QString spannungsstr;            //String for the entered voltage
QByteArray Daten;                //QByteArray Daten for transmission
QString Adresse="00000101"; //QString Address for transmission
```

Source Code 9.14 Definition of variables in the "on_pushButton_clicked()" function of DAC "AD5686" field within the "MojoSerial" class

Subsequently, the content of the text field labeled "lineEdit_3," intended for inputting the four command bits for the "AD5686" DAC type, is extracted and assigned to the string variable "commandstr". To accommodate later operations requiring a decimal representation, the binary command bits entered in "commandstr" are converted into an integer value known as "commandbits". The source code for this resembles the following structure:

```
commandstr = ui->lineEdit_3->text();
int commandbits=commandstr.toInt(&ok,2);
```

Source Code 9.15 Save the input of the command bits and convert to an integer value in the "on_pushButton_clicked()" function of DAC "AD5686"

The buttons for the DAC channels are each queried with an if condition. If a button is activated, a string with the same name as the selected DAC channel is filled with "1", otherwise with "0". After the if-queries all strings are combined to a string "DCBA". This string is then appended to the string with the command bits and passed to the string "commchanstring". Thus, the first eight MSBs for the input shift register (see chapter 3.1.3 in [9]) are in one string. The source text for this is in the following lines:

```
if (ui->radioButton->isChecked()) {     //if radioButton is true,
   D="1";                          //Fill string D with "1",
} else {                           //otherwise
   D="0";                          //with "0"
}
if (ui->radioButton_2->isChecked()) {   //if radioButton_2 is true,
   C="1";                          //Fill string C with "1
} else {                           //otherwise
   C="0";                          //with "0"
}
if (ui->radioButton_3->isChecked()) {   //if radioButton_3 is true,
   B="1";                          //Fill String B with "1" ,
} else {                           //otherwise
   B="0";                          //with "0"
}
if (ui->radioButton_4->isChecked()) {   //wenn radioButton_4 true ist,
   A="1";                          //Fill String A with "1"S,
} else {                           //otherwise
   A="0";                          //with "0"
}

//Combining the command bits and the selected DAC channels to a string (binary)
DCBA=D+C+B+A;
QString commchanstring=commandstr+DCBA;
```

Source Code 9.16 Form a string based on the selected channels in the "on_pushButton_clicked()" function of the DAC "AD5686" field

The maximum possible output voltage at a reference voltage of 3 V according to the calculation with the transfer function is:

$$V_{OUT} = 3V * \frac{65535}{65536} = 2{,}99995\ V \tag{9.1}$$

The smallest possible adjustable output voltage after 0 V is:

$$V_{OUT} = 3V * \frac{1}{65536} = 0{,}00005\ V \tag{9.2}$$

Due to the 16-bit resolution, there is therefore a voltage difference of at least 0.00005 V between two differently set output voltages. For this reason, users should be able to enter mV values into the application. The maximum adjustable output voltage is therefore 2999 mV. For the later transmission of the 16 data bits the decimal equivalent of this input must be determined with the transfer function [9]. For this purpose, the formula must be converted to "D" which results in:

$$D = \frac{V_{OUT}}{V_{REF}} * 2^N \quad (7.3)$$

Once the content of the voltage text field has been captured in the string "spannungsstr", it undergoes conversion into an integer value, which is subsequently assigned to the variable "spannungseingabe". Utilizing this transformed value, the 16 data bits decimal equivalent is computed using a specific formula, and this result is stored in the integer variable "Di". Following this, the value of "Di" is transformed into a hexadecimal representation, then further into a string that finds its place in the variable "Distr".

An "if" query is employed to inspect whether the string "Distr" comprises four characters. If it falls short, the requisite number of zeros is affixed to the beginning of the string, ensuring it attains a length of precisely four characters. This step bears significance as the hexadecimal string must encapsulate the full 16 data bits and adhere to a length of four characters. It's pertinent to recognize that a four-character hexadecimal number corresponds to a 16-character binary counterpart.

```
  spannungsstr = ui->lineEdit_4->text();

  int spannungseingabe = spannungsstr.toInt(&ok, 10);
  int Di=((spannungseingabe*65536)/3000);
  //Conversion of the decimal transfer value into a hexadecimal value and a string to be able to fill the
QByteArray data
  QString Distr = QString::number(Di,16);
  //if Distr is smaller than four characters, it is added with so many zeros that Distr contains exactly four
characters
  if (Distr.size()<2) {
    Distr= "000" + Distr;
  } else if (Distr.size()<3) {
    Distr= "00" + Distr;
  } else if (Distr.size()<4) {
    Distr= "0" + Distr;
  }
```

Source Code 9.17 Converting captured voltage input to a 16-bit hexadecimal string, transforming into the complete 16 data bits with proper zero-padding in the "on_pushButton_clicked()" function of the DAC "AD5686" field

The initial definition of the string “address” involves the command code utilized for selecting the DAC of “AD5686” type. Within the string “commchanstring” the eight most significant bits (MSBs) are stored in binary form. In parallel, the string “Distr” captures the 16 data bits in hexadecimal format, intended for the shift register of the “AD5686” DAC.

When necessary, these strings are furnished to the function “sendDACdata(QString address, QString commandandchannel, QStringdata)”, facilitating the transmission of data to the FPGA. However, this transmission should only occur if the provided input is accurate. Any discrepancies should prompt the display of an “Error Message” dialog.

An “if” query is employed to verify input errors. The string “DCBA” should not consist solely of zeros, as this would indicate the absence of a selected channel. The size of the “commandstr” string must be at least four, as there are four command bits in total. If non-binary characters are input as command bits, their integer equivalent is zero. The integer equivalent can only be zero if four zeros have been specifically entered. Similarly, if non-numeric characters are used for the voltage input, the integer equivalent of the voltage input becomes zero.

Furthermore, the voltage input is only considered zero if at least one zero has been entered, and no other characters are present. Additionally, the integer equivalent must not exceed 2999, as this is the maximum output voltage permissible.

If any of these conditions are not met, an error message dialog will appear, deactivating the main window. Until this dialog is closed, no further changes can be made to the main window inputs. Upon correcting entries, the function “sendDACdata(QString address, QString commandandchannel, QString data)” is executed, transmitting the data to the FPGA. If the data transmission is successful,

the voltage input field is automatically cleared. The source code for this is as following:

```
   if (DCBA == "0000" or commandstr.size()<4 or (commandbits==0 and commandstr !="0000") or
spannungseingabe>2999
   or((spannungsstr!="0" and spannungsstr!="00" and spannungsstr!="000" and spannungsstr!="0000") and
spannungseingabe==0)) {
       FensterZwei fensterzwei;
       fensterzwei.setModal(true);
       fensterzwei.exec();
   } else {
   Kommunikation->sendDACdata(Adresse, commchanstring, Distr);
   ui->lineEdit_4->clear();
   }
}
```

Source Code 9.18 Validation of input conditions in the "on_pushButton_clicked()" function of the DAC "AD5686" field

9.2.2.7 Functions for the "Mode selection" field of the GUI

Upon clicking the "Mode_set" button in the GUI, the "on_Mode_set_clicked()" function is invoked. This function is responsible for configuring modes for ADCs based on user interaction with continuous mode or normal mode in the GUI.

Initially, some boolean variables are defined to initialize with false values. These variables are employed for control purposes, ensuring they do not disrupt various execution processes that can occur in either normal mode or continuous mode within the program. Subsequently, it determines the selected mode between "normal mode" and "continuous mode" based on the states of the radio buttons in the GUI. Depending on the selected mode, appropriate flags ("normalmode" or "contmode") are set to control the program's subsequent execution logic. If no mode is selected, a warning message is displayed to prompt the user to make a choice. The source code for this function has the following structure:

```
void MojoSerial::on_Mode_set_clicked()   //define the mode of ADCs(continious mode or normal mode)
{
   lesen=false;
   linearity_test_flag=false;
   flag_measure=false;
   polarization_flag=false;
   angle_calculation=false;
```

```
  if (ui->normalmode->isChecked()) {
     normalmode=true;
     contmode=false;
  }
  else if(ui->continuousmode->isChecked()) {
     normalmode=false;
     contmode=true;
  }
  else
  {
     msgbox->warning(this,"warning","please select one mode.");
  }
}
```

Source Code 9.19 Define the mode of ADCs in the “on_Mode_set_clicked()” function within the “MojoSerial” class

Clicking the “reset” button in the GUI, the “on_reset_clicked()” function is invoked. This function is designed to call the AD5544 function multiple times with different parameters and to configure the DAC settings to their initial values at zero, to clear text fields, and to reset the program mode and other flags to their initial values.

```
void MojoSerial::on_reset_clicked()
{
  AD5544(0, "00000000", "00000000");
  AD5544(0, "00000000", "00000001");
  AD5544(0, "00000000", "00000010");
  AD5544(0, "00000000", "00000011");

  ui -> textP0  -> clear();
  ui -> textP90 -> clear();
  ui -> textP45 -> clear();
  ui -> textP135-> clear();
  ui -> textI0  -> clear();
  ui -> textI2  -> clear();
  ui -> textI1  -> clear();
  ui -> textI3  -> clear();

  linearity_test_flag=false;
  flag_measure=false;
  lesen=false;
  normalmode=false;
  contmode=false;
  angle_calculation=false;
  lightsource_flag=false;
  polarization_flag=false;
  polarizer_sweep_flag =false;

}
```

Source Code 9.20 Configure of the DAC settings to initial values at zero in the “on_reset_clicked()” function within the “MojoSerial” class

Furthermore, to readout and display the voltages in continuous mode there is a “Angle_measurement” button which by clicking invokes the function “on_Angle_meaurment_clicked()”. This function manages the setup and interaction with the angle measurement functionality in the program. It ensures that the proper conditions are met for continuous mode operation before connecting signals and displaying the measurement interface.

Since this button opens the new form, an initial is checked if the “angle_measurment” object exists. If it does not exist, it is created using the new operator and associated with the “Angle_measurment” class. This step ensures that there is a valid instance of the angle measurement object.

```
if (!angle_measurment)
    angle_measurment = new Angle_measurment(this);
```

Source Code 9.21 Instantiate an object of the “Angle_measurment” class in the “on_Angle_meaurment_clicked()” function

then, the flags lesen and “linearity_test_flag” are set to false and the flag “flag_measure” is set to true.

```
lesen=false;
linearity_test_flag=false;
flag_measure=true;
```

Source Code 9.22 Set flags for measurement control in the “on_Angle_meaurment_clicked()” function

If the “contmode” flag is true indicating continuous mode is selected, the code establishes several connections using the “connect()” function. These connections are set up to link signals from the “angle_measurment” object to slots in the MojoSerial class.

The connections are related to starting and stopping data transmission ("send_startcontdata()" and "send_stopcontdata()" signals), as well as resetting the angle form interface ("reset_angleform()" signal).

The "get_startcontdata()", "get_stopcontdata()", and "angleform_exit()" slots in the MojoSerial class are connected to these signals. The "angle_measurment" interface is displayed using the "show()" method. If contmode is false, a warning message is displayed using the "msgbox->warning()" function. The warning message prompts the user to choose an appropriate mode ("continuous mode") and to click the "Mode_set" button before proceeding. The code for this process is as follows:

```
void MojoSerial::on_Angle_meaurment_clicked()
{
  if (!angle_measurment)
    angle_measurment = new Angle_measurment(this);

  lesen=false;
  linearity_test_flag=false;
  flag_measure=true;

  if(contmode ){
    connect(angle_measurment,SIGNAL(send_startcontdata()),this,SLOT(get_startcontdata()));
    connect(angle_measurment,SIGNAL(send_stopcontdata()),this,SLOT(get_stopcontdata()));
    connect(angle_measurment,SIGNAL(reset_angleform()),this,SLOT(angleform_exit()));
    angle_measurment ->show();
   }
   else {
    msgbox->warning(this,"warning","please select continuous Mode and click Mode_set button.");
   }
}
```

Source Code 9.23 Definition of the "on_Angle_meaurment_clicked()" function within the "MojoSerial" class

The "get_startcontdata()" function is defined within the "MojoSerial" class. This function is related to starting the continuous data acquisition process.

It sets the flag_measure flag to true. This flag controls the measurement process and invokes the "startreadcont()" function on the "Kommunikation" object

(“Kommunikation” is an instance of the “communication_EvalBoard()” class) for initializing the continuous data reading operation.

```
void MojoSerial::get_startcontdata(){
   flag_measure=true;
   Kommunikation->startreadcont();
 }
```

Source Code 9.24 Definition of the “get_startcontdata()” function within the “MojoSerial” class

The “get_stopcontdata()” function is involved in stopping the ongoing continuous data acquisition process. It updates the “flag_measure” flag to the false value and invokes “stopreadcont()” function that manages the cessation of continuous data reading.

```
void MojoSerial::get_stopcontdata(){
   flag_measure=false;
   Kommunikation->stopreadcont();
 }
```

Source Code 9.25 Definition of the “get_stopcontdata()” function within the “MojoSerial” class

The “angleform_exit()” function is responsible for stopping the ongoing continuous data acquisition process. It invokes the “stopreadcont()” function, which is designed to manage the termination of continuous data reading when the “Angle_meaurment” window is closed.

```
void MojoSerial::angleform_exit(){

      Kommunikation->stopreadcont();

 }
```

Source Code 9.26 Definition of the “angleform_exit()” function within the “MojoSerial” class

9.2.2.8 Functions for the “AD7980” field of the GUI

Clicking the “Lesen” button within the “AD7980” field triggers the execution of the function “on_Buttonlesen_clicked()”. In this function, a sequence of actions is considered. Initially, several boolean variables are initialized to ensure that the reading voltages are related to voltage reading operations in the “AD7980” field. Before proceeding with further actions, the function checks if the flags for “Angle_measurement” and “Linearity_Test” are set to false, the “lesen” flag is set to true. This action indicates that the data transmitted from the FPGA must be forwarded to the “AD7980” field.

This arrangement ensures that the reading of voltages is linked to the “AD7980” field in the main GUI. The careful initialization of variables and the conditional checks helps to establish a controlled and accurate process for the voltage measurement operations.

If “normalmode” is set, a check is performed to determine which ADCs have been chosen. If a button is activated, a string with the same name as the selected ADC in the “AD7980” field is populated with “1”; otherwise, it is filled with “0”. These strings, corresponding to the status of the buttons, are combined into a sequence. To this sequence, the string “0110” is appended, resulting in the creation of the “ADCAdresse” string.

When two or more ADCs are chosen, this “ADCAdresse” string takes on the role of the command code in the communication protocol. This ensures the selection of the appropriate ADCs for data retrieval, with the acquired data subsequently sent to the PC. Consequently, if two or more ADCs are selected, the function “sendADCdata(ADCAdresse)” is invoked. The “ADCAdresse” string is then passed to this function to facilitate the transmission of data to the FPGA.

In cases where no button is activated, an error message dialog emerges, deactivating the primary window. This window remains inactive until the error message dialog is closed.

The subsequent lines showcase the structure of the "on_ Buttonlesen _clicked()" function:

```
void MojoSerial::on_Buttonlesen_clicked()
{
    QString ADCAdresse;       //String for the address of the ADCs
    QByteArray Daten;

    polarizer_sweep_flag =false;
    angle_calculation=false;
    polarization_flag=false;

    if(!(flag_measure or linearity_test_flag)){
        lesen=true;
        if (normalmode){
            //checking whether the respective buttons for the channels are true or false and filling the associated
strings
            if (ui->ButtonP0P90->isChecked()) {      //if ButtonP0P90 is true,
                P0P90="1";                       //Fill string P0P90 with "1",
            } else {                             //otherwise
                P0P90="0";                       //with "0"
                ui -> textP0 -> clear();
                ui -> textP90 -> clear();
            }

            if (ui->ButtonP45P135->isChecked()) {    //if ButtonP45P135 is true,
                P45P135="1";                     //Fill string P45P135 with "1",
            } else {                             //otherwise
                P45P135="0";                     //with "0"
                ui -> textP45 ->  clear();
                ui -> textP135 ->  clear();


            }
            if (ui->ButtonI0I2->isChecked()) {    //wenn ButtonI0I2 true ist,
                I0I2="1";                        //Fill String I0I2 with "1" ,
            } else {                             //otherwise
                I0I2="0";                        //with "0"
                ui -> textI0 -> clear();
                ui -> textI2 -> clear();
            }
            if (ui->ButtonI1I3->isChecked()) {    //wenn ButtonI1I3 true ist,
                I1I3="1";                        //Fill String I1I3 with "1" ,
            } else {                             //otherwise
                I1I3="0";                        //with "0"
                ui -> textI1 -> clear();
                ui -> textI3 -> clear();
             }
            ADCAdresse= P0P90 + P45P135 + I0I2 + I1I3 +"0110";
```

```
        // qDebug() << ADCAdresse;
        if (ADCAdresse=="00000110") {  //if no.......ErrorMESSAGE, otherwise send data
          FensterZwei fensterzwei;
          fensterzwei.setModal(true);
          fensterzwei.exec();
        } else {
          //qDebug() << ADCAdresse;
          Kommunikation->sendADCdata(ADCAdresse);
        }
      }
      else{
        msgbox->warning(this,"warning","please select normal Mode and click Mode_set button.");
      }
  }
}
```

Source Code 9.27 Definition of the “on_Buttonlesen_clicked()” function within the “MojoSerial” class

9.2.2.9 Functions for the “AD5544” field of the GUI

For every channel of the “AD5544” DAC type, there exists both a slider and a progress bar. To ensure synchronization between the slider and its corresponding progress bar, the “valueChanged(int)” action has been incorporated into each slider within the main window's form. Within each of these action functions, the programming dictates that the linked progress bar should reflect the exact value of the slider. An example of such a function's source code is depicted below:

```
void MojoSerial::on_SliderU6A_valueChanged(int value)
{
  ui->BarU6A->setValue(value);
}
```

Source Code 9.28 Definition of the “on_SliderU6A_valueChanged(int value)” function within the “MojoSerial” class to update the value of “BarU6A” to match the changed value of “SliderU6A” in response to the user's interaction

Upon dragging a slider with the mouse and subsequently releasing it, the chosen channel of the designated DAC is configured to match the percentage indicated by the associated progress bar. To facilitate this, the “sliderReleased()” action has been appended to each slider in the main window's form. Within each of these action functions, the “AD5544(int progress, QString address, QString channel)”

function is invoked. This function receives the value from the respective progress bar, the command code corresponding to the selected DAC in the communication protocol, and a byte where the two least significant bits (LSBs) correspond to the selected channel. The subsequent lines present an illustrative example of the "sliderReleased()" function and the invocation of the aforementioned function:

```
void MojoSerial::on_SliderU6A_sliderReleased()
{
   AD5544(ui->BarU6A->value(), "00000000", "00000000");
}
```

Source Code 9.29 Definition of the "on_SliderU6A_sliderReleased()" function within the "MojoSerial" class

The function "AD5544(int progress, QString address, QString channel)" is responsible for computing a hexadecimal representation based on the input received from a progress bar. In the case of "AD5544" type DACs, a total of 16 data bits are utilized. When all 16 bits are set to one, the corresponding decimal value is 65535. Considering that a progress bar operates with integer values ranging from zero to 100, a conversion process is required.

To achieve this conversion, the value obtained from the progress bar is divided by 100 and then multiplied by 65535, reflecting the scale of the DAC's available range. This ensures that the progress bar's integer values are appropriately mapped onto the 16-bit range of the "AD5544" DACs.

The purpose is to determine the decimal equivalent that aligns with the 16 data bits, configuring the desired value for the chosen DAC. This decimal value is then converted into a hexadecimal representation. If the resulting hexadecimal value spans fewer than four characters, it is filled with zeros to ensure a total length of four characters, thus covering all 16 data bits. Subsequently, the function "sendDACdata(QString address, QString commandandchannel, QString data)" is invoked with the appropriate parameters to transmit this data to the FPGA. The following lines show the source code for this:

```
void MojoSerial::AD5544(int progress, QString Adresse, QString Kanal) {
   int datenint = (progress * 65535)/100;
  QString datenhex = QString::number(datenint,16);
   if (datenhex.size()<2) {
      datenhex= "000" + datenhex;
   }  else if (datenhex.size()<3) {
      datenhex= "00" + datenhex;
   }  else if (datenhex.size()<4) {
      datenhex= "0" + datenhex;
   }
   Kommunikation->sendDACdata(Adresse, Kanal, datenhex);

}
```

Source Code 9.30 Definition of the “AD5544(int progress, QString Adresse, QString Kanal)” function within the “MojoSerial” class

9.2.2.10 Functions for the “Measurement and Test” field of the GUI

For each button within the “Measurement and Test" field, there is a corresponding function that gets triggered when the button is clicked. The functions associated with these buttons are described as follows.

9.2.2.10.1 “on_linearity_test_clicked()” function

This function prepares the linearity test interface for use by creating a form, setting flags and variables, showing the form to the user, and establishing connections to manage interactions with the linearity test interface.

Firstly, it is verified whether the “linearity_window” object has been instantiated. If it has not been created, a new instance of the “linearity_test” form is created.

The “lesen” flag is set to false, to prevent data reading during the linearity test. The flags “flag_measure”, “polarization_flag” and “polarizer_sweep_flag” are set to false to ensure that the measurement, polarization and polarizer sweep processes are inactive, respectively. The “linearity_test_flag” is set to true to indicate that the linearity test interface is active.

The “linearity_window” form is shown to the user and the connections between signals and slots are established.

The “DAC_settings_linearity” signal is emitted by the “linearity_window” with the parameters (int, QString, QString) and is connected to the “AD5544(int , QString , QString)” function in the “Mojoserial” class to allows for communication between these components and to configure the DAC settings.

The “reset_linearity_flag” signal from the “linearity_window” is connected to the “on_reset_clicked()” function. This connection is used to reset flags and settings related to the last configuration of DAC channels in the “linearity_window”.

The “ADC_settings” signal emitted by the “linearity_window” is connected to the “ADC_setting()” function. This connection is used to configure ADC settings.

```
    void MojoSerial::on_linearity_test_clicked()
{
  if (!linearity_window)
      linearity_window = new linearity_test(this);

   lesen=false;
   flag_measure=false;
   polarization_flag=false;
   polarizer_sweep_flag=false;
   linearity_test_flag=true;
   linearity_window ->show();
   connect(linearity_window,SIGNAL(DAC_settings_linearity(int,QString,QString)),this,SLOT(AD5544(int ,
QString , QString )));
   connect(linearity_window,SIGNAL(reset_linearity_flag()),this,SLOT(on_reset_clicked()));
   connect(linearity_window,SIGNAL(ADC_settings()),this,SLOT(ADC_setting()));
}
```

Source Code 9.31 Definition of the “on_linearity_test_clicked()” function within the “MojoSerial” class

The purpose of the “ADC_setting()” function is to configure the program for reading the voltage values from all ADCs in normal mode. For this purpose, a QString variable named “ADCAddress” is assigned the value “11110110”. Then, the “sendADCdata()” function of the “Kommunikation” object is invoked with “ADCAddress” as an argument to send data to the FPGA related to the specified ADC address.

```
    void MojoSerial::ADC_setting() {
      QString ADCAddress="11110110";
      Kommunikation->sendADCdata(ADCAddress);
```

```
}
```

Source Code 9.32 Definition of the “ADC_setting()” function within the “MojoSerial” class

9.2.2.10.2 “on_polarization_measurement_clicked()” function

This function serves the purpose to enable the DAC selection to control the Gain and and Offset of the selected channel, as well as to simplify the interaction with the polarizer and to set a specific polarization angle. Additionally, it handles the retrieval of output signals from all ADC channels. The function begins by verifying the existence of a “polarization_window”. If this window is not yet instantiated, a new instance of the “polarization_measurement” class is created and linked to the “polarization_window” pointer. This link establishes a connection between the main window and the newly created instance. By setting the “polarization_flag” to true and concurrently setting other flags to false, the functionality for signal reading within this function becomes operational. Next the GUI window associated with the “polarization_window” instance is made visible to the user showing interface elements for controlling and monitoring polarization-related measurements.

The connect lines establish connections between signals emitted by the “polarization_window” instance and corresponding slots (functions) in the “MojoSerial” class. These connections allow communication between the GUI components and the underlying logic of the program. When certain actions occur in the GUI such as changing DAC or ADC settings, the corresponding slots will be executed.

The first connect statement connects the signal named “DAC_settings” with three parameters (an integer and two QStrings) from the “polarization_window” to the slot named “AD5544” in the “MojoSerial” class to set the selected DAC with the specified settings.

The second connect statement establishes a connection between a signal named “ADC_settings” emitted by the “polarization_window” and a slot named

“ADC_setting” within the "MojoSerial" class. This connection is made to specify the specific readout signals of the ADCs that need to be read and processed. The source code of this function is as follows.

```
  if (!polarization_window)
      polarization_window = new polarization_measurement(this);

   lesen=false;
   linearity_test_flag=false;
   flag_measure=false;
   polarizer_sweep_flag =false;
   angle_calculation=false;
   polarization_flag=true;
   polarization_window ->show();
   connect(polarization_window,SIGNAL(DAC_settings(int,QString,QString)),this,SLOT(AD5544(int , QString ,
QString )));
   connect(polarization_window,SIGNAL(ADC_settings()),this,SLOT(ADC_setting( )));

}
```

Source Code 9.33 Definition of the “on_polarization_measurement_clicked()” function within the “MojoSerial” class

9.2.2.10.3 “on_polarizer_sweep_clicked()” function

This function is responsible for displaying the readout voltage of the chosen channel with specified gain and offset while sweeping the polarizer.

It creates a new instance of the “polarizer_sweep” class and assigns it to “polarizer_test_window” if it does not already exist. The function also initializes various flags (lesen, linearity_test_flag, flag_measure, polarization_flag, polarizer_sweep_flag, and angle_calculation) that are essential for the polarizer sweep functionality to ensure that the readout data will pass to this window. It shows the “polarizer_test_window” by calling the “show()” function. This action makes the window visible to the user.

It establishes signal-slot connections using the “connect()” functions. These connections allow objects to communicate with each other when certain events or signals occur. The first connect statement connects a signal from the “polarizer_measurement” class to a slot named “AD5544(int, QString, QString)”

in the “MojoSerial” class to configure the selected DAC with the specified settings of Gain and Offset.

The second connect statement establishes a connection between a signal named “ADC_setting()” emitted by the “polarizer_measurement” class to a slot named “ADC_setting()” within the "MojoSerial" class to configure ADC settings.

The third connect statement is used to reset flags and settings related to the last configuration of the “polarizer_measurement”. It connects the signal “reset_polarizer_flag()” to the slot “on_reset_clicked()” and resets all configurations to default values. The last connect statement connects the “movement_completed()” signal from the “polarizer_measurement” class to the “polarizer_settled()” slot to perform actions when the polarizer movement is completed. The source code of this function is as follows.

```
void MojoSerial::on_polarizer_sweep_clicked()
{
   if (!polarizer_test_window)
       polarizer_test_window = new polarizer_sweep(this);

    lesen=false;
    linearity_test_flag=false;
    flag_measure=false;
    polarization_flag=false;
    polarizer_sweep_flag=true;
    angle_calculation=false;

    polarizer_test_window ->show();

connect(polarizer_test_window,SIGNAL(signal_DACsettings_polarizer(int,QString,QString)),this,SLOT(AD5544(i
nt , QString , QString )));
    connect(polarizer_test_window,SIGNAL(ADC_settings()),this,SLOT(ADC_setting()));
    connect(polarizer_test_window,SIGNAL(reset_polarizer_flag()),this,SLOT(on_reset_clicked()));
    connect(polarizer_test_window, SIGNAL(movement_completed()), this, SLOT(polarizer_settled()));

}
```

Source Code 9.34 Definition of the “on_polarizer_sweep_clicked()” function within the “MojoSerial” class

9.2.2.10.4 "on_angle_calculation_clicked()" function

The "on_angle_calculation_clicked()" function is responsible for initializing and showing the "Angle_Calculation" window, setting up and configuring the "Angle Calculation" mode of the application, connecting signals to appropriate slots for handling user interactions and events within that mode.

A new instance of the "Angle_calculation" class is created if it does not already exist. To ensure the readout data transfer to this window, the "angle_calculation" flag is set to true, while other flags associated with different windows are set to false. The "connect()" functions allow objects to communicate with each other when certain signals occur. The first connect statement connects the "signal_DACsettings_angle(int,QString,QString)" signal to the slot "AD5544(int, QString, QString)". It is used to configure the selected DAC with the specified settings of gain and offset. The second connect statement is used to reset flags, variables and all configurations.

The third connect statement is used to stop continuous mode during angle calculations. This is achieved by emitting the "stop_cont()" signal from the "Angle_calculation" class and establishing a connection to the "stopcontmode()" slot within the "MojoSerial" class by calling the "stopreadcont()" function of the Kommunikation object.

The next connect () function is used to ensure that the polarizer is settled and that the signals can be read. It establishes a connection between the "movement_completed()" signal emitted by the "Angle_calculation" class and the "polarizer_settled()" slot within the current class. This connection indicates that when the polarizer movement is completed (settled), the "polarizer_settled()" slot should be executed, allowing signal readings to take place.

The last connect statement is responsible for configuring the system differently based on whether the "normal" or "continuous" mode is selected, including settings related to data acquisition and communication. It establishes a connection

between the “mode_set(QString)” signal emitted by the “Angle_calculation” class and the “defined_mode(QString)” slot within the “MojoSerial” class.
The “defined_mode(QString mode)” slot is responsible for handling the defined mode based on the input mode parameter. If the mode is set to "normal" it sets the “set_normalmode” flag to true, “set_contmode” flag to false, and calls the “ADC_setting()” function for configuring ADC settings and switching to operate in normal mode. If "continuous" mode is selected, the system is configured for continuous or streaming data acquisition by setting the “set_normalmode” flag to false and “set_contmode” flag to true. Also the “startreadcont()” function is used to initiate continuous data reading.The source code of this function is as follows.

```
void MojoSerial::on_angle_calculation_clicked()
{

    if (!angle_calculation_window)
        angle_calculation_window = new Angle_calculation(this);

    // Initialize flags and states
    angle_calculation=true;
    lesen=false;
    flag_measure=false;
    lightsource_flag=false;
    polarization_flag=false;
    linearity_test_flag=false;
    polarizer_sweep_flag =false;

    // Show the Angle Calculation window
    angle_calculation_window ->show();

    // Connect signals and slots
  connect(angle_calculation_window,SIGNAL(signal_DACsettings_angle(int,QString,QString)),this,SLOT(AD55
  44(int , QString , QString )));
  connect(angle_calculation_window,SIGNAL(reset_polarizer_flag()),this,SLOT(on_reset_clicked()));
  connect(angle_calculation_window,SIGNAL(stop_cont()),this,SLOT(stopcontmode()));
  connect(angle_calculation_window,SIGNAL(movement_completed()), this, SLOT(polarizer_settled()));
  connect(angle_calculation_window,SIGNAL(mode_set(QString)),this,SLOT(defined_mode(QString)));

}
```

Source Code 9.35 Definition of the “on_angle_calculation_clicked()” function within the “MojoSerial” class

9.2.2.10.5 “on_light_source_test_clicked()” function

This function shows a window dedicated to light source testing. It configures flags for controlling the passage of intensity signals to “Light Source Test” window, displays the window, and establishes connections to effectively handle events or signals emitted by the "light_source_test" class.

It checks if the “lightsource_window” object already exists. If it does not, it creates a new instance of the “light_source_test” class. To ensure that the intensity signals pass to the “Light Source Test” window, it sets the “lightsource_flag” flag to true.

The “connect()” functions facilitate communication between the “light_source_test” class and the “MojoSerial” class when specific signals occur. The signals emitted from the “light_source_test” class are utilized for configuring DAC settings and ADC settings, respectively. The source code of this function is as follows.

```
void MojoSerial::on_light_source_test_clicked()
{
  if (!lightsource_window)
      lightsource_window = new light_source_test(this);

    lesen=false;
    linearity_test_flag=false;
    angle_calculation=false;
    flag_measure=false;
    polarizer_sweep_flag =false;
    polarization_flag=false;
    lightsource_flag=true;
    lightsource_window ->show();


connect(lightsource_window,SIGNAL(signal_DACsettings_lightsource(int,QString,QString)),this,SLOT(AD5544(in
t , QString , QString )));
    connect(lightsource_window,SIGNAL(ADC_settings()),this,SLOT(ADC_setting()));

 }
```

Source Code 9.36 Definition of the “on_light_source_test_clicked()” function within the “MojoSerial” class

9.2.2.10.6 “on_WL_Sweep_clicked()” function

This function is responsible for initiating a window dedicated to wavelength sweep testing. To accomplish this, it first checks if the “WL_sweep_window” object already exists; if not, it creates a new instance of the “Wavelength_sweep” class associated with this window. Subsequently, it sets the “WL_sweep_flag” to true and other relevant flags to false to control the passage of readout data to specific windows, and then displays the “WL_sweep_window”. Additionally, it establishes connections to manage signals emitted by the “Wavelength_sweep” class, such as configuring DAC settings and ADC settings. The source code of this function is as follows.

```
void MojoSerial::on_WL_Sweep_clicked()
{
  if (!WL_sweep_window)
     WL_sweep_window = new Wavelength_sweep(this);

  WL_sweep_flag=true;

  lesen=false;
  flag_measure=false;
  WL_Pol_flag=false;
  lightsource_flag=false;
  polarization_flag=false;
  angle_calculation=false;
  linearity_test_flag=false;
  polarizer_sweep_flag =false;
  WL_sweep_window ->show();

connect(WL_sweep_window,SIGNAL(signal_DACsettings_angle(int,QString,QString)),this,SLOT(AD5544(int
, QString , QString )));
  connect(WL_sweep_window,SIGNAL(ADC_settings()),this,SLOT(ADC_setting()));
}
```

Source Code 9.37 Definition of the “on_WL_Sweep_clicked()” function within the “MojoSerial” class

9.2.2.10.7 “on_WL_Polarizer_Sweep_clicked()” function

This function serves to initialize a window specifically designed for wavelength and polarizer sweep testing. To achieve this, it first checks whether the “WL_Polarizer_window” object exists. If not, it creates a new

instance of the “Wavelength_polarizer_sweep” class associated with this window. Subsequently, it sets the “WL_Pol_flag” flag to true and other relevant flags to false, controlling the passage of readout data to specific windows, and then displays the “WL_Polarizer_window”. Additionally, it establishes connections to handle signals emitted by the “Wavelength_polarizer_sweep” class, including configuring DAC settings, ADC settings, and detecting when polarizer movement is completed. The source code of this function is as follows.

```
void MojoSerial::on_WL_Polarizer_Sweep_clicked()
{

   if (!WL_Polarizer_window)
       WL_Polarizer_window = new Wavelength_polarizer_sweep(this);

   WL_Pol_flag=true;
   lesen=false;
   flag_measure=false;
   lightsource_flag=false;
   polarization_flag=false;
   angle_calculation=false;
   linearity_test_flag=false;
   polarizer_sweep_flag =false;

   WL_Polarizer_window ->show();

  connect(WL_Polarizer_window,SIGNAL(signal_DACsettings_angle(int,QString,QString)),this,SLOT(AD5544(int ,
QString , QString )));

   connect(WL_Polarizer_window,SIGNAL(ADC_settings()),this,SLOT(ADC_setting()));
   connect(WL_Polarizer_window, SIGNAL(movement_completed()), this, SLOT(polarizer_settled()));
}
```

Source Code 9.38 Definition of the “on_WL_ Polarizer _Sweep_clicked()” function within the “MojoSerial” class

9.2.3 The “Angle_measurement” class

The “Angle_measurment” class is designed to display and update real-time angle measurement data in a QwtPlot widget while also providing functionality for starting and stopping continuous data reading. below is an explanation of the actions carried out by this class:

- **Constructor**: In the constructor, the class initializes the user interface defined in ui_angle_measurment, creates a QwtPlot widget for real-time plotting of angle data, and sets up various properties of the plot, such as its title and axis labels. It also creates a QTimer named timer_angle and connects its "timeout()" signal to two slots: "realtimeplot()" and "updateLDC()".
- **Destructor:** the destructor cleans up and deletes the UI.
- **closeEvent() Function:** this function is called when the user closes the dialog. It clears the data vectors x, y1, and y2, and emits a signal named "reset_angleform()" before closing the dialog. This signal is connected to the "MojoSerial" class, where it triggers a corresponding function responsible for stopping the continuous data reading process.
- **on_readcont_clicked() Function:** this function is called when a button with the object name "readcont" is clicked to start continuous data reading. It calls a function "send_startcontdata()" and starts the "timer_angle" to update the plot at regular intervals.
- **on_stopcont_clicked() Function:** this function is called when a button with the object name "stopcont" is clicked to stop continuous data reading. It calls a function "send_stopcontdata()" and stops the "timer_angle", effectively stopping real-time plotting.
- **calculate_measurement(int angle90_0, int angle135_45) Function:** this function is responsible for converting angle values from their raw representation (radians) to degrees. It takes two parameters: "angle90_0" and "angle135_45" and calculates the corresponding angles in degrees, storing them in the variables "angle90" and "angle135".

- **updateLDC() Function:** this function updates the values displayed on LCD widgets (“ui->lcdNumber_angle135” and “ui->lcdNumber_angle90”) with the current angles in degrees. It uses the “angle90” and “angle135” variables to update the display.
- **Realtimeplot() Function:** this function handles real-time plotting of angle data. It is connected to the “timer_angle” and is called at regular intervals. It updates a QwtPlot widget with the current angle data, ensuring that the plot is continuously updated as new data becomes available. The function creates two plot curves (blue and red) for “angle90_0” and “angle135_45” and appends new data points to the curves for real-time visualization.

9.2.4 The “linearity_test” class

This class is designed for evaluating the linearity of the system by configuring DAC channels, data acquisition, and plotting the acquired data in real-time. It also handles user interactions and provides the ability to reset and reconfigure the test parameters. Additionally, it manages communication with other classes through signals and slots.

- **Constructor:** the class constructor initializes the user interface and the Qwtplot widget required for plotting data. It sets up a timer (DACsettings_timer) to periodically trigger the “TimerSlot_DACsettings()” function.
- **closeEvent(QCloseEvent *event):** when the user closes the dialog, this function is called. It stops the timer (DACsettings_timer) and clears the data vectors (x and y). It emits a signal “reset_linearity_flag()” to perform some action in the “MojoSerial” class. Finally, it closes the dialog.

- **on_stopBtn_clicked():** this function pauses the ongoing process by stopping the timer. It also facilitates the possibility of resuming the process by storing the current DAC settings, sweep type, which can be either "Gain" or "Offset".
- **on_runBtn_clicked():** this function is executed upon clicking the “Run” button and ensures that all essential parameters are correctly configured before initiating a sweep operation. It includes checks and warnings to assist users in configuring the test effectively. The function begins by checking the input fields for start, stop, and step values. It verifies whether these fields are empty, and if any of them is empty, a warning message is displayed using the “msgbox->warning()” mechanism. This serves as a prompt to the user to provide the required values for these fields, ensuring that the “sweep_configuration()” function can proceed with valid parameters. This function is responsible for extracting and configuring the essential sweep parameters and preparing them for the subsequent operation. In addition to validating sweep parameters, the function undertakes a critical examination of the “Gain/Offset” box. It verifies whether any of the channels within this box are selected and whether values are provided for the selected channels. If none of the channels are chosen or if the corresponding text fields remain empty, the function responds with a warning message. This message urges the user to select at least one channel and input the necessary values for the selected channels. Following the channel selection and validation process, the function proceeds to execute the “Gain_Offset_DACsetting()” function. This function plays a pivotal role in extracting and configuring the DAC settings based on the selected channels and their associated values. It ensures that the DAC settings align with the user's specified preferences for the sweep

operation. An additional task of the function is to ascertain the sweep type selection. This is accomplished by examining the states of the radio buttons associated with the sweep type, typically labeled as "ui->gainBtn" and "ui->offsetBtn". If neither of these radio buttons is selected, the function intervenes by displaying a warning message. This message prompts the user to make a clear choice regarding the sweep type, which can be either "Gain" or "Offset." When a sweep type is chosen, the function initiates the "sweep_type()" function to configure the sweep type and its associated settings. In the final execution, stages it checks whether any of the essential parameters, such as DAC settings, sweep type, or start value have changed. If any of these parameters have changed, the function responds by clearing the existing data vectors (commonly referred to as x and y), resetting the sweep counter, and reconfiguring the DAC settings to match the updated parameters.

Upon successfully completing all these validation checks, the function proceeds to the culmination of its task. It ensures that the necessary conditions, represented by "gain_offset_flag", "sweep_flag", and "configuration_DAC_flag" are met. When these conditions align, the function starts the "DACsettings_timer" timer. This timer serves as the mechanism to initiate the sweep process, ensuring that it commences only when all essential parameters are properly configured, thereby safeguarding the accuracy and reliability of the test.

- **Gain_Offset_DACsetting():** this function is responsible for configuring and setting DAC values based on user selections and input. The function first checks the state of various radio buttons (e.g., "ui->p0", "ui->p90") to determine which channels are selected for DAC configuration. For each selected channel, it sets specific binary values (e.g., "p0," "p90") and binary addresses based on the channel's position.

then it checks if corresponding text fields (e.g., “ui->textP0”) contain values. If a field is empty when the channel is selected, a warning message is displayed via “msgbox->warning,” and the “gain_offset_flag” is set accordingly. then it concatenates the binary representations of all selected channels into the “new_DAC_setting” variable. If “new_DAC_setting” is empty or contains all zeros warning is displayed otherwise the DAC value is calculated as a percentage (“DAC_value_f”) based on the user-provided DAC value.

- **Sweep Configuration ():** this function is used to read and process input values the from user interface (UI) and prepare them for a linearity test.

It reads the text values (“start_value”, “stop_value” and “step_value”) from the UI and converts them to an integer using the “toInt()” method. Then the function scales them to a floating-point value in the range of [0, 100] by dividing it by 65535, and then the result is multiplied by 100. These scaled values are stored and rounded to integer values. This ensures that the value remains an integer, depending on the requirements of the test. Finally, to indicate that the configuration process has been completed the “configuration_DAC_flag” flag is set to true.

- **sweep_type() :** this function determines whether the sweep is a “Gain” or “Offset” sweep based on user selections and updates relevant variables accordingly.

The sweep type is determined by inspection of the radio buttons in the user interface. If the “gainBtn” radio button is checked, the “gain_sweep” boolean variable is set to true by means of “isChecked()” function while marking the “offset_sweep” variable as false. This signifies that the test will be a “Gain” sweep. If, alternatively, the “offsetBtn” radio button is checked, it sets “offset_sweep” to true, indicating an “Offset” sweep.

Subsequently, the value of “DAC_value_f” is computed by rounding the floating-point value. The “sweep_flag” boolean variable is then set to true, signifying the completion of test configuration for the sweep. Lastly, the “new_sweep_type” string variable is assigned the value "Gain" or "Offset" to specify the type of sweep.

- **TimerSlot_DACsettings():** this function is used for controlling and configuring DACs for a sweep operation and is periodically called by the timer. Depending on the values of “gain_sweep” and “offset_sweep”, it configures the DACs with specific values and increments the counter until a stopping condition is met.

Firstly, an “ADC_settings()” signal is emitted to perform ADC settings, and this signal is received in the “MojoSerial” class. Then the type of operation is controlled by checking the boolean variables “gain_sweep” and “offset_sweep”.

The “DAC_settings_linearity()” signal is emitted to configure DAC channels, and like the “ADC_settings()” signal, it is received in the “MojoSerial” class. Depending on whether “gain_sweep” or “offset_sweep” is selected, the appropriate parameters are sent. For instance, if “gain_sweep” is chosen, it use the function “DAC_settings_linearity(offset_value, address, offset_channel)” to configure the DAC. Then, by monitoring the “counter” variable, it calls the function “DAC_settings_linearity(counter, address, gain_channel)” to configure the DAC required for sweeping.

For the DAC sweeping the counter is incremented by “step_value” until it reaches the “stop_value”. Subsequently, the counter is reset to “start_value” and both “gain_sweep’ and “offset_sweep” are set to false, indicating the completion of the sweep operation.

- **update_values():** this function receive the readout values, updates the "readout_value" variable based on the user's selection channel and then triggers "plot()" function to reflect the selected value in a graphical representation.
- **Plot():** this function is responsible for creating a curve plot on a QwtPlot widget. The x-axis represents the step values, and the y-axis represents the readout values. Data points are continuously added to the curve, creating a visual representation of the relationship between step values and readout values over time.
- **on_reset_clicked():** this function is triggered when a "Reset" button is clicked in the user interface. It resets various variables and user interface elements to their initial or default values, allowing the user to start a new or reconfigure the application.
- **reset_DACsetting():** this function is responsible for resetting the DAC settings for different channels and configurations to their initial values. It emits the "DAC_settings_linearity()" signal with specific parameters for each channel and configuration to set the DAC values to 0. By setting these DAC values to 0, it effectively resets the DAC channels to their initial states, ensuring that subsequent operations start with the default configurations.

9.2.5 The "Angle_calculation" class

The purpose of this class is to create a GUI application that can interface with a KDC101 device to control and monitor the polarizer angle, perform angle calculations and measurements in software in normal mode or calculate angles in the hardware with continuous mode. Also, it displays angle data in real-time using a Qt-based GUI with a plot. It provides a user-friendly interface for configuring settings, initiating measurements, and

visualizing measurement results in real-time. Additionally, it logs measurement data for further analysis.

- **Constructor:** the constructor sets up the basic components, timers, and signal-slot connections needed for the application.

  Firstly, a QDialog is created to serve as the primary user interface. It was designed using the “Ui::Angle_calculation” class via Qt Designer. Additionally, essential components are initialized, including an instance of the “KDC101” class for interfacing with the external hardware, a user messaging QMessageBox named msgbox, and a visual angle data display QwtPlot widget named qwtPlot. The qwtPlot widget is customized with dimensions, visual attributes like background color and grid lines, and axis titles and scales. To enable time-based actions, three timers are set up with specific intervals in milliseconds, facilitating scheduled tasks. Furthermore, signal-slot connections establish links between various objects: “timer_NormalMode” is connected to the “timerSlot_settings” slot, the KDC101 object's “move_completed” signal connects to the “polarizer_movement()” slot, and “timer_ContMode” is associated with the “contMode_sampling” slot. these functions get executed when the “timeout ()” signal of related timer is emitted. Finally, the KDC device is initialized by opening it for communication and identifying its properties, signifying the completion of the setup phase for the application.

- **Destructor:** the destructor is responsible for performing cleanup operations related to the destruction of an “Angle_calculation” object. It ensures that any resources associated with the object are properly released before the object is removed from memory.

- **timerSlot_settings():** this function emits signals responsible for configuring settings related to the "Gain" and "Offset" of the selected DAC, as well as setting the operation mode.
- **polarizer_movement():** this function begins by emitting a signal named “movement_completed()” to indicate that the polarizer's movement process has finished, and it's now time to read out the data. Subsequently, it checks whether the continuous mode is active. If continuous mode is enabled, it proceeds to invoke the “timerSlot_settings()” function, which manages various settings associated with the polarizer's movement.
- **closeEvent(QCloseEvent *event):** this function is responsible for handling the closure of the application's window and performs various cleanup and reset operations when the application's window is closed.
- **on_stopBtn_clicked():** this function is used to stop various processes associated with normal and continuous modes of operation and to reset relevant variables when the “Stop” button is clicked. Additionally, the polarizer is moved to a specified start position.
- **on_reset_clicked():** this function is associated with a click event, of the button labeled “Reset” and performs a comprehensive reset of various parameters, settings, and data storage variables in the application, effectively clearing and initializing them to default or initial values.
- **reset_DACsetting():** It is responsible for resetting DAC settings to their default or zero values for various channels.
- **on_homeBTN_clicked():** by clicking the “Home” button in the user interface a process is initiated to move the polarizer controlled by the “kdc” object to a position specified by the “start_pos” value. This action is often used to reset or return a device to a predefined reference position.

- **on_normal_modeBTN_clicked():** this function is associated with a button click event labeled "Normal Mode". Its primary objective is to configure and initialize various parameters and flags required for normal operation mode. Additionally, it verifies the user input, specifically the "averaged_number" field, to ensure proper configuration. In cases where the user fails to provide a non-zero value, a warning message is displayed to prompt the user for a valid input.
- **on_cont_modeBTN_clicked():** this function is associated with a button click event labeled "Continuous Mode". Its purpose is to set various operational flags and parameters to enable continuous mode while disabling normal mode. It also clears data collections and calls a configuration function to prepare the application for continuous operation.
- **configuration_function():** the "configuration_function()" is responsible for the configuration of parameters and flags based on the user inputs. It checks if a channel is selected and validates the "Gain" and "Offset" values. Then, it calculates the DAC settings within the valid range. Next, it calls the "polarizer_sweep_configuration()" function to prepare data collection and initializes various variables for further data acquisition and processing. It sets flags for successful configuration and triggers related timers based on the selected mode (normal or continuous).
- **polarizer_sweep_configuration():** this function prepares the application to perform polarizer sweeps using user-specified start, stop, and step values. It calculates the necessary step count for the sweep and activates a flag to confirm the successful configuration.
- **DACsettings():** this function configures the DAC settings based on the user's selection of channels (P0, P90, P45, P135, I0, I1, I2, I3) for signal generation. It sets appropriate binary values for the selected channels,

specifies the DAC address, offset channel, and gain channel accordingly. The function then combines these settings into a single binary string, representing the new DAC configuration. Finally, it sets a flag to confirm that DAC settings have been successfully configured.

- **realtimeplot():** this function is responsible for real-time plotting and logging of angle measurements during continuous mode operation, on a QwtPlot widget. It initializes two QwtPlotCurve objects in case they do not exist. It configures their appearance and attaches them to the QwtPlot widget. The function appends new data points (angles) to the respective data series and updates the curves with the new data. After updating the plot, it checks if the specified number of steps has been completed. If so, it stops the continuous mode and schedules a delay before moving to the initial position. If not, it continues with the next step. Additionally, the function records angle measurements to a text file for data logging.

- **plot():** this function is responsible for plotting and updating real-time data on a QwtPlot widget. It initializes two QwtPlotCurve objects for displaying data series related to angles "angle90_0" and "angle135_45" during normal mode operation which are calculated within the software. The function then appends new data points to the respective data series and updates the curves with the new data. After updating the plot, it saves the "angle90_0" data to a text file.

- **step_delay() :** this function is designed to introduce controlled delay between each step of a polarizer sweep, allowing for precise measurements or adjustments to be made. It calculates the new position based on the start position, the step position and current step count. After determining the new position, it instructs the polarizer to move to this position.

- **contMode_sampling():** this function is part of a continuous mode operation. It is responsible for managing the steps involved in the continuous mode of data sampling.
- **calculate_angles (float, float, float, float):** this function is used to calculate angles based on voltage measurements in normal mode and involves sampling and angle calculation phases. It computes angles based on voltage differences.

If the “sampling” flag is true, the measurement is in the sampling phase. The code then aims to find maximum and minimum values by sweeping a polarizer from 0 to 180 degrees. It checks if “averaged_number” is set to 1. If so, it finds max/min values and moves the polarizer step by step to sample the values. If “averaged_number” is greater than 1, the code performs multiple iterations, which helps in collecting averaged data for better accuracy. Once all steps are completed, the code sets “sampling” to false, resets the “curr_steps” counter, and moves the polarizer back to the initial position.

In the angle calculation phase, “sampling” flag is false and the code calculates angles based on measured voltage differences (“I90_0” and “I135_45”) and the maximum and minimum values obtained during the sampling phase. It computes “Arg90_0” and “Arg135_45” based on these values which represent angles. The code determines the sign of “Arg90_0” and “Arg135_45” and stores them in the variables “sign90_0” and “sign135_45” respectively. Finally, it calls the “define_region” function for further processing based on the calculated angles.

- **find_Max_Min():** this function finds and updates the maximum (Max) and minimum (Min) values of the two variables, “I90_0” and “I135_45”. It does so by comparing the current values of these variables with the existing Max and Min values:

If “I90_0” is greater than the current Max90_0 value, it updates Max90_0 with the value of “I90_0” ensuring that Max90_0 always stores the maximum value encountered during the sampling process. If “I90_0” is less than the current Min90_0 value, it updates Min90_0 with the value of “I90_0” ensuring that Min90_0 always stores the minimum value encountered during the sampling process.

The same logic is applied to “I135_45” and the corresponding Max135_45 and Min135_45 variables. This function ensures that Max and Min values are updated correctly as data is sampled, allowing subsequent angle calculations to be based on the correct maximum and minimum values.

- **define_region(float, bool, float, bool):** this function aims to define regions and calculate angles based on the arcsine values, signs, and certain conditions. It categorizes the angles into different regions and computes angle values in degrees:

The angles are then converted from radians to degrees. After plotting the data, the function checks if there are more steps to be processed. If so, it calls “step_delay()” to calculate the next position. If all steps are completed, it stops the timer, moves the polarizer to the initial position, and resets the “curr_steps” counter.

9.2.6 The “hyperchromator” class

This class is designed to manage the operation of the Hyperchromator light source connected through a serial port. It offers control over various functionalities, including wavelength configuration, speed adjustment, filter control, and position reading. To interact with the Hyperchromator, the class employs specific TML commands documented in [23]. The commands are not sent directly as ASCII strings but encoded into binary command strings for proper communication with the device. Through the

functions described below, this class offers essential control and interaction with the Hyperchromator device via the serial port, allowing users to configure settings and retrieve information.

- **Constructor**: Initializes some variables and connects the “read_serial()” function to the “readyRead()” signal of a serial port. This signal is emitted when data is available for reading from the serial port.
- **read_serial():** Handles reading data from the serial port. It reads data in chunks and processes it to extract relevant information, such as position values. It also manages the handling of incomplete messages.
- **open(QString cal_path):** Opens the serial port and reads calibration data from a file specified by “cal_path”. It sets up the serial port with specific settings and initiates other setup actions.
- **initSpeed():** Sets the initial speed for the device. It is part of the initialization process.
- **initWL():** Sets the initial wavelength for the device. It's also part of the initialization process.
- **close():** Closes the serial port when done using the device.
- **openShutter():** Sends a command to open a shutter, possibly controlling some optical path.
- **closeShutter():** Sends a command to close the shutter.
- **setWL(QString wl_input_str):** Sets the wavelength for the device based on an input string. It converts the desired wavelength to device-specific position values and sets the position.
- **getPos(float wl):** Retrieves the position associated with a given wavelength from calibration data.

- **getWL():** Retrieves and prints the current wavelength.
- **setPos(float pos):** Sets the device's position based on the provided position value. It converts the position value to a byte array and sends it to the device.
- **updatePos():** Sends a command to update the position.
- **readPos():** Sends a command to read the current position of the device.
- **reset():** Sends a command to reset the device.
- **setSpeed(QString speed_str):** Sets the speed of the device based on the input speed value.
- **filter1(), filter2(), filter3(), filter4():** Each of these functions sends a specific command to the device to select a filter.
- **increment(), decrement():** These functions send commands to increment or decrement offsets.
- **clear():** Sends a command to clear offsets.

9.2.7 The "kdc101" class

This class is an interface for controlling a KDC101 device through a serial port, and it's structured to handle different commands and responses from the device. The class includes several functions which perform various tasks related to configuring and controlling the device. Each function sends specific commands to the device via the serial port. The commands are explained in detail in Appendix B.

- **Constructor**: This constructor sets up initial values for the class. It connects a slot function ("read_serial") to the "readyRead" signal of a "serial_port" object for handling incoming data from the serial port. When data becomes available in the serial port's input buffer (e.g., data received

from the KDC101 device), the “readyRead()” signal is emitted. The connected “read_serial()” slot is then executed to read and process this data.

- **Open():** This function initializes and opens the serial port for communication with the device. It looks for the correct USB port based on vendor and product identifiers. If the port is successfully opened, it sets the communication parameters like baud rate and flow control.
- **read_serial():** This slot function is triggered whenever there is data available to read from the serial port. It reads data from the port and processes it based on certain message types (e.g., confirming homing, reading positions, getting velocity parameters). The purpose is to ensure that data is correctly received, processed, and actions are taken accordingly, all in an asynchronous and event-driven manner.
- **move_rel() and move():** These are movement functions which are used to move the stage to a specified relative or absolute positions in degrees, respectively by converting it to encoder counts. They construct the appropriate command messages and send them to the device through the serial port.
- **step_delay():** This function is used in a scenario where the KDC101 device is supposed to perform a sequence of movements with a specified step size and a step delay in between. It calculates the next position to move to, initiates the movement, and increments the step count for subsequent steps.
- **identify():** The “identify()” function sends an identification message or command to the KDC101 device, requesting the device to provide some form of identification or status information.
- **enable():** This function is responsible for enabling the drive channel of the KDC101 device. The “enable()” function sends a command to the KDC101

device to enable its drive channel, which allows the device to start operating or moving as needed.

- **home():** The "home()" function initiates the homing process of the KDC101 stage by sending a specific command through the serial port. Homing is essential to establish a known reference point for subsequent positioning operations. Every time the device is turned on it is used "home()" after "open()" the port.
- **req_pos():** It is responsible for requesting the position of a KDC101 stage by sending a specific command message to the device through the serial port. It also checks for any write operation failures and provides debugging information if necessary.
- **Jog():** This function is used to perform a jogging operation on the KDC101 stage. It takes the initial and final positions, step size, delay, and speed as input parameters, and then it calculates the number of steps required to reach the final position. It initiates the movement and increments the position until the final position is reached. Debugging messages are provided for tracking the progress of the operation.
- **set_vel():** This function is used to set the velocity parameters of the KDC101 stage. It takes the maximum velocity as input, calculates the raw values for velocity-related parameters, converts them to the appropriate format, and sends them to the device. The function is designed to ensure that the parameters are correctly formatted and scaled before transmission.
- **req_vel():** This function is used to send a request for velocity parameters to the KDC101 stage. It constructs a specific message and sends it to the device via the serial port. If the write operation fails, it provides debugging information, and if successful, it confirms that the request has been sent.

- **Disable():** This function is used to send a command which disables the drive channel of the KDC101 stage. It constructs a specific message and sends it to the device via the serial port. If the write operation fails, it provides debugging information, and if successful, it confirms that the device has been disabled.
- **close():** This function is responsible for closing the serial port associated with the KDC101 device. It ensures that the port is properly closed, and it provides a message for debugging and tracking purposes to indicate that the port closure was successful.

9.2.8 The "light_source_sweep" class

This class serves as the control center for a GUI application that allows users to control hardware components (e.g., Hyperchromator, KDC101) and set up data acquisition parameters for a wavelength sweep experiment. It provides an interface for configuring and starting the experiment while visualizing acquired data in real-time using a QwtPlot.

- **Constructor:** The constructor initializes the class. It sets up the user interface based on the UI defined in the "light_source_sweep.ui" file, creates instances of other classes such as Hyperchromator, KDC101, and QMessageBox. It also initializes a QwtPlot for data visualization.
- **Destructor:** The destructor performs cleanup when the object is destroyed, such as deleting the user interface and stopping timers.
- **closeEvent():** The closeEvent function is an event that is triggered when the user attempts to close the "light_source_sweep" window. In this function, several actions are taken to ensure that resources are properly released, and the application is prepared for closure.This includes stopping timers, closing hardware components, and clearing data vectors to ensure a clean exit.

- **Button Click Handlers:** The class contains slots that respond to button clicks from the user interface. These buttons click handlers perform various actions, including opening/closing hardware components (HC->open, HC->close, HC->openShutter, HC->closeShutter), configuring the light source with selected filters (HC->filter1(), HC->filter2(), HC->filter3() and HC->filter4()).

  When the “openBut” button is clicked, this function initiates the opening or initialization of the hyperchromator device using the Hyperchromator object “HC”. It does so by providing a path to a configuration file required operate correctly.

- **Configuration Functions:** Functions like “polarizer_sweep_configuration()” and “DAC_settings()” handle setting up parameters for a polarizer sweep and configuring a DAC based on user input.

- **on_setWLBut_clicked():** this function is intended to perform an action related to setting the wavelength and set the “WL_set” flag to true, which is used to track the state of the wavelength setting.

- **on_readvoltageBtn_clicked():** when the “readvoltageBtn” button is clicked, this function checks if the wavelength has been set (“WL_set” is true), and if so, it emits a read() signal to initiate a reading operation. This signal is received and processed in the “MojoSerial” class.

- **on_read_plot_clicked():** this function handles user input and settings related to a light source and polarizer configuration. When the user clicks the “read_plot” button, it ensures that all required settings are provided. It validates the user input for gain and offset settings and it configures the polarizer sweep based on user-defined start, stop and step values. If conditions are met, it initiates the polarizer sweep operation using the

"kdc" object and starts a timer ("timer_lightsource") to read data periodically for plotting.

9.2.9 The "light_source_test" class

- This class is used for light source testing and measurement purposes. It provides a user interface for controlling and monitoring a light source and related components, with options for selecting channels, setting wavelength values, and reading voltage values. This class utilizes the button click handlers and configuration functions like "DAC_settings()" to handle the setting of parameters for configuring a DAC based on user input. Also the class includes the "on_setWLBut_clicked()" function to set the wavelength which is described in the "light_source_sweep" class (Section 7.2.8).
- **constructor:** In the constructor, an instance of the "Hyperchromator" class is created and initialized. Additionally, the "WL_set" flag is set to false. This constructor is called when an object of this class is instantiated, and it sets up the initial state and user interface for the class.
- **on_readvoltageBtn_clicked():** This function is responsible for handling button clicks, checking various conditions related to wavelength setting, DAC settings and input values. It checks which channels are selected and calculates DAC values accordingly, and then potentially emitting signals for DAC settings if all conditions are satisfied. If any condition is not met, it displays a warning message.
- **update_values():** This function is responsible for updating the displayed values in the user interface line edit fields with the provided voltages.
- **on_get_wl_clicked() and get_wl_position(QString):** These functions are used to request and display the wavelength position from the Hyperchromator in the user interface of the "light_source_test" class. The

"on_get_wl_clicked()" function establishes a connection between the HC (Hyperchromator) object and "get_wl_position(QString)" to retrieve the wavelength position. The "get_wl_position(QString wl_input)" receives the wavelength position from the Hyperchromator and displays it.

9.2.10 The "polarization_measurement" class

This class provides a user interface for controlling polarization measurements, configuring DAC settings and updating measurement values. It also handles events such as window closure and data validation.

- **Constructor and Initialization**: The constructor sets up the GUI for polarization measurements by creating an instance of the "polarization_measurement" class. It initializes an instance of the "KDC101" class, which is an interface of the linear polarization filter with adjustable angle polarization measurements.
- **update_values():** The function is responsible for updating the values displayed in the graphical user interface (GUI). It takes eight QString parameters representing values for different channels and ensuring that the user can see the most recent measurements in the interface.
- **on_moveBut_clicked():** When the "Move" button is clicked, this function extracts the desired position from the GUI input field and instructs the "KDC101" object to move the hardware device to that specified angle.
- **on_homeBut_clicked():** This function is designed to move the device to a predefined "home" or reference position.
- **on_reqPosBut_clicked():** When the "Request Position" button is clicked, this function requests the current position from the hardware device controlled by the "kdc" object and sets up a connection to receive and display that position using the "get_position" slot. This mechanism ensures

that the GUI can continuously update and show the current position of the device in real-time as it changes.

- **get_position(QString):** This function is a slot that is called in response to a signal emitted by the "kdc" object when it has the current position information available. This function is responsible for updating the displayed position in the GUI.
- **on_identBut_clicked():** When the "Identify" button is clicked, this function sends a command to the "KDC101" object (kdc) to initiate the identification process of the hardware device it controls.
- **on_read_pushButton_clicked():** When the "Read" button is clicked in the GUI, this function performs several checks and actions related to configuring and reading settings. It checks for valid gain and offset values, DAC channel selection, and a valid position angle before proceeding to configure the device. If any of the checks fail, warning messages are displayed to guide the user.
- **Gain_Offset_DACsetting():** The "Gain_Offset_DACsetting()" function is responsible for configuring the address settings related to a DAC based on the user's radio button selections in the GUI.
- **on_resetButton_clicked() and reset_DACsetting():** These functions are used to reset various settings and clear fields in the GUI. When the "Reset" button is clicked in the GUI, these functions clear user-entered values, reset flags, and set the DAC settings to default values, ensuring that the system is in a known reset state for the next operation or measurement.

9.2.11 The "polarizer_sweep" class

The "polarizer_sweep" class provides a user interface for configuring and executing polarization sweeps through different polarization

configurations, measuring corresponding voltage outputs, and logging data. It interfaces with a KDC101 device and controls polarization-related hardware or experiments.

- **Constructor**: The constructor initializes the GUI for the polarizer sweep. It creates a new instance of the KDC101 class, which is responsible for controlling the polarizer. It sets up a QwtPlot widget for displaying measurement data. A timer ("timer_polarizer") is created to control the polarization sweep process.
- **polarizer_movement():** This function is a slot that is called when the polarizer movement is completed.
- **on_ReadvoltageBtn_clicked():** This function is called when "Read Voltage" button is clicked. It performs a series of checks on user inputs and configurations, including DAC settings, polarization configuration, and sweep parameters. If all the inputs are valid, it initiates a polarization sweep process using the timer "timer_polarizer". During the sweep, it continuously collects measurement data and displays it on the QwtPlot widget for real-time monitoring. Additionally, this function takes care of logging the acquired data into a file for future reference or analysis.
- **DAC_settings():** This function is used to configure DAC settings based on the user-selected channel.
- **on_stopBtn_clicked():** This function is called when the "Stop" button is clicked. It stops the sweep process and resets relevant flags and variables.
- **on_reset_clicked():** This function is called when a "Reset" button is clicked. It resets DAC settings, flags, and clears data for a fresh start.
- **polarizer_sweep_configuration():** This function sets up the parameters for the polarization sweep, such as start and stop angles and step size.

- **plot(float):** This function updates and displays measurement data on the QwtPlot widget.
- **update_values():** This function receives and processes measurement data for display and logging.
- **timerSlot_polarizersweep():** This slot is triggered by a time-out event of the "timer_polarizer" timer. It updates both DAC and ADC settings, facilitating the progression of the sweep, and managing the completion of the sweep process.
- **step_delay():** This function introduces a delay between sweep steps.
- **reset_DACsetting():** It resets DAC settings to default values.
- **on_home_clicked():** This function is called when a "Home" button is clicked. It returns the polarization to a home position.

9.2.12 The "Wavelength_polarizer_sweep" class

This class is designed to manage and control a complex measurement setup involving both a polarizer angle and a wavelength sweep while ensuring precise control over measurement parameters and data collection. It provides a graphical user interface (UI) for configuring various aspects of the system and executing measurement steps. It incorporates the following functions:

- **Constructor:** This function initializes and establishes connections with the KDC101 (polarizer control) and Hyperchromator (wavelength control) devices for controlling a measurement process. It also sets up the graphical user interface.
- **closeEvent():** This function performs various cleanup and reset actions before closing the application's window. It ensures that timers are stopped,

data is cleared, and devices (such as the polarizer and wavelength controller) are properly closed or reset to their initial states.

- **wl_pol_sweep():** This function manages the sequencing of wavelength and polarizer angle sweeps. for each wavelength step, the polarizer sweeps from its initial position to the stop position, following a predefined step size. Throughout this sweeping operation, it sets DAC and ADC settings, handles delays and step transitions, and resets the system to initial positions after the last sweep cycle.
- **wl_step_delay():** The purpose of this function is to incrementally adjust the wavelength during the sweep process by calculating the next wavelength position based on the step size and current step count and then updating the Hyperchromator to set the new wavelength position. This function is typically called as part of the process to sweep through a range of wavelengths during data collection.
- **pol_step_delay():** This function is responsible for calculating and setting the next position of the polarizer during a wavelength-polarizer sweep.
- **polarizer_movement():** The purpose of this function is to handle the completion of the polarizer movement. When the polarizer has completed its movement to a specific angle or position, it typically triggers a signal (e.g., the “move_completed” signal). The “polarizer_movement()” function is connected to this signal, and its main role is to inform the “MojoSerial” class to indicate that the polarizer movement has completed.
- **DACsettings():** This function is responsible for configuring and setting the DAC settings based on the user's selection of the channel and parameters for subsequent use in data acquisition and control processes.
- **polarizer_sweep_configuration():** This function is responsible for configuring the polarizer angle sweep parameters, ensuring that the system

is ready to perform the sweep as defined by the user's input values. It reads the user inputs from the user interface, specifically the starting angle (retrieved from “polarizer_start_value”), the stopping angle (from “polarizer_stop_value”), and the step size (from “polarizer_step_value”). These values are then converted to floating-point numbers and used to calculate the total number of steps or positions the polarizer will sweep through. Additionally, it sets a flag called “polarizer_configuration_flag” to indicate that the polarizer angle sweep has been configured with the user-specified parameters.

- **wavelength_sweep_configuration():** This function is responsible for configuring the parameters for the wavelength sweep, ensuring that the system is ready to perform the sweep. It reads the user inputs from the user interface, specifically the start, stop, and step values of wavelength positions. These values are then converted to floating-point numbers and used to calculate the total number of steps or positions the wavelength will sweep through. Additionally, it sets a flag called “wl_configuration_flag” to indicate that the wavelength sweep has been configured with the user-specified parameters.

- **reset_DACsetting():** This function is responsible for resetting the DAC settings for different channels to their initial values. The DAC settings control offset and gain values for specific channels used in the system. The function sets these settings to zero, effectively resetting them. By setting these DAC values to zero, the function ensures that the system starts with a clean slate for subsequent measurements or sweeps, where the user can configure these settings as needed.

- **on_stopBtn_clicked():** This function is associated with a button click event. Its purpose is to terminate an ongoing wavelength and polarizer sweep process, reset several flags, including “wl_configuration_flag” and

"polarizer_configuration_flag" flag which indicate the validity of the wavelength sweep and polarizer sweep configuration, and the "DAC_setting_flag" flag which indicates whether the DAC settings are properly configured. It also resets the step counters for both the wavelength and the polarizer sweeps ("wl_curr_steps" and "pol_curr_steps") to their initial values. Additionally, it commands the KDC101 controller (kdc) to move the polarizer to the initial position specified in the user interface (ui->polarizer_start_value) and sets the wavelength to the initial position specified in the UI (ui->wl_start_value). These actions prepare the system for a new measurement or sweep configuration.

- **on_clearBut_clicked():** This function responds to a button named "clearBut" click event, and its purpose is to clear the text or values entered into three specific input fields for the wavelength sweep within the user interface. This can be useful when the user wants to reset or clear the user's input in these fields, to start with fresh values or to remove the previous input for a new operation.
- **on_openBut_clicked():** The purpose of this function call is to open a calibration file with the given file path for use with the Hyperchromator.
- **on_closeBut_clicked():** This function is associated with a button click event and is used to close the communication port of Hyperchromator.
- **on_readvoltageBtn_clicked():** This function is responsible for configuring settings by calling the "configuration_function()" function and preparing the system for reading the voltage data.
- **configuration_function():** This function plays a pivotal role in ensuring that all essential settings and configurations are meticulously arranged before embarking on a measurement or sweep operation. It validates user input, monitors the status of various flags, and calls the execution of

specific functions for configuring the polarizer and light source and initiating the operation when all conditions are met.

One of its primary checks involves inspecting the selection of DAC channels (channels p0, p90, p45, p135, i0, i1, i2, i3). If none of them are selected, the function promptly responds by displaying a warning message asking the user to make a choice. If at least one channel is chosen, it checks whether the user has provided values for gain and offset in the GUI (“gain_value” and “offset_value”). If either or both of these values are absent, the function displays a warning message prompting the user to enter the required information. If both gain and offset values are provided, it performs calculations based on the provided values, calls the “DACsettings()” function subsequently to implement required configurations.

Furthermore, this function checks the availability of values essential for configuring the polarizer angle sweep. It verifies whether the user has thoughtfully provided values for the sweep's starting point, stopping point, and step size within the GUI. If any of these values are missing, it displays a warning message requesting the user to furnish these necessary inputs. If all values are provided, the function calls the “polarizer_sweep_configuration()” function. This function handles the configuration of a polarizer sweep for the measurement.

Moreover, the function assesses the presence of values requisite for configuring the wavelength settings. these values for the wavelength configuration (start, stop, and step) are provided in the GUI (“wl_start_value”, “wl_stop_value”, “wl_step_value”). If any of these values are missing, it prompts a warning message asking the user to provide them. In the event that the user provides all the essential values, the function is called the “wavelength_sweep_configuration()” function. This

function handles the configuration of a wavelength sweep for the measurement.

Furthermore, the function conducts a final examination, checking specific flags labled as “wl_configuration_flag”, “DAC_setting_flag” and “polarizer_configuration_flag”. The simultaneous setting of these flags to “true” signifies the attainment of all indispensable configurations. Upon detecting this state, the function proceeds and orchestrates the movement of the polarizer to the designated start position, sets the initial wavelength position accordingly, and initiates a timer to seamlessly initialize the forthcoming measurement.

- **update_values(float spannung_0 ,float spannung_90,float spannung_45,float spannung_135,float spannung_i0 , float spannung_i1,float spannung_i2,float spannung_i3):** The purpose of this function is to update the displayed voltage reading (“readout_value”) in the GUI based on the user's specific channel selection. It allows for real-time monitoring and visualization of voltage data acquired from different channels or configurations within the system. The function takes eight float values as input, which represent different voltage readings associated with various channels. Depending on the selected channel, it sets the “readout_value” to the voltage reading associated with that channel. For example, if channel p0 is selected, “readout_value” is set to spannung_0. Finally, the function calls the “plot()” function with the “readout_value”. By this means the plot in the GUI is updated to display the voltage reading associated with the selected channel.
- **Plot():** The purpose of the plot function is to update and display a real-time plot in a graphical user interface (GUI) based on incoming data, particularly voltage readings (“readout_value”), along with saving measurement-related data to a file for logging.

- **on_home_kdc_clicked():** this function is designed to initiate the movement of the KDC101 controller to its home or reference position by simply clicking a button in the GUI. This action can be useful for initializing or resetting the controller's position before starting a measurement or sweep operation.
- **on_openBut_kdc_clicked():** this function is designed to handle a button click event in a graphical user interface (GUI) application. When the associated button is clicked, this function performs an action related to opening a connection to a polarizer by calling the open method on "kdc" object.
- **on_reset_clicked():** when the associated "Reset" button is clicked, this function performs several actions related to resetting various settings or variables. These actions prepare the system for a fresh start or a new measurement.

Additionally, there are functions which are employed for tasks such as opening and closing the shutter and configuring filter settings.

9.2.13 The "Wavelength_ sweep" class

This class manages the configuration of the polarizer and light source components, controls data acquisition, and visualizes data. It provides a user-friendly interface for users to interact with and control various aspects of the underlying hardware and measurement processes.

The functionality of this class is like the "wavelength_polarizer_sweep" class, with the key difference being that it measures intensity signals at a specified polarizer angle, and the polarizer itself is not sweeping.

- **Constructor:** The constructor initializes the "Wavelength_sweep" class. It sets up the user interface (UI), creates instances of two objects, kdc and

HC, which are used to control the polarizer and the light source. It creates and configures a QwtPlot for data visualization, calls functions of the kdc object, which establishes a connection to the polarizer and identifies it. Additionally it sets up a QTimer named "timer_wl" and connects it to the "wl_sweep()" slot to control periodic actions for a wavelength sweep.

- **Destructor:** The destructor (~Wavelength_sweep()) is responsible for cleaning up resources when the class instance is destroyed. It deletes the UI and any dynamically allocated objects.
- **closeEvent():** This function is responsible for cleaning up and resetting various components and variables before closing the GUI window associated with the "Wavelength_sweep" class. It ensures that the hardware and data are appropriately configured for termination.
- **on_moveBut_clicked():** This function is responsible for taking the polarizer position specified by the user in the GUI, commanding the polarizer represented by the "kdc" object to move to that position, and setting a flag to indicate that a valid position has been set.
- **on_Read_intensityBtn_clicked():** This function is a user-triggered action that initiates the configuration and preparation steps required for reading and recording intensity values. It ensures that the necessary setting in place for accurate intensity measurements.
- **configuration_function():** This function ensures that the necessary settings related to polarization angle, DAC channel, and wavelength configuration are in place before initiating a measurement. If any required settings are missing, it provides warnings to the user. Once all conditions are met, it prepares the system for data acquisition or measurement.
- **DACsettings():** The purpose of this function is to prepare the DAC configuration based on the user's selections, allowing the system to apply

the correct settings when generating analog signals for measurements. Once this function is executed, the DAC settings are updated, and the “DAC_setting_flag” is set to indicate that the settings are ready for use.

- **Wavelength_sweep_configuration():** Sets up wavelength sweep parameters based on user input.
- **on_reset_clicked():** This function is designed to provide a comprehensive reset of the system's state, including DAC settings, motor positioning, UI fields, and various internal variables. It ensures that the system is returned to a known state or configuration, allowing users to start fresh or reconfigure it as needed.
- **reset_DACsetting():** This function resets DAC settings for all channels and polarization angles to zero, effectively resetting them to their initial or default values. It ensures that the DAC is in a known state before further configuration or operation.
- **on_reset_wl_But_clicked():** This function is a slot function that is triggered when a button (named “reset_wl_But”) is clicked. This function activates filter 1 of the hyperchromator and sets the wavelength to the value in “wl_start_pos”.
- **wl_sweep():** This function is called by the timer at regular intervals. It performs actions related to DAC settings, ADC settings, and steps of a wavelength sweep.
- **step_delay():** This function updates the wavelength position during a sweep by incrementing it based on the step size.
- **on_openBut_clicked(), on_closeBut_clicked(), on_f1But_clicked(), on_f2But_clicked(), on_f3But_clicked(), on_f4But_clicked(), on_openShutterBut_clicked(), on_closeShutterBut_clicked():** These

functions provide an interface for controlling the behavior of the light source represented by the “HC” object and are linked to button clicks, indicating user-initiated actions which are explained in section 9.2.12.

- **on_reqPosBut_clicked**(): When the “reqPosBut” button is clicked, this function requests the position information from the polarizer represented by the “kdc” object and sets up a connection to receive and process the position data once it is available.
- **get_position(QString pos_str):** This function is responsible for updating a UI element with a received position value and formatting it with the degree symbol. It allows the user to see the current position information in the UI.
- **on_homeBut_clicked():** This function is responsible for triggering a “home” operation of the polarizer represented by an “kdc” object when the user clicks the “Home” button in the user interface.
- **on_stopBtn_clicked():** This function is associated with a user interface button click event. When the user clicks the “Stop” button in the user interface, this function is executed. This function is responsible for stopping a timer, resetting some variables and flags, and potentially resetting the wavelength to its initial value.
- **update_values(...):** This function dynamically updates and visualizes data on the GUI based on the user's selection of channels and is an essential component for monitoring and displaying real-time data in the application.
- **plot(float readout_value):** This function is responsible for real-time data plotting on a graph in the GUI and logging data to a text file.

10 Measurement results

In this chapter, the measures are presented which have been acquired using the system described in the chapters before. The measurement results are structured into subsections, which correspond to the buttons or functions of the main window. The functionality of these buttons has been explained in the previous chapters, and now the results which are obtained using these functions are described.

10.1 Single Readout of sensor channels

After configuring the DAC channels and selecting "Normal Mode" in the "Mode Selection" field, clicking the "Lesen" button will trigger the measurement of the AD7980 ADCs corresponding to the P0-P135 sensor channels and the I0-I3 intensity signals voltages, which will then be displayed in the "MojoSerial" GUI. Figure 10.1 shows the result of an experiment where the "RELNEGREF" DAC channel of readout channel "P90" is set to 100 percent. By this means the offset of channel "P90" is set to the largest possible value. After applying the mentioned settings, the ADC output "P90" is observed to be 2.3V and the ADC "P0" displays negligible voltage. As the remaining DACs are unconnected to the board, all ADCs read the maximum value of 2.99V which is the input voltage of the board.

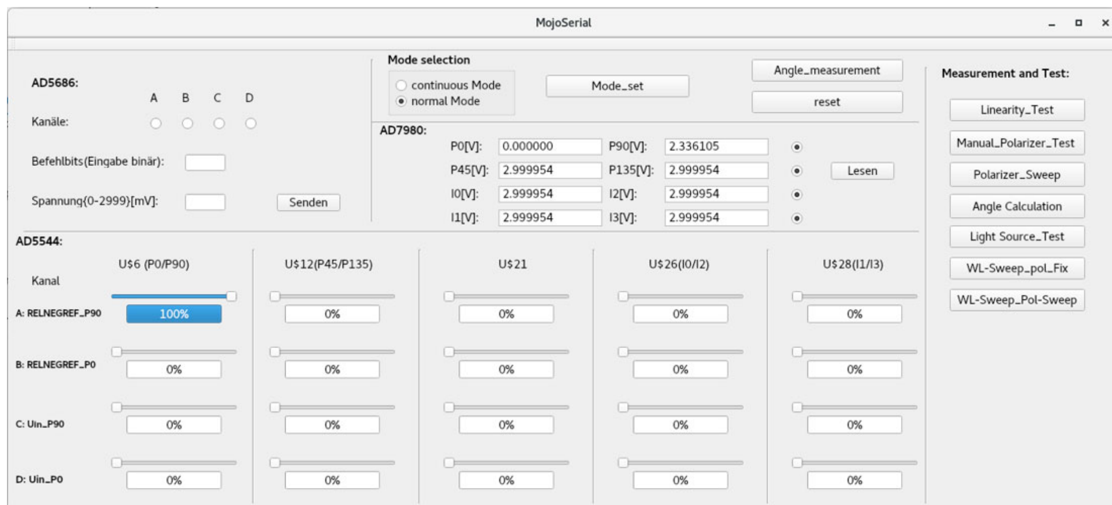


Figure 10.1 The display of readout voltages

10.2 Continuous Angle Measurement

This measurement is utilized to validate angle measurement data obtained from the FPGA without the application of the polarizer and light source. After configuration of the DAC channels and selecting "Continuous Mode" in the "Mode Selection" field, clicking the "Angle_measurement" button will trigger the opening of a new window.

In Figure 10.2, the calculated angles are displayed in a real-time plot, with the "RELNEGREF" channel of DAC "P90" set to 100 percent and for DAC "P0", both "RELNEGREF" and "Uin" channels are set to 0 percent. The same applies to DAC "P90" for the "Uin" channel. The x-axis of the plot represents time in milliseconds [ms], while the y-axis corresponds to the measured angles in degrees [°]. It can be seen that initially, the measured angles are zero, indicating sampling during the early phase of data acquisition. Subsequently, after a certain duration, the measured angles stabilize and remain constant over time. The absence of the light source and polarizer during the measurement, leading to consistent readings for the measurement angles.

By mathematically calculating angles from equations (2.16)- (2.23) in Chapter 2 and comparing them with the measured angles, it can be deduced that the results align with expectations for both angles.

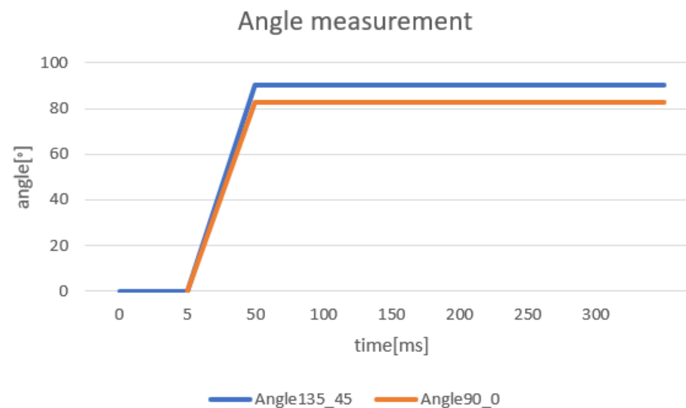


Figure 10.2 Plot of calculated angles in continuous mode

10.3 “Linearity_Test” button

In this section, linearity testing is conducted by selecting “Sweep Type” gain for one test and “Sweep Type” offset for another. Voltage data is plotted in both scenarios. For these tests, channel “P0” is selected, the “Sweep Configuration DAC” is set with a start value of 0, stop value of 65535, and step size of 1000. Figure 10.3 illustrates the readout voltages under the “gain” sweep type, while Figure 10.4 presents the readout voltages under the “offset” sweep type.

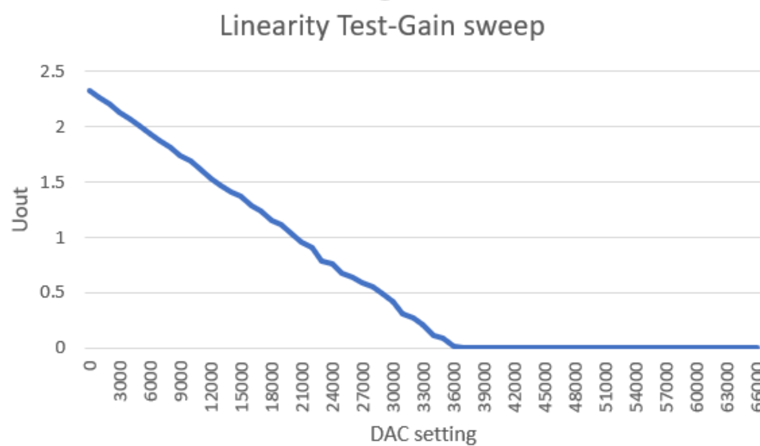


Figure 10.3 Linearity test for gain sweep

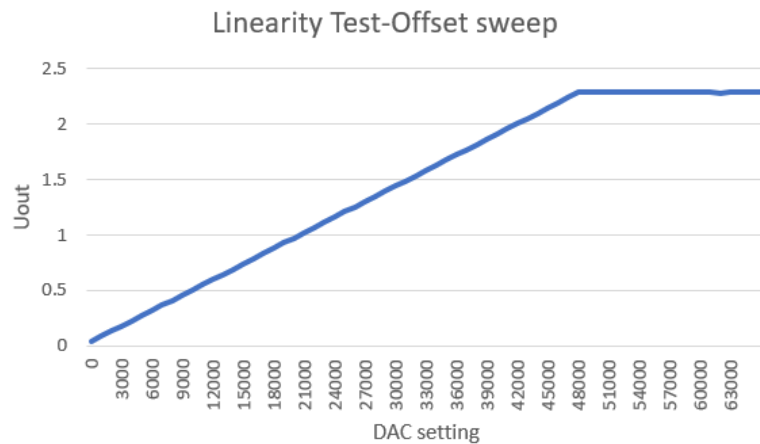


Figure 10.4 Linearity test for offset sweep

As the gain value increases, a discernible decrease in the output voltage is observed, as illustrated in Figure 10.3. This reduction is a consequence of the

increasing prominence of negative values, as evidenced in formulas (5.35) and (5.36). While the ADC's input voltage range being limited to zero volts, the minimum value is constrained to zero volts in the measurements.

In contrast, as the offset value increases, the output voltage also rises proportionally. The output voltage eventually stabilizes at 2.4V, representing the maximum attainable value, as depicted in Figure 10.4.

The correctness of the linearity test is evident in these figures, aligning with the expectations and the explanation provided in section 5.3.1.3.

10.4 Test with Manual Polarizer Angle Setting

This measurement is designed to illustrate the single readout voltages of all channels under the specified polarizer angle and DAC configurations. Figure 8.5 presents the measurement results obtained from a manual polarizer test, where DAC channel P0 was selected with an offset value of 65535, a gain value of 0, and the polarizer configured at 90°.

To verify the correctness of the configuration, the position of the polarizer can be requested. By pressing the 'Request Position' button the specified position is displayed in the field box.

Subsequently, clicking the 'Read' button reveals the readout voltages of channels in the corresponding fields. Due to the DAC settings of gain and offset values for DAC P0, the readout voltage for this channel reaches 2.33V, closely approaching the maximum limit of 2.4V. For channel P90, as no specific DAC settings are applied, the readout voltage remains at zero volts. Since the other channels are not connected on the board, they display a readout voltage of 2.999954V.

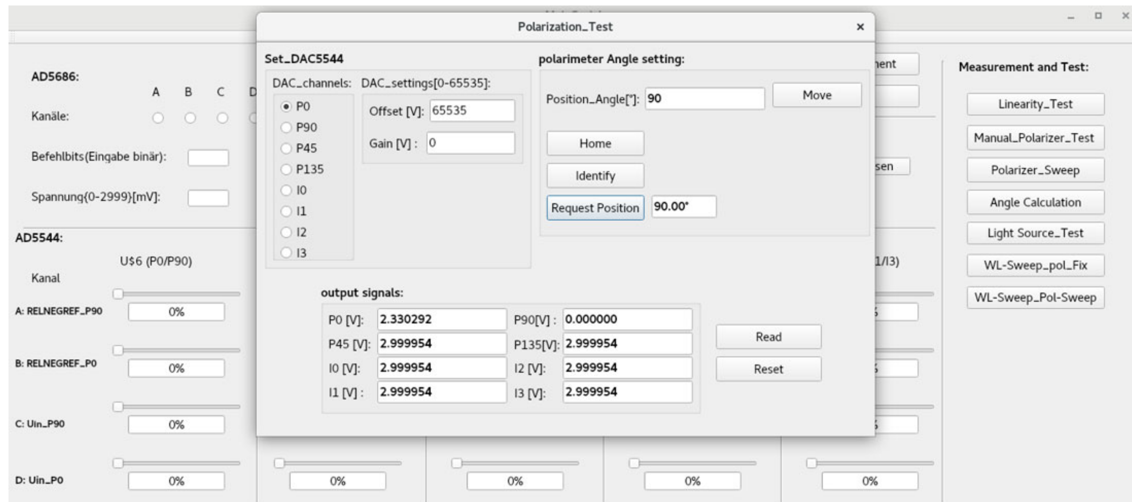


Figure 10.5 Manual polarizer test

As depicted in Figure 10.5, the measurement results meet expectations by accurately reflecting the anticipated behaviors associated with the selected DAC configurations and the readout voltage of other channels. Additionally, the system demonstrates its ability to communicate with the polarizer, configuring it at the specified angle.

10.5 Sweep of Polarizer Angle

This measurement is used to assess the sensor performance as the polarizer varies smoothly its angle from the initial value to the final value, using a predetermined step value. To evaluate the precision of this function, DAC channel "P0" is configured with an offset value of 65535 and a gain value of 13107. Subsequently, the polarizer undergoes a sweep from 0 to 180 degrees in 10-degree increments. The measured voltage values are plotted in Figure 10.6. As observed during the sweep, the readout voltage changes in response to angle change reaching its minimum at 0° and 180° degree and its maximum around 90°.

This behavior aligns with expectations, as the polarizer, when aligned with the polarization filter at 0 or 180 degrees, blocks or significantly attenuates light that

is not aligned with this polarization. Consequently, at these angles, minimal light is anticipated to pass through the polarizer, resulting in a lower readout voltage.

Conversely, when the polarizer is aligned with the polarization filter at 90 degrees, it allows light polarized in this direction to pass through most effectively. This leads to a higher readout voltage at this angle due to increased light transmission through the polarizer.

For intermediate polarizer angles, between 0 and 90 degrees or 90 and 180 degrees, the readout voltage is expected to display intermediate values. This is because these configurations allow partial transmission of light through the filters at 45 and 135 degrees.

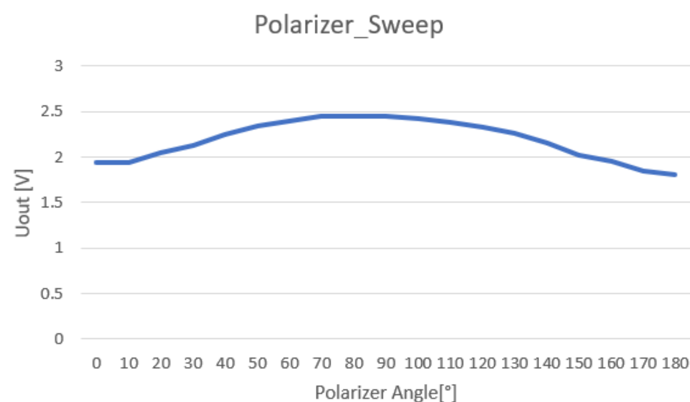


Figure 10.6 Polarizer sweep test

In summary, the measurement result aligns with expectations based on the principles of polarimetry. The configured settings and the observed voltage changes during the sweep are consistent with the anticipated behavior of the system under the specified DAC configurations and polarizer angle variations.

10.6 Angle Measurement

As previously explained, angles can be calculated both in software and hardware. Consequently, tests have been conducted to assess the measurement accuracy in both modes. Channel P0 was chosen with a gain value of 13107, an offset value

of 65535, and polarizer angles ranging from 0 to 180 degrees, with a step value of 10 degrees.

Figure 10.7 presents the result of angle calculation in software. In this mode, a configurable number of samples is taken from the sensor intensity signals for each polarizer angle. The average and in turn the corresponding angle is calculated and plotted in the GUI.

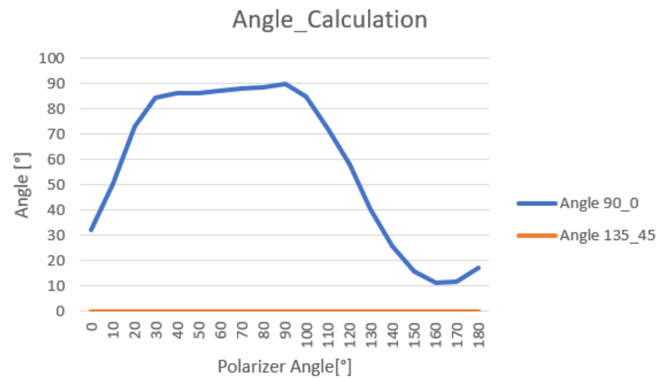


Figure 10.7 Angle Measurement for a polarizer angle sweep from 0° to 180° degrees with angle calculation in software

Figure 10.8 shows the result of angles computed by digital logic in the FPGA. In this mode, the polarizer performs a sweep for sampling across the 0 to 180-degree range with a 10-degree step, after which the calculated angles are transmitted from the FPGA and plotted.

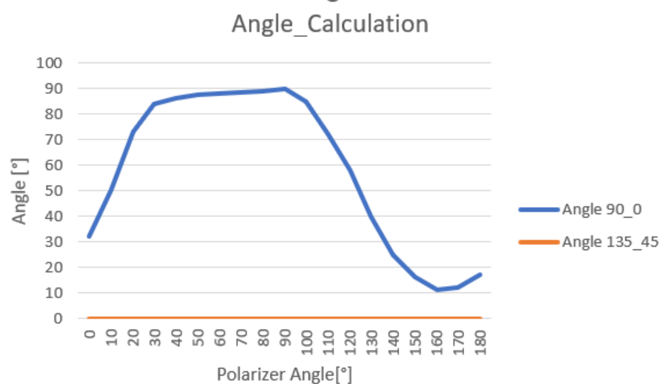


Figure 10.8 Angle Measurement for *a* polarizer angle sweep from 0° to 180° degrees with angle calculation in the FPGA hardware

10.7 Light Source Test

The "Light Source_Test" button serves the purpose of establishing a connection via the serial port between the PC running the Qt application and the light source. Afterwards the light source can be configured including selecting filters and wavelengths. Figure 10.9 shows the GUI associated with this measurement process. During this test, the following steps are carried out:

By pressing the "open port" button, a connection via the serial port between Qt and the light source is established. A wavelength within the range of 250 to 1100 nm. The button "set wavelength" is pressed to finalize the light source wavelength configuration. In the shown case, the wavelength is set to 650 nm.

The "Get wavelength" button is used to retrieve and display the wavelength value in a text box. Once the DAC channels are selected and configured, pressing the "read" button will acquire intensity signals from all ADC channels and display them accurately in their respective channels.

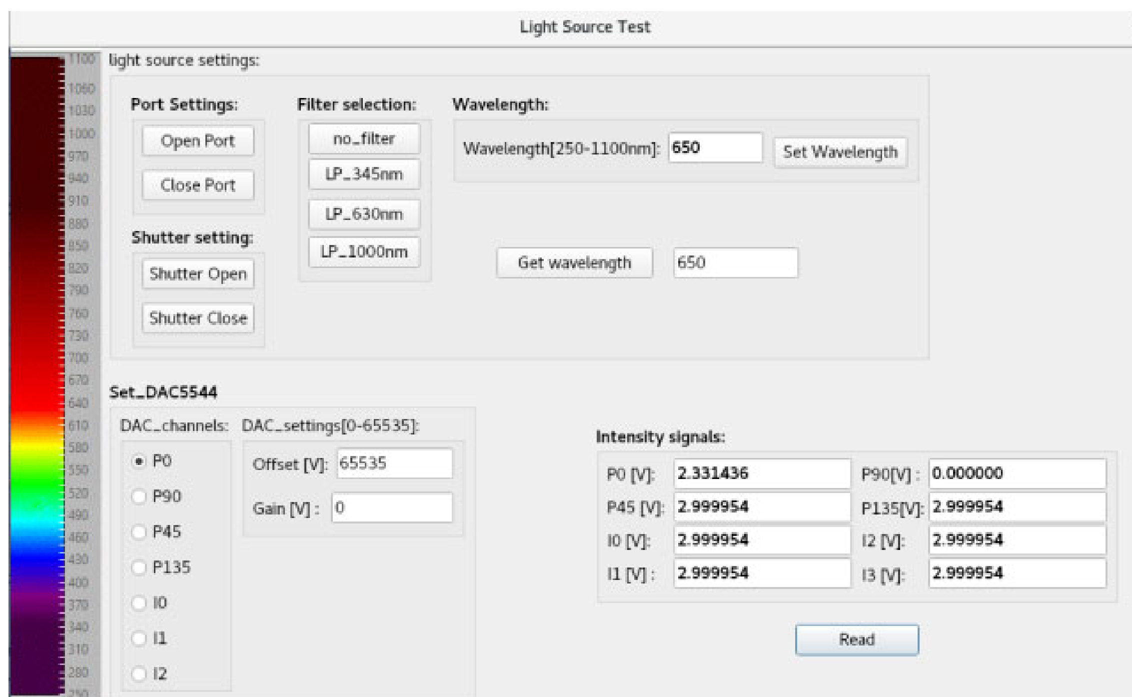


Figure 10.9 Light Source communication and configuration to read intensity signals

10.8 Wavelength Sweep

A Wavelength sweep between 250 to 1100 nm is performed in steps of 50nm using the “WL_Sweep_pol_fix” window while the polarizer angle is set at 90°, and the DAC channel P0 is configured with a gain value of 13107 and an offset value of 65535. Figure 10.10 displays the measurement results obtained with these configurations.

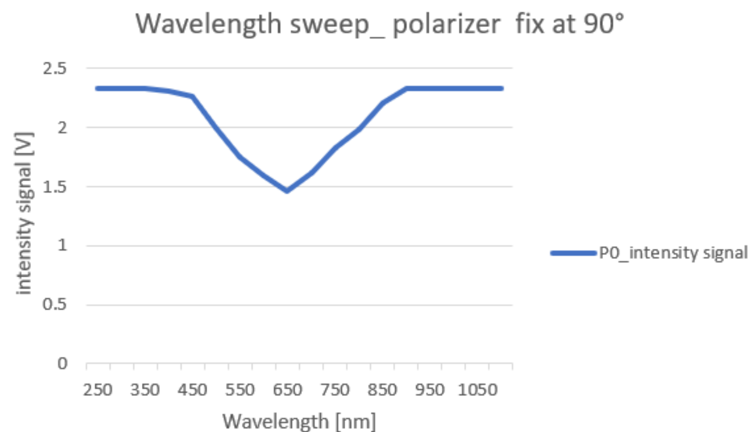


Figure 10.10 Light source test with wavelength sweeps and polarizer at 90°

The sensor seems to have its highest spectral sensitivity in the wavelength range of visible light between 380nm and 750nm.

10.9 Wavelength and Polarizer Angle Sweep

Using the “WL_Sweep_Pol_Sweep” window the sensor sensitivity can be measured as a function of both the polarizer and the wavelength. In the test result shown in figure 10.11, the DAC channel P0 is configured with a gain value of 13107 and an offset value of 65535. Additionally, for each wavelength ranging from 250 to 1100 nm in 50 nm steps, the polarizer sweeps through an angle range from 0 to 180 in 10-degree steps. Throughout this procedure, the readout intensity signals are plotted as a function of the polarizer's angle position.

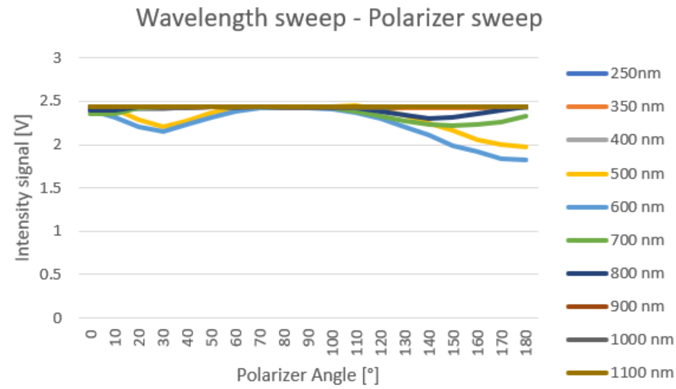


Figure 10.11 Sensor sensitivity as a function of the polarizer angle and for light with different wavelengths

As depicted in Figure 10.11, the measurement results closely adhere to the expected values. The sensor exhibits highest spectral sensitivity within the wavelength spectrum of visible light, specifically ranging from 380nm to 750nm, as described in section 10.8. Furthermore, the configured settings, coupled with the discerned voltage alterations throughout the sweep, demonstrate a remarkable coherence with the expected behavior of the system. This alignment holds true under the defined DAC configurations and variations in polarizer angles, affirming the reliability and accuracy of the experimental setup. The confluence of these factors not only validates the sensor's performance within the specified wavelength range but also attests to the robustness of the system in responding predictably to dynamic experimental conditions.

11 Conclusion & Outlook

This master thesis presents the design and implementation of a signal processing chain for the optical determination of rotation angles. This is achieved through the utilization of four sensors, which are realized as photodiodes equipped with integrated polarization filters and a hardware setup which amplifies, corrects for offsets and digitizes the sensor signals. Additionally, a high-precision CORDIC (Coordinate Rotation Digital Computer) hardware design is implemented on an FPGA using Verilog. Moreover, the integration of a monochromatic light source with configurable wavelength and polarizer with configurable plarization angle, is deployed to perform automated measurements through an QT application executed on a Linux PC. Via the GUI of the created QT application, the user can control all DACs of the hardware setup, configure the polarizer angle, and the light source wavelength and readout the digitized intensity voltages or calculated angles.

In summary, it can be said that the thesis has effectively achieved its primary objective. Nonetheless, it's noteworthy that only the two sensors channel P0 and P90 are readout in the current setup and the hardware setup can be extended for two more sensors channels P180 and P270.

Bibliography

[1] Volder, J. E., “The CORDIC Trigonometric Computing Technique,” IRE Transactions on Electronic Computers, vol.EC-8, pp.330-334, 1959. DOI:10.1109/TEC.1959.5222693.

[2] S. Tijani “VHDL-Implementierung der Arkussinusfunktion und der Division von Festkommazahlen nach dem CORDIC Algorithmus“ M.S. thesis, Dept. Electron. Eng., Fachhochschule Dortmund, Dortmund, Germany, 2016.

[3] Universität Paderborn, Bachelorarbeit, Sergej Gummer: Entwurf von SignalverarbeitungsElektronik und Algorithmen zur hochgenauen Verarbeitung von Winkel-Sensor-Daten.

[4] Mazenc, C. Merrheim, X. Muller, J.-M. Computing Functions cos/sup -1/ and sin/sup -1/ Using CORDIC. IEEE Transactions on Computers 42 (1993), Nr. 1, S. 118-122. https://ieeexplore.ieee.org/document/192222.

[5] Andraka, Ray: A survey of CORDIC algorithms for FPGA based computers. In: Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays ACM, 1998, S. 191-200

[6] J. S. Walther, „A unified algorithm for elementary functions “, in Proceedings of the May 18-20, 1971, spring joint computer conference, Atlantic City, New Jersey, Mai 1971, S. 379–385, DOI: 10.1145/1478786.1478840.

[7] Grau, Dr. G. ‚POLDI_Signalverarbeitung.LG

[8] M. Beuler, „CORDIC-Algorithmus zur Auswertung elementarer Funktionen in Hardware“, Fachhochschule Gießen-Friedberg, FH-Report 11/08, 2008. Zugegriffen: Apr. 29, 2020. [Online]. Verfügbar unter:http://digdok.bib.thm.de/volltexte/2009/4148/pdf/CORDIC_Algorithmus.pdf.

[9] C. Demske, “Entwurf eines VHDL-Designs und einer Applikationssoftware zur Konfiguration und Kalibrierung eines optischen Winkelgebers,” FH Dortmund, Dortmund, 2019.

[10] A. Engelbrecht, “ Entwicklung einer Messelektronik zur Bestimmung des Drehwinkels mittels POLDI-Sensor”. (2014)

[11] Mountain Photonics - the hyperchromator - TLS - te lintelo systems (2019) TLS. Available at: https://www.tlsbv.nl/mountain-photonics/ (Accessed: 07 July 2023).

[12] Double Glan-Taylor calcite polarizers (no date) Thorlabs, Inc. - Your Source for Fiber Optics, Laser Diodes, Optical Instrumentation and Polarization Measurement & Control. Available at: https://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=1775 (Accessed: 07 July 2023).

[13] Thorlabs.de. Available at: https://www.thorlabs.de/newgrouppage9.cfm?objectgroup_id=2875 (Accessed: 07 July 2023).

[14] Entwurf von Leiterplatten für die Versorgung und Auslesung eines ... Available at: https://opus.bsz-bw.de/fhdo/files/3352/Bachelorarbeit_Oguz_Eroglu.pdf (Accessed: 02 August 2023).

[15] Data sheet AD780 - Analog devices. Available at: https://www.analog.com/media/en/technical-documentation/data-sheets/AD780.pdf (Accessed: 02 August 2023).

[16] 500ma peak output LDO regulator - microchip technology. Available at: https://ww1.microchip.com/downloads/en/DeviceDoc/MIC5219-500mA-Peak-Output-LDO-Regulator-DS20006021A.pdf (Accessed: 02 August 2023).

[17] TSV91x rail-to-rail input/output, 8-mhz operational amplifiers ... Available at: https://www.ti.com/lit/ds/symlink/tsv912.pdf?HQS=dis-dk-null-digikeymode-dsf-pf-null-wwe&ts=1671702512280&ref_url=https%253A%252F%252Fwww.ti.com%252Fgeneral%252Fdocs%252Fsuppproductinfo.tsp%253FdistId%253D10%2526gotoUrl%253Dhttps%253A%252F%252Fwww.ti.com%252Flit%252Fgpn%252Ftsv912 (Accessed: 04 August 2023).

[18] ADR510 1.0 v precision low noise shunt voltage reference data sheet ... Available at: https://www.analog.com/media/en/technical-documentation/data-sheets/ADR510.pdf (Accessed: 04 August 2023).

[19] TSV91x rail-to-rail input/output, 8-mhz operational amplifiers ... - ti.com. Available at: https://www.ti.com/lit/ds/symlink/tsv912.pdf (Accessed: 04 August 2023).

[20] NLASB3157 - SPDT, 3 ohms Ron Switch - Onsemi. Available at: https://www.onsemi.com/pdf/datasheet/nlasb3157-d.pdf (Accessed: 05 August 2023).

[21] 16-bit, 1 msps, Pulsar ADC in MSOP/LFCSP Data Sheet AD7980 - Analog Devices. Available at: https://www.analog.com/media/en/technical-documentation/data-sheets/AD7980.pdf (Accessed: 05 August 2023).

[22] CMOD A7 reference manual - digilent. Available at: https://digilent.com/reference/_media/reference/programmable-logic/cmod-a7/cmod_a7_rm.pdf (Accessed: 06 August 2023).

[23] Hyperchromator Low-level Remote Control.

[24] L. Alaee "Design and implementation of signal processing chain for the optical determination of the rotation angle in Verilog" research thesis, Dept. Electron. Eng., Fachhochschule Dortmund, Dortmund, Germany, 2022.

[25] Oguz Eroglu, Entwurf von Leiterplatten für die Versorgung und Auslesung eines optischen Winkelgebers, Bachelorarbeit, FH Dortmund, 2023

Appendix A: Controlling the light source

The control of the light source is achieved through the transmission of high-level command or low-level command.

In the high-level remote control, the Hyperchromator program is running and through the LabVIEW Shared Variable Interface allows to send high-level command such as “set_wave”, where allowing direct specification of the desired wavelength. During this automated process, the system evaluates wavelength calibration, adjusts the motor accordingly, and switches to the correct long-path order sorting filter.

On the other hand, the low-level remote control provides users with manual control capabilities, eliminating the need to run the Hyperchromator program in the background. In this mode, the user can command the grating to specific motor positions and configure filters or the shutter according to their experimental requirements. This manual control option offers greater flexibility and hands-on customization, allowing user to adapt the light source to specific experimental conditions and parameters. In the scope of this thesis, the chosen method for manipulating the light source is the low-level remote control.

1 Principle of low-level remote control

After switching on the Hyperchromator, the internal motor controller initiates motor control loops, performs motor indexing and subsequently transitions to an idle state, waiting to receive commands through its RS-232 interface.

The internal motor controller is a high-performance motion controller which uses the proprietary programming language TML. Through the RS-232 interface, users can send direct TML commands at any time, thereby superseding the

internal program loops. The TML commands are used to describe the functions and they are encoded into binary command strings.

The most important function is to command the grating motor to a new position for a desired wavelength. There is a calibration table (wavelength_calib) in the configuration directory. In these editable text files, the first column is the wavelength (nm), the second is FWHM (nm), the third column is the motor position in internal motor units (“counts”).

Target wavelength = 440nm -> target motor position = 140000 (counts)

The motion controller, after driving to some target position in the previous move, is now in idle mode and waits for commands. The relevant parameter here is CPOS, the “commanded position”. The controller is in “absolute position mode” (CPA), thus CPOS is given as absolute position. Changing CPOS, followed by the update command UPD will cause the controller to resume motion with the pre-set parameters for speed and acceleration, until the new target position is reached:

Example:

CPOS = 140000; //target position for 440nm in the case of the calibration table above

UPD; // go to it!

The order-sorting filter wheel (“Mot2”) as well as the shutter (“MOT1”) are driven via function calls without parameters. For example, to open or close the shutter, two independent functions, labeled Mot1Pos1 and Mot1Pos2 are used.

Example:

CALL Mot1Pos1; // opens the shutter

CALL Mot1Pos2; // closes the shutter

Likewise, the filterwheel may be called as Mot2Pos1. Mot2Pos4 with positions for 4 filters.

The filter and shutter positions are fixed within the controller. However, sometimes it may be necessary to slightly tune the positions, e.g. to avoid vignetting by a filter's edge. This can be accomplished by functions MotINC and MotDEC, incrementing or decrementing the position of the currently selected motor and position (i.e. the last issued MotxPosy command). These offsets are stored internally until the ClearOffsets function is called. In the Hyperchromator program, all offsets are set once during initialization.

2 Communication Protocol

The communications protocol is based on a binary command message structure where each byte has a designated role in conveying information about the message length, axis/group ID, operation code, data, and a checksum for error checking.

2.1 Binary command message structure

Byte 1: Message length (number of bytes-2)

Byte 2: Axis/Group ID – high byte

Byte 3: Axis/Group ID – low byte

Byte 4: Operation code – high byte

Byte 5: Operation code – low byte

Byte 6: Data (1) – high byte

Byte 7: Data (1) – low byte

Byte 8: Data (2) – high byte

…

Byte13: Data (4) – low byte

Last byte: Checksum CS (modulo 256)

- **Byte 1: Message length (number of bytes-2):** This byte indicates the total length of the message in terms of the number of bytes.
- **Byte 2-3: Axis/Group ID – high byte and low byte:** These two bytes use a complicated bit-wise encoding. For the Hyperchromator, axisID=255, these bytes are 0F and F0.
- **Byte 4-5: Operation code – high byte and low byte:** These two bytes together represent either a command code of the TML language or the register address of a TML register or a combination.
- **Byte 6-13: Data (1) to Data (4) – high byte and low byte:** These bytes carry the actual data associated with the operation. Depending on the specific message and operation, the data bytes can represent parameters, values, or other relevant information.
- **Last Byte: Checksum CS (modulo 256):** The last byte serves as a checksum, calculated using modulo 256.

2.2 List of commands

In this section the employed commands are explained. Binary data is given as hexadecimal numbers.

The "hwhb", "hwlb", "lwhb" and "lwlb" are Abbreviations of high_word_high_byte, high_word_low_byte, low_word_high_byte and low_word_low_byte respectively.

2.2.1 Set Target Position

the target position in internal units (counts) for the next grating motor move. The move will not be started until the UPD command is sent.

TML: CPOS = position (Integer);

Register address (re-coded): 24 9E

Binary: 08 0F F0 24 9E lwhb lwlb hwhb hwlb CS

2.2.2 Update Position

start motor move to last set CPOS.

TML: UPD;

Command code: 01 08

Binary: 04 0F F0 01 08 0C

2.2.3 Set Speed

sets the motor speed in internal units for the next grating motor move. The change will be effective with the next CPOS and UPD command. Speed is a fixed point number, with the first word describing the decimals (in multiples of 1/65536, for example 0.1 will be shown as 0.1*65536 rounded to 6554 = 0x199A) and the second word describing the integer part. Usually, it should be OK to use only the integer part, thus setting the lower word to zero. NOTE: make sure that speed is not above the safe maximum value, which is given by the key "speed" in config.txt.

TML: CSPD = speed (fixed point);

Register address (re-coded): 24 A0
Binary: 08 0F F0 24 A0 lwhb(0) lwlb(0) hwhb hwlb CS

2.2.4 Reset drive

reset the drive and index the motor after control errors. The behavior is similar to power-on of the hardware.

TML: RESET;

Command code: 04 02

Binary: 04 0F F0 04 02 09

2.2.5 Read current pos

reads the current position of the motor. If the system is an open loop, there is no position feedback and one needs to rely on the current target position parameter TPOS of the trajectory generator. If no pulses are lost (overspeed!), this position corresponds to the motor position. If the system has closed loop feedback, the encoder provides the actual motor position APOS. Most Hyperchromators are closed-loop. Again, we need to provide the correct register address to either read TPOS or APOS.

TML: ?APOS ;

Command code: B0 05

Axis requested: 0F F0 (code for axisID=255, standard for the Hyperchromator)

Register address: 02 28 (APOS reg_h reg_l)

Expeditor address: 0F F1 (master axis for communication in a multi-axis system)

Binary: 08 0F F0 B0 05 0F F1 reg_h reg_l CS

The command returns the current APOS value in the following form:

Binary: 4C 0C 0F F1 B4 05 0F F0 02 28 pos_lwhb pos_lwlb pos_hwhb pos_hwlb CS

2.2.6 Call function

This command calls a function stored in non-volatile memory with the motion controller. The function label is associated to an internal memory address, which is being transferred. To find the memory addresses related to all available functions, open the motion controller configuration, which is of the type *.t.zip (e.g. Hyperchromator_enc.t.zip) within the configs/calibration directory of your Hyperchromator. Within this ZIP file, you can view the file variables.cfg and locate the list of function labels close to the bottom of this file.

LABEL MOT2POS1 @0x407D

LABEL MOT2POS2 @0x4086

LABEL MOT2POS3 @0x408F
LABEL MOT2POS4 @0x4098
…

TML: CALL function_label;
Command code: 74 01
Binary: 06 0F F0 74 01 reg_high reg_low CS
Example:
CALL Mot2Pos3: 06 0F F0 74 01 40 8F 49

Function label	Action
MOT1POS1	Shutter OPEN
MOT1POS2	Shutter CLOSE
MOT2POS1	Filter 1
MOT2POS2	Filter 2
MOT2POS3	Filter 3
MOT2POS4	Filter 4
RELAISOFF	Open the relais switch (optional! for Lamp ON via the extension connector)
RELAISON	Close the relais switch (optional! for Lamp OFF via the extension connector)
MOTINC	Increment the motor position offset of the last active motor and position (last MOTxPOSy command)
MOTDEC	Decrement the motor position offset of the last active motor and position (last MOTxPOSy command)
CLEAROFFSETS	Clear ALL offsets

Table 1: List of public functions in Hypercromator

Example: switching shutter between open/close

06 0F F0 74 01 40 A1 5B

06 0F F0 74 01 40 AA 64

Appendix B: Polarizer motor control

1 Communications Protocol

The communications protocol is based on a message structure that always starts with a fixed-length, 6-byte message header which, in some cases, is followed by a variable-length data packet. For simple commands, the 6-byte message header is sufficient to convey the entire command. For more complex commands that involve passing a set of parameters, the 6-byte header is inadequate and in this case the header is followed by the data packet.

The header part of the message always contains information that indicates whether a data packet follows the header and if so, specifies the number of bytes that the data packet contains. This allows the receiving process to accurately determine the start and end of messages, facilitating proper message handling.

The 6-bytes in the message header are shown below:

Byte:	byte 0	byte 1	byte 2	byte 3	byte 4	byte 5
Meaning if no data packet to follow	message ID		param1	param2	dest	source
Meaning if data packet to follow	message ID		data packet length		dest | 0x80	source

Figure 1: The 6-bytes message header structure

The meaning of some of the fields depends on whether the message is followed by a data packet or not. This is indicated by the most significant bit in byte 4, called the destination byte, therefore the receiving process must first check if the MSB of byte 4 is set. If this bit is not set, then the message is a header-only message, and the interpretation of the bytes is as follows:

- **message ID:** describes what the action the message requests.
- **param1:** first parameter (if the command requires a parameter, otherwise 0)

- **param2:** second parameter (if the command requires a parameter, otherwise 0)
- **dest:** the destination module
- **source:** the source of the message

If the MSB of byte 4 is set, then the message will be followed by a data packet and the interpretation of the header is the following:

- **message ID:** describes what the action the message requests
- **data packet length:** number of bytes to follow header

Note: although this is a 2-byte long field, currently no data packet exceeds 255 bytes in length.

- **dest | 0x80**: the destination module logic OR'd with 0x80 (noted by d|)
- **source:** the source of the data

The source and destination fields are used to indicate the source and destination of the message. When the host sends a message to the module, it uses the source identification byte of 0x01 (meaning host controller (i.e., control PC)) and the destination byte of 0x50 (meaning "generic USB unit"). In messages that the module sends back to the host, the content of the source and destination bytes is swapped.

1.1 General message exchange rules

The type of messages used in the communications exchange between the host and the sub-modules can be divided into 4 general categories:

(a) Host issues a command, sub-module carries out the command without acknowledgement (i.e., no response is sent back to the host).

(b) Host issues a command (message request) and the sub-module responds by sending data back to the host.

(c) Following a command from the host, the sub-module periodically sends a message to the host without further prompting.

These messages are referred to as *status update messages*. These are typically sent automatically every 100 msec from the sub-module to the host, showing, amongst other things, the position of the stage the controller is connected to. The meters on the Thorlabs User GUI rely on these messages to show the up-to-date status of the stage.

(d) Rarely – error messages, exceptions. These are spontaneously issued by the sub-module if some error occurs. For example, if the power supply fails in the sub-module, a message is sent to the host PC to inform the user.

Apart from the last two categories (status update messages and error messages), in general the message exchanges follow the SET -> REQUEST -> GET pattern. The SET part of the trio is used by the host to set some parameter or other. If the host requires some information from the sub-module, then it may send a REQUEST for this information, and the sub-module responds with the GET part of the command.

1.2 Conversion between position, velocity and acceleration values in standard physical units and their equivalent Thorlabs Software parameters.

The physical units needed to describe position, velocity and acceleration are related to position and time measurement units (millimetres/degrees and seconds). In motion controllers, however, normally the system only knows the distance travelled in encoder counts (pulses) as measured by an encoder fitted to the motor shaft. In most cases the motor shaft rotation is also scaled down further by a gearbox and a leadscrew. In any case, the result is a scaling factor between encoder counts and position. The value of this scaling factor depends on the stage. In the section below this scaling factor will be represented by the symbol EncCnt.

Time is related to the sampling interval of the system, and as a result, it depends on the motion controller. Therefore, this value is the same for all stages driven by

a particular controller. In the sections below the sampling interval will be denoted by T.

The sections below describe the position, velocity, and acceleration scaling factors for all the controllers and stages that are used with these controllers. The symbols POSAPT, VELAPT and ACCAPT are used to denote the position, velocity and acceleration values used in Thorlabs commands, whereas the symbols Pos, Vel and Acc denote physical position, velocity and acceleration values in mm, mm/sec and mm/sec2 units for linear stages and degree, degree/sec and degree/sec2 for rotational stages.
As Thorlabs parameters are integer values, the Thorlabs values calculated from the equations need to be rounded to the nearest integer.

Brushed DC Controller driven stages:

Mathematically:

$POS_{APT} = EncCnt \times Pos$

$VEL_{APT} = EncCnt \times T \times 65536 \times Vel$

$ACC_{APT} = EncCnt \times T_2 \times 65536 \times Acc$

where $T = 2048 / (6 \times 10^6)$

The value of "EncCnt" and the resulting conversion factors are listed below for KPRM1E/M stage:

Stage	EncCnt per mm Or EncCnt per °	Scaling Factor	
		Velocity	Acceleration
KPRM1E/M	1919.6418578623391	42941.66 (°/s)	14.66 ($°/s^2$)

2 Motor Control Messages

The 'Motor' messages provide the functionality required for a client application to control one or more of the Thorlabs series of motor controller units. The motor

messages can be used to perform activities such as homing stages, absolute and relative moves, changing velocity profile settings and operation of the solenoid state (on solenoid control units). In the following the messages which are used in this project are explained. As KDC101 is a single-channel controller, the addressed channel is identified in the Chan Ident byte and is always set to channel 1 (0x01).

2.1 MGMSG_MOD_IDENTIFY

This message instructs the hardware unit to identify itself by flashing its front panel LEDs. The command structure of this is 6 bytes as follow:

0	1	2	3	4	5
Header only					
23	02	Chan Ident	00	d	s

2.2 MGMSG_MOD_SET_CHANENABLESTATE

This message is sent to enable or disable the specified drive channel. The command structure of this message is 6 bytes:

0	1	2	3	4	5
Header only					
10	02	Chan Ident	Enable State	d	s

To enable the channel, the Enable State parameter is set to “0x01” and to disable channel, this parameter is set to “0x02”.

2.3 MGMSG_MOT_GET_POSCOUNTER

This message is intended for configuring the real-time position count within the controller. The stage is homed immediately after power-up during which its position is unknown since the stage can freely move when power is off. Once the

homing process is completed the position counter is automatically adjusted to display the actual position. From this point onward, the position counter always shows the actual absolute position. The command structure in this message is 12 bytes, 6-byte header followed by 6-byte data packet as follows:

0	1	2	3	4	5	6	7	8	9	10	11
Header						Data					
12	04	06	00	d	s	Chan Ident		position			

The updated value of the position counter is provided as a 32-bit signed integer, encoded in the Intel format.

2.4 MGMSG_MOT_SET_VELPARAMS, MGMSG_MOT_REQ_VELPARAM, MGMSG_MOT_GET_VELPARAMS

These functions are used to set the trapezoidal velocity parameters for the specified motor channel.

SET:

The Command structure of this function has 20 bytes, which is 6-byte header followed by 14-byte data packet as follows:

0	1	2	3	4	5	6	7	8	9	10	11
Header						Data					
13	04	0E	00	d	s	Chan Ident		Min Velocity			

12	13	14	15	16	17	18	19
Data							
Acceleration				Max Velocity			

The minimum (start) velocity is 4-byte value and always set to zero. The acceleration is 4-byte unsigned long value. The maximum (final) velocity is 4-bytes unsigned long value.

REQUEST:

Command structure of this function is 6 bytes in the following format:

0	1	2	3	4	5
Header only					
14	04	Chan Ident	0X	d	s

GET:

Response structure of Get function is 20 bytes. The 6 bytes header followed by 14 bytes data packet in the following format:

0	1	2	3	4	5	6	7	8	9	10	11
Header						Data					
15	04	0E	00	d	s	Chan Ident		Min Velocity			

12	13	14	15	16	17	18	19
Data							
Acceleration				Max Velocity			

2.5 MGMSG_MOT_MOVE_HOME

This command is used to initiate a home movement sequence on the motor channel. It has a TX structure in 6 bytes with the following format:

0	1	2	3	4	5
Header only					
43	04	Chan Ident	0X	d	s

2.6 MGMSG_MOT_MOVE_HOMED

In this command, once the home sequence is completed, the controller sends a "homing completed" message. However, there is no response to the initial message. This message is received as an RX structure consisting of 6 bytes in the following format:

0	1	2	3	4	5
Header only					
44	04	Chan Ident	0X	d	s

2.7 MGMSG_MOT_MOVE_RELATIVE

This command can be used to start a relative move. There are two versions of this command: a short version (6-byte header only) and a long version (6-byte header plus 6 data bytes). When the short version is used, the relative distance parameter used for the move will be the parameter sent previously by a MGMSG_MOT_SET_MOVERELPARAMS command and the relative distance is encoded in the data packet that follows the header.

Short version:

TX structure is 6-bytes, and the structure is as following:

0	1	2	3	4	5
Header only					
48	04	Chan Ident	0X	d	s

Long version:

This command appends the relative move params structure (MOT_SET_MOVERELPARAMS) to this message header. The command structure is 12 bytes, 6-byte header followed by 6-byte data packet as follows:

0	1	2	3	4	5	6	7	8	9	10	11
Header						Data					
48	04	06	00	d	s	Chan Ident		Relative Distance			

In this structure “Relative Distance” is the distance to move. This is a 4-byte signed integer that specifies the relative distance in position encoder counts. After the relative move is finished, the controller transmits a message indicating the completion of the movement. This message, known as the "Move Completed" message, is sent as described below.

2.8 MGMSG_MOT_MOVE_COMPLETED

This command does not elicit an immediate response upon the initial message. However, once the relative or absolute move sequence is finished, the controller transmits a “move completed” message. The RX structure is in 20 bytes as follow:

0	1	2	3	4	5	6	7	8	9	10	11
Header						Data					
64	04	0E	00	d	s	Chan Ident		position			

12	13	14	15	16	17	18	19
Data							
Velocity		Motor Current		Status Bits			

The position bytes indicate the measured position in encoder counts, which are controller units. However, the relationship between encoder counts and physical units, such as millimeters or degrees, varies depending on both the controller and the specific stage being used. The actual velocity is measured in controller units, which are specific to the motor and controller being used.

The motor current is measured in milliamperes (mA) and is represented by a signed 16-bit integer, ranging from -32768 to +32767.

“Status Bits” is 32-bit variable providing status information. The command “0x00000200” indicates that the motor is performing a homing move and command “0x00000400” indicates that the motor has completed the homing move, the absolute position is known and therefore the position count is now valid.

2.9 MGMSG_MOT_MOVE_ABSOLUTE

This command is used to start an absolute move on the specified motor channel (using the absolute move position parameter above). As previously described in the “MOVE RELATIVE” command, there are two versions of this command: a shorter (6-byte header only) version and a longer (6-byte header plus 6 data bytes) version. When the first one is used, the absolute move position parameter used for the move will be the parameter sent previously by a MGMSG_MOT_SET_MOVEABSPARAMS command. If the longer version of the command is used, the absolute position is encoded in the data packet that follows the header.

Short version:

The short version has the TX structure in 6 bytes:

0	1	2	3	4	5
Header only					
53	04	Chan Ident	0X	d	s

Long version:

Another method of employing this command is by attaching the MOTABSMOVEPARAMS (absolute move parameters structure) to the message header. The command structure consists of a total of 12 bytes, with the initial 6

bytes representing the header, followed by a 6-byte data packet structured as follows:

0	1	2	3	4	5	6	7	8	9	10	11
Header						Data					
53	04	06	00	d	s	Chan Ident		Absolute Distance			

The Absolute Distance represents the distance to be moved and is expressed as a 4-byte signed integer, indicating the absolute distance in position encoder counts.

Appendix C: all command codes of the communication protocol

Command code	selected DAC	adjustable signals
00000000	U$6 (AD5544)	SIN0, SIN90
00000001	U$12 (AD5544)	SIN45, SIN135
00000010	U$26 (AD5544)	VOH0, VOH2
00000011	U$28 (AD5544)	VOH1, VOH3
00000100	U$21 (AD5544)	V_FREE, REF_INTENSITY
00000101	U$7 (AD5544)	SHIELD, VT_INTEN, VT_POLA
Command codes for continuous readout of all ADCs without sending the readout data to the PC.		
Command codes	Command	
10000111	Start of continuous readout	
00000111	Stop continuous readout	
Command codes for reading out various ADCs once and then sending the read-out data to the PC. The data is sent to the PC in the order in which the ADCs and the signals are arranged in the table (from left to right).		
Command codes	selected ADCs	Signals connected to the "V+" pins of the ADCs
10000110	U$13, U$5	SIN0, SIN90
01000110	U$15, U$14	SIN45, SIN135
11000110	U$13, U$5, U$15, U$14	SIN0, SIN90, SIN45, SIN135
00100110	U$17, U$16	VOH0, VOH2
10100110	U$13, U$5, U$17, U$16	SIN0, SIN90, VOH0, VOH2
01100110	U$15, U$14, U$17, U$16	SIN45, SIN135, VOH0, VOH2

11100110	U$13, U$5, U$15, U$14, U$17, U$16	SIN0, SIN90, SIN45, SIN135, VOH0, VOH2
00010110	U$19, U$18	VOH1, VOH3
10010110	U$13, U$5, U$19, U$18	SIN0, SIN90, VOH1, VOH3
01010110	U$15, U$14, U$19, U$18	SIN45, SIN135, VOH1, VOH3
11010110	U$13, U$5, U$15, U$14, U$19, U$18	SIN0, SIN90, SIN45, SIN135, VOH1, VOH3
00110110	U$17, U$16, U$19, U$18	VOH0, VOH2, VOH1, VOH3
10110110	U$13, U$5, U$17, U$16, U$19, U$18	SIN0, SIN90, VOH0, VOH2, VOH1, VOH3
01110110	U$15, U$14, U$17, U$16, U$19, U$18	SIN45, SIN135, VOH0, VOH2, VOH1, VOH3
11110110	U$13, U$5, U$15, U$14, U$17, U$16, U$19, U$18	SIN0, SIN90, SIN45, SIN135, VOH0, VOH2, VOH1, VOH3

Sworn declaration:

I declare under oath that I have prepared the paper at hand independently and without the help of others and that I have not used any other sources and resources than the ones stated. Parts that have been taken literally or correspondingly from published or unpublished texts or other sources have been labeled as such. This paper has not been presented to any examination board in the same or similar from before.

Date, signature