HAW HAMBURG

BACHELORTHESIS
Jacob Ernsting

Funktionsdemonstrator für magnetische Sensor-Arrays auf Basis des Mikrocomputers Raspberry PI

FAKULTÄT TECHNIK UND INFORMATIK
Department Informations- und Elektrotechnik

Faculty of Computer Science and Engineering
Department of Information and Electrical Engineering

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Jacob Ernsting

Funktionsdemonstrator für magnetische Sensor-Arrays auf Basis des Mikrocomputers Raspberry PI

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Elektro- und Informationstechnik*
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr.-Ing. Karl-Ragmar Riemschneider
Zweitgutachter: Prof. Dr. Paweł Buczek

Eingereicht am: 25. Juni 2020

Jacob Ernsting

Thema der Arbeit

Funktionsdemonstrator für magnetische Sensor-Arrays auf Basis des Mikrocomputers Raspberry PI

Stichworte

Mikrocontroller, Mikrocomputer, Raspberry Pi, Linux, Raspbian, Sensorarray, Magnetfeld, Sensor, Multiplexer, UART, Hardware-Flusskontrolle, OpenGL, GTK 3, ISAR, Demonstrator, C-Quelltext, Vektorfeld, Farbmatrix, grafische Benutzeroberfläche

Kurzzusammenfassung

In dieser Bachelorarbeit wird ein Demonstrator entwickelt, mit dem die Funktionsweise eines Magnetfeld-Sensor-Arrays und wichtige Schritte der Signalverarbeitung grafisch dargestellt werden können. Handbetätigt soll der Demonstrator die Funktionsweise des Sensor-Arrays intuitiv veranschaulichen und Experimente vereinfachen.

In Vorarbeiten ist ein Funktionsmodell der Sensor-Hardware im vergrößerten Maßstab entstanden. Es wird Software erstellt, welche mit Hilfe eines Mikrocontrollers zunächst die Sensordaten des Arrays erfasst und diese über eine serielle Schnittstelle bereitstellt. Ein Mikrocomputer steuert den Mikrocontroller über die serielle Schnittstelle, sammelt die erfassten Messwerte, vollzieht die Signalverarbeitung und stellt die Ergebnisse der Auswertung in anschaulicher Form dar.

Jacob Ernsting

Title of Thesis

Demonstration device for magnetic senor arrays based on the microcomputer Raspeberry PI

Keywords

Microcontroller, Microcomputer, Raspberry Pi, Linux, Raspbian, sensor-array, magnetic feld, sensor, multiplexer, UART, hardware-flow-control, OpenGL, GTK 3, ISAR, demonstrator, C-Code, vectorfield, heatmap, grafical user interface

Abstract

In this thesis a demonstration device is developed that can visualize the function of a magnetic-field-sensor-array and important steps of the signal processing. Handoperated, this device shall make it easy to experiment with and intuitively understand the concept.

An upscaled functional model of a sensor-array was the result of previous theses. To evaluate the sensor-signals of the array software is written to run on a microcontroller. This microcontroller prepares the signals and provides them over a serial interface. A microcomputer uses this interface to control the microcontroller and collect the sensordata which is then processessed and visualized on a graphical user interface.

Inhaltsverzeichnis

1 Einleitung

1.1 Thema

Bei der Erfassung von Drehwinkeln rotierender Achsen mit Magnetfeldsensoren wird ein Gebermagnet auf dem rotierenden Objekt platziert, dessen Magnetfeld im rechten Winkel zur Rotationsachse ausgerichtet ist. Der Magnetfeldsensor erfasst das Magnetfeld in zwei zueinander und zur Rotationsachse orthogonal stehenden Richtungen. Mit einfacher Vektorrechnung lässt sich anhand der Feldstärke der jeweiligen Richtung der Winkel des rotierenden Objekts bestimmen. Die Anordnung des Sensors und Gebermagneten ist in der Abbildung 1.1 dargestellt. Anwendungsbereiche liegen zum Beispiel in der Automobilindustrie zur Erfassung der Rotorlage von Elektromotoren oder der Stellung von Drosselklappen.

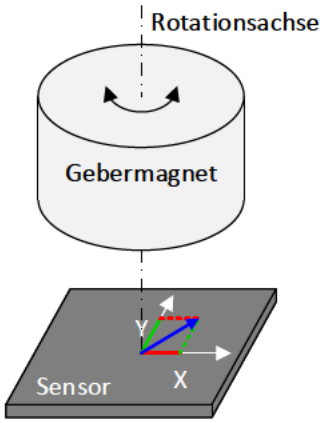


Abbildung 1.1: Anordnung des Gebermagneten und Magnetfeldsensors zur Erfassung von Drehwinkeln. Der Gebermagnet sitzt in horizontaler Ausrichtung über dem Magnetfeldsensor. Die Rotationsachse des Magneten verläuft durch den Mittelpunkt des Sensors. Die Addition der Magnetfeldvektoren des Sensors zur Winkelberechnung ist angedeutet.

In der Praxis auftretende externe Störmagnetfelder stellen für dieses Konzept jedoch ein Problem dar, denn sie verfälschen den vom Sensor ermittelten Winkel. Das externe Magnetfeld verhält sich additiv mit dem des Gebermagneten, sodass der Sensor nicht zwischen dem Gebermagnetfeld und dem der Störquellen unterscheiden kann. Übersteigt die Störfeldstärke im Sensor die des Gebermagneten, wird das Messergebnis gänzlich unbrauchbar. Abbildung 1.2 zeigt, welchen Einfluss ein homogenes Störfeld auf den ermittelten Winkel hat.

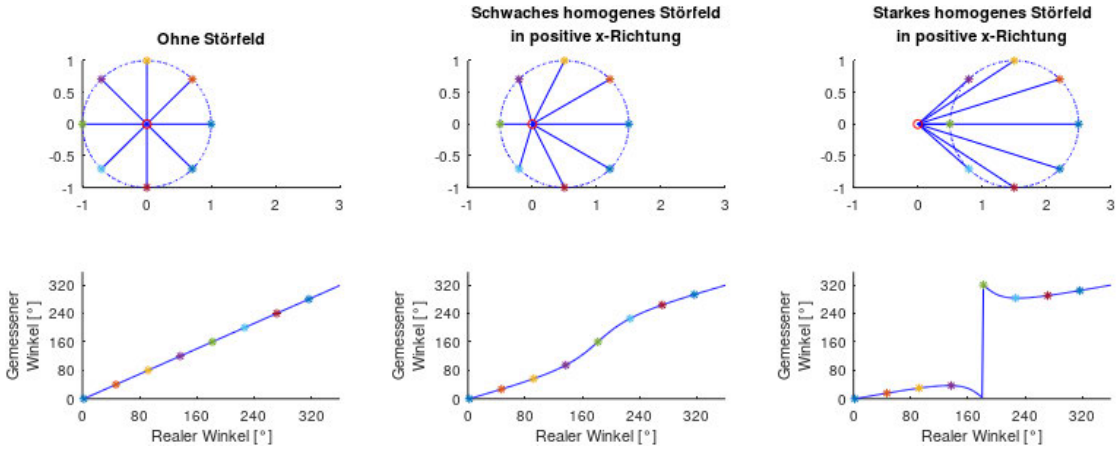


Abbildung 1.2: Einfluss eines Störmagnetfeldes auf den gemessenen Winkel eines einzelnen Magnetfeldsensors. Drei Graphenpaare zeigen den Einfluss eines homogenen Störmagnetfeldes auf den gemessenen Winkel. Der obere Graph zeigt den Magnetfeldvektor am rot markierten Sensor und die Kreisbahn, die dessen Spitze bei rotierendem Gebermagneten beschreibt. Der untere Graph stellt das Verhalten des gemessenen Winkels dar. Die linke Spalte zeigt den idealen Fall. Es liegt kein Störmagnetfeld an und der gemessene Winkel entspricht exakt dem realen. Die mittlere Spalte zeigt den Fall, in dem ein Störmagnetfeld anliegt, das die Messung zwar verfälscht, die Winkelmessung jedoch nicht gänzlich unbrauchbar macht. In der rechten Spalte ist der Fall dargestellt, der ein Messen des Rotationswinkels unmöglich macht. Die Feldstärke des Störmagnetfeldes verschiebt den durch den Magnetfeldvektor beschriebenen Kreis so weit, dass der Sensor außerhalb liegt.

Um Störfelder herauszufiltern und zuverlässig Aussagen über den Drehwinkel und die Ausrichtung des Sensors und Gebermagneten zur Rotationsachse zu treffen, kann ein Array der Magnetfeldsensoren eingesetzt werden, das zusammen mit digitaler Signalverarbeitung genaue Ergebnisse liefern kann. Die Forschungsgruppe Sensorik des Instituts für Informationstechnik und verteilte Systeme der Hochschule für Angewandte Wissenschaf-

ten (HAW) Hamburg arbeitet an der Entwicklung und Implementierung von Algorithmen und Sensoren in Hardware [18], um das Konzept praktisch umzusetzen.

1.2 Stand und Vorarbeiten

Aus einem Bachelor-Projekt [4] entstand bereits ein Demonstrationsaufbau, dessen Mechanik die Grundlage für diese Arbeit bildet. Er besteht aus einer Grundplatte aus Holz, auf der das Magnetfeld-Sensor-Array sowie eine Mikrocontroller-Entwicklungsplatine und ein Farbbildschirm montiert sind. An einer Plexiglasplatte, welche mit Gewindestangen parallel zur Grundplatte befestigt ist, hängt über dem Sensor-Array ein Pendel mit Handknauf. Das Pendel trägt ein Halbach-Permanentmagnet-Array, welches für ein nahezu homogenes Magnetfeld über dem Sensor-Array sorgt. Der Benutzer kann das Pendel mit dem Handknauf und damit den Gebermagneten über dem Sensor-Array schwenken und rotieren. Abbildung 1.3 zeigt die bisherige Konstruktion auf der linken und die in dieser Arbeit überarbeitete auf der rechten Seite. Das Sensor-Array entstand zusammen mit einem Mikrocontrollerprogramm in einer Bachelor-Arbeit [6]. Zum Auslesen des Arrays und zur Signalverarbeitung existiert C-Quelltext aus zwei weiteren Bachelor-Arbeiten, die sich zum einen mit der Erarbeitung eines Vorgängers des aktuellen Sensor-Arrays befassten [1] und zum anderen mit der Signalverarbeitung der aufgenommenen Magnetfelddaten [11]. Die Signalverarbeitung wird in dieser Arbeit jedoch nur zur Einbindung vorbereitet und ist nicht Bestandteil.

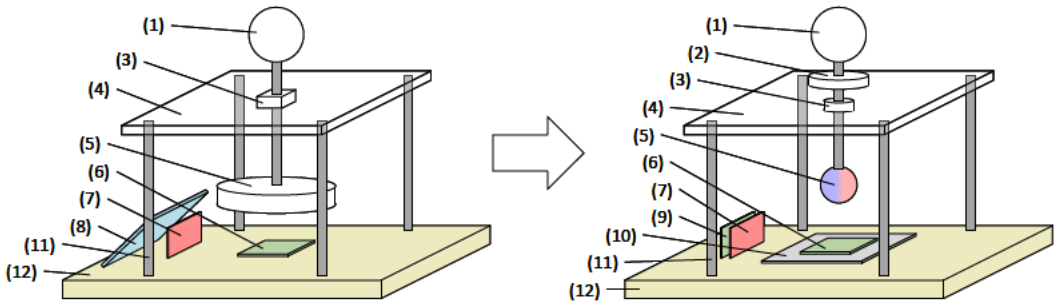


Abbildung 1.3: Die Darstellung zeigt schematisch links den bisherigen Demonstrationsaufbau und rechts den zukünftigen. Die Komponenten sind mit Nummern versehen, um sie einfach benennen zu können: (1) Pendelachse mit Handknauf, (2) Höhenstellrad, (3) Pendellager, (4) Trägerplatte, (5) Gebermagnet, (6) Sensor-Array, (7) Mikrocontroller, (8) Bildschirm, (9) Mikrocomputer, (10) Montageplatte, (11) Gewindestangen, (12) Grundplatte

1.3 Inhalt und Ziel dieser Arbeit

Ziel dieser Arbeit ist es, das Konzept des Sensor-Arrays mit Hilfe des handbetätigten Demonstrationsaufbaus zu veranschaulichen und die Messung und Signalverarbeitung auf einem Bildschirm in Echtzeit anzuzeigen. Dies soll dem Entwicklungsteam die Möglichkeit geben, ihre Konzepte zu verifizieren und Ergebnisse vorzuführen.

Der Demonstrationsaufbau wird um ein Höhenstellrad ergänzt, das Pendellager überarbeitet und ein Mikrocomputer nimmt den Platz des Bildschirms ein. Das Permanent-Magnet-Array mit einem möglichst homogenen Magnetfeld über dem Sensor-Array weicht einem Kugelmagneten mit möglichst inhomogenen Magnetfeld. Das inhomogene Magnetfeld des kugelförmigen Gebermagneten erlaubt, durch seine starke Krümmung das Herausfiltern des homogenen Störfeldes, ohne das Gebersignal zu verlieren. Aufgrund der kleinen Größe des geplanten Sensor-Arrays von wenigen Millimetern und des verhältnismäßig großen Abstands zur potentiellen Störquelle, kann das Störmagnetfeld als nahezu homogen angenommen werden.

Im Folgenden dieser Arbeit werden die Themen Systemkonzipierung, Kommunikation, Software, Grafikdarstellung und Mechanik bearbeitet. Externe Quellen, Vorlagen und Materialien sind entsprechend gekennzeichnet.

2 Grundlagen

2.1 Auswahl der Komponenten

Das Datenerfassungs- und Verarbeitungssystem besteht aus den drei Hauptkomponenten Sensor-Array, Mikrocontroller und Mikrocomputer, welche am Demonstrationsaufbau montiert sind. Diese Komponenten tauschen digitale und analoge Signale aus, um im Verbund die Gesamtfunktion zu erfüllen. Als Nutzerschnittstelle dienen ein Bildschirm, eine Computer-Maus und eine Tastatur. Die zur Durchführung des Projekts gestellten Baugruppen sind in der Tabelle 2.1 aufgelistet.

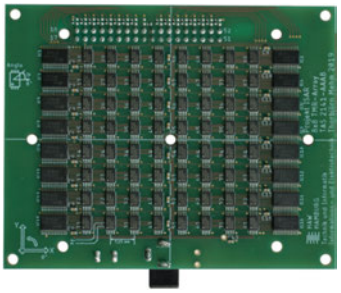
Komponente	Beschreibung	Abbildung
Sensor-Array: 8x8 TMR-Sensor-Array *TAS 2141-AAAB*	Platine mit 64 Tunnel Magnetoresistance (TMR)-Sensoren, angeordnet in einer 8x8-Matrix. Die Sensoren gibt vier Analogsignale aus. Je zwei für die x- und y-Achse, deren Differenz dem gemessenen Magnetfeld der Achse entspricht. Über 16 Analog-Multiplexer lassen sich alle Analogsignale durch Adressierung per parallelem Adressbus auf die Analogpins des Arrays schalten.	 [6]
Mikrocontroller: Texas Instruments Tiva C Series TM4C1294 Connected LaunchPad Evaluation Kit *EK-TM4C1294XL Rev.D*	Mikrocontroller-Entwicklungsplatine mit GPIO-Pins, RJ-45- und Micro-AB-USB-Buchse, sowie einem Reset- und drei Tastern. Über einen Debug-Chip lässt sich der Mikrocontroller per Micro-B-USB-Anschluss programmieren und debuggen.	 [13]
Mikrocomputer: Raspberry Pi Foundation *Raspberry Pi 3 Model B V1.2*	ARM-basierter Mikrocomputer mit Konnektivität für GPIO-Pins, USB, HDMI, Analog-Audio, serielle Kamera und Monitor, sowie Netzwerk, Bluetooth und WLAN.	 [8]

Tabelle 2.1: Liste der Hauptkomponenten für das Datenerfassungs- und Verarbeitungssystem

2.2 Verteilung der Grundfunktionen auf die Komponenten

Welche Funktionen mit welcher Komponente realisiert wird, ist in der Abbildung 2.1 dargestellt.

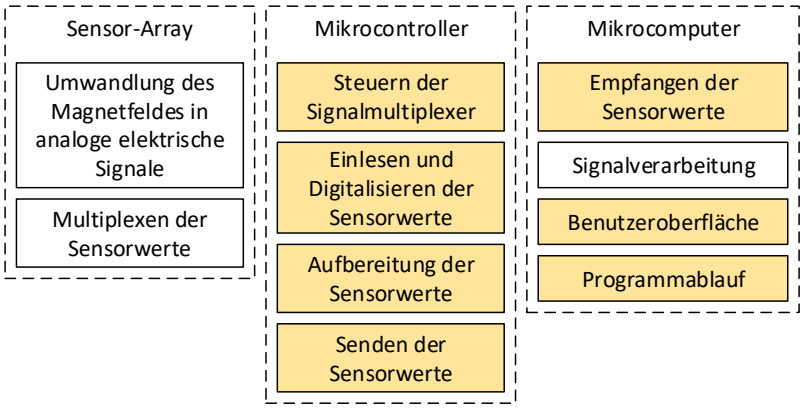


Abbildung 2.1: Funktionen nach Komponente aufgelistet. Der gestrichelte Rahmen mit der Überschrift markiert die verschiedenen Komponenten des Aufbaus. Darin aufgelistet sind die jeweiligen Aufgaben und Funktionen. Die farbig markierten Funktionen sind Teil dieser Arbeit.

Die Erfassung und Wandlung des Magnetfeldes in analoge elektrische Signale geschieht mit dem Sensor-Array. Dessen analoge Multiplexer werden durch den Mikrocontroller gesteuert, welcher die Sensoren des Arrays sequentiell ausliest, die Messwerte digitalisiert und aufbereitet. Sendet der Mikrocomputer eine Anfrage über den seriellen Bus an den Mikrocontroller, führt dieser die Anfrage aus und antwortet dem Mikrocomputer mit den entsprechenden Daten. Sind die Daten am Mikrocomputer angekommen, führt dieser die Signalverarbeitung aus und bietet dem Bediener mit Hilfe eines Bildschirms und einer Maus verschiedene Möglichkeiten der Steuerung und Visualisierung.

2.3 Signalaustausch zwischen den Komponenten

Der Mikrocontroller kommuniziert sowohl analog als auch digital mit dem Sensor-Array, um das Magnetfeld in digitale Werte umzuwandeln. Die Schnittstelle besteht dabei aus einem parallelen Adressbus, der zur Auswahl der analogen Messwerte über die Multiplexer des Sensor-Arrays dient, und einem parallelen analogen Bus, welcher die Analogwerte an

den Mikrocontroller überträgt. Über einen seriellen Bus werden die digitalisierten Analogwerte des Arrays auf Anfrage des Mikrocomputers vom Mikrocontroller übertragen. Abbildung 2.2 zeigt die Systemübersicht mit den Datenverbindungen.

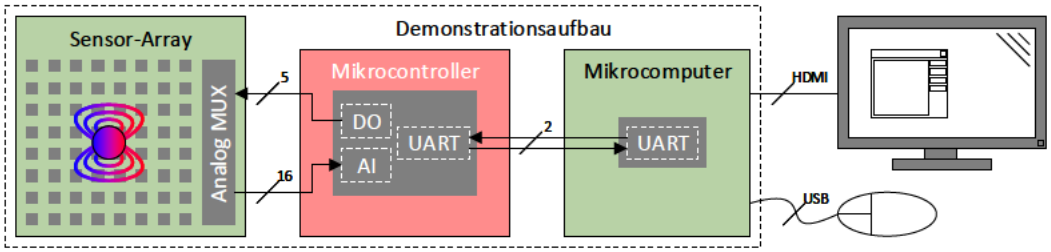


Abbildung 2.2: Übersicht über die Komponenten für das Datenerfassungs- und Verarbeitungssystem und deren Kommunikationsverbindungen

Die serielle Kommunikation zwischen Mikrocomputer und Mikrocontroller erfolgt mittels Universal Asynchronous Receive-Transmit (UART)-Protokoll. Der Mikrocontroller empfängt Befehle und ausführt diese aus. Der Mikrocomputer sendet Befehle und wartet auf die Antwort. Dauert die Übertragung länger als erwartet oder wird ein Übertragungsfehler festgestellt, ignoriert der Mikrocomputer die Antwort, verwirft alle bisher empfangenen Daten und sendet den Befehl erneut.

2.4 Kommunikation zwischen Mikrocomputer und Mikrocontroller

2.4.1 Auslegung der seriellen Schnittstelle

Der Mikrocontroller stellt eine serielle UART-Schnittstelle zur Verfügung, welche es ihm ermöglicht, Befehle zu empfangen und Daten zu senden. Die Übertragung geschieht nach der Data/Parity/Stop (DPS)-Notation in der Form 8E1 mit einer Bit-Rate von 460800 $\frac{b}{s}$. Die Abkürzung beschreibt die Übertragung mit acht Datenbits, einem geraden Paritätsbit und einem Stoppbit. Abbildung 2.3 zeigt die Struktur der DPS-Notation. Die Herleitung der Übertragungsrate ist in Kapitel 2.5 erläutert.

Die serielle Schnittstelle verfügt neben den Datenleitungen über Hardware-Flusskontrolle mittels Clear To Send (CTS)- und Ready To Send (RTS)-Signal. Diese Signale werden,

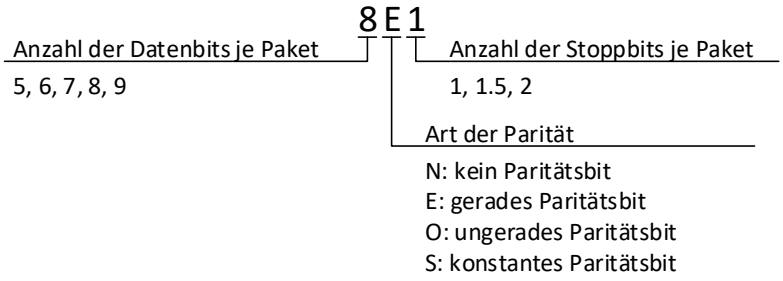


Abbildung 2.3: DPS-Notation. Die erste Ziffer nennt die Anzahl an Datenbits, die je übertragenem Paket zwischen dem Start- und den Stoppbits übermittelt werden. Der Buchstabe nennt die Art der Parität je Paket. Ein N (Abkürzng für englisch none) bedeutet, dass im Gegensatz zu den anderen Optionen kein Paritätsbit vorhanden ist. Ein E (von englisch even) bedeutet, dass das Paritätsbit so übertragen wird, dass die Anzahl an logisch wahren Bits exklusive der Start- und Stoppbits gerade ist. Das Gegenteil trifft bei dem Buchstaben O (Abkürzung für englisch odd) zu. Hier wird die Anzahl der logisch wahren Bits exklusive der Start- und Stoppbits ungerade gehalten. Ein S (Abkürzung für englisch stick) bedeutet, dass das Paritätsbit unabhängig vom Rest des übertragenen Pakets einen konstanten Wert hat. Das Paritätsbit ist in diesem Fall immer logisch wahr oder unwahr und ändert seinen Wert nicht. Die zweite Ziffer nennt die Anzahl der Stoppbits, die das Ende des übertragenen Pakets markieren. Der Wert ein-ein-halb entspricht der ein-ein-halb-fachen Länge eines Bits.

wie auch die Leitungen Receive-Data (RxD) und Transmit-Data (TxD), über Kreuz zwischen beiden Kommunikationspartnern verbunden. Die Verbindung ist in der Abbildung 2.4 dargestellt.

Schaltet der Empfänger den RTS-Ausgang auf den elektrischen low-Pegel, erkennt der Sender durch Beobachten dessen CTS-Eingang, dass der Empfänger bereit ist, Daten zu empfangen. Es ist möglich, die Kommunikation ohne aktive Beschaltung der jeweiligen CTS-Eingänge zu betreiben, indem diese dauerhaft mit dem elektrischen low-Pegel verbunden werden. Für den fehlerfreien Betrieb sollten die Flusskontroll-Signale jedoch aktiv genutzt werden, da die First In First Out (FIFO)-Puffer der Kommunikationspartner ansonsten volllaufen und Daten verloren gehen können.

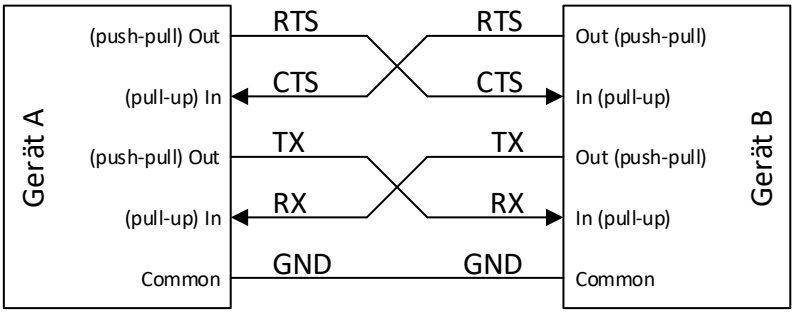


Abbildung 2.4: Elektrische Verbindung der Kommunikationspartner für die serielle Übertragung mittels UART-Protokoll mit Hardware-Flusskontrolle. Die Anschlussbeschriftung der Geräte beschreibt die elektrische Funktion der Anschlüsse.

Die Steuerung durch das Verfahren der Hardware-Flusskontrolle ist im Vergleich zu den Alternativen mit größerem Verdrahtungsaufwand verbunden. In der Tabelle 2.2 sind Möglichkeiten aufgelistet, die für die Schnittstelle in Betracht gezogen werden.

Die zwei Lösungen, die für dieses Projekt in Frage kommen sind der erste Lösungsansatz ohne Paketflusssteuerung und der dritte mit Hardware-Flusskontrolle. Da die Übertragungszeit in diesem Projekt aufgrund der hohen Bilderneuerungsrate kritisch ist, erfordert der erste Ansatz eine sehr hohe Übertragungsrate und schnelle Reaktionszeit, welche insbesondere beim betriebssystemgesteuerten Mikrocomputer nicht gewährleistet werden kann. Der dritte Lösungsansatz ist damit am besten geeignet.

Lösungsansatz	Vor-/Nachteile
1. Lösungsansatz **Keine Paketflusssteuerung:** Die Kommunikation geschieht im strengen Wechsel. Der Sender überträgt nur ein Paket und wartet dann auf Antwort des Empfängers, welcher den Empfangspuffer nach dem Empfangen beim Auslesen leert.	**Nachteile:** Es ist viel Steuerkommunikation zwischen den Kommunikationspartnern nötig. Dies verlängert die Übertragungszeit im Vergleich zu anderen Lösungen erheblich, da die Partner jedes empfangene Paket bestätigen müssen. Bei der Übertragung von großen Datenmengen in nur einer Richtung, würde das besonders ins Gewicht fallen. Ein weiterer Nachteil ist der, dass die Kommunikationspartner nicht über die gegenseitige Verfügbarkeit wissen. Der Sender muss einen Sendeversuch unternehmen und auf eine Antwort oder Zeitablauf warten. **Vorteile:** Bei einem Paritätsfehler kann das fehlerhafte Paket einzeln erneut angefordert werden.
2. Lösungsansatz **Software-Flusskontrolle:** Der Sender wertet die Pakete des Empfängers auf bestimmte Signale aus, die den Datenfluss steuern.	**Nachteile:** Für diese Auswertung ist es erforderlich, dass Pakete als American Standard Code for Information-Interchange (ASCII)-Zeichen ausgewertet werden. Damit ist nicht der gesamte Wertebereich von 0x00 bis 0xFF für Daten und Befehle verfügbar, sodass Zahlenwerte zifferweise übertragen werden müssen. Somit nimmt diese Lösung verhältnismäßig viel Zeit zur Übertragung in Anspruch. **Vorteile:** Dieser Ansatz spart Steuerkommunikation im Vergleich zum vorherigen 1. Lösungsansatz. Zusätzliche Pakete zur Flusskontrolle sind jedoch weiterhin notwendig.
3. Lösungsansatz **Hardware-Flusskontrolle:** Beide Kommunikationspartner verfügen wie zuvor beschrieben über separate Steuersignale neben den Datensignalen.	**Nachteile:** Anstatt drei (RX, TX, GND) sind fünf Leiter (RX, TX, RTS, CTS, GND) zur Kommunikation nötig. Dies erhöht den Verdrahtungsaufwand. **Vorteile:** Im Gegensatz zur Software-Flusskontrolle sind keine Flusskontrollpakete nötig. Dadurch kann der Wertebereich und das verfügbare Zeitfenster für den Datenaustausch vollständig für Daten genutzt werden. Weiterhin erkennt der Sender, wann der Empfänger empfangsbereit ist und startet erst dann einen Sendeversuch.

Tabelle 2.2: Lösungsansätze mit Vor- und Nachteilen für die Paketflusskontrolle der seriellen Schnittstelle

2.4.2 Kommunikationsprotokoll

Die Kommunikation funktioniert nach dem Master-Slave-Prinzip. Der Slave (englisch Sklave) erhält Befehle, die der Master (englisch Meister oder Herr) erteilt. Für die Realisierung dieser Arbeit ist der Mikrocomputer der Master, der die Daten anfragt und verarbeitet, und der Mikrocontroller der Slave, welcher sie auf Anfrage sammelt und bereitstellt. Wie zuvor beschrieben, geschieht die Übertragung in der Form 8E1. Die Auswertung der übertragenen Bits geschieht bitweise. Dadurch stehen alle acht Datenbits zur Verfügung.

Master

Der Master sendet ein Anfragepaket an den Mikrocontroller, welches das Paket wie in der Abbildung 2.5 dargestellt in zwei Bereiche unterteilt: Den Befehl mit 3 Bit Länge und das Argument mit 5 Bit Länge.

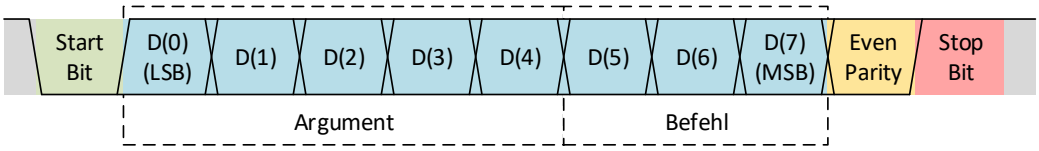


Abbildung 2.5: Struktur des Anfragepakets, das vom Master an den Slave gesendet wird. Es besteht aus einem Startbit, acht Datenbits, einem Paritätsbit und einem Stoppbit. Die Datenbits sind in zwei Bereiche unterteilt: 3 Bit für den Befehl und 5 Bit für das Argument des Befehls.

Hierbei ist die Position beider Teile frei gewählt. Das UART-Protokoll sendet das Least Significant Bit (LSB) zuerst, sodass die Befehlsbits innerhalb eines Pakets zeitlich nach den Informationsbits gesendet werden. Der Befehl kann eine von acht möglichen Kombinationen annehmen, welche die Aufgabe des Mikrocontrollers bestimmt. Das Argument enthält Informationen zur Ausführung des Befehls. Die entsprechende Zuordnung kann der Tabelle 2.3 entnommen werden.

Wird ein Befehl an den Slave gesendet während dieser einen iterativen Prozess ausführt, wie beispielsweise das Senden von mehreren Datenpaketen, bricht er diesen ab und bearbeitet die neue Anfrage des Masters.

Befehl [3]		**Argument [5]**
Beschreibung	Übertragung (MSB→LSB)	Beschreibung
Keine Aktion	**000**	**X XXXX**	beliebig; wird ignoriert
Array-Daten aufnehmen	**001**	**X XXXX**	beliebig; wird ignoriert
Array-Daten aufn. & senden	**010**	**X XXXX**	beliebig; wird ignoriert
nicht vergeben	**011**	**X XXXX**	nicht vergeben
Array-Daten senden	**100**	**X XXXX**	beliebig; wird ignoriert
Array-Daten-Zeile senden	**101**	**0 0000 … 1 1111**	Zeilennummer
nicht vergeben	**110**	**X XXXX**	nicht vergeben
nicht vergeben	**111**	**X XXXX**	nicht vergeben

Tabelle 2.3: Gültige Anfragepakete des Übertragungsprotokolls zwischen Master und Slave.

Slave

Die Antwort des Slaves auf eine Anfrage ist nach Beendigung der angefragten Aufgabe das empfangene Anfragepaket, gegebenenfalls gefolgt von den angefragten Daten.

Für Antworten auf Befehle, die lediglich eine Befehlsbestätigung des Slaves benötigen und keine zurückzusendenden Daten haben, ist die Anzahl der Pakete nach Gleichung 2.1 eins.

$$n_{Pakete/Befehl} = n_{Bestätigung} = 1 \tag{2.1}$$

Die Anfrage einer Zeile des Arrays benötigt ein Paket für die Bestätigung addiert zu dem Produkt aus der Anzahl der Sensoren in x-Richtung, der Anzahl an Werten pro Sensor und der Anzahl an Paketen pro Wert. Gleichung 2.2 zeigt die Rechnung.

$$\begin{aligned} n_{Pakete/Zeile} &= n_{Bestätigung} + n_{xSensoren} \cdot n_{Signale/Sensor} \cdot n_{Pakete/Wert} \\ &= 1 + 8 \cdot 2 \cdot 2 \\ &= 33 \end{aligned} \tag{2.2}$$

Die Anzahl der übertragenen Pakete pro Array-Daten-Anfrage, auch mit Aufnahme der Daten, ergibt sich mit eins für das wiederholte Anfragepaket zur Bestätigung addiert zu dem Produkt der Anzahl der Sensoren in x- und y-Richtung, der Anzahl an Werten pro

Sensor und der Anzahl an Paketen pro Wert. Gleichung 2.3 zeigt die Rechnung.

$$n_{Pakete/Array} = n_{Bestätigung} + n_{xSensoren} \cdot n_{ySensoren} \cdot n_{Signale/Sensor} \cdot n_{Pakete/Wert} \tag{2.3}$$
$$= 1 + 8 \cdot 8 \cdot 2 \cdot 2$$
$$= 257$$

Der Faktor $n_{Pakete/Wert}$ ist notwendig, da das UART-Protokoll maximal acht Datenbits zulässt. Daten, deren Länge acht Bit überschreitet, müssen daher aufgespalten werden. Für das Sensor-Array heißt dies, dass die x- und y-Werte, welche im 16 Bit langen Datentyp *signed integer* gespeichert sind, zwei Übertragungspakete pro eigentlichem Wert benötigen. Nach der Aufteilung wird das höherwertige Byte zuerst gesendet. Abbildung 2.6 veranschaulicht die Aufteilung eines 16 Bit Wertes.

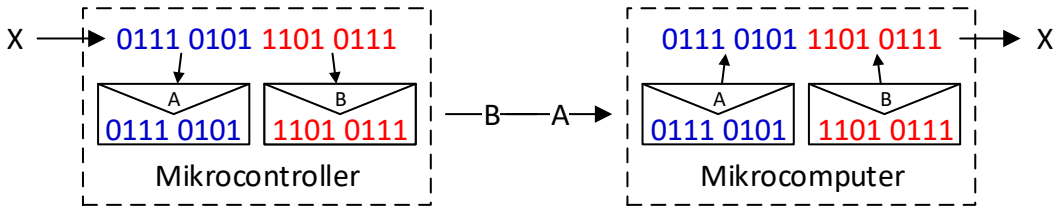


Abbildung 2.6: Prinzip der Aufteilung von Daten zur Übertragung in kleineren Paketen

2.5 Übertragungsrate und Zeitverhalten

Die Aufgaben der Prozessoren sind das Aufnehmen, Senden und Empfangen von Daten über die serielle Schnittstelle, Verarbeiten und Visualisieren von Daten, sowie die Bereitstellung einer Benutzeroberfläche zur Bedienung. Diese Aufgaben werden sequentiell abgearbeitet. Abbildung 2.7 zeigt einen beispielhaften Ablauf.

Die Zykluszeit ergibt sich aus der Summe der Zeiten der sequentiellen Einzelaufgaben. Der ungünstigste Fall für die Zeitanalyse mit fehlerfreier Übertragung ist der, dass die Array-Daten durch einen separaten Befehl aufgenommen und reihenweise angefragt werden. In diesem Fall entsteht der größte Kommunikationsaufwand zwischen Master und

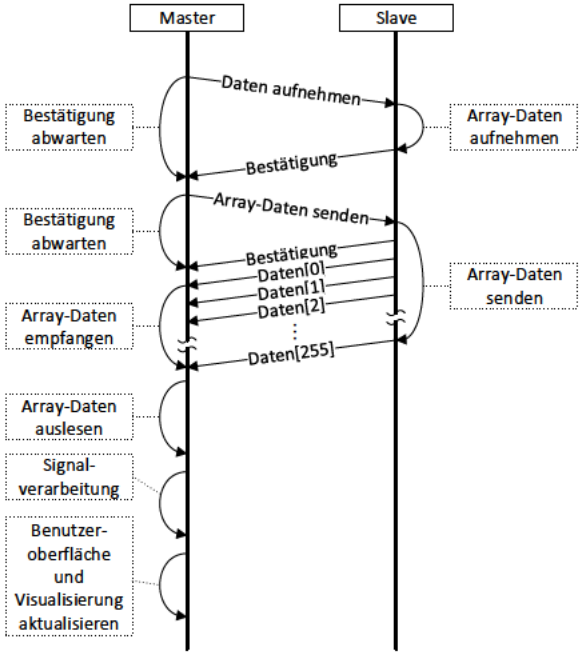


Abbildung 2.7: Beispielhafter Kommunikationsablauf zwischen Master und Slave für den Hauptprogrammzyklus. Der Master sendet den Befehl zum Aufnehmen der Sensor-Array-Daten an den Slave und wartet, bis dieser antwortet. Hat der Slave geantwortet, sind die Daten aufgenommen. Nun sendet der Master den Befehl zum Senden der Array-Daten. Der Slave folgt dem Befehl. Der Master liest die empfangenen Daten aus seinem Empfangspuffer aus und unterzieht sie der Signalverarbeitung. Zuletzt aktualisiert er die Visualisierung und der Ablauf beginnt von vorn.

Slave. Dieser Ablauf ist in der Abbildung 2.8 dargestellt. Den günstigsten Fall, dass die Aufnahme und das Senden mit einem Befehl angefragt werden, zeigt die Abbildung 2.9.

Die Zeiten zur Ausführung der Programmteile hängen von den Prozessortakten, den Quelltexten und der Prozessorauslastung des Mikrocomputers durch andere Programme ab. Da die Prozessortakte festgelegt sind und keine weiteren Programme auf dem Mikrocomputer neben dem Programm für den Demonstrationsaufbau ausgeführt werden sollen, können die Programmzeiten als konstant angenommen werden. Die Ermittlung der jeweiligen Zeiten ist im Kapitel 6 ausgeführt. Da die Signalverarbeitung nicht Teil dieser Arbeit ist, kann hierfür keine Zeit festgestellt werden. Variabel ist auch die Zeit für die Übertragung der Daten zwischen Master und Slave durch Anpassung der Übertragungsrate. Die Zykluszeit ist also abhängig von der Signalverarbeitungszeit und Übertragungsrate

der seriellen Schnittstelle. Für den in der Abbildung 2.9 beschriebenen günstigsten Fall ergibt sich die Zykluszeit in Gleichung 2.4.

$$\begin{aligned} t_{Zyklus} = {} & t_{Array-Daten\ aufnehmen\ \&\ senden} + (1 + n_{Paket/Befehl}) \cdot t_{Paket} + \\ & t_{Array-Daten\ auslesen} + (1 + n_{Pakete/Zeile}) \cdot t_{Paket} + \\ & t_{Signalverarbeitung} + t_{Benutzeroberfläche\ aktualisieren} \end{aligned} \tag{2.4}$$

Die Aktualisierungsrate ist der Kehrwert der Zykluszeit. Es wird eine Aktualisierungsrate $f_{Zyklus} = 30\ Hz$ angestrebt, um eine flüssige Animation und kurze Reaktionszeiten zu erzielen. Wird die Gleichung 2.4 umgestellt, ergibt sich die Gleichung 2.5 für die Signalverarbeitungszeit.

$$\begin{aligned} t_{Signalverarbeitung} = {} & t_{Zyklus} - \\ & t_{Array-Daten\ aufnehmen\ \&\ senden} - (1 + n_{Paket/Befehl}) \cdot t_{Paket} - \\ & t_{Array-Daten\ auslesen} - (1 + n_{Pakete/Zeile}) \cdot t_{Paket} - \\ & t_{Benutzeroberfläche\ aktualisieren} \end{aligned} \tag{2.5}$$

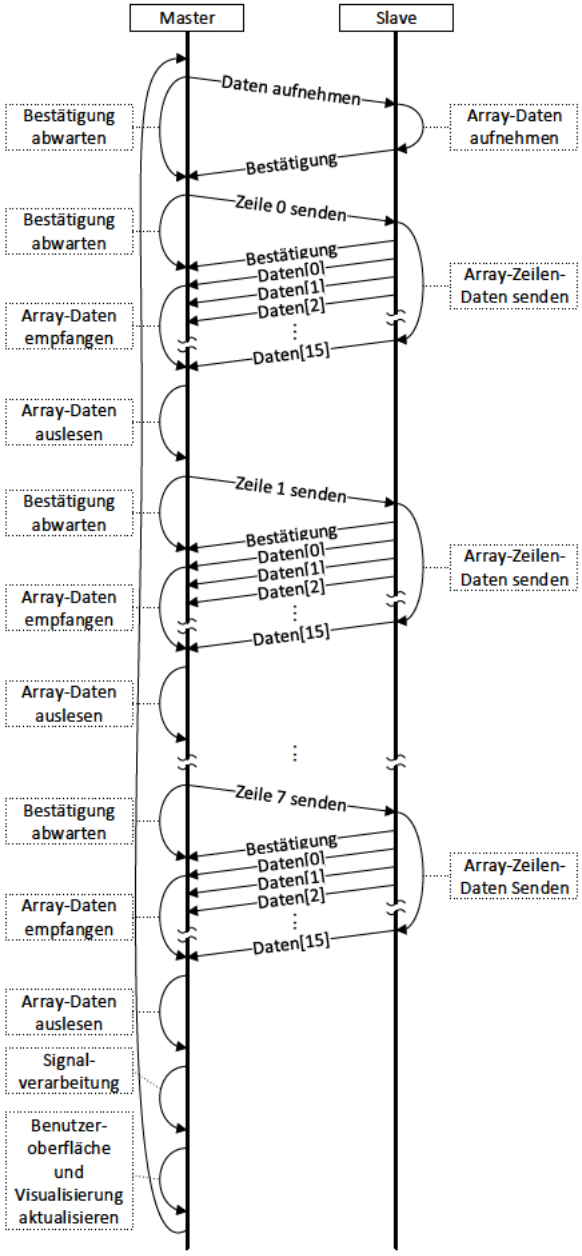


Abbildung 2.8: Beispielhafter Kommunikationsablauf zwischen Master und Slave für den Hauptprogrammzyklus mit längster Zykluszeit. Der Master sendet den Befehl zum Aufnehmen der Sensor-Array-Daten an den Slave und wartet, bis dieser antwortet. Hat der Slave geantwortet, fragt der Master die Array-Daten zeilenweise an. Hat der Master alle Daten vom Slave empfangen, unterzieht er diese der Signalverarbeitung und aktualisiert schließlich die Benutzeroberfläche und Visualisierung.

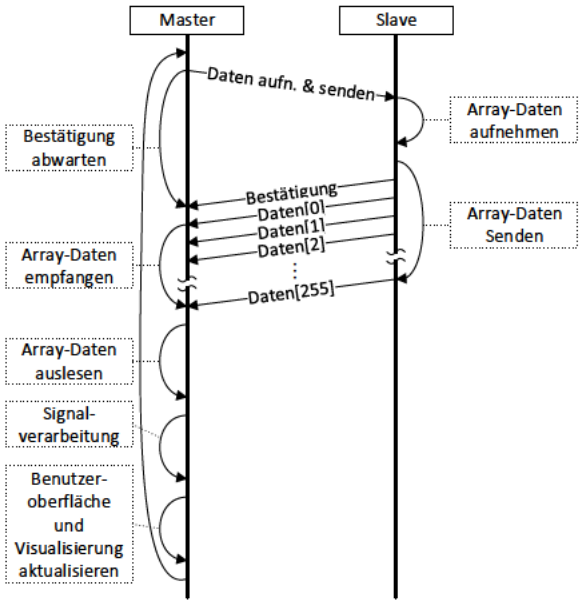


Abbildung 2.9: Beispielhafter Kommunikationsablauf zwischen Master und Slave für den Hauptprogrammzyklus mit kürzester Zykluszeit. Der Master sendet den Befehl zum Aufnehmen der Sensor-Array-Daten an den Slave und wartet, bis dieser antwortet. Hat der Slave geantwortet, sind die Daten aufgenommen und werden zum Master gesendet. Der Master liest die empfangenen Daten aus seinem Empfangspuffer aus und unterzieht sie der Signalverarbeitung. Zuletzt aktualisiert er die Visualisierung und der Ablauf beginnt von vorn.

3 Implementierung auf dem Mikrocontroller

3.1 Zusammenfassende Funktionsbeschreibung

Der Mikrocontroller agiert als Slave. Empfängt er ein Paket über die serielle Schnittstelle, bearbeitet er den im Paket enthaltenen Befehl ohne Verzögerung. Ist die Bearbeitung abgeschlossen, antwortet er mit dem von ihm empfangenen Befehlspaket. Erfordert der Befehl weitere Daten als Antwort, folgen diese dem ersten Antwortpaket. Stimmt der Inhalt des ersten Antwortpakets mit dem des Befehlspakets des Masters überein, hat der Slave den richtigen Befehl ausgeführt. Wird während der Bearbeitung eines Befehls ein neues Paket an den Slave gesendet, stoppt dieser die aktuelle Bearbeitung und führt stattdessen den neuen Befehl aus. Eine Liste der gültigen Befehle ist in der Tabelle 2.3 zu finden.

Für die Aufnahme der Sensor-Daten beschaltet der Slave den Adressbus der Multiplexer des Arrays, um sequentiell alle Analogwerte der Sensoren auf den Analogbus zu schalten. Der Analogbus wird mit Hilfe der integrierten Multiplexer des Mikrocontrollers auf die internen Module der Analog-to-Digital-Converter (ADC) geschaltet, ausgewertet und gespeichert. Aus den differenziellen Rohsignalen der Sensoren errechnet der Mikrocontroller dann die Sensorwerte. Auf den entsprechenden Befehl hin, werden die Sensorwerte wie im Kapitel 2.4.2 beschrieben über die serielle Schnittstelle an den Master gesendet.

3.2 Elektrischer Anschluss

Der Anschluss des Mikrocontrollers erfolgt mit steckbaren Verbindungsleitungen. Der Mikrocomputer versorgt den Mikrocontroller, welcher wiederum das Sensor-Array mit der Betriebsspannung versorgt. Alle Verbindungen aus Sicht des Mikrocontrollers sind in der

Tabelle 3.1 aufgelistet. Die erste Spalte benennt die Pinleiste mit dem zugehörigen Pin, die zweite den entsprechenden Port des Mikrocontrollers und Spalte drei die Funktion des Pins.

Zur Veranschaulichung ist in der Abbildung 3.1 die Platine des Mikrocontrollers mit den Anschlussleitungen und Steckbrücken skizziert. Die Steckbrücken sind so konfiguriert, dass die Spannungsversorgung des Mikrocontrollers über die Anschlusspins geschieht.

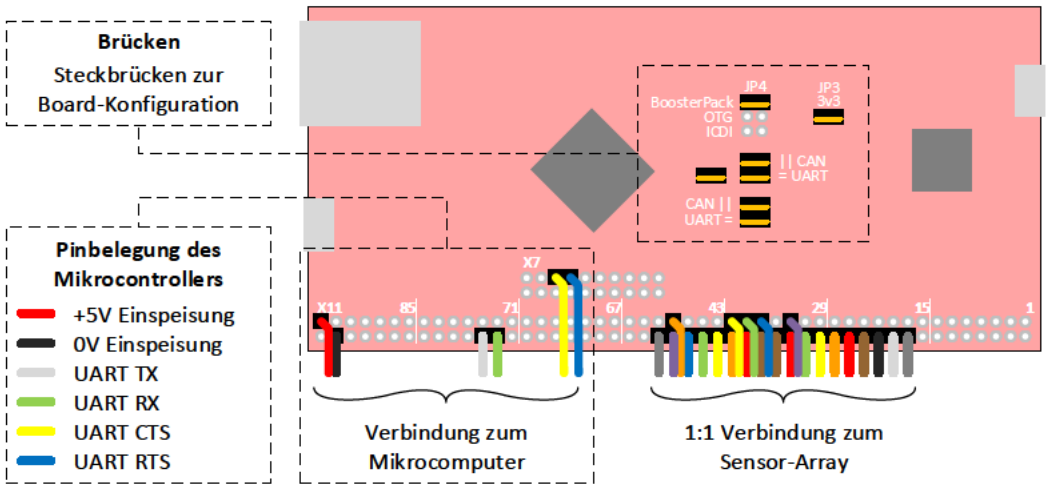


Abbildung 3.1: Schematische Darstellung der Platine des Mikrocontrollers mit Signal- und Versorgungsverbindungen

Die Verbindung zum Sensor-Array ist aus dessen Bachelor-Arbeit [6] übernommen und entspricht exakt der Pinbelegung des Arrays. So ist es möglich, die Platine des Arrays mit einem Buchse-zu-Buchse-Adapter auf die Platine des Mikrocontrollers zu stecken. Für den in dieser Arbeit realisierten Demonstrationsaufbau werden jedoch Leitungen verwendet, um nur das Sensor-Array in unmittelbarer Nähe des Gebermagnetfeldes zu positionieren und den Einfluss des Magnetfeldes auf den Mikrocontroller zu reduzieren.

Anschluss-Nr.	Port	Beschreibung
X7:13	PH0	Mikrocontroller UART RTS (Alternativ X11:9)
X7:15	PH1	Mikrocontroller UART CTS (Alternativ X11:11)
X11:18	PE2	Sensor-Array-Multiplexer 15 Ausgang
X11:20	PE3	Sensor-Array-Multiplexer 13 Ausgang
X11:22	PE4	Sensor-Array-Multiplexer 11 Ausgang
X11:24	PE5	Sensor-Array-Multiplexer 9 Ausgang
X11:26	PK0	Sensor-Array-Multiplexer 7 Ausgang
X11:28	PK1	Sensor-Array-Multiplexer 5 Ausgang
X11:30	PK2	Sensor-Array-Multiplexer 3 Ausgang
X11:32	PK3	Sensor-Array-Multiplexer 1 Ausgang
X11:33	PM3	Sensor-Array-Multiplexer Adressbus Bit 3
X11:34	REF	Sensor-Array Versorgung +3,3 V Dieser Pin wird vom +3,3 V Spannungsregler der Entwicklungsplatine gespeist und ist mit dem REF-Pin des Mikrocontrollers verbunden, der den internen ADC-Modulen als Referenzspannung dient. Da die Versorgung des Sensor-Arrays gleichzeitig die maximal mögliche Signalspannung der Sensoren ist, eignet sich dieser Pin gut als Referenz.
X11:36	-	Sensor-Array Versorgung 0 V
X11:37	PM2	Sensor-Array-Multiplexer Adressbus Bit 2
X11:38	PD5	Sensor-Array-Multiplexer 2 Ausgang
X11:39	PM1	Sensor-Array-Multiplexer Adressbus Bit 1
X11:40	PD4	Sensor-Array-Multiplexer 4 Ausgang
X11:41	PM0	Sensor-Array-Multiplexer Adressbus Bit 0
X11:42	PD7	Sensor-Array-Multiplexer 6 Ausgang
X11:44	PD6	Sensor-Array-Multiplexer 8 Ausgang
X11:46	PD3	Sensor-Array-Multiplexer 10 Ausgang
X11:48	PD1	Sensor-Array-Multiplexer 12 Ausgang
X11:49	PL3	Sensor-Array-Multiplexer Adressbus Bit 3 invertiert
X11:50	PD0	Sensor-Array-Multiplexer 14 Ausgang
X11:52	PD2	Sensor-Array-Multiplexer 16 Ausgang
X11:74	PA0	Mikrocontroller UART RxD
X11:76	PA1	Mikrocontroller UART TxD
X11:96	-	Mikrocontroller Versorgung 0 V
X11:97	-	Mikrocontroller Versorgung +5 V

Tabelle 3.1: Übersicht der Signalverbindungen des Mikrocontrollers und Zuordnung der Funktion

3.3 Entwicklungs-Software

Als Entwicklungsumgebung für den Mikrocontroller bietet der Hersteller ein Quelltext-Editor und -Übersetzer mit Debug-Funktionalität [15] unter Windows an, für den eine Treiberbibliothek [16] zur Hardware-Konfiguration und -Bedienung sowie eine Header-Datei [17] mit Register-Adress-Makros installiert werden können. Die Installation der Entwicklungsumgebung geschieht nach dem Herunterladen unter Windows 10 per Ausführungsdatei. Die Softwarebibliothek und Header-Datei müssen im Anschluss in gleicher Weise installiert werden. Nach dem Starten des Programms und Anlegen eines neuen Projekts muss die richtige Debug-Schnittstelle konfiguriert werden, um den kompilierten Quelltext auf den Mikrocontroller übertragen zu können. Die Programmiersprache ist C.

3.4 Laufzeit-Software

3.4.1 Überblick

Das Programm des Mikrocontrollers ist in einer C-Quelltext-Datei realisiert. Die Einzelaufgaben sind auf Funktionen aufgeteilt, die von der Hauptfunktion aufgerufen werden. Die implementierten Funktionen sind in der Tabelle 3.2 aufgelistet. Funktionen, die der Konfiguration von Hardware-Modulen des Mikrocontrollers dienen, sind mit dem Präfix *setup_* versehen.

Funktion	Beschreibung
adc0_ss0_interrupt_handler	Interrupt-Service-Routine (ISR) für die Sample-Sequence (SS) 0 des ADC 0 Moduls. Löscht die Interrupt-Flag für die SS 0 des ADC 0 Moduls und setzt eine globale Flag zur Auswertung in der Hauptfunktion.
adc1_ss0_interrupt_handler	ISR für die SS 0 des ADC 1 Moduls. Löscht die Interrupt-Flag für die SS 0 des ADC 1 Moduls und setzt eine globale Flag zur Auswertung in der Hauptfunktion.
uart0_interrupt_handler	ISR für das UART 0 Modul. Löscht alle anstehenden Interrupt-Flags des UART 0 Moduls und setzt eine globale Flag zur Auswertung in der Hauptfunktion.
setup_system	Initialisiert den Systemtakt.
setup_mux_address_bus	Konfiguriert die General-Purpose Input / Output (GPIO)-Module L und M für den Adressbus der Analogmultiplexer des Sensor-Arrays.
setup_adc	Konfiguriert die ADC-Module 0 und 1 mit dem jeweiligen SS 0, sowie die GPIO-Module D, E und K für den Analogbus der Analogmultiplexer des Sensor-Arrays.
setup_uart	Konfiguriert das UART-Modul 0 nach der Beschreibung in dem Kapitel 2.4, sowie die GPIO-Module A und H für die serielle Schnittstelle zum Mikrocomputer.
sample_array	Steuert den Adressbus für die Analogmultiplexer des Sensor-Arrays, nimmt mit Hilfe der SS 0 der ADC-Module 0 und 1 die Sensorwerte über den Analogbus auf und speichert diese in ein globales Array.
compute_diff	Bildet das Differenzsignal aus den gemessenen Sensorwerten und speichert diese in einem globalen Array ab.
send_row(row)	Sendet die Differenzsignale einer Zeile des Arrays über das UART 0 Modul an den Mikrocomputer.
send_array	Sendet die alle Differenzsignale zeilenweise über das UART 0 Modul an den Mikrocomputer.

Tabelle 3.2: Funktionsübersicht und -Beschreibung für das Programm des Mikrocontrollers

3.4.2 Hauptfunktion main

Die Hauptfunktion steuert den Programmablauf in Abhängigkeit der globalen Flags. Die Abbildungen 3.2 und 3.3 zeigen die Implementierung anhand eines Flussdiagramms.

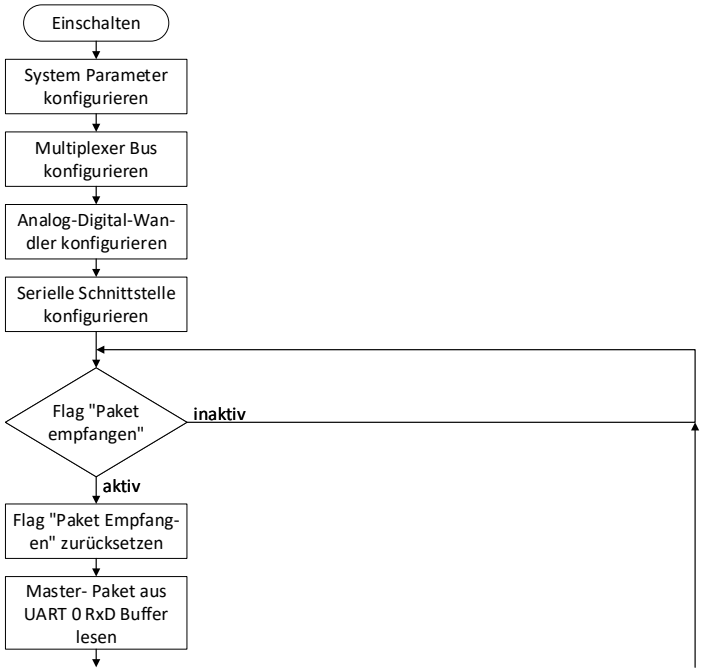


Abbildung 3.2: Flussdiagramm für das Hauptfunktion des Mikrocontrollers (Abschnitt 1 von 2). Zunächst werden die Konfiguration für System, den Adressbus der Sensor-Array-Multiplexer, Analogwertaufnahme und -Digitalwandlung, sowie den seriellen Bus ausgeführt. Dann beginnt eine ewige Schleife. Jeden Zyklus wird die globale Flag für ein empfangenes Master-Paket überprüft. Ist diese gesetzt, wird sie zurückgesetzt und das empfangene Paket aus dem Empfangspuffer gelesen.

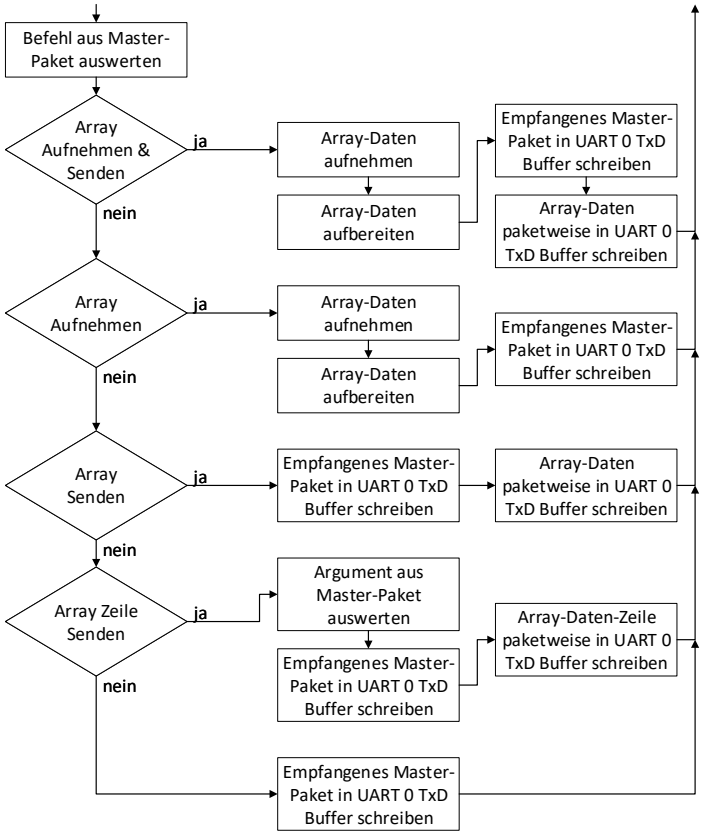


Abbildung 3.3: Flussdiagramm für das Hauptfunktion des Mikrocontrollers (Abschnitt 2 von 2). Abhängig vom empfangenen Befehl entschiedet eine switch-Anweisung über die Reaktion entsprechend der Tabelle 2.3.

3.4.3 Konfigurationsfunktionen

Konfiguration der Systemparameter
setup_system

Der Systemtakt wird auf 120 MHz eingestellt. Die Synchronisationstaktquelle ist der 25 MHz Oszillator der Entwicklungsplatine. Abbildung 3.4 zeigt die Implementierung anhand eines Flussdiagramms.

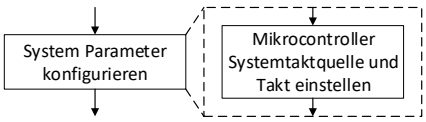


Abbildung 3.4: Flussdiagramm für die Konfigurationsfunktion der Systemparameter des Mikrocontrollers. Der Systemtakt wird eingestellt.

Konfiguration des Adressbusses der Sensor-Array-Multiplexer setup_mux_address_bus

Der Adressbus für die Multiplexer des Sensor-Arrays besteht aus fünf parallelen Leitungen. Er deckt einen Adressraum von vier Bit beziehungsweise 16 Adressen mit vier der Leitungen ab. Die fünfte Leitung trägt das gleiche Signal wie die vierte und ist aufgrund der Auslegung des Sensor-Arrays notwendig. Der Bus wird über GPIO-Pins der Ports L und M getrieben. Entsprechend sind die Pins als digitale Ausgänge konfiguriert.

Die Multiplexer des Sensor-Arrays benötigen eine Einschwingzeit von 1 μs [6] nachdem der Adressbus geändert wurde. Die Verzögerung wird mit Hilfe einer Funktion der Treiberbibliothek [16] realisiert, welche vom Hersteller mit drei Zyklen pro einem Inkrement des Arguments beschrieben wird. Entsprechend wird das Argument mit dem Systemtakt ausgerechnet. Abbildung 3.5 zeigt die Implementierung anhand eines Flussdiagramms.

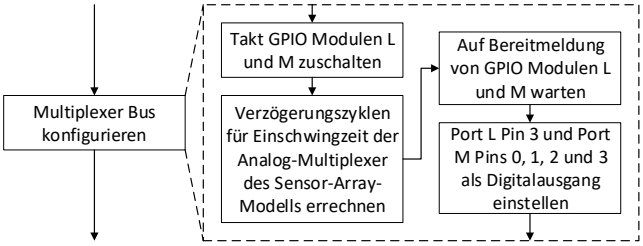


Abbildung 3.5: Flussdiagramm für die Konfigurationsfunktion des Mikrocontrollers des Adressbusses der Analogmultiplexer des Sensor-Arrays. Zur Konfiguration der Ausgänge wird den GPIO-Modulen zunächst der Takt zugeschaltet. Um die Einschaltzeit der GPIO-Module zu nutzen, wird zunächst das Argument einer Verzögerungsfunktion für die Einschwingzeit der Analog-Multiplexer nach Ändern der Adresse berechnet. Dann wird auf die Bereitschaft der GPIO-Module gewartet. Melden diese Bereitschaft, werden die entsprechenden Pins als Digitalausgang konfiguriert.

Konfiguration der Analog-Digital-Wandler und des Analogbusses setup_adc

Zum Auslesen der Analogwerte des Sensor-Array-Modells über den analogen Bus werden 16 der verfügbaren 20 Analogeingänge des Mikrocontrollers für die 16 Multiplexer des Arrays genutzt. Dazu wird der Digitalteil der GPIO-Module für Port D, E und K abgeschaltet, sodass die Pins den zwei internen analogen ADC-Multiplexern des Mikrocontrollers zur Verfügung stehen. Die achtkanalige SS des jeweiligen ADC-Moduls steuert den zugehörigen internen Multiplexer des Mikrocontrollers, um die Analogeingänge nacheinander zuzuschalten. Ist eine SS nach dem Start seines Zyklus beendet, löst diese einen Interrupt aus. Die damit aufgerufene ISR löscht zuerst die Flag des entsprechenden Registers und setzt dann eine globale Steuervariable, welche in der Hauptfunktion verarbeitet wird.

Diese Handhabung ist in allen ISRs auf Empfehlung des Herstellers [14] umgesetzt um sicherzustellen, dass die Interrupt-Flag im Register nach Beendigung der ISR gelöscht ist. Anderenfalls besteht die Möglichkeit, dass die ISR im Anschluss erneut ausgeführt wird. Der minimalistische Quellcode innerhalb der ISRs soll außerdem die Bearbeitungszeit kurz halten und so das Risiko verkleinern, dass für die Programmteile, die außerhalb von ISRs ausgeführt werden, zu wenig Prozessorzeit übrig bleibt. Abbildung 3.6 zeigt die Implementierung anhand eines Flussdiagramms.

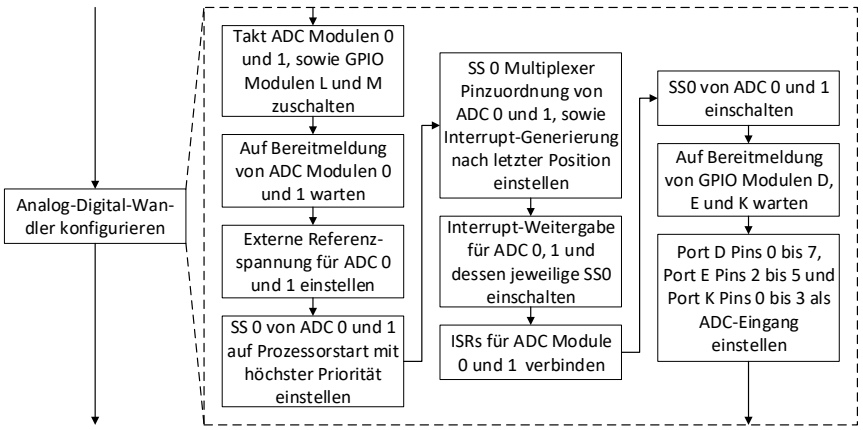


Abbildung 3.6: Flussdiagramm für die Konfigurationsfunktion des Analogbusses des Sensor-Arrays und der ADC-Module des Mikrocontrollers. Nachdem der Takt für die ADC- und GPIO-Module eingeschaltet ist, wird gewartet, bis die ADC-Module Bereitschaft melden. Sind diese bereit, wird die Referenzspannung auf extern eingestellt. Danach werden die SS der ADC-Module konfiguriert, die ISRs den Interrupt-Signalen zugeordnet und die SS eingeschaltet. Zuletzt werden nach deren Bereitschaftsmeldung die GPIO-Module als Analogeingang eingestellt.

Konfiguration der seriellen Schnittstelle setup_uart

Das UART 0 Modul des Mikrocontrollers ist mit Pins der GPIO-Module A und H als TxD-, RxD-, CTS- und RTS-Signale konfiguriert und setzt die in Kapitel 2.4 beschriebenen Einstellungen für die serielle Schnittstelle um. Die zugehörige ISR löscht zuerst alle Flags des Interrupt Registers des UART 0 Moduls und setzt dann eine globale Steuervariable, die von der Hauptfunktion verarbeitet wird. Abbildung 3.7 zeigt die Implementierung anhand eines Flussdiagramms.

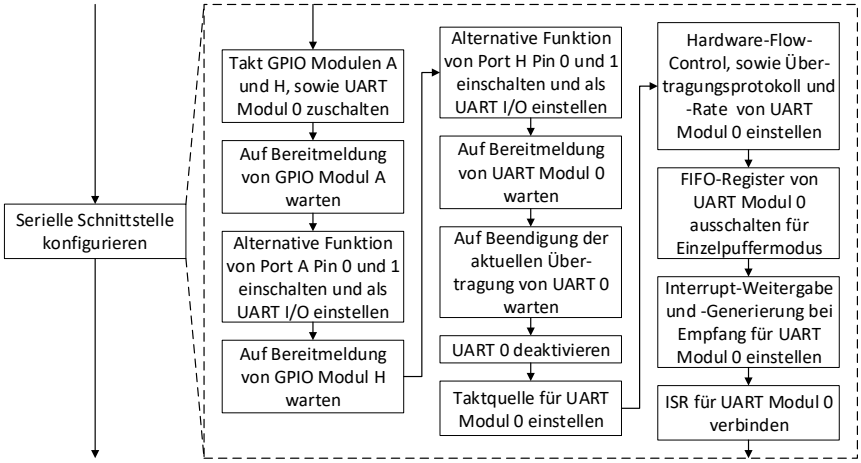


Abbildung 3.7: Flussdiagramm für die Konfigurationsfunktion des seriellen Busses und des UART 0 Moduls des Mikrocontrollers. Den UART-0- und GPIO-A- und -H-Modulen wird der Takt zugeschaltet und auf die Bereitschaft des GPIO-A-Moduls gewartet. Ist das Modul bereit, werden die entsprechenden Pins für das UART-0-Modul konfiguriert. Die Pins des GPIO-H-Moduls werden in gleicher Weise konfiguriert. Dann wird auf die Bereitschaft des UART-0-Moduls gewartet, auf dessen Bereitmeldung hin die serielle Schnittstelle entsprechend des Kapitels 2.4 eingestellt wird. Zuletzt wird die ISR zugewiesen.

3.4.4 Laufzeitfunktionen

Aufnahme der Sensor-Array-Werte sample_array

Die Aufnahme der Sensor-Array-Wert geschieht durch Ansteuern der MUXs über den parallelen Adressbus und Auslesen der MUX-Ausgänge über den parallelen analogen Bus des Arrays. Dazu wird zuerst mit den GPIO Pins aus Kapitel 3.4.3 die gewünschte Adresse eingestellt. Um die Verzögerung zwischen den zwei GPIO-Ports möglichst gering zu halten, nutzt der Quelltext direkten Registerzugriff mit Hilfe der Register-Macros [17] anstatt der Funktionen der Treiberbibliothek [16]. Nach der Einschwingzeit starten die SS 0 der ADCs die Aufnahme. Mit einer while-Schleife wird der Status der SS überprüft. Ist einer fertig, werden die Werte zugehörigen gespeichert. Sind beide fertig, wird die Schleife verlassen und die Funktion ist beendet. Abbildung 3.8 zeigt die Implementierung anhand eines Flussdiagramms.

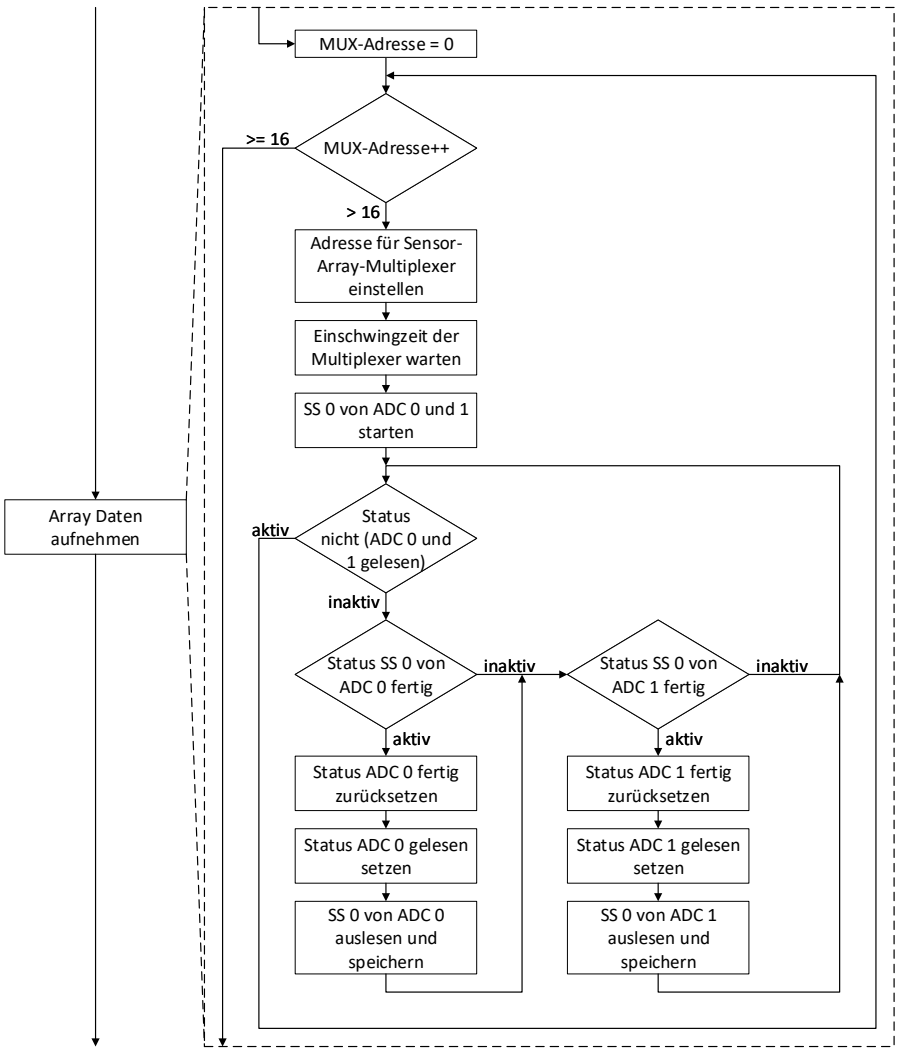


Abbildung 3.8: Flussdiagramm für die Funktion zur Aufnahme der Sensor-Array-Daten durch den Mikrocontroller. Die Adresse des Multiplexer-Adressbusses für das Sensor-Array wird auf den Wert null gesetzt. In einer for-Schleife wird die Adresse dann hochgezählt, um alle Sensorwerte zu erfassen. In der Schleife werden zuerst der Adressbus auf die aktuelle Adresse gesetzt, dann die Einschwingzeit der Signale abgewartet und schließlich die SS zur Aufnahme der Analogwerte gestartet. In einer while-Schleife wird dann auf die Fertigmeldung der SS gewartet. Meldet ein SS, dass sein Zyklus beendet ist, werden die Meldung zurückgesetzt, eine lokale Statusmeldung zum Beenden der while-Schleife gesetzt und die Analogwerte aus dem entsprechenden Register ausgelesen und gespeichert. Sind beide Statusmeldungen zum beenden der while-Schleife gesetzt und damit die Analogwerte beider ADC gespeichert, beginnt die for-Schleife von vorn.

Aufbereitung der Sensor-Array-Werte
compute_diff

Die Aufbereitung der Sensor-Array-Werte erfolgt ohne Filter in Form der Differenzbildung je Achse und Sensor. Zwei geschachtelte for-Schleifen iterieren über die Werte und speichern die errechnete Differenz ab. Sind alle Daten gespeichert, ist die Funktion beendet. Abbildung 3.9 zeigt die Implementierung anhand eines Flussdiagramms.

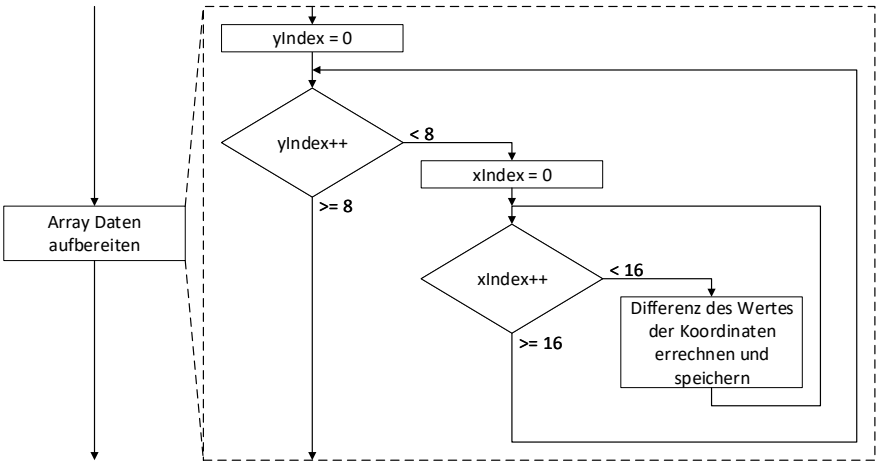


Abbildung 3.9: In einer einfach verschachtelten for-Schleife wird je Achse und je Sensor die Differenz der aufgenommenen Analogwerte des Sensor-Arrays errechnet und abgespeichert. Ist zu Beginn jeder Iteration die globale Steuervariable gesetzt, die den Empfang einer seriellen Nachricht anzeigt, wird die Schleife abgebrochen.

Senden der Sensor-Array-Daten
send_array

Das Senden der Sensor-Array-Daten geschieht sequentiell mit dem UART 0 Modul, angefangen vom Sensor, der den Koordinatenkreuz unten links am nächsten ist, und zuerst entlang der x-Achse und dann der y-Achse. Kapitel 2.4.2 beschreibt die Aufteilung der Daten in kleinere Pakete zur Übertragung per acht-Bit-UART. Sind alle Daten gesendet, ist die Funktion beendet. Abbildung 3.10 zeigt die Implementierung anhand eines Flussdiagramms.

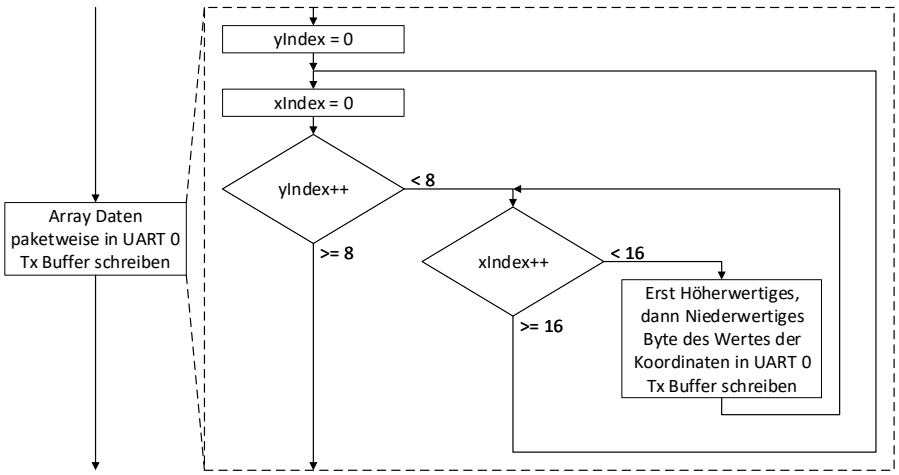


Abbildung 3.10: In einer einfach verschachtelten for-Schleife wird je Sensor die Differenz der aufgenommenen Analogwerte des Sensor-Arrays byteweise in den Sendepuffer des UART 0 Moduls geschrieben. Dabei wird zuerst das höherwertige und dann das niederwertige Byte zum Senden übergeben. Ist zu Beginn jeder Iteration die globale Steuervariable gesetzt, die den Empfang einer seriellen Nachricht anzeigt, wird die Schleife abgebrochen.

Zeilenweises Senden der Sensor-Array-Daten
send_row

Im Vergleich zum Senden aller Sensor-Array-Daten aus Kapitel 3.4.4 wird hier nur die als Argument übergebene Zeile (in x-Richtung) auf gleiche Weise gesendet. Sind alle Daten der Zeile gesendet, ist die Funktion beendet. Abbildung 3.11 zeigt die Implementierung anhand eines Flussdiagramms.

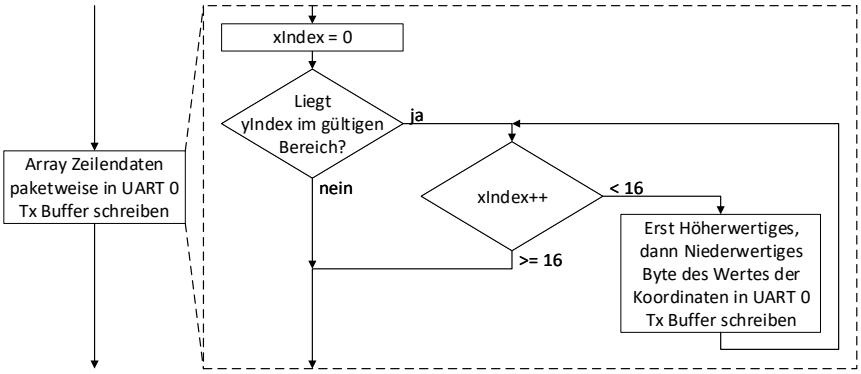


Abbildung 3.11: In einer for-Schleife wird je Sensor die Differenz der aufgenommenen Analogwerte des Sensor-Arrays für die als Funktionsargument übergebene Zeile byteweise in den Sendepuffer des UART 0 Moduls geschrieben. Dabei wird zuerst das höherwertige und dann das niederwertige Byte zum Senden übergeben. Ist zu Beginn jeder Iteration die globale Steuervariable gesetzt, die den Empfang einer seriellen Nachricht anzeigt, wird die Schleife abgebrochen.

4 Implementierung auf dem Mikrocomputer

4.1 Zusammenfassende Funktionsbeschreibung

Der Mikrocomputer stellt die Daten des Sensor-Arrays mit verschiedenen Darstellungsvarianten visuell dar. Er agiert als Master für den Mikrocontroller und steuert damit das Sensor-Array. Das Programm des Mikrocomputers bietet eine Grundlage für die Implementierung beliebiger Signalverarbeitung in der Programmiersprache C.

4.2 Elektrischer Anschluss

Der Mikrocomputer wird über den Universal Serial Bus (USB)-Micro-B-Anschluss mit +5 V versorgt. Die serielle Schnittstelle zwischen Mikrocomputer und Mikrocontroller dient neben der Signalverbindung auch als Versorgung des Mikrocontrollers und ist mit einem USB-zu-Seriell-Adapter realisiert. Die Computermaus und Tastatur werden per USB-A-Anschluss und der Bildschirm per High-Definition Multimedia Interface (HDMI)-Anschluss verbunden.

Serielle Schnittstelle

Da sich die folgend erläuterte Konfiguration der integrierten seriellen Schnittstelle des Mikrocomputers mit Hardware-Flusskontrolle als komplex und fehleranfällig erwies, wird hierfür ein kostengünstiger USB-zu-Seriell-Adapter verwendet, welcher keine Konfiguration außerhalb des Programms dieser Arbeit benötigt. Der eingesetzte Adapter basiert auf einem *Prolific PL-2303H* Integrated Circuit (IC) [7] und verfügt über alle nötigen Signale zur Realisierung der Hardware-Flusskontrolle. Die Leitungsfarben entsprechen

den in der Tabelle 4.1 gelisteten Funktionen. Erfolgreich getestet wurde außerdem ein *Silabs CP 2102E* [12] USB-zu-Seriell-Adapter.

Um stattdessen eine der zwei integrierten seriellen Schnittstellen des Mikrocomputers mit dessen GPIO-Pins verwenden zu können, muss über das *Startmenü* > *Preferences* > *Raspberry Pi Configuration* > *Interfaces* die Option *Serial Port* eingeschaltet werden. Die *Mini-UART* Schnittstelle, die mit der Standardkonfiguration des Systems nach dem Einschalten auf die GPIO-Pins geführt ist, unterstützt neben den RxD- und TxD-Signalen keine weiteren. Da dies für die vorgesehene Implementierung der Hardware-Flusskontrolle notwendig ist, muss stattdessen die zweite vollwertige serielle Schnittstelle des Mikrocomputers verwendet werden. Diese wird jedoch mit der Standardkonfiguration für die Bluetooth- und WLAN-Module verwendet und ist nicht über die GPIO-Pins zugänglich. Es bedarf daher der Abschaltung oder Umleitung der WLAN- und Bluetooth-Module auf die *Mini-UART*-Schnittstelle des Prozessors, während die vollwertige integrierte serielle Schnittstelle auf die GPIO-Pins geschaltet wird. Dazu muss in der Systemdatei */boot/config.txt* der Eintrag ***dtoverlay=disable-bt*** zum Abschalten oder ***dtoverlay=miniuart-bt*** zum Umleiten ergänzt werden. Weiterhin muss sichergestellt werden, dass die Ausgabe von Systemmeldungen über die serielle Schnittstelle auf den GPIO-Pins abgeschaltet ist. In der Systemdatei */boot/cmdline.txt* wird dazu der Eintrag ***console=serial0,115200*** gelöscht. Zur Verwendung der CTS- und RTS-Signale müssen nach jedem Systemstart die entsprechenden GPIO-Pins konfiguriert werden. Zu diesem Zweck steht im Internet ein Programm [5] zur Verfügung, welches die Hardware-Version des Mikrocomputers erkennt und die entsprechenden Einstellungen vornimmt, um die CTS- und RTS-Signale mit den GPIO-Pins verwenden zu können. Nachdem die GPIO-Pins konfiguriert sind, muss die Hardware-Flusskontrolle der seriellen Schnittstelle per Eingabe des Textes ***sudo stty -F /dev/ttyAMA0 crtscts*** in eine Konsole eingeschaltet werden.

Funktion	**Leitungsfarbe**
+5 V Spannungsversorgungsausgang	Rot
+3,3 V Spannungsversorgungsausgang	Orange
0 V Bezugspotential	Schwarz
RxD-Signal	Weiß
TxD-Signal	Grün
CTS-Signal	Blau
RTS-Signal	Gelb

Tabelle 4.1: Zuordnung der Leitungsfarben zu den Funktionen des USB-zu-Seriell-Adapters

4.3 Entwicklungs-Software

Der Quelltext des Mikrocomputers ist für das Betriebssystem *Raspbian Buster with Desktop* [10], einer Debian-basierten und für den Mikrocomputer optimierten Unix-Distribution, in der Programmsprache C geschrieben. Die Installation des Betriebssystems erfolgt nach einer Anleitung auf der Internetseite des Herstellers [9]. Als Compiler für den Quelltext wird die GNU Compiler-Collection (GCC) genutzt, welche bereits vorinstalliert ist. Als Software für die Benutzeroberfläche wurden die in der Tabelle 4.2 aufgelisteten Optionen in Erwägung gezogen. Die Umsetzung in dieser Arbeit nutzt das GNU Image-Manipulation-Program-Tool-Kit (GTK) 3, eine Bibliothek zur Erstellung von grafischen Benutzeroberflächen, und die Open Graphics Library (OpenGL), eine Programmierschnittstelle zur Entwicklung von Computergrafikanwendungen [20]. Die Ausführung von Grafikanwendungen mit der OpenGL geschieht abhängig von der Implementierung auf dem Zentral- oder Grafikprozessor des Mikrocomputers. Das in dieser Arbeit umgesetzte Programm nutzt den Zentralprozessor. Die GTK 3 Bibliothek lässt sich mit einer Internetverbindung per Eingabe des Textes ***sudo apt-get install libgtk-3-dev*** in eine Konsole installieren.

Option	Argumentation
Gnuplot mit Gnuplot_i-Bibliothek: Gnuplot ist ein Kommandozeilenprogramm zur Erstellung von wissenschaftlichen Grafiken. Mit der Software-Bibliothek Gnuplot_i lässt es sich auch im C-Quelltext einbinden. Die Bedienung der Visualisierung geschieht per Tasten, die mit Hilfe der WiringPi-Bibliothek über GPIO des Mikrocomputers eingelesen werden können oder mit einer der folgenden Graphical User Interface (GUI)-Optionen.	Gnuplot ist für die Erstellung von Einzelbildern ausgelegt. Für die Visualisierung sind viele Einzelbilder in kurzer Zeit nötig, um eine fließende Animation und kurze Reaktionszeiten zu erzielen. Da dies jedoch sehr große Datenträgeraktivität hervorruft, kann es vorkommen, dass Bilder unvollständig dargestellt werden, wenn ein anderer Prozess auf den Datenträger zugreift und somit die verfügbare Zugriffsrate für die Erstellung der Bilder nicht ausreichend hoch ist. Als Lösungsansatz wurde eine Random Access Memory (RAM)-Disk, ein virtuelles Speichermedium im Arbeitsspeicher des Mikrocomputers, erprobt, was nicht den gewünschten Erfolg erzielte. Zudem nutzt die Gnuplot_i-Bibliothek eine virtuelle Kommandozeile, um die Befehle des aufrufenden Quelltextes an Gnuplot weiterzugeben. Dies führt bei hoher Prozessorauslastung dazu, dass die Kommandozeilenbefehle, die an Gnuplot gesendet werden, teilweise unvollständig übergeben und somit nicht ausgeführt werden. Es ist zudem erforderlich eine weitere separate Bibliothek für die Benutzerschnittstelle zu implementieren. Da die Option, wiringPi zu verwenden, die Möglichkeiten der Bedienung auf physikalische Taster beschränkt, ist die Nutzung einer der folgenden Optionen vorteilhaft.
Qt: Qt ist eine Bibliothek zur Erstellung von grafischen Benutzeroberflächen, welche in C++ programmiert wird. Sie ermöglicht es, zur Grafischen Visualisierung die OpenGL zu verwenden.	Da die Programmiersprache des Projekts vorzugsweise C ist, wurde die nachfolgende Software vorgezogen.
GTK 3: Wie Qt ist das GTK eine Bibliothek zur Erstellung von grafischen Benutzeroberflächen. Sie wird im Gegensatz zu Qt in der Programmiersprache C programmiert. Auch mit GTK 3 ist es möglich, die OpenGL zu verwenden.	Aufgrund der vorgezogenen Programmiersprache und umfangreicher, im Internet verfügbarer Dokumentation und Anleitung wurde diese Option gewählt.

Tabelle 4.2: Optionen für die Software der Benutzerschnittstelle und Wahl

4.4 Laufzeit-Software

4.4.1 Überblick

Die Programmquellen des Mikrocomputers sind in einer Verzeichnisstruktur angelegt, welche die Dateien in Konfigurations- und Quelltextdateien sortiert. Der Oberordner enthält neben den Ordnern der Konfigurations- und Quelltextdateien eine Datei [3], die die Informationen für das vorinstallierte Kompilierwerkzeug *Make* enthält. Wird diese sogenannte *makefile* mit der Eingabe des Textes ***make*** in einer im Verzeichnis geöffneten Konsole ausgeführt, wird der Quelltext automatisch kompiliert. Abbildung 4.1 zeigt die jeweiligen Speicherorte der Dateien im Oberordner.

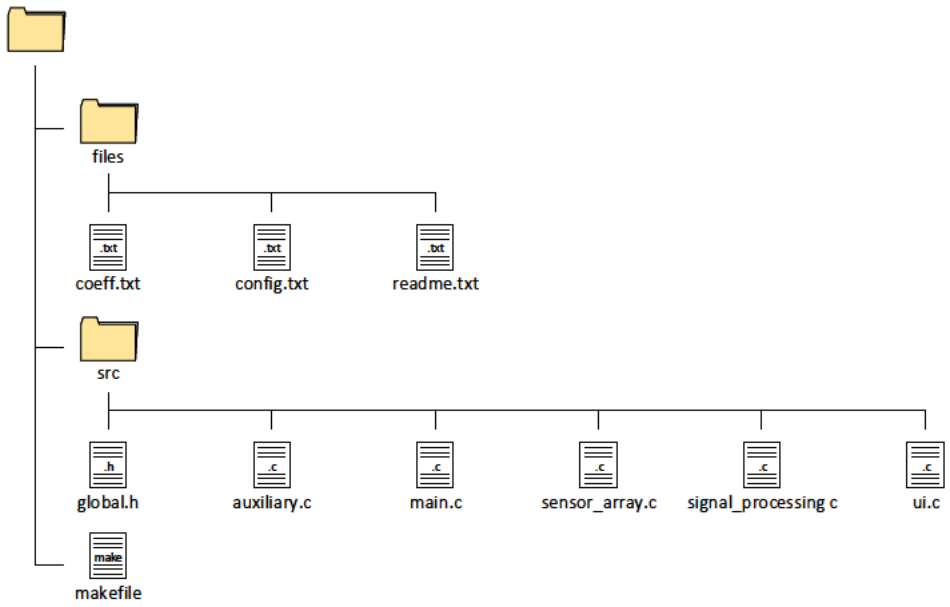


Abbildung 4.1: Verzeichnisstruktur der Konfigurations- und Quelltextdateien des Programms für den Mikrocomputer. Im Hauptordner befinden sich zwei Unterordner, die die Konfigurationsdateien vom Quelltext trennen sowie die Datei *makefile*, welche die Informationen für das Kompilieren durch das Werkzeug *Make* enthält.

Im Ordner *files* sind drei Textdateien gespeichert. Die Datei *coeff.txt* enthält Filterparameter für die Signalverarbeitung der Sensor-Array-Daten durch den Mikrocomputer. *config.txt* enthält einstellbare Programmparameter für die serielle Schnittstelle des Mi-

krocomputers, die Anzahl an Sensoren des Sensor-Arrays und die Größe der Animationsfläche in Pixeln. Diese beiden Dateien werden vom Programm des Mikrocomputers ausgelesen. Der jeweilige Inhalt ist in der *readme.txt* Textdatei erläutert.

Die Quelltexte sind im Ordner *src* (Abkürzung, englisch source) abgelegt und teilen die Funktionen in die in der Tabelle 4.3 aufgelisteten Aufgabenbereiche auf.

Datei	**Aufgabenbereich**
global.h	Bereitstellen von globalen Macros, Structures, Konstanten und Variablen
auxiliary.c	Unterstützung der Hauptfunktion
main.c	Hauptfunktion zum Steuern des Programmablaufs
sensor_array.c	Steuern des Sensor-Arrays
signal_processing.c	Signalverarbeitung (Implementierung vorbereitet)
ui.c	Benutzeroberfläche und Visualisierung

Tabelle 4.3: Aufgabenbereiche zur Unterteilung des Mikrocomputerprogramms

Die in den Quelltexten enthaltenen Funktionen des Programms werden von der Hauptfunktion aufgerufen und sind in der Tabelle 4.4 aufgelistet.

Funktion	Beschreibung
ui_thread	Funktion für die Konfiguration und Verwaltung der Benutzeroberfläche. Sie ist für den Aufruf als separater Prozess vorgesehen.
read_configuration_file	Liest die Konfigurationsdatei.
read_coefficient_file	Liest die Koeffizientendatei für die Signalverarbeitung.
set_up_uart	Konfigurationsfunktion zum Öffnen der seriellen Schnittstelle.
set_down_uart	Wendet die Einstellungen der übergebenen Konfigurationsvariable auf die serielle Schnittstelle an und schließt diese.
sensor_array_send_request	Sendet einen übergebenen Befehl und Argument als Paket mit der seriellen Schnittstelle an den Mikrocontroller.
sensor_array_collect_array_data	Sammelt die empfangenen Daten der seriellen Schnittstelle und prüft, ob der Mikrocontroller alle Sensor-Array-Werte erfolgreich übertragen hat.
gl_init_shaders	Initialisiert die Shader für die Visualisierung mit der OpenGL.
gl_create_shaders	Legt die Shader für die Visualisierung mit der OpenGL an.
gl_init_buffers	Initialisiert die Vertex-Puffer für die Visualisierung mit der OpenGL.
gl_draw_vector_field	Zeichnet ein Vektorfeld mit den übergebenen Daten und der OpenGL.
gl_draw_heatmap	Zeichnet eine Farbmatrix mit den übergebenen Daten und der OpenGL.
gl_draw_vector	Zeichnet einen Vektor mit der OpenGL, welcher durch seinen Winkel zur x-Achse und der Abweichung seines Startpunktes vom Mittelpunkt definiert ist.

Tabelle 4.4: Funktionsübersicht für das Programm des Mikrocomputers

4.4.2 Gestaltung der Benutzeroberfläche

Die Benutzeroberfläche ist mit dem GTK 3 erstellt und besteht aus Widgets (englisches Kunstwort von window und gadget, vergleichbar mit Bedienelement), die in einem Fenster angeordnet sind. Die verschiedenen Widgets sind die Animationsfläche, Schaltflächen zum Laden der Konfigurations- und Filterparameter aus der jeweiligen Datei, das Auswahlmenü der Darstellung und das Häkchen zum Ein- und Ausschalten der Interpolation der darzustellenden Daten. Ein Entwurf der Benutzeroberfläche auf dem Mikrocomputer ist in der Abbildung 4.2 dargestellt.

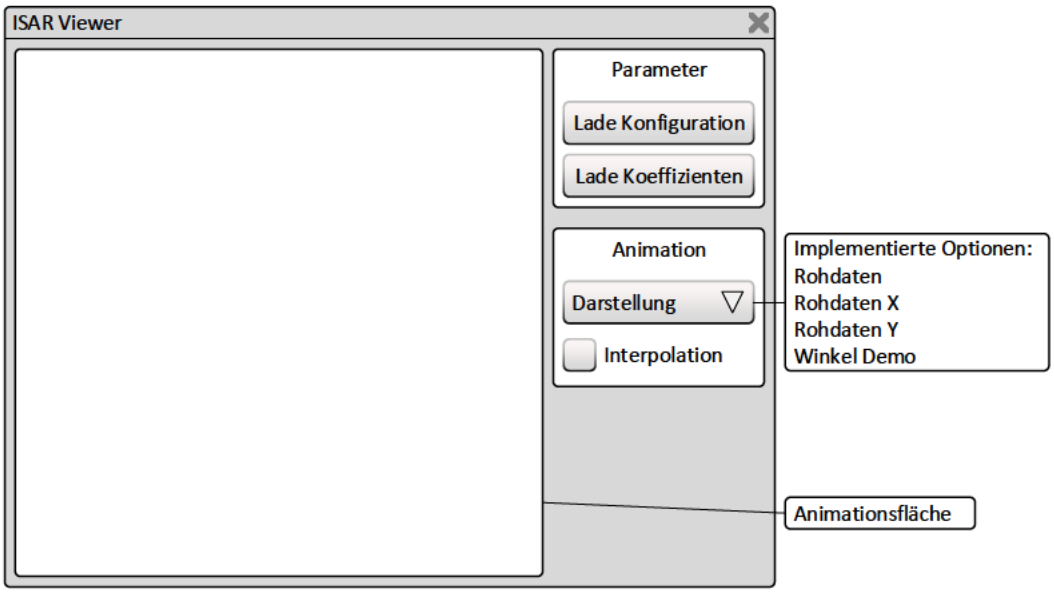


Abbildung 4.2: Entwurf der Benutzeroberfläche auf dem Mikrocomputer. Sie besteht aus den Widgets der Animationsfläche, Schaltflächen zum Laden der Konfigurations- und Filterparameter aus der jeweiligen Datei, dem Auswahlmenü der Darstellung und Häkchen zum Ein- und Ausschalten der Interpolation der darzustellenden Daten.

Das Fenster verfügt über die Schaltfläche zum Schließen, die bei Betätigung die Benutzeroberfläche und das Hauptprogramm beendet. Die Größe des Fensters ist abhängig von der Größe der Animationsfläche, die in der Konfigurationsdatei festgelegt ist, und kann nicht per Mauszeiger vergrößert oder verkleinert werden. Wird die Konfigurationsdatei während der Programmausführung geändert, müssen die Einstellungen mit Betätigen der Schaltfläche *Lade Konfiguration* eingelesen werden. Gleiches gilt für die Filterkoeffizien-

ten mit der Schaltfläche *Lade Koeffizienten*. Welche Daten und Verarbeitungsstufe auf der Animationsfläche dargestellt werden, kann über das Auswahlmenü eingestellt werden. Mit Setzen des Häkchens für die Interpolation, werden die Daten interpoliert.

4.4.3 Hauptfunktion main

Die Hauptfunktion steuert alle Abläufe zur Programmausführung anhand der Steuersignale der Benutzeroberfläche. Für die Sensor-Daten-Anfrage vom Mikrocontroller wird der im Kapitel 2.5 mit der Abbildung 2.9 erläuterte Ablauf implementiert. Treten bei der zyklischen Kommunikation Fehler auf, wird ein Fehlerzähler um den Wert zwei erhöht. Jeden Zyklus wird dieser Zähler bis zum Wert null um eins verringert. Überschreitet der Fehlerzähler einen Grenzwert, wird das Programm beendet. Diese Methode begrenzt die Anzahl an Fehlern in einem gewissen Zeitintervall. Bei gelegentlichen Übertragungsfehlern kann das Programm so weiter ausgeführt werden, häufen sich die Fehler jedoch an, wie zum Beispiel beim Ausfall der seriellen Schnittstelle, wird das Programm beendet. Die Abbildung 4.3 bis 4.7 zeigen die Implementierung anhand eines Flussdiagramms. Die Funktion zur Interpolation der darzustellenden Array-Daten, die in der Hauptfunktion verwendet wird, stammt aus einer Vorarbeit [11] und ist daher nicht näher beschreiben.

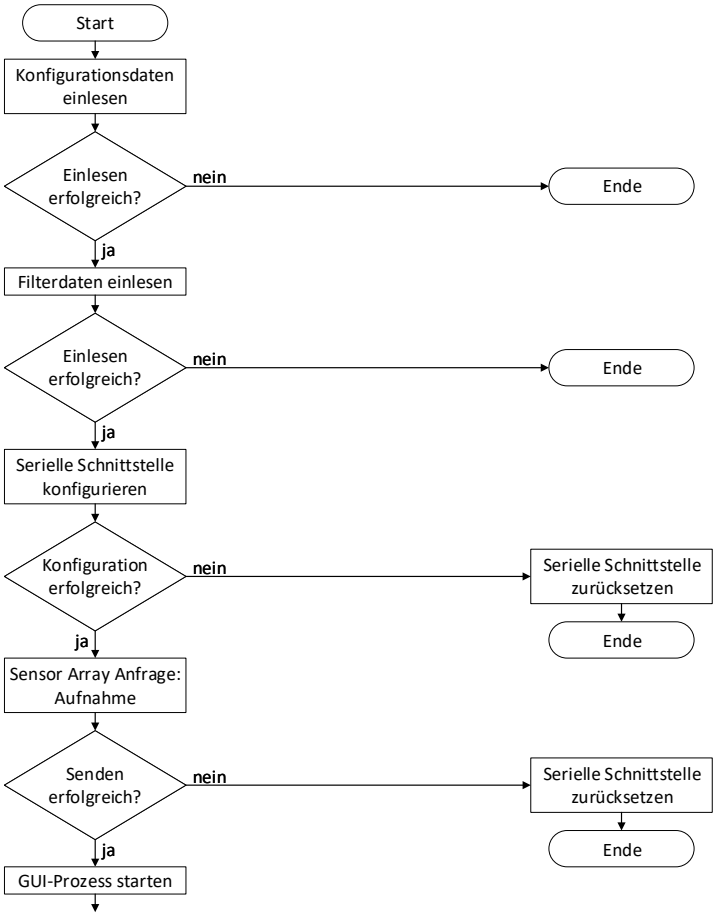


Abbildung 4.3: Flussdiagramm für das Hauptprogramm des Mikrocomputers (Abschnitt 1 von 5). Nach dem Start werden zunächst die Dateien der Konfigurationsdaten und Filterkoeffizienten eingelesen. Geschieht dies fehlerfrei wird die serielle Schnittstelle eingestellt. Tritt bei dabei ein Fehler auf, wird sie serielle Schnittstelle zurückgesetzt und das Programm beendet, anderenfalls wird eine Anfrage an den Mikrocontroller gesendet und überprüft, ob dieser erwartungsgemäß antwortet. Empfängt der Mikrocomputer keine oder eine fehlerhafte Antwort wird die serielle Schnittstelle zurückgesetzt und das Programm beendet. Dann wird der Benutzerschnittstellen-Prozess, kurz GUI, gestartet und der Hauptzyklus des Programms betreten.

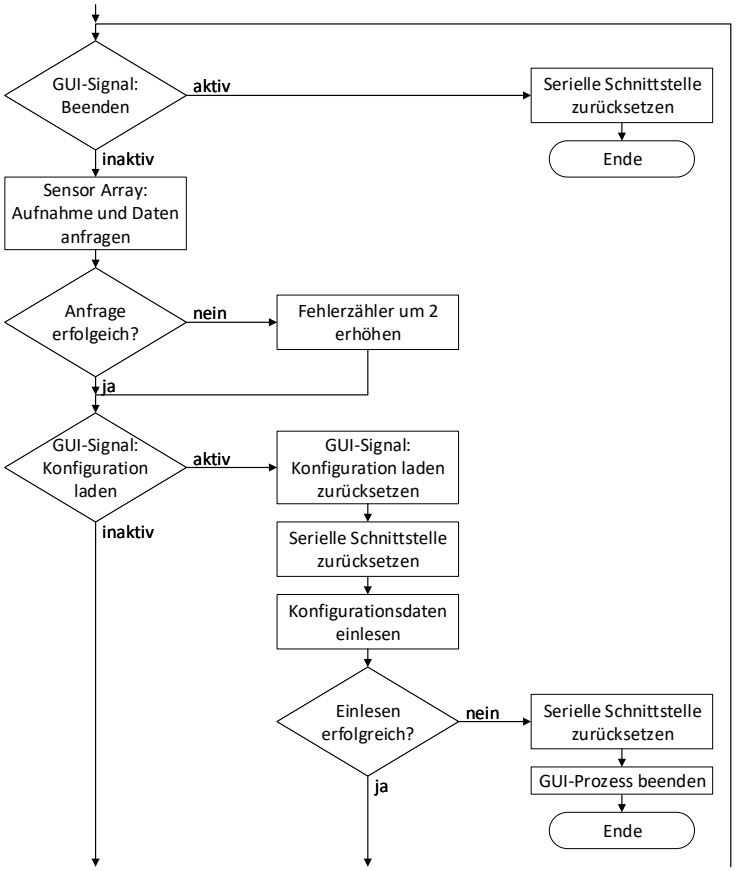


Abbildung 4.4: Flussdiagramm für das Hauptprogramm des Mikrocomputers (Abschnitt 2 von 5). Der Hauptzyklus wird verlassen, die serielle Schnittstelle zurückgesetzt und das Programm beendet, wenn das Steuersignal zum Beenden von der GUI gesetzt wird. Im Hauptzyklus wird zunächst der Befehl zum Aufnehmen und Senden der Sensor-Array-Daten über die serielle Schnittstelle an den Mikrocontroller gesendet. Schlägt dies fehl, wird ein Fehlerzähler erhöht. Es folgt die Auswertung der Steuersignale der GUI. Ist das Signal zum Einlesen der Konfigurationsdatei gesetzt, wird die serielle Schnittstelle zurückgesetzt und die Konfigurationsdatei eingelesen. Schlägt das Einlesen fehl, wird die serielle Schnittstelle zurückgesetzt und der GUI-Prozess und das Hauptprogramm beendet.

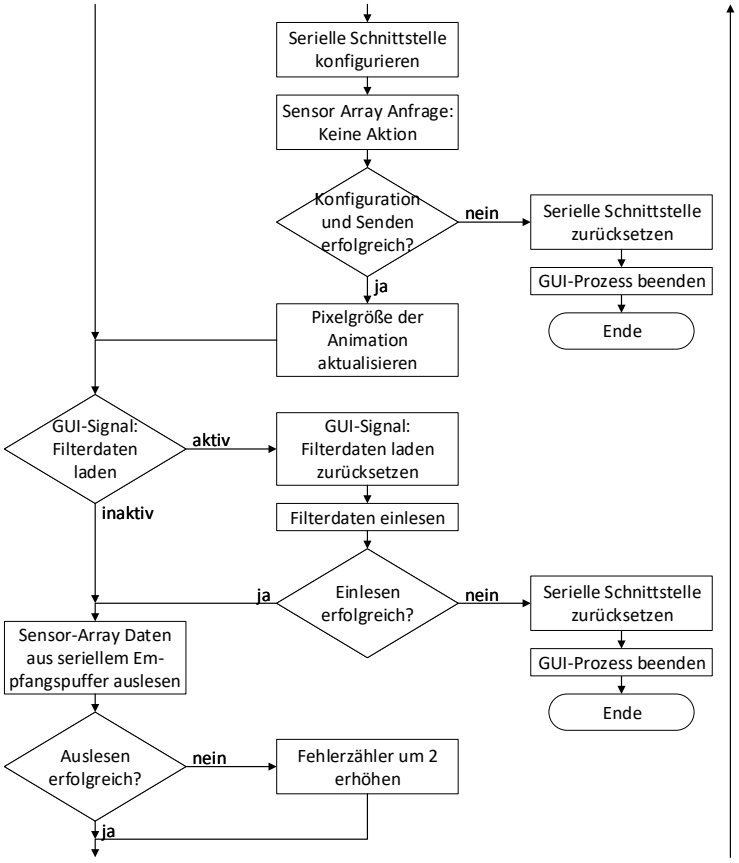


Abbildung 4.5: Flussdiagramm für das Hauptprogramm des Mikrocomputers (Abschnitt 3 von 5). Nach erfolgreichem Einlesen wird die serielle Schnittstelle mit den neu geladenen Einstellungen konfiguriert und eine Anfrage über die serielle Schnittstelle an den Slave gesendet. Tritt dabei ein Fehler auf, wird die serielle Schnittstelle zurückgesetzt und der GUI-Prozess und das Hauptprogramm beendet. Bei Erfolg wird die Größe der Animationsfläche aktualisiert, bevor der Hauptzyklus weitergeht. Ist das Steuersignal der GUI aktiv werden die Filterdaten aus der Koeffizientendatei eingelesen. Tritt dabei ein Fehler auf, wird die serielle Schnittstelle zurückgesetzt und der GUI-Prozess und das Hauptprogramm beendet. Anderenfalls wird der Hauptzyklus fortgesetzt mit dem Auslesen der Sensor-Array-Daten aus dem Empfangspuffer der seriellen Schnittstelle. Bei einem Fehler wird ein Fehlerzähler erhöht.

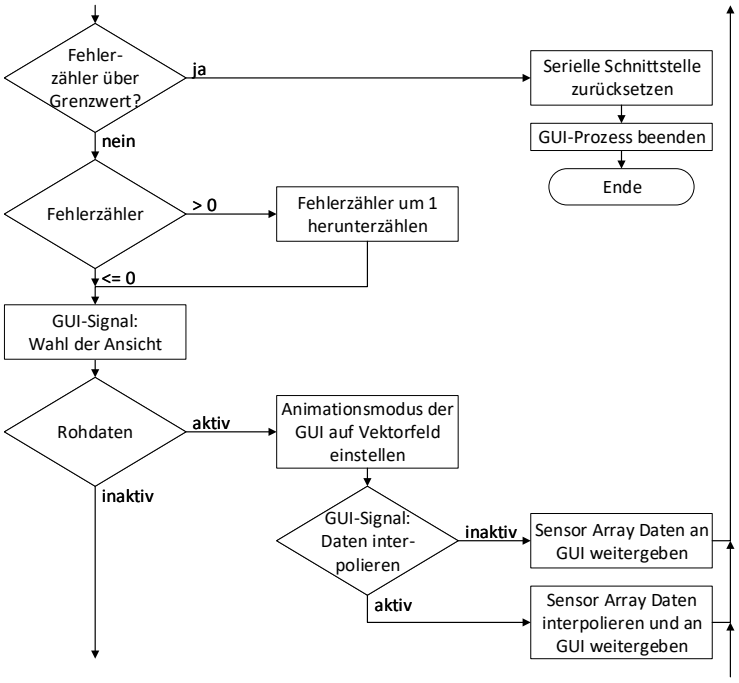


Abbildung 4.6: Flussdiagramm für das Hauptprogramm des Mikrocomputers (Abschnitt 4 von 5). Ist der Fehlerzähler über einem Grenzwert, wird die serielle Schnittstelle zurückgesetzt und der GUI-Prozess und das Hauptprogramm beendet. Ist der Fehlerzähler größer als null, wird dieser heruntergezählt. Abhängig vom GUI-Steuersignal der gewählten Ansicht geschieht im Folgenden die Signalverarbeitung und Aktualisierung der GUI-Daten für die Animation. Das Steuersignal der GUI zur Interpolation wird für die Vektorfeld- und Matrixansichten ausgewertet.

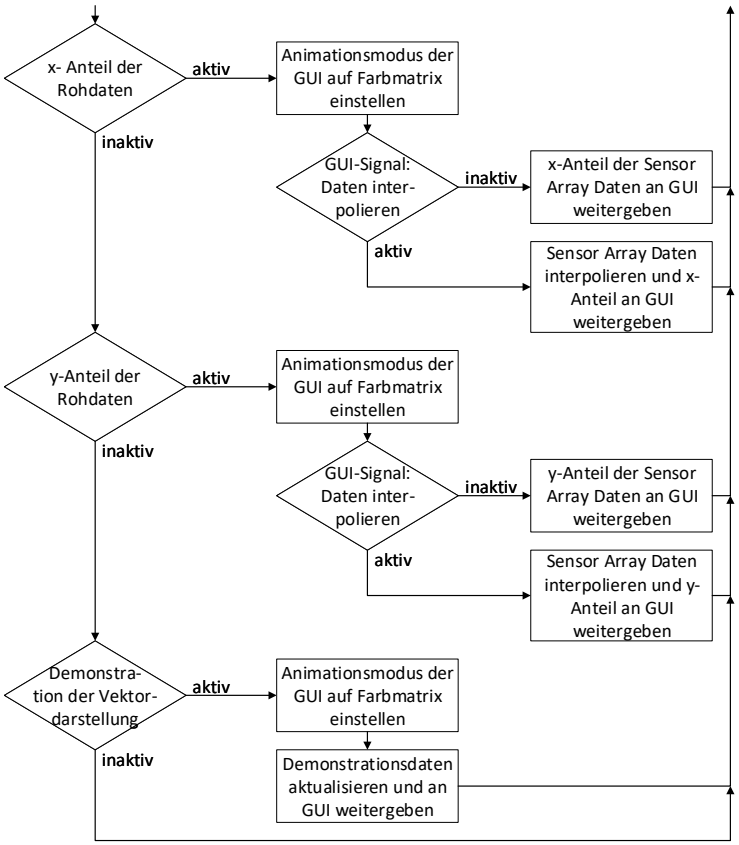


Abbildung 4.7: Flussdiagramm für das Hauptprogramm des Mikrocomputers (Abschnitt 5 von 5). Die Aktualisierung der Animationsdaten wird fortgesetzt. Danach beginnt der Hauptzyklus von vorne.

4.4.4 Funktionen für die Hauptfunktion

Einlesen der Konfigurationsdatei read_configuration_file

Die Einlesefunktion der Konfigurationsdatei liest die *files/config.txt*-Datei zeilenweise. Die Anzahl der Sensoren wird zur Limitation des Speicheraufwandes begrenzt. Abbildung 4.8 zeigt die Implementierung anhand eines Flussdiagramms.

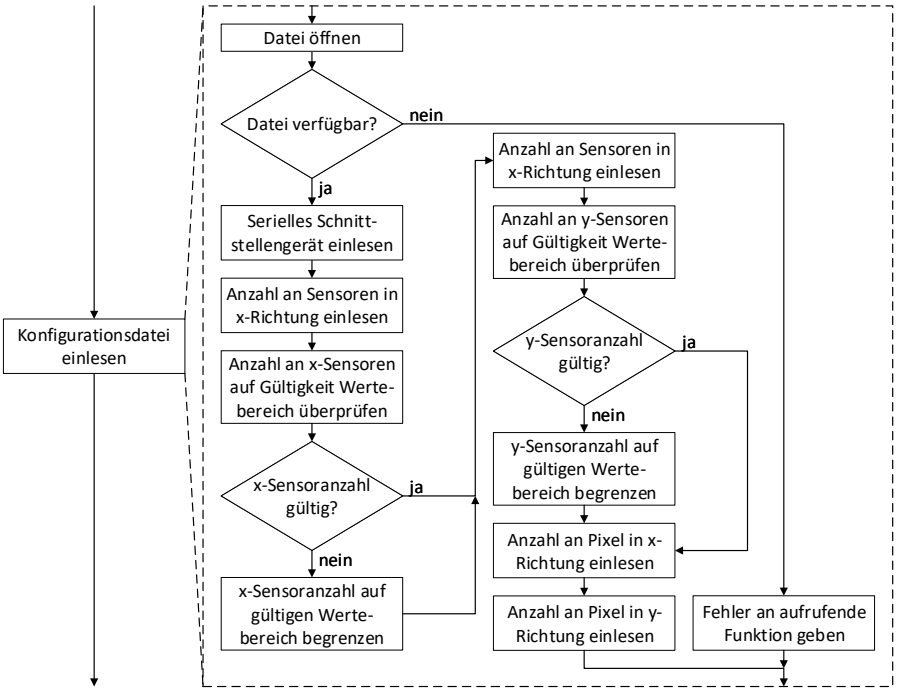


Abbildung 4.8: Flussdiagramm für das Einlesen der Konfigurationsdatei des Mikrocomputers. Zuerst wird der Gerätepfad der seriellen Schnittstelle aus der Konfigurationsdatei eingelesen. Danach wird die Anzahl an Sensoren in x-Richtung des Sensor-Arrays eingelesen und, wenn nötig, auf den gültigen Wertebereich begrenzt. Der gleiche Ablauf geschieht für die Anzahl der Sensoren in y-Richtung. Zuletzt werden die Anzahl der Pixel der Animationsfläche in x- und danach in y-Richtung gelesen.

Einlesen der Filterkoeffizientendatei read_coefficient_file

Die Funktion zum Einlesen der Filterkoeffizienten funktioniert wie die zum Einlesen der Konfigurationsdatei zeilenweise. In der *files/coeff.txt*-Datei wird in jeder Zeile ein Wert erwartet. Die Anzahl entspricht der Anzahl an Sensoren, die in der Konfigurationsdatei angegeben sind. Abbildung 4.9 zeigt die Implementierung anhand eines Flussdiagramms.

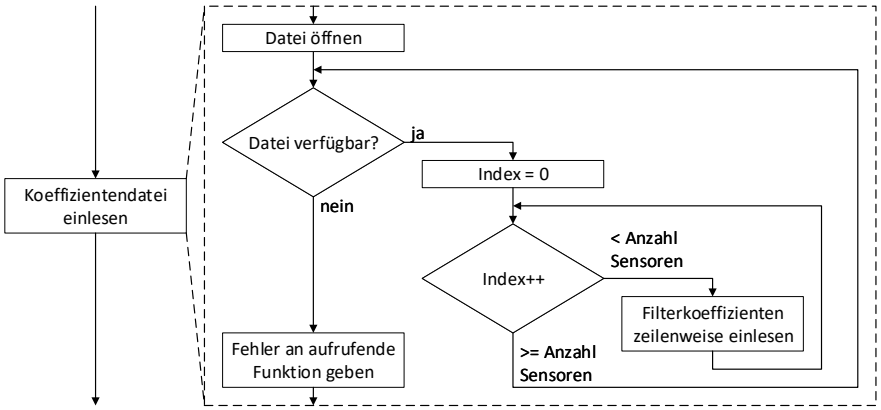


Abbildung 4.9: Flussdiagramm für das Einlesen der Koeffizientendatei des Mikrocomputers. Die Filterkoeffizienten für die Signalverarbeitung werden zeilenweise eingelesen. Die Anzahl der erwarteten Koeffizienten entspricht der Anzahl der Sensoren, die in der Konfigurationsdatei angegeben ist.

4.4.5 Funktionen für das Sensor-Array

Öffnen und Konfigurieren der seriellen Schnittstelle set_up_uart

Die Konfiguration der seriellen Schnittstelle geschieht nach den in Kapitel 2.4 beschriebenen Einstellungen. Abbildung 4.10 zeigt die Implementierung anhand eines Flussdiagramms.

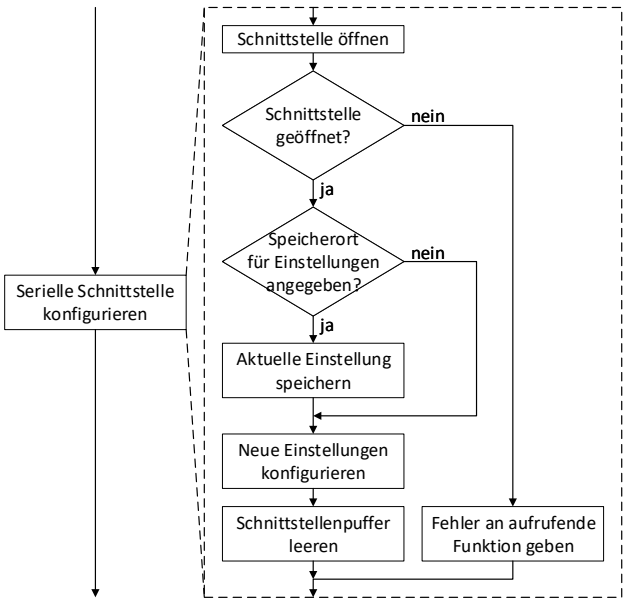


Abbildung 4.10: Flussdiagramm für die Konfiguration der seriellen Schnittstelle des Mikrocomputers. Zunächst wird versucht, die Schnittstelle mit dem als Argument übergebenen Gerät zu öffnen. Gelingt dies nicht, wird die Funktion beendet und der aufrufenden Funktion ein Fehler signalisiert. Bei Erfolg werden die aktuellen Einstellungen der Schnittstelle in der als Argument übergebenen Variable gespeichert. Ist diese Variable nicht angegeben, wird das Speichern übersprungen. Dann werden die übergebenen Einstellungen konfiguriert und damit der Sende- und Empfangspuffer geleert.

Wiederherstellen und Schließen der seriellen Schnittstelle set_down_uart

Beim Aufruf dieser Funktion wird die serielle Schnittstelle mit den Einstellungen der übergebenen Variable für konfiguriert und bei Erfolg geschlossen. Abbildung 4.11 zeigt die Implementierung anhand eines Flussdiagramms.

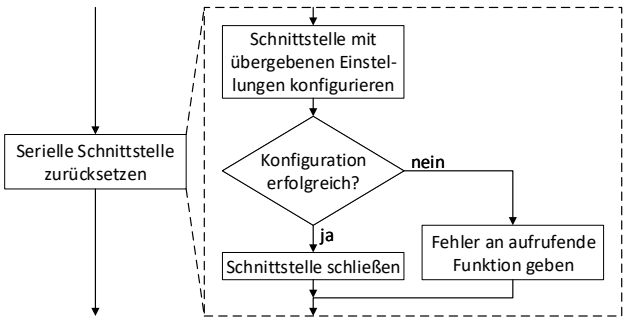


Abbildung 4.11: Flussdiagramm für das Rücksetzen der seriellen Schnittstelle des Mikrocomputers. Die der Funktion als Argument übergebenen Einstellungen werden eingestellt und gespeichert. Danach wird die serielle Schnittstelle geschlossen.

Senden einer Anfrage an den Mikrocontroller sensor_array_send_request

Sendet der Mikrocontroller Daten über die serielle Schnittstelle, werden diese in einem Puffer gespeichert. Diese Funktion liest den Puffer aus und prüft, ob die Übertragung fehlerfrei verlaufen ist. Vor dem Senden eines Befehls über die serielle Schnittstelle muss der Empfangspuffer des Masters leer sein, damit das nächste empfangene Paket die Bestätigung durch den Slave ist. Wurde bei der Übertragung ein Paritätsfehler festgestellt, wird das fehlerhafte Paket nicht in den Empfangspuffer geschrieben und die Anzahl der im Puffer enthaltenen Pakete fällt niedriger aus, als durch die Anfrage des Masters an den Slave erwartet. Diese Methode setzt voraus, dass die serielle Schnittstelle von keinem anderen Programm genutzt wird. Abbildung 4.12 zeigt die Implementierung anhand eines Flussdiagramms.

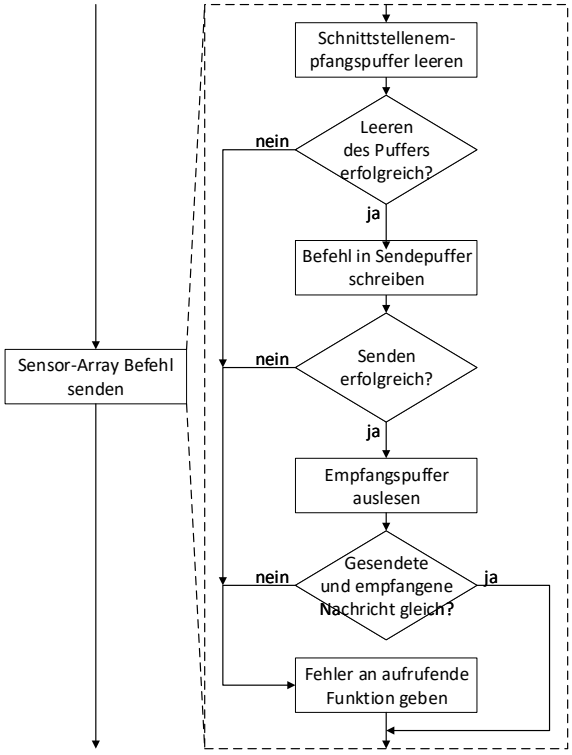


Abbildung 4.12: Flussdiagramm für das Senden eines Pakets des Mikrocomputers an den Mikrocontroller. Zuerst wird der Empfangspuffer des Mikrocomputers geleert. Tritt dabei ein Fehler auf, wird die Funktion mit Benachrichtigung der aufrufenden Funktion beendet. Ist das Leeren erfolgreich, wird der Befehl, welcher der Funktion als Argument übergeben wird, in den Sendepuffer der seriellen Schnittstelle geschrieben um das Paket zu senden. Stimmen die als Reaktion folgende Antwort des Mikrocontrollers und das gesendete Paket überein, ist die Funktion erfolgreich beendet. Anderenfalls wird die Funktion mit einer Fehlernachricht an die aufrufende Funktion beendet.

Empfangen der Sensor-Array-Daten vom Mikrocontroller sensor_array_collect_array_data

Wird eine Anfrage an den Mikrocontroller gesendet, die neben der Bestätigung auch Daten als Antwort verlangt, werden diese wie in dem Kapitel 2.4.2 beschreiben als Antwort gesendet. Diese Funktion übernimmt die Aufgabe der Auswertung der empfangenen Sensor-Array-Daten. Hierzu wird der Empfangspuffer der seriellen Schnittstelle des Mikrocomputers vollständig ausgelesen und überprüft, ob die Anzahl der empfangenen Pakete der Erwartung entspricht. Die Herleitung der erwarteten Anzahl von Paketen ist im oben genannten Kapitel zu finden. Wurde bei der Übertragung ein Paritätsfehler festgestellt, wird das fehlerhafte Paket nicht in den Puffer geschrieben und die Anzahl der im Puffer enthaltenen Pakete fällt niedriger aus, als durch die Anfrage des Masters an den Slave erwartet. Diese Methode setzt voraus, dass die serielle Schnittstelle von keinem anderen Programm genutzt wird. Sind die Pakete vollzählig, werden die Daten zusammengesetzt und gespeichert. Bei einem Fehler wird die aufrufende Funktion benachrichtigt. Abbildung 4.13 zeigt die Implementierung anhand eines Flussdiagramms.

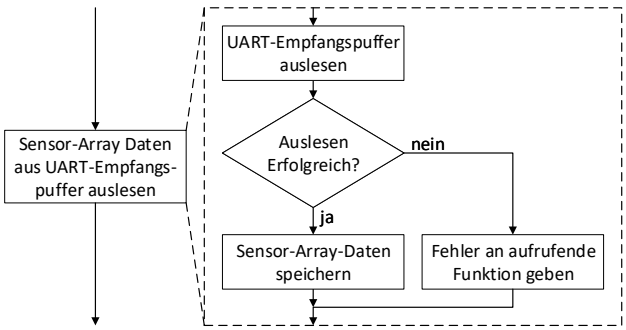


Abbildung 4.13: Flussdiagramm für die Auswertung der empfangenen Pakete des Mikrocomputers, wenn alle Sensor-Array-Daten angefragt werden. Der Empfangspuffer der seriellen Schnittstelle wird ausgelesen. Tritt dabei ein Fehler auf, wird die aufrufende Funktion benachrichtigt und die Funktion beendet. Bei erfolgreichem Empfang werden die Daten gespeichert.

4.4.6 Funktionen für die Benutzeroberfläche und Visualisierung

Erstellen und Verwalten der Benutzeroberfläche ui_thread

Die Erstellung von Benutzeroberflächen mit dem GTK 3 geschieht objektorientiert mit Widgets. Jedes Widegt verfügt über Eigenschaften und Signale, die dessen Erscheinungsbild und Funktion bestimmen. Mit Hilfe von speziellen Funktionen der GTK 3 lassen sich die Objektparameter anpassen. Objekte, die vom Nutzer bedient werden können, werden einer Funktion zugeordnet, die bei Aktivität des Objekts aufgerufen und im Quelltext definiert werden. Abhängig vom Objekttyp kann der Status des Widgets ausgelesen werden. Für die Implementierung dieser Arbeit werden den Widget-Funktionen die Steuervariablen zur Kommunikation zwischen Benutzeroberfläche und Hauptprogramm übergeben, sodass das Hauptprogramm reagieren kann. Der Gebrauch von globalen Variablen ist hier nicht möglich, denn die Funktion des GTK 3 zur Verarbeitung der Nutzeringaben, die zuletzt in dieser Funktion aufgerufen wird, endet erst mit Schließen des Fensters der Benutzeroberfläche. Daher ist diese Funktion ausgelegt, um vom Hauptprogramm als separater Prozess gestartet zu werden. Die Kommunikationsvariablen werden der Funktion der Benutzeroberfläche als Pointer übergeben, sodass die Prozesse auf die selben Speicherorte der Variablen zugreifen. Abbildung 4.14 zeigt die Implementierung anhand eines Flussdiagramms.

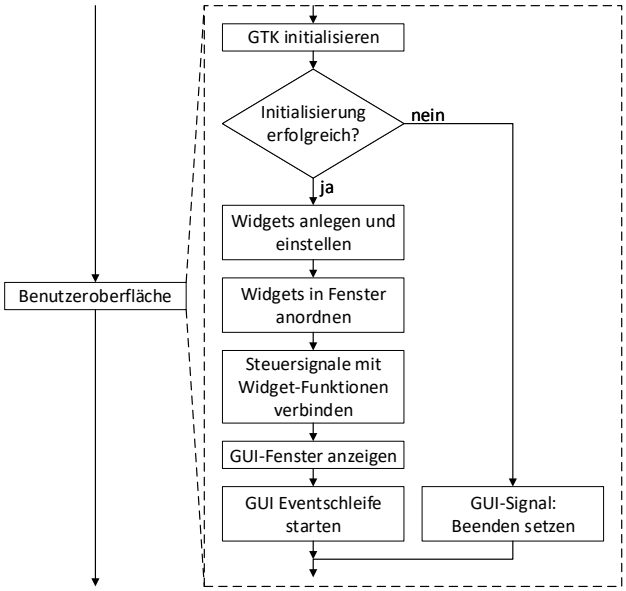


Abbildung 4.14: Flussdiagramm für die Erstellung und Verwaltung der Benutzeroberfläche des Mikrocomputers. Die Funktion beginnt mit der Initialisierung der GTK Bibliothek. Tritt dabei ein Fehler auf, wird das Signal zum Beenden des Programms gesetzt und die Funktion beendet. Ist die Initialisierung erfolgreich, werden die Widgets angelegt und im Fenster angeordnet. Die Funktionen der Widgets, die eine Interaktion mit dem Benutzer erlauben, werden anschließend mit den Steuersignalen zwischen Hauptfunktion und Benutzeroberfläche verbunden. Sie setzen beim Aufruf das jeweilige Steuersignal, sodass die Hauptfunktion darauf reagieren kann. Zuletzt wird das Fenster angezeigt und die Eventschleife gestartet, die die Widgets auf Nutzereingaben prüft. Diese Schleife wird durch Schließen des Fensters verlassen.

Initialisieren der OpenGL Shader
gl_init_shader

Der Quelltext dieser Funktion stammt aus einem Internetforum [2].

Um mit der OpenGL Grafiken darzustellen, müssen Teile der Grafik-Pipeline (englisch Verarbeitungskette) im Quelltext erstellt werden. Die nötigen sogenannten Shader sind der Vertex-Shader, der zur Verarbeitung der Koordinaten des darzustellenden Objekts dient, und Fragment-Shader, welcher die Farbwiedergabe bestimmt. Für das Programm dieser Arbeit reicht es aus, den Vertex-Shader ohne weitere Verarbeitung mit den Koordinaten im dreidimensionalen Raum auszustatten. Da der Vertex-Shader die erste Stufe der Grafik-Pipeline ist, müssen die Argumente aller folgenden Shader ebenfalls an den Vertex-Shader übergeben werden. Somit erhält dieser die Farbwerte für rot, grün und blau, die direkt an den Fragment-Shader weitergegeben werden. Der Fragment-Shader ist die letzte Stufe der Grafik-Pipeline und ist so ausgelegt, dass die Farbwerte zur Darstellung nicht verändert werden. Die Shader müssen bei jedem Start des Programms neu kompiliert und danach in einem Shader-Programm verknüpft werden. Abbildung 4.15 zeigt die Implementierung anhand eines Flussdiagramms.

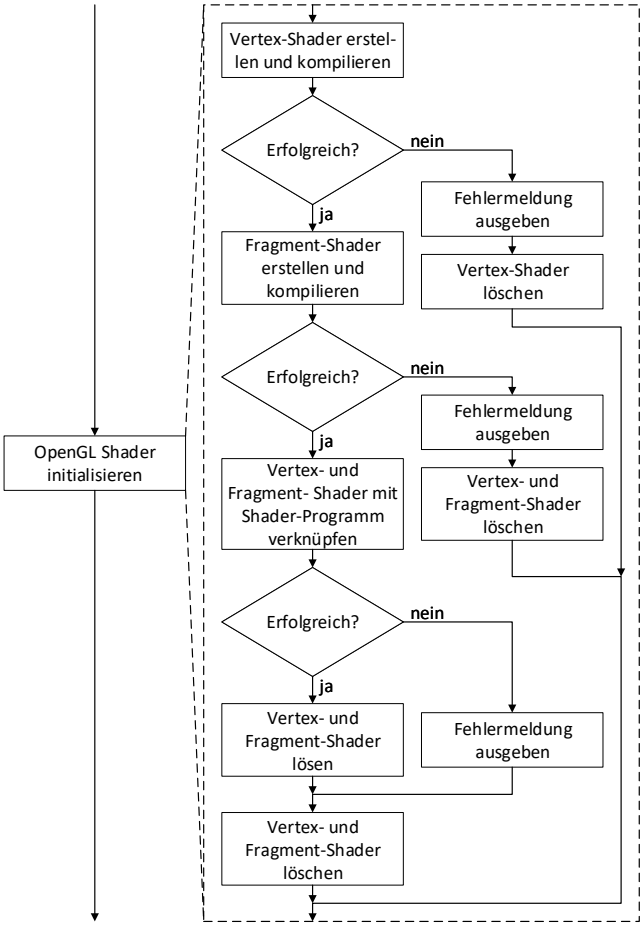


Abbildung 4.15: Flussdiagramm für die Initialisierung der OpenGL-Shader des Mikrocomputers. Nacheinander werden der Vertex- und Fragment-Shader angelegt und kompiliert. Schlägt das Kompilieren fehl, werden die angelegten Shader gelöscht und eine Fehlermeldung ausgegeben, bevor die Funktion beendet wird. Sind beide Shader erfolgreich kompiliert, werden Sie mit dem Shader-Programm verknüpft. Geschieht dies fehlerfrei, werden die Shader wieder gelöst. Bei einem Fehler wird eine entsprechende Fehlermeldung ausgegeben. Zuletzt werden die Shader gelöscht.

Erstellen und Kompilieren der OpenGL Shader gl_create_shader

Der Quelltext dieser Funktion stammt aus einem Internetforum [2].

Da der Vorgang zum Kompilieren der Vertex- und Fragment-Shader identisch ist, dient diese Funktion als Hilfsfunktion für das Initialisieren der Shader. Abbildung 4.16 zeigt die Implementierung anhand eines Flussdiagramms.

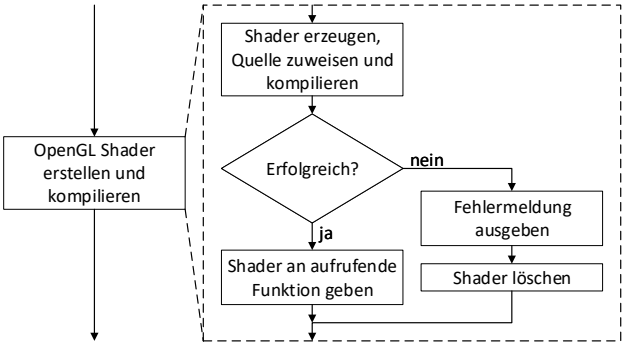


Abbildung 4.16: Flussdiagramm für das Kompilieren der OpenGL-Shader des Mikrocomputers. Zunächst wird der Shader erzeugt und mit dem Quellcode kompiliert. Bei Erfolg wird der Shader an die aufrufende Funktion gegeben. Tritt ein Fehler auf, wird stattdessen eine Fehlermeldung ausgegeben und der Shader gelöscht.

Initialisieren der OpenGL Puffer gl_init_buffers

Der Quelltext dieser Funktion stammt aus einem Internetforum [19].

Die Vertices zur Darstellungen mit der OpenGL werden als dicht gepackter Datenblock, einem sogenannten Vertex-Buffer-Object (VBO), in den Speicher des Grafikprozessors kopiert. Um die Datenstruktur des VBO zu definieren, wird diese aus dem Shader-Programm gelesen und in einem Vertex-Array-Object (VAO) gespeichert, welches vor jeder Aktualisierung der Daten des Grafikprozessors ausgewählt werden muss. Abbildung 4.17 zeigt die Implementierung anhand eines Flussdiagramms.

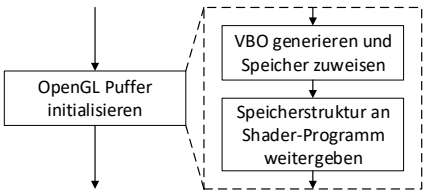


Abbildung 4.17: Flussdiagramm für die Initialisierung der OpenGL-Puffer des Mikrocomputers. Das VBO wird generiert und einem Speicherplatz zugewiesen. Dann wird die Speicherstruktur des VBO aus dem Shader-Programm gelesen und im VAO gespeichert.

Zeichnen des Vektorfeldes mit OpenGL
gl_draw_vector_field

Das Vektorfeld besteht aus Einzelvektoren, die mit gleichmäßigem Abstand in der Animationsfläche verteilt sind. Der Abstand zwischen den Vektoren und zum Außenrand der Animationsfläche entspricht der maximalen Länge eines Vektors. Die Farbe eines Vektors hängt von seinem Betrag ab und reicht von Blau für kurze bis Rot für lange Vektoren. Abbildung 4.18 zeigt den Farbverlauf als Funktion des Betrags des Vektors.

Jeder Vektor besteht aus Punkten, die die Anfangs- und Endposition des Vektors im dreidimensionalen Raum bestimmen. Die Position eines Punktes wird mit drei Vertices für die x-, y- und z-Position angegeben, die Farbe des Punktes mit einem Vertix für je Rot, Grün und Blau. Abbildung 4.19 zeigt die Implementierung anhand eines Flussdiagramms.

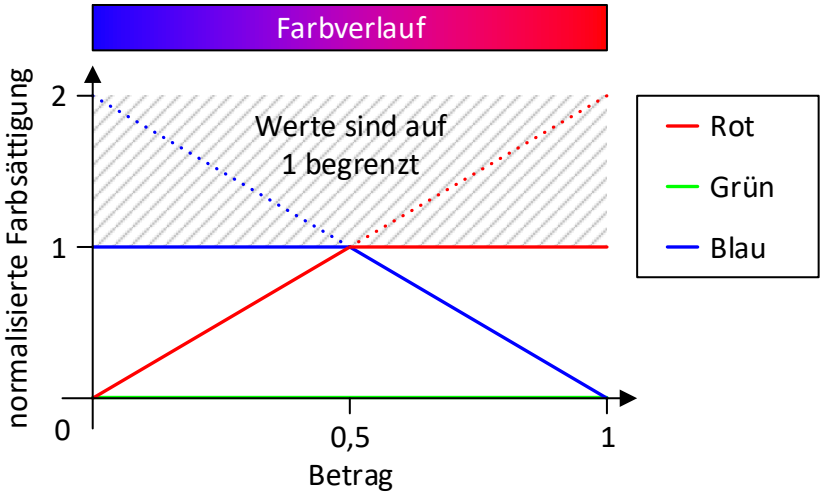


Abbildung 4.18: Farbverlauf für die Visualisierung des Vektorfeldes des Mikrocomputers. Die Farbe jedes einzelnen Vektors hängt von seiner Länge ab. Ein betragsmäßig kurzer Vektor hat eine blaue Färbung, ein langer eine rote.

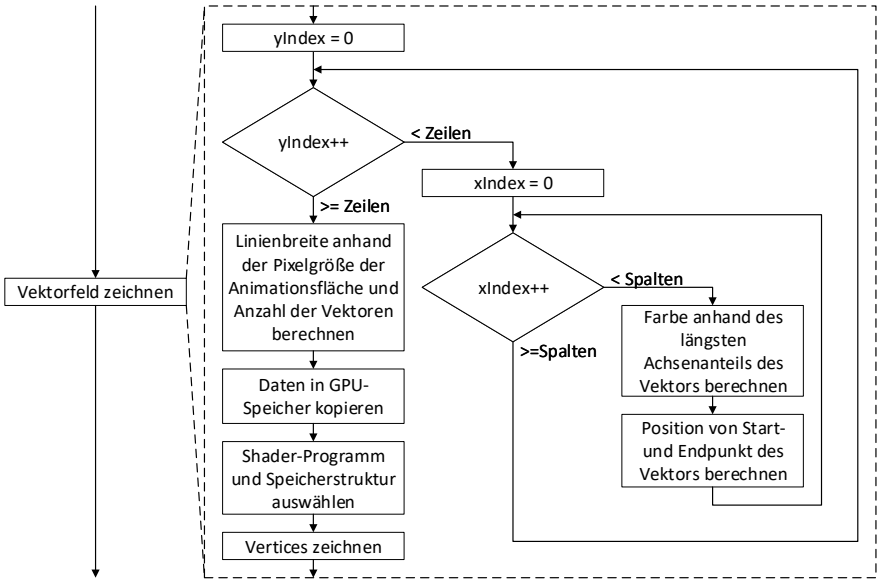


Abbildung 4.19: Flussdiagramm für die Darstellung des Vektorfelds mit dem Mikrocomputer. Die Sensoren des Arrays werden zeilen- und spaltenweise durchlaufen. Dabei wird der Startpunkt des Vektors anhand der Position des Sensors auf dem Array gewählt und die x- und y-Werte für den Endpunkt auf den Startpunkt addiert. Die Farbe wird anhand des größeren der x- und y-Werte je Vektor bestimmt. Sind alle Vektoren berechnet, wird die Linienbreite anhand der Animationsflächengröße eingestellt und die Daten in den Speicher des Grafikprozessors geschrieben. Zur Darstellung werden dann die Shader und die Datenstruktur der Vertices gewählt und zuletzt der Zeichenbefehl an den Grafikprozessor gesendet.

Zeichnen der Farbmatrix mit OpenGL
gl_draw_heatmap

Die Farbmatrix besteht aus rechteckigen gleichmäßig verteilten Feldern, die anhand ihrer Färbung nach der Abbildung 4.20 die Ausprägung des jeweiligen Werts veranschaulichen.

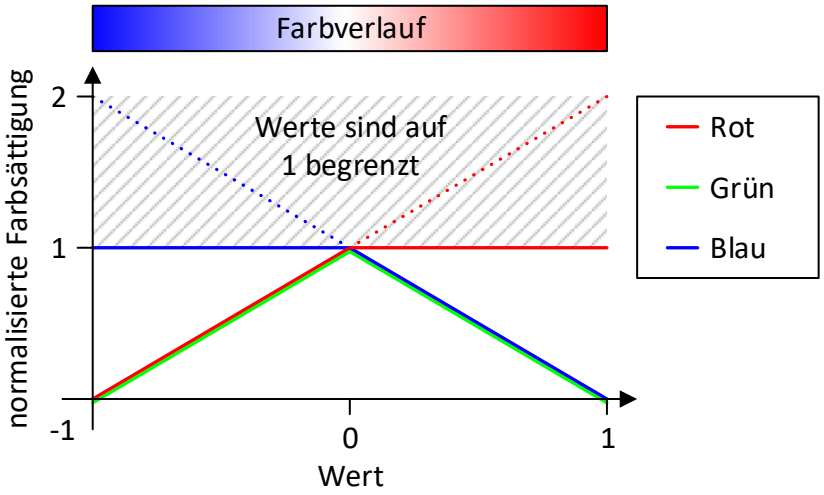


Abbildung 4.20: Farbverlauf für die Visualisierung der Farbmatrix des Mikrocomputers. Die Farbe jedes einzelnen Feldes hängt vom jeweiligen Wert ab. Ein negativer Wert hat eine blaue Färbung, ein positiver eine rote. Ist der Wert nahe null, ist die Färbung weiß.

Die Position und Farbe der Felder wird anhand der Vertices seines Punktes festgelegt. Die Zuordnung der Vertices je Punkt entspricht exakt der gleichen, wie beim Vektorfeld. Da je Feld jedoch nur ein Punkt benötigt wird, ist deren Anzahl im Vergleich zum Vektorfeld halbiert. Abbildung 4.21 zeigt die Implementierung anhand eines Flussdiagramms.

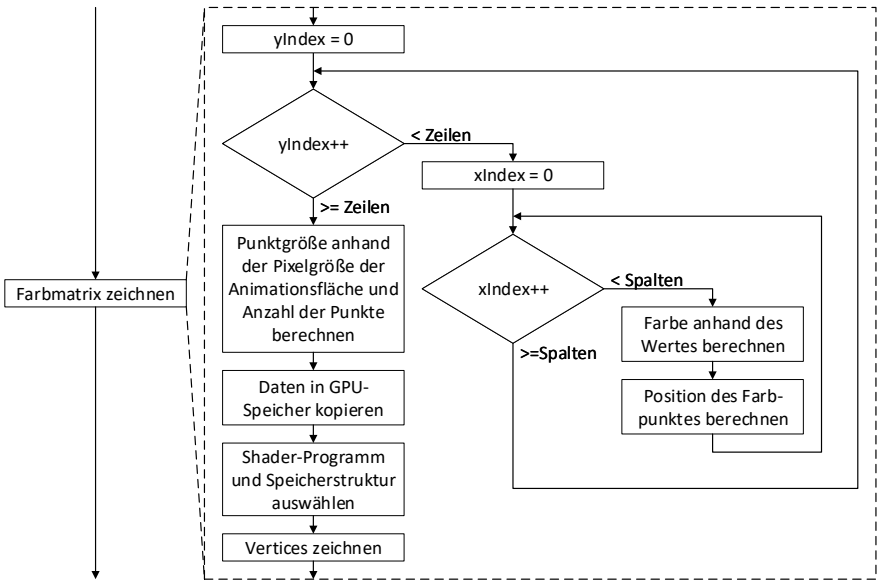


Abbildung 4.21: Flussdiagramm für die Darstellung der Farbmatrix mit dem Mikrocomputer. Die Sensoren des Arrays werden zeilen- und spaltenweise durchlaufen. Dabei wird die Position des Farbpunktes anhand der Position des Sensors auf dem Array gewählt. Die Farbe wird anhand des Sensorwertes der gewählten Achse je Punkt bestimmt. Sind alle Punkte berechnet, wird die Punktgröße anhand der Animationsflächengröße eingestellt und die Daten in den Speicher des Grafikprozessors geschrieben. Zur Darstellung werden dann die Shader und die Datenstruktur der Vertices gewählt und zuletzt der Zeichenbefehl an den Grafikprozessor gesendet.

Zeichnen des Vektors mit OpenGL
gl_draw_vector

Die Vektoranzeige zeigt einen Vektor mit unbestimmter Länge, der einen Versatz seines Startpunktes vom Mittelpunkt der Animationsfläche und einen Winkel zu seiner x-Achse besitzt. Damit lässt sich der Winkel und die Achse eines Rotierenden Objektes darstellen. Je größer der Betrag des Versatzes ist, desto stärker ist die Färbung des Vektors ins Rote. Weicht der Startpunkt des Vektors nur wenig vom Mittelpunkt ab, ist die Färbung blau. Abbildung 4.18 zeigt den Farbverlauf anhand des Betrages des Versatzes. Der Winkel des Vektors wird in positive Richtung von der x-Achse im ersten Quadranten aus gewertet. Abbildung 4.22 zeigt die positive Richtung.

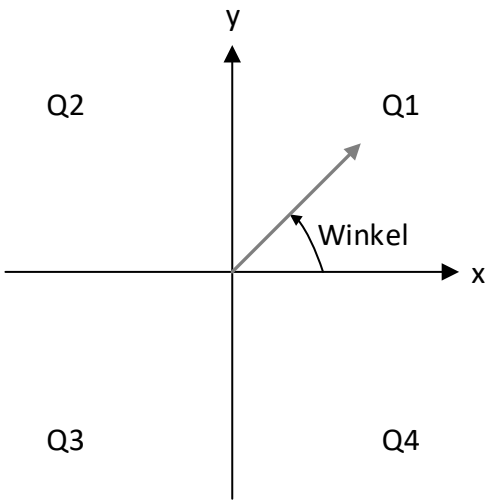


Abbildung 4.22: Winkeldefinition für die Visualisierung des Mikrocomputers

Wie beim Vektorfeld wird der Vektor hier ebenfalls mittels zweier Punkte definiert. Positionierung und Färbung sind in gleicher Weise mit den Vertices strukturiert. Die Position des Startpunkts des Vektors hängt vom der Funktion übergebenen Versatz ab. Der Endpunkt wird mit dem übergebenen Winkel bestimmt und liegt immer außerhalb des sichtbaren Bereichs. Abbildung 4.23 zeigt die Implementierung anhand eines Flussdiagramms.

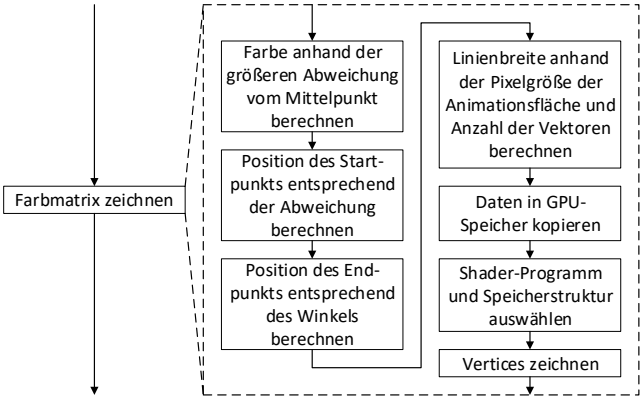


Abbildung 4.23: Flussdiagramm für die Darstellung des Vektors mit dem Mikrocomputer. Die Farbe des Vektors wird anhand der Abweichung vom Mittelpunkt der Animationsfläche bestimmt. Der Startpunkt des Vektor entspricht dem Versatz, der Endpunkt liegt außerhalb der Animationsfläche und wird anhand des Winkels berechnet. Ist der Vektor berechnet, wird die Linienbreite anhand der Größe der Animationsfläche eingestellt und die Daten in den Speicher des Grafikprozessors geschrieben. Zur Darstellung werden dann die Shader und die Datenstruktur der Vertices gewählt und zuletzt der Zeichenbefehl an den Grafikprozessor gesendet.

5 Demonstrationsaufbau

Die Mechanik des Demonstrationsaufbaus besteht aus den in Kapitel 1.2 beschriebenen Komponenten. Um den Aufbau dem Systemkonzept dieser Arbeit anzupassen, muss die Platinenbefestigung überarbeitet werden.

Zusätzlich wird die Pendellagerung neu konstruiert. Die bisherige Lösung des Schwenklagers ist nicht robust und besitzt weder eine Höhenverstellung noch mechanische Endanschläge. Zudem neigt die Aluminiumbuchse, die im Schwenklager sitzt und die Pendelstange umfasst, dazu, Aluminiumstaub zu erzeugen, welcher sich auf das Sensor-Array absetzen kann. Außerdem liegen die Stahl Schwenkkugelpfanne sowie die Befestigungsmuttern der Aluminiumhalterung des Schwenklagers ohne Kraftverteilung auf dem spröden Kunststoff der Trägerplatte. Stöße, wie sie bei üblicher Handhabung auftreten können, werden über die kleine Auflagefläche übertragen und könnten langfristig zu Rissen im Kunststoff führen.

5.1 Platinenmontage

Die Platinen des Mikrocontrollers und Mikrocomputers werden jeweils rückseitig an einem stehenden Aluminium-L-Profil befestigt. Abbildung 5.1 zeigt die Anordnung. Die Platine des Sensor-Arrays wird auf einer Aluminiumplatte montiert, deren Position unter der Pendel einstellbar ist.

Distanzbuchsen aus Kunststoff mit M3-Gewinde befestigen die Platinen. Da die Befestigungslöcher des Mikrocomputers zu klein für M3 Schrauben sind, werden Sie von 2,75 mm auf 3 mm aufgebohrt. Dies ist ohne Probleme möglich, da die Platine im unmittelbaren Umfeld der Befestigungslöcher keine Komponenten oder Leiterbahnen besitzt. Das Winkelprofil des Mikrocomputers und Mikrocontrollers wird mit zwei Holzschrauben am Außenrand und die Montageplatte des Sensor-Arrays im Zentrum der Grundplatte befestigt wie die Abbildung 1.3 zeigt.

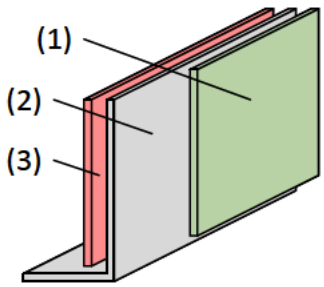


Abbildung 5.1: Anordnung der Platinenmontage. Die Platinen des Mikrocontrollers und Mikrocomputers sind mit der Rückseite an einem stehenden Aluminum-L-Profil befestigt.
(1) Platine des Mikrocomputers
(2) Aluminium-L-Profil
(3) Platine des Mikrocontrollers

5.2 Pendellagerung

Für eine bessere Stabilität, Haptik und Funktionalität wird die Aufnahme des Schwenklagers neu konstruiert. Wichtige Anforderungen sind dabei, dass das Pendel höhenverstellbar, rotations- und neigungsfähig ist, grober Handbedienung standhält und keine elektrisch leitfähigen Späne produziert.

Die Einzelteile der überarbeiteten Konstruktion sind in der Abbildung 5.2 als Explosionszeichnung und in der Abbildung 5.3 als Schnittbild dargestellt und umfassen die in der Tabelle 5.1 aufgelisteten Positionen mit den zugehörigen Funktionen.

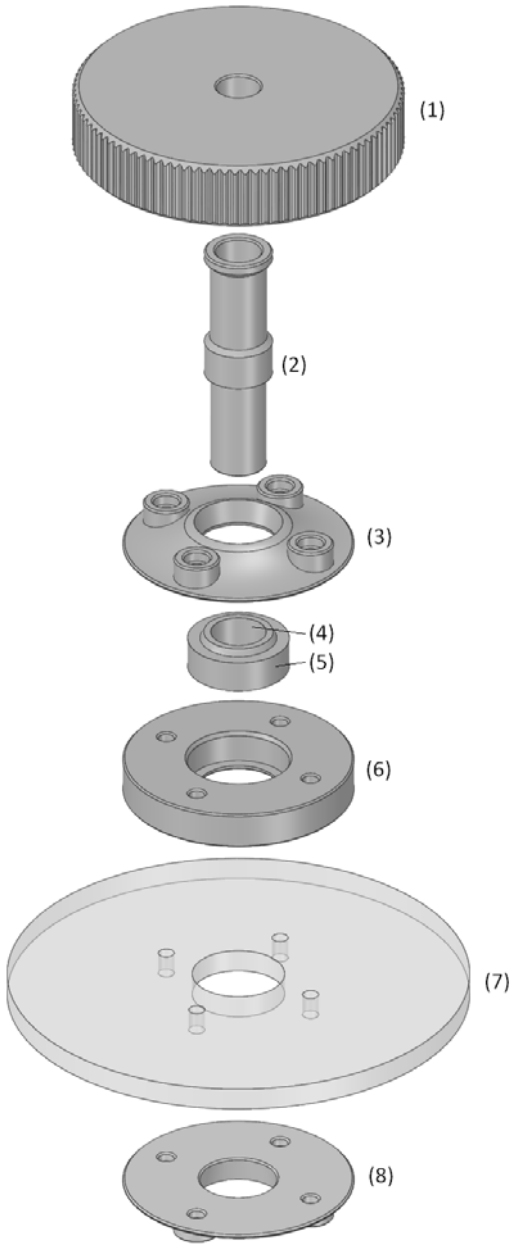


Abbildung 5.2: Explosionszeichnung der Pendellagerung. Zu sehen sind von oben aus das Höhenstellrad (1), die Lagerbuchse (2), die Oberplatte (3), das Schwenklager (4, 5), der Lagerblock (6), ein Ausschnitt der Trägerplatte (7) und die Unterplatte (8). Nicht gezeigt sind die vertikal eingeführte Pendelachse, die Höhenstellmutter des Höhenstellrades und die Befestigungsschrauben und -Muttern des Lagerblocks sowie der Rest des Demonstrationsaufbaus.

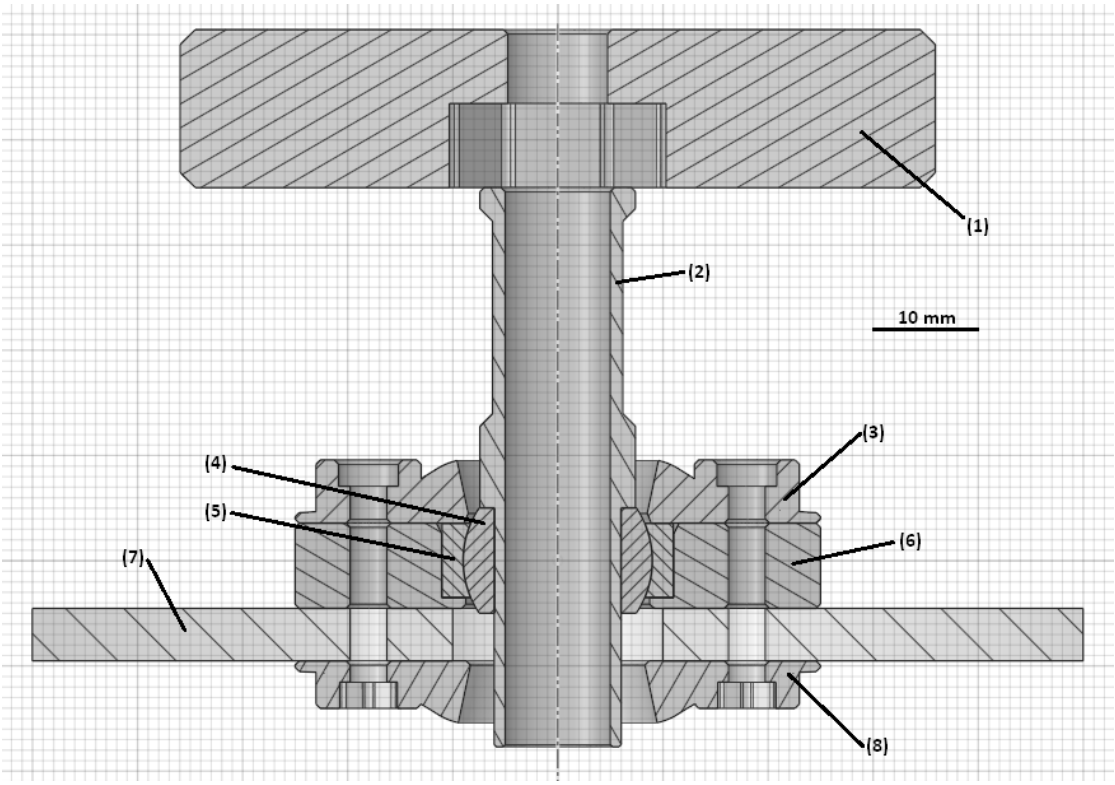


Abbildung 5.3: Schnittbild der Pendellagerung. Zu sehen ist der fertig montierte Aufbau abzüglich der vertikal eingeführten Pendelachse, der Höhenstellmutter des Höhenstellrades und der Befestigungsschrauben und -Muttern des Lagerblocks. Die Einzelteile sind mit Nummern versehen, um eine eindeutige Zuordnung zu ermöglichen.
(1) Höhenstellrad
(2) Lagerbuchse
(3) Oberplatte
(4) Schwenkkugel
(5) Schwenkkugelpfanne
(6) Lagerblock
(7) Trägerplatte
(8) Unterplatte

Einzelteil	Bezeichnung	Funktion
(1)	**Höhenstellrad** (Kunststoff, 3d-Druck)	Zylinder mit Rändelung an der Außenflanke für Handbedienung. Eine sechseckige Tasche auf Rotationsachse nimmt eine Höhenstellmutter auf, die auf die Pendelachse geschraubt ist. Das Durchgangsloch über der Tasche und auf der Rotationsachse bietet der Drehung auf der Gewindeachse durch Reibung Widerstand, sodass sich das Höhenstellrad nicht verstellt, wenn die Pendelachse in der Lagerbuchse (2) rotiert.
(2)	**Lagerbuchse** (Kunststoff, 3d-Druck)	Eingepresst in die Schwenkkugel (4) des Schwenklagers bietet die Lagerbuchse (2) bei Neigung der Pendelachse zusammen mit der Ober- (3) und Unterplatte (8) des Lagerblocks (6) einen mechanischen Endanschlag. Die Lagerbuchse (2) dient außerdem als Radialgleitlager der Pendelachse und Axialgleitlager der Höhenstellmutter des Höhenstellrades (1). Durch die Länge des verjüngten Schafts der Lagerbuchse (2) wird ein praktikabler Abstand zwischen dem Höhenstellrad (1) und der Trägerplatte (7) gehalten.
(3)	**Oberplatte** (Kunststoff, 3d-Druck)	Die Oberplatte (3) fixiert die Schwenkkugelpfanne (5) im Lagerblock (6) und nimmt Klemmlast zwischen ihr (3) und der Unterplatte (8) auf. Sie dient weiter als Neigungsanschlag für die Lagerbuchse (2) der Pendelachse. Der maximale Neigungswinkel des Schwenklagers beträgt 14°. Die mechanischen Anschläge begrenzen die Neigung auf 12°.
(4)	**Schwenkkugel** (Stahl)	Teil des bestehenden Schwenklagers. Die Schwenkkugel (4) kann mit geringer Reibung in der Schwenkkugelpfanne (5) in allen Achsen frei rotiert werden.
(5)	**Schwenkkugelpfanne** (Stahl)	Teil des bestehenden Schwenklagers. Die Schwenkkugelpfanne (5) nimmt die Schwenkkugel (4) auf.
(6)	**Lagerblock** (Kunststoff, 3d-Druck)	Der Lagerblock (6) trägt die Schwenkkugelpfanne (5) des Schwenklagers und verteilt die Kräfte der Pendelachse großflächig auf die Trägerplatte (7).
(7)	**Trägerplatte**	Die Trägerplatte (7) nimmt die Tragelast der Pendelachse auf und ist Teil des bestehenden Aufbaus.
(8)	**Unterplatte** (Kunststoff, 3d-Druck)	Die Unterplatte (8) nimmt die Klemmlast zwischen ihr (8) und der Oberplatte (3) auf und verteilt sie großflächig auf die Trägerplatte (7). Sie (8) dient wie auch die Oberplatte (3) als mechanischer Neigungsendanschlag für die Lagerbuchse (2) der Pendelachse.

Tabelle 5.1: Einzelteile des Schwenklagers für das Pendel

6 Erprobung und Test

6.1 Allgemeine Tests

Zum allgemeinen Test aller Komponenten und des Programms für den Mikrocomputer werden die in der Tabelle 6.1 aufgelisteten Tests durchgeführt. Tritt während des Programmablaufs ein Fehler auf, wird im Konsolenfenster, mit welchem das Programm gestartet wurde, eine entsprechende Warnung oder Fehlermeldung ausgegeben. Die implementierten Ausgaben sind in den Tabellen 6.2, 6.3 und 6.4 aufgelistet und beschrieben.

Test	Kommentar
Start des Programms: Das Programm wird mit dem Befehl ***./ISAR_Viewer*** im Oberordner gestartet. Das Fenster der Benutzeroberfläche öffnet sich. Die Animationsfläche zeigt die Rohdaten des Sensor-Arrays in Vektorfelddarstellung.	Die Bedienfläche ist schwarz und kein Bedienelement wird angezeigt. Wird das Fenster mit der Bedienfläche über den Bildschirmrand hinaus verschoben und danach wieder vollständig auf den Bildschirm zurück, ist der Grafikfehler behoben. Während der Ausführung des Programms wird für die Gesamtauslastung des Prozessors im Mittel 81 % angezeigt.
Beenden des Programms: Das gestartete Programm wird mit Mausklick auf das Kreuz des Fenster geschlossen. Mit dem Linux-Task-Manager wird beobachtet, dass das Hauptprogramm und das Fenster der Bedienoberfläche beendet werden.	Hauptprogramm und Fenster der Bedienoberfläche werden wie beschrieben beendet.
Zweites Starten und Beenden des Programms: Das Programm wird wie zuvor gestartet und beendet.	Das Verhalten des Hauptprogramms und des Fensters der Bedienoberfläche ist unverändert.
Verfügbarkeit aller Sensorsignale des Arrays: Nach dem Programmstart wird mit einem Magneten über das Sensor-Array gefahren. Die Vektorfelddarstellung zeigt, dass alle Sensoren in x- und y-Richtung positiv und negativ ausgesteuert werden.	Alle Sensoren werden auf beiden Achsen in beide Richtungen ausgesteuert.
Position der Sensorsignale des Arrays im Animationsfeld: Nach dem Programmstart wird mit einem Magneten über das Sensor-Array gefahren. Die Vektorfelddarstellung zeigt, dass die Position der Sensor-Daten mit der Position der Sensoren auf dem Array übereinstimmen.	Die Positionen stimmen überein.
Ausführung des Programms unter Belastung durch Videowiedergabe: Im Chromium-Web-Browser wird ein Video mit einer Auflösung von 1280 x 720 Pixeln und progressiver Aktualisierung abgespielt. Das Programm wird gestartet und die Bedienbarkeit der Bedienelemente sowie die Gleichmäßigkeit und Rate der Bilderneuerung des Animationsfeldes getestet.	Die Bedienelemente lassen sich im Vergleich zur Einzelausführung des Programms mit kurzer Verzögerung bedienen. Die Aktualisierungsrate der Animation ist gleichmäßig und merklich niedriger, reicht zur flüssigen Wiedergabe der Sensor-Array-Daten jedoch aus. Das wiedergegebene Video stockt beim Vorladen häufig. Die Gesamtauslastung des Prozessors beträgt im Mittel 99 %.

Tabelle 6.1: Liste der allgemeinen Programmtests

Fehlermeldung	**Beschreibung**
Error in read_configuration_file: Inaccessible configuration file at path %s.	Die Konfigurationsdatei am Pfad *%s* ist zum Einlesen der Parameter für das Programm nicht verfügbar.
Error in read_configuration_file: Incomplete configuration file. Missing serial interface device path.	Die Konfigurationsdatei enthält keine Daten. Der erste Eintrag des Gerätepfads der seriellen Schnittstelle fehlt.
Error in read_configuration_file: Incomplete configuration file. Missing number of sensors of the array in x-direction.	Die Konfigurationsdatei enthält nur einen Eintrag. Der zweite Eintrag der Anzahl der Sensoren auf dem Array in x-Richtung fehlt.
Warning in read_configuration_-file: Requested sensor array size in x direction of %u is too high and will be set to MAX_X_NUMBER_OF_-SENSORS.	Der zweite Eintrag der Anzahl der Sensoren des Arrays in x-Richtung mit dem Wert *%u* liegt außerhalb des gültigen Bereichs und wurde auf den Maximalwert *MAX_X_NUMBER_OF_SENSORS* korrigiert.
Error in read_configuration_file: Incomplete configuration file. Missing number of sensors of the array in y-direction.	Die Konfigurationsdatei enthält nur zwei Einträge. Der dritte Eintrag der Anzahl der Sensoren auf dem Array in y-Richtung fehlt.
Warning in read_configuration_-file: Requested sensor array size in y direction of %u is too high and will be set to MAX_Y_NUMBER_OF_-SENSORS.	Der dritte Eintrag der Anzahl der Sensoren des Arrays in y-Richtung mit dem Wert *%u* liegt außerhalb des gültigen Bereichs und wurde auf den Maximalwert *MAX_Y_NUMBER_OF_SENSORS* korrigiert.
Error in read_configuration_file: Incomplete configuration file. Missing number of pixels of the animation area in x-direction.	Die Konfigurationsdatei enthält nur drei Einträge. Der vierte Eintrag der Anzahl der Pixel auf der Animationsfläche in x-Richtung fehlt.
Error in read_configuration_file: Incomplete configuration file. Missing number of pixels of the animation area in y-direction.	Die Konfigurationsdatei enthält nur vier Einträge. Der fünfte Eintrag der Anzahl der Pixel auf der Animationsfläche in y-Richtung fehlt.
Error in read_coefficient_file: Inaccessible coefficients file at path %s.	Die Filterkoeffizientendatei am angegebenen Pfad *%s* ist zum Einlesen der Parameter für das Programm nicht verfügbar.
Error in read_configuration_file: Incomplete coefficients file. Expected %d floating point values. The values must be separated by a "\n" character.	Die Konfigurationsdatei enthält nicht genügend Einträge. Die Erwartete Anzahl ist *%d*.

Tabelle 6.2: Fehlermeldungen der Hilfsfunktionen des Programms für den Mikrocomputer

Fehlermeldung	Beschreibung
Warning in sensor_array_send_request: Unexpected data in receive buffer. Clearing buffer ... done.	Ist der Empfangspuffer der seriellen Schnittstelle nicht leer, wird der Empfangspuffer vor dem Senden geleert.
Error in sensor_array_send_request: Request was not answered.	Auf die Anfrage des Masters hin hat der Slave nicht rechtzeitig geantwortet.
Error in sensor_array_send_request: Erroneous answer received. Should be %2x not %2x	Die Antwort des Slaves in Form der Befehlsbestätigung mit dem Wert *%2x* stimmt nicht mit der Anfrage des Masters mit dem Wert *%2x* überein.
Error in sensor_array_collect_array_data: Timeout while receiving data.	Der Master hat in der längsten zu erwartenden Antwortzeit des Slaves nur fehlerhafte oder keine Pakete erhalten.
Warning in sensor_array_collect_array_data: Data was out of range %d times.	Die vom Slave gesendeten Sensor-Array-Daten waren *%d*-mal außerhalb des zu erwartenden Wertebereichs und wurden auf das Maximum korrigiert.
Error in sensor_array_collect_row_data: Timeout while receiving data.	Der Master hat in der längsten zu erwartenden Antwortzeit des Slaves nur fehlerhafte oder keine Pakete erhalten.

Tabelle 6.3: Fehlermeldungen der Sensor-Array-Funktionen für das Programm des Mikrocomputers

Fehlermeldung	Beschreibung
gl_create_shader - Compile failure in %s shader: %s	Der *%s*-Shader für die Funktionen der Visualisierung mit der OpenGL konnte nicht kompiliert werden. *%s* gibt die Fehlerursache an.
gl_init_shaders - Couldn't create vertex shader	Der Vertex-Shader für die Funktionen der Visualisierung mit der OpenGL konnte nicht generiert werden.
gl_init_shaders - Couldn't create fragment shader	Der Fragment-Shader für die Funktionen der Visualisierung mit der OpenGL konnte nicht generiert werden.
gl_init_shaders - Linking failure: %s	Die Vertex- und Fragment-Shader konnten nicht mit dem Shader-Programm verbunden werden. *%s* gibt die Fehlerursache an.

Tabelle 6.4: Liste der Fehlermeldungen der Funktionen für die Visualisierung des Mikrocomputers

6.2 Zeitmessung

6.2.1 Messung des seriellen Busses und des Mikrocontrollers

Zum Test der seriellen Kommunikation und Messung der Zeiten des Mikrocontrollers werden die Signale RxD, TxD, CTS und RTS des seriellen Busses zwischen Mikrocomputer und Mikrocontroller mit einem Oszilloskop aufgezeichnet. Tabelle 6.5 zeigt die Zuordnung vom jeweiligen Kanal des Oszilloskops zum Signal des seriellen Busses. Der Mikrocontroller agiert als Slave und der Mikrocomputer als Master, der alle implementierten Anfragen aus der Tabelle 2.3 zur Aufzeichnung mit dem Oszilloskop sendet.

Kanal	Kennzeichnung	Beschreibung
1	M_RTS	RTS-Signal des Masters
2	M_CTS	CTS-Signal des Masters
3	M_RX	RxD-Signal des Masters
4	M_TX	TxD-Signal des Masters

Tabelle 6.5: Kanalbeschreibung des Oszilloskops für die Zeitmessung des seriellen Busses und des Mikrocontrollers

Anfrage zum Aufnehmen der Sensor-Array-Daten

Abbildung 6.1 zeigt die Aufzeichnung des Oszilloskops der gesamten Kommunikation zwischen Master und Slave, wenn der Befehl zum Aufnehmen der Sensor-Array-Daten vom Master an der Slave gesendet wird. Die Bearbeitungszeit des Slaves ist $t_{Bearbeitung} = 606\ \mu s$, die Antwortzeit von $t_{Antwort} = 23,9\ ms$ entspricht der erwarteten Anzahl von $n_{Bestätigung} = 1$ Paket.

$$n_{Pakete} = \frac{t_{Antwort}}{t_{Paket,\ 460800\ Hz,\ 8E1}} = \frac{23,9\ \mu s}{23,872\ \mu s} = 1,0 \rightarrow 1 Paket$$

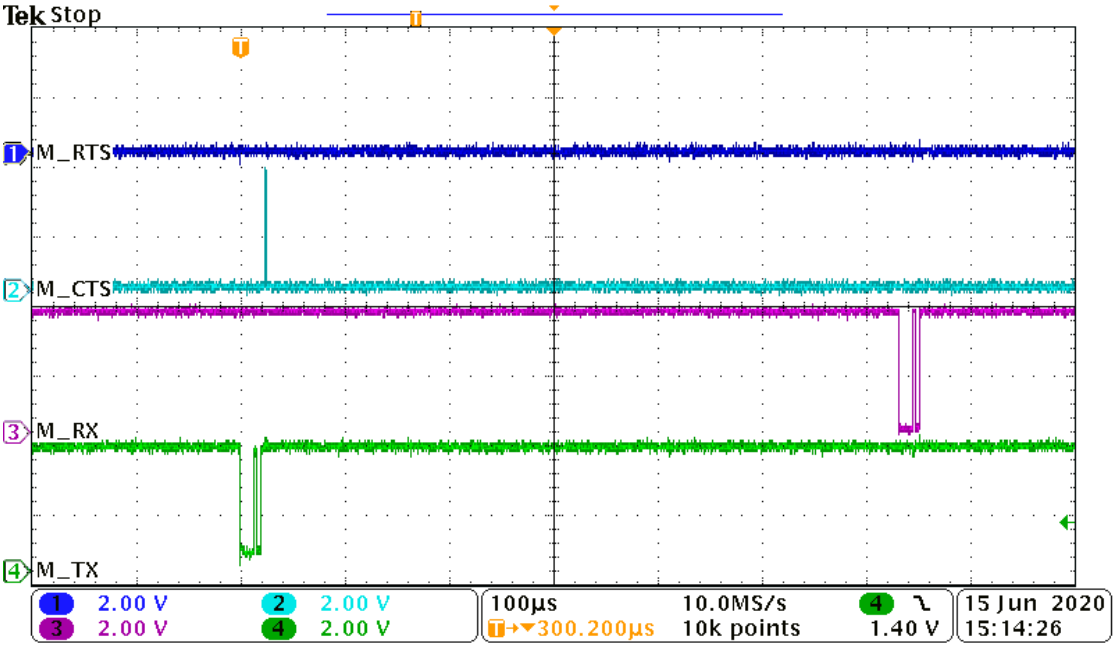


Abbildung 6.1: Aufzeichnung des Oszilloskops für den Befehl zum Aufnehmen der Sensor-Array-Daten. Das RTS-Signal des Masters behält im aufgezeichneten Bereich den low-Pegel. Sendet der Master mit dem TxD-Signal die Anfrage an den Slave, steigt das CTS-Signal des Masters durch den Slave auf den high-Pegel, bis dieser das Paket aus seinem Empfangspuffer ausgelesen hat. Die Bearbeitung des Befehls durch den Slave beginnt mit dem Ende des Pakets des Masters. Die Bearbeitungszeit dauert bis zur Antwort des Slaves, welche aus dem wiederholten Befehlspaket des Masters besteht. Die Antwortzeit beginnt mit dem ersten gesendeten Paket des Slaves und endet nach dem letzten.

Anfrage zum Aufnehmen und Senden der Sensor-Array-Daten

Abbildung 6.2 zeigt die Aufzeichnung des Oszilloskops der gesamten Kommunikation zwischen Master und Slave, wenn der Befehl zum Aufnehmen und Senden der Sensor-Array-Daten vom Master an der Slave gesendet wird. Die Bearbeitungszeit des Slaves ist $t_{Bearbeitung} = 606\ \mu s$, die Antwortzeit von $t_{Antwort} = 6,13\ ms$ entspricht der erwarteten Anzahl von $n_{Pakete/Array} = 257$ Paketen.

$$n_{Pakete} = \frac{t_{Antwort}}{t_{Paket,\ 460800\ Hz,\ 8E1}} = \frac{6,13\ ms}{23,872\ \mu s} = 256,8 \rightarrow 257 Pakete$$

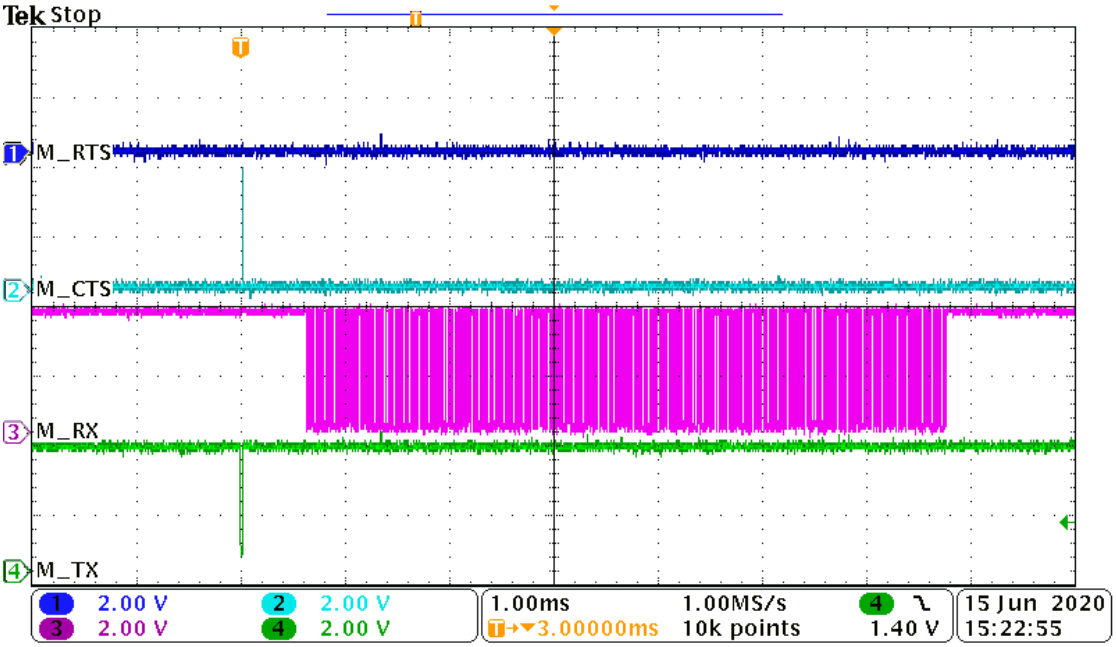


Abbildung 6.2: Aufzeichnung des Oszilloskops für den Befehl zum Aufnehmen und Senden der Sensor-Array-Daten. Das RTS-Signal des Masters behält im aufgezeichneten Bereich den low-Pegel. Sendet der Master mit dem TxD-Signal die Anfrage an den Slave, steigt das CTS-Signal des Masters durch den Slave auf den high-Pegel, bis dieser das Paket aus seinem Empfangspuffer ausgelesen hat. Die Bearbeitung des Befehls durch den Slave beginnt mit dem Ende des Pakets des Masters. Die Bearbeitungszeit dauert bis zur Antwort des Slaves, welche aus dem wiederholten Befehl des Masters und den Sensor-Array-Daten besteht. Die Antwortzeit beginnt mit dem ersten gesendeten Paket des Slaves und endet nach dem letzten.

Anfrage zum Senden der Sensor-Array-Daten

Abbildung 6.3 zeigt die Aufzeichnung des Oszilloskops der gesamten Kommunikation zwischen Master und Slave, wenn der Befehl zum Senden der Sensor-Array-Daten vom Master an der Slave gesendet wird. Die Bearbeitungszeit des Slaves ist $t_{Bearbeitung} = 1,83\ \mu s$, die Antwortzeit von $t_{Antwort} = 6,14\ ms$ entspricht der erwarteten Anzahl von $n_{Pakete/Array} = 257$ Paketen.

$$n_{Pakete} = \frac{t_{Antwort}}{t_{Paket,\ 460800\ Hz,\ 8E1}} = \frac{6,14\ ms}{23,872\ \mu s} = 257,2 \rightarrow 257 Pakete$$

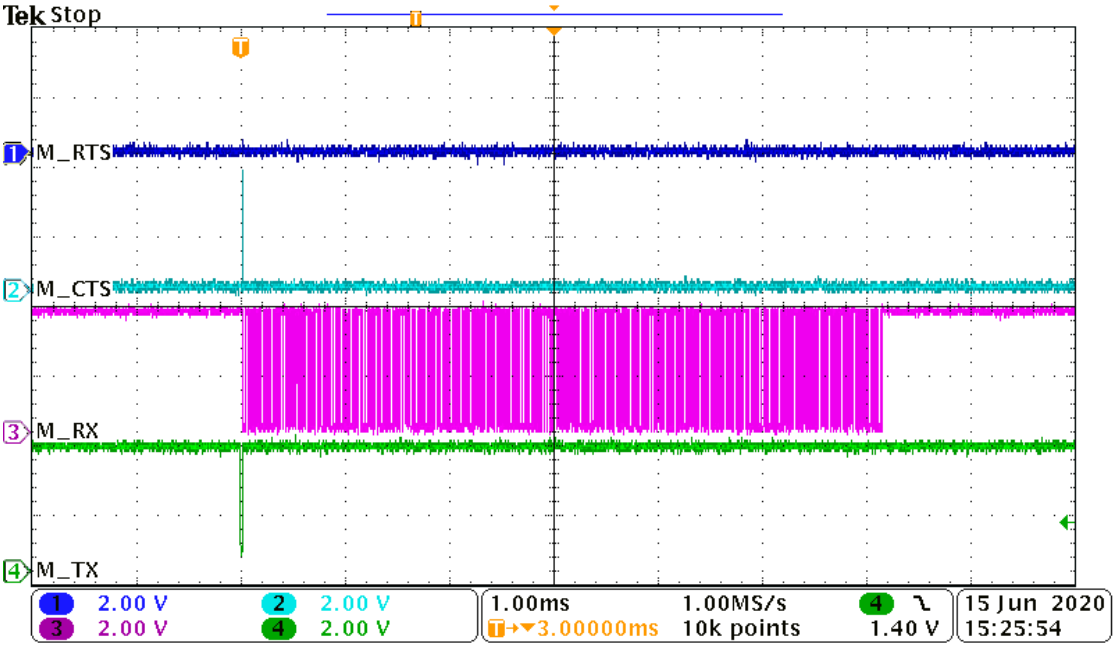


Abbildung 6.3: Aufzeichnung des Oszilloskops für den Befehl zum Senden der Sensor-Array-Daten. Das RTS-Signal des Masters behält im aufgezeichneten Bereich den low-Pegel. Sendet der Master mit dem TxD-Signal die Anfrage an den Slave, steigt das CTS-Signal des Masters durch den Slave auf den high-Pegel, bis dieser das Paket aus seinem Empfangspuffer ausgelesen hat. Die Bearbeitung des Befehls durch den Slave beginnt mit dem Ende des Pakets des Masters. Die Bearbeitungszeit dauert bis zur Antwort des Slaves, welche aus dem wiederholten Befehl des Masters und den Sensor-Array-Daten besteht. Die Antwortzeit beginnt mit dem ersten gesendeten Paket des Slaves und endet nach dem letzten.

Anfrage zum Senden einer Zeile der Sensor-Array-Daten

Abbildung 6.4 zeigt die Aufzeichnung des Oszilloskops der gesamten Kommunikation zwischen Master und Slave, wenn der Befehl zum Senden einer Zeile der Sensor-Array-Daten vom Master an der Slave gesendet wird. Die Bearbeitungszeit des Slaves ist $t_{Bearbeitung} = 1,78\ \mu s$, die Antwortzeit von $t_{Antwort} = 786\ \mu s$ entspricht der erwarteten Anzahl von $n_{Pakete/Zeile} = 33$ Paketen.

$$n_{Pakete} = \frac{t_{Antwort}}{t_{Paket,\ 460800\ Hz,\ 8E1}} = \frac{786\ \mu s}{23,872\ \mu s} = 32,9 \rightarrow 33 Pakete$$

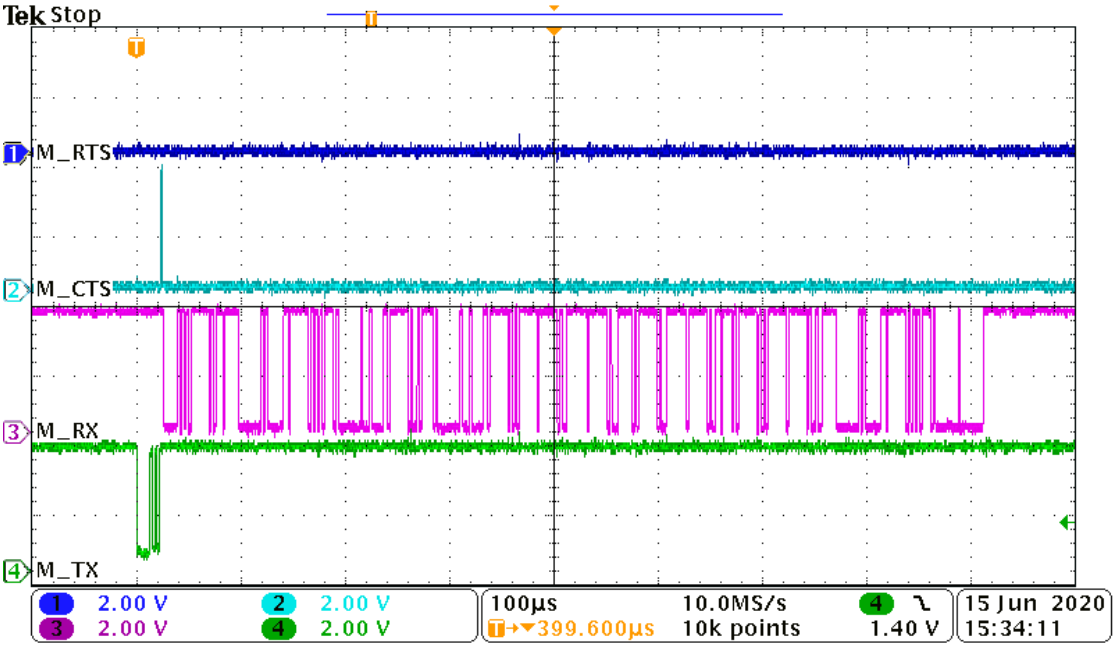


Abbildung 6.4: Aufzeichnung des Oszilloskops für den Befehl zum Senden einer Zeile der Sensor-Array-Daten. Das RTS-Signal des Masters behält im aufgezeichneten Bereich den low-Pegel. Sendet der Master mit dem TxD-Signal die Anfrage an den Slave, steigt das CTS-Signal des Masters durch den Slave auf den high-Pegel, bis dieser das Paket aus seinem Empfangspuffer ausgelesen hat. Die Bearbeitung des Befehls durch den Slave beginnt mit dem Ende des Pakets des Masters. Die Bearbeitungszeit dauert bis zur Antwort des Slaves, welche aus dem wiederholten Befehl des Masters und einer Zeile der Sensor-Array-Daten besteht. Die Antwortzeit beginnt mit dem ersten gesendeten Paket des Slaves und endet nach dem letzten.

6.2.2 Messung des Mikrocomputers

Zur Zeitmessung mit dem Mikrocomputer wird in den Quelltext der Hauptfunktion mit Hilfe der Funktion *clock()* der Bibliothek *time.h* des Betriebssystems die Zeit der in Tabelle 6.6 aufgelisteten Funktionen gemessen. Je Funktion werden die ermittelten Zeitwerte für die Auswertung in einer Datei gespeichert. Die Messung wird je Funktion über einen Zeitintervall von etwa zwei Minuten durchgeführt. Zur Auswertung wird der Mittelwert der Messwerte gebildet. Messwerte für Funktionen, die in ihrer Zeit abhängig von der Übertragungsrate sind, werden durch Subtraktion der Zeit korrigiert, die für Übertragung der Pakete über den seriellen Bus benötigt wird. Neben dem Messprogramm führt

der Mikrocomputer während jeder Messung ein Konsolenfenster und das Programm *Geany* zur Bearbeitung des Quelltextes aus.

Aktion des Mikrocomputers	**Benötigte Zeit [μs]**
Hauptzyklus ohne Kommunikationsfunktionen mit interpolierter Vektorfelddarstellung der Sensor-Array-Daten und ohne Signalverarbeitung	$t_{Benutzeroberfläche\ aktualisieren} = 23,1$
Hauptzyklus mit interpolierter Vektorfelddarstellung der Sensor-Array-Daten und ohne Signalverarbeitung	$t_{Zyklus} = 20286,1$
Array-Daten aus Empfangspuffer auslesen	$t_{Array-Daten\ auslesen} = 4755,1$
Zeit zwischen senden der Anfrage *Array-Daten aufnehmen und senden* und Auslesen der Antwort aus dem Empfangspuffer ohne Übertragungszeit	$t_{Array-Daten\ aufnehmen\ \&\ senden} = 15500,1$

Tabelle 6.6: Zeitmessung des Mikrocomputers für die Kommunikation mit dem Slave

Die Zeit zwischen Senden der Anfrage *Array-Daten aufnehmen und senden* und Auslesen der Antwort aus dem Empfangspuffer ohne Übertragungszeit macht einen Großteil der Hauptzykluszeit aus. Die Bearbeitungszeit des Slaves für diesen Befehl ist der Messung aus dem Kapitel 6.2.1 nach 606 μs. Der Mikrocomputer benötigt die übrigen $14894,1\ \mu s$. Mit der vereinfachten Gleichung 2.5 aus dem Kapitel 2.5 ergibt sich die maximale Signalverarbeitungszeit in Gleichung 6.1.

$$\begin{aligned} t_{Signalverarbeitung} &<= \frac{1}{f_{Zyklus,soll}} - t_{Zyklus,ist} \\ &<= \frac{1}{30\ Hz} - 20286,1\ \mu s \\ &<= 13047,2\ \mu s \end{aligned} \tag{6.1}$$

6.3 Benutzeroberfläche

Wird das Programm per Eingabe ***./ISAR_Viewer*** in die im Oberordner geöffnete Konsole gestartet, öffnet sich die Benutzeroberfläche in einem Fenster. Die Standardansicht ist das Vektorfeld, das die Rohdaten des Sensor-Arrays darstellt. Wie die Abbildung 6.5

zeigt, ist das Bedienfeld rechts neben der Animationsfläche schwarz und zeigt kein oder nur das erste Bedienelement. Wird mit dem Mauszeiger über das Bedienfeld gefahren, zeigen sich die Bedienelemente und bleiben sichtbar. Durch Schieben des Fensters mit dem Bedienfeld über den Bildschirmrand hinaus und wieder zurück, werden alle Bedienelemente, Texte und der Hintergrund sichtbar. Abbildung 6.6 zeigt die richtig dargestellte Benutzeroberfläche nach Programmstart.

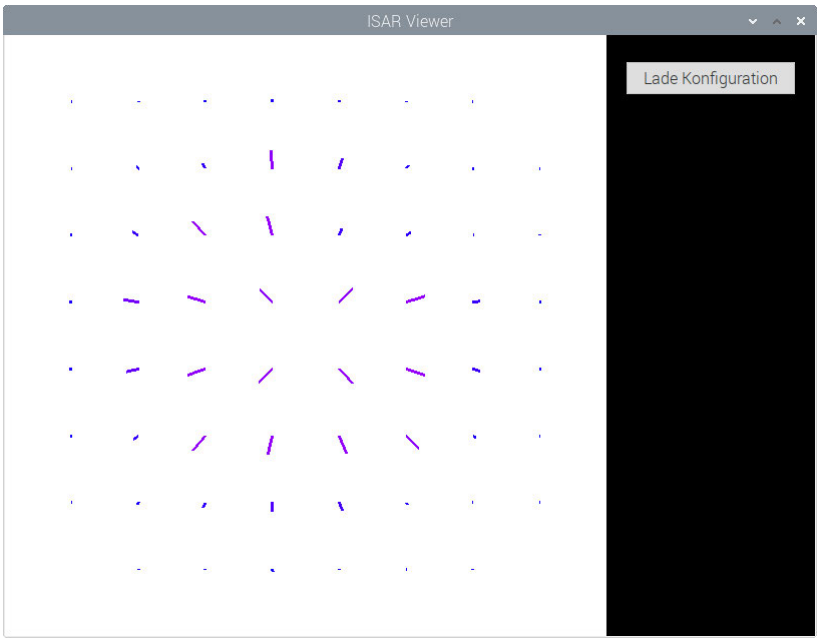


Abbildung 6.5: Benutzeroberfläche nach dem Programmstart. Das Fenster zeigt die Animationsfläche links und das fehlerhaft in schwarz dargestellte Bedienfeld mit nur einem sichtbaren Bedienelement.

Das Auswahlmenü der Ansichtenwahl zeigt keinen Eintrag und der Haken zur Interpolation ist nicht gesetzt. Der Haken kann jederzeit gesetzt und rückgesetzt werden. Die jeweilige Animation auf der Animationsfläche passt sich entsprechend an. Wird über das Auswahlmenü eine andere Ansicht gewählt, wird die Animation anhand der Wahl geändert und fortan im Menüfeld angezeigt. Die Animation auf der Animationsfläche wirkt flüssig und direkt. Alle verfügbaren Ansichten sind in den Abbildungen 6.7, 6.8, 6.9 und 6.10 dargestellt. Durch Ändern und Speichern der Textdatei für die Konfigurationsdaten und Betätigen der Schaltfläche *Lade Konfiguration* wird zum Beispiel die Größe der Animationsfläche entsprechend den Einstellungen angepasst. Werden die Einstellungen für die serielle Schnittstelle geändert und geladen, bricht das Programm ab, wenn die neue Schnittstelle nicht verfügbar ist. Wird die Größe des Sensor-Arrays geändert, sodass es

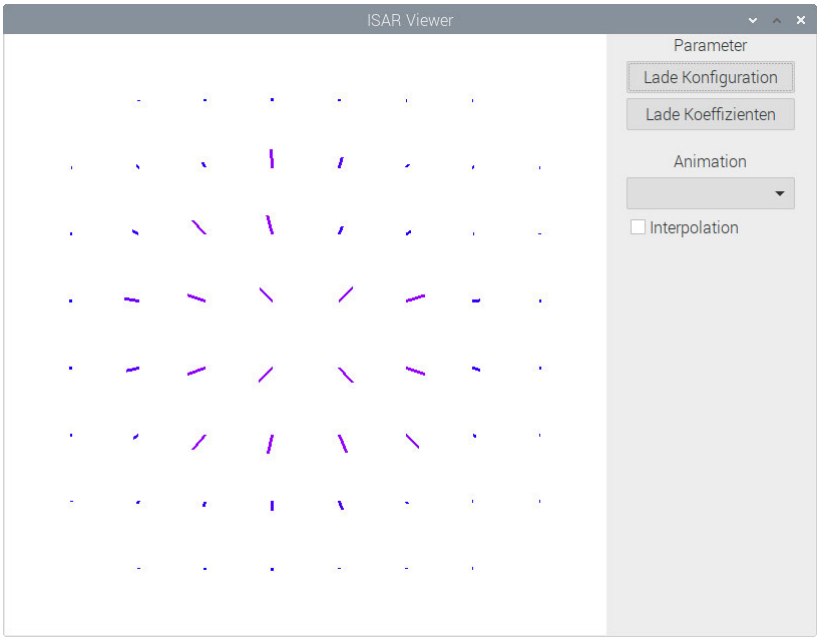


Abbildung 6.6: Startansicht der Benutzeroberfläche. Das Fenster zeigt die Animationsfläche links und das Bedienfeld rechts.

nicht mehr mit der tatsächlichen übereinstimmt, bricht das Programm ebenfalls nach kurzer Zeit ab, da die vom Mikrocomputer erwartete Anzahl an gesendeten Datenpaketen nicht mehr mit der des Mikrocontrollers übereinstimmt.

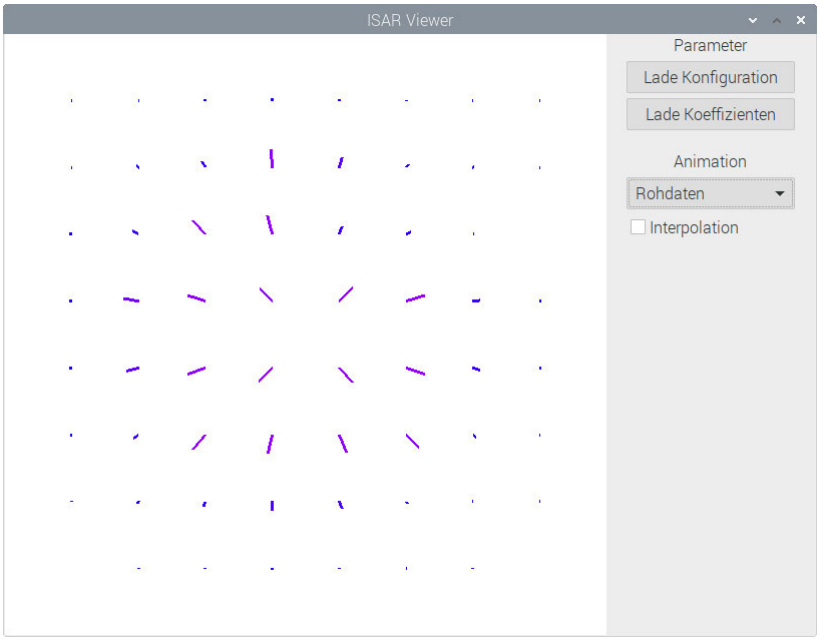


Abbildung 6.7: Vektorfeldansicht der Benutzeroberfläche. Die Vektorfelddarstellung zeigt hier die Rohdaten des Sensor-Arrays ohne Interpolation.

Abbildung 6.8: Farbmatrixansicht der Benutzeroberfläche für die Rohdaten in x-Richtung des Sensor-Arrays

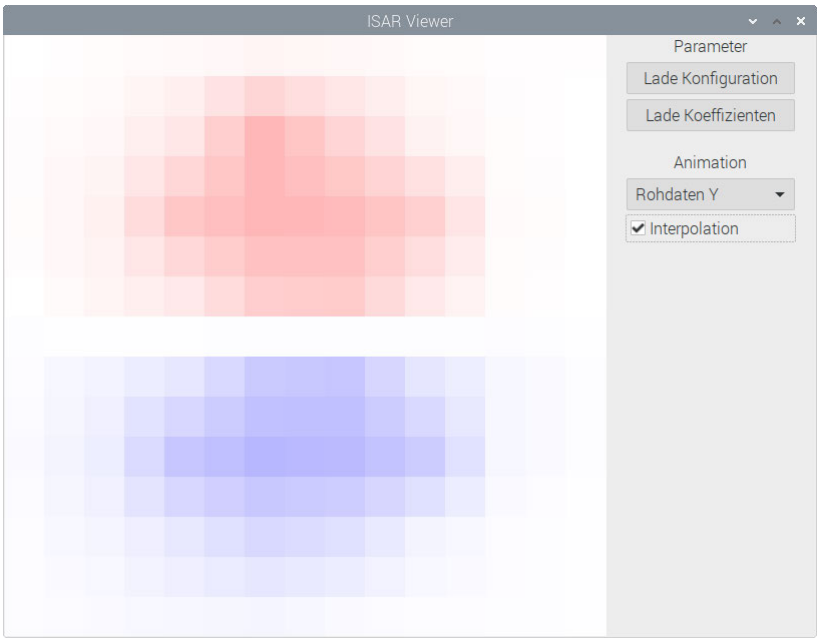


Abbildung 6.9: Farbmatrixansicht der Benutzeroberfläche für die Rohdaten in x-Richtung des Sensor-Arrays

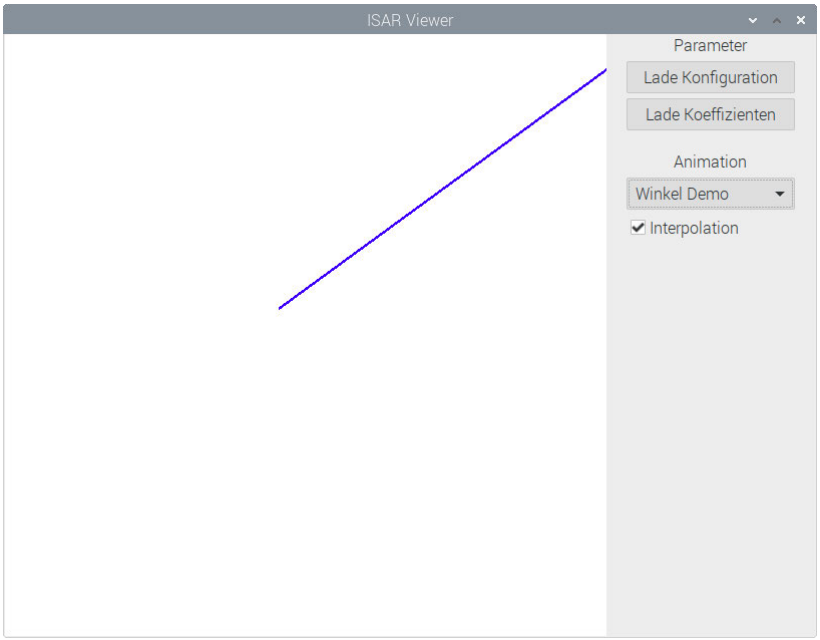


Abbildung 6.10: Vektoransicht der Benutzeroberfläche. Der angezeigte Vektor wird zur Demonstration mit künstlichen Daten versorgt. Mit dieser Darstellung lässt sich zum Beispiel eine rotierende Achse mit einem Winkel darstellen, die einen wandernden Versatz zum Mittelpunkt des Sensor-Arrays hat.

6.4 Festgestellte Mängel und Fehlerquellen

Abgesehen von den zuvor beschriebenen Grafikfehlern bei der Darstellung der grafischen Benutzeroberfläche, kann es nach dem Systemstart des Mikrocomputers sein, dass der USB-zu-UART-Adapter nicht ordnungsgemäß funktioniert. Wird der Adapter kurzzeitig aus dem USB-Sockel entfernt und neu angesteckt, ist der Fehler behoben.

Es kann außerdem zu einem Systemabsturz kommen, wenn der Micro-A-USB-Stecker des versorgenden Netzteils nicht fest in der Micro-A-USB-Buchse des Mikrocomputers steckt. Der Anschluss bietet nur dann eine zuverlässige Versorgung, wenn die mechanische Verbindung kein Spiel aufweist.

7 Zusammenfassung und Ausblick

Das Ziel dieser Arbeit war es, mit Hilfe des handbetriebenen Demonstrationsaufbaus die Daten des Sensor-Arrays darzustellen und die Signalverarbeitung zu ermöglichen. Zu diesem Zweck wurde ein Mikrocontroller programmiert, der die Sensorwerte des Arrays aufnimmt. Über eine serielle Schnittstelle als Slave agierend, kann der Mikrocontroller Befehle annehmen, die die Aufnahme und das Senden der Daten durch den Mikrocontroller steuern. Um die Daten des Arrays verarbeiten und grafisch visualisieren zu können, erfüllt der Mikrocomputer die Aufgabe des Masters, der den Slave über die serielle Schnittstelle steuert. Mittels einer grafischen Benutzeroberfläche, die auf dem Mikrocomputer ausgeführt wird, können Daten als Vektorfeld, in einer Farbmatrix oder als Winkel mit Versatz zum Mittelpunkt dargestellt werden. Welche Daten in welcher Form verarbeitet und grafisch wiedergegeben werden, ist im C-Quelltext des Mikrocomputers einfach anzupassen und zu erweitern. Der aktuelle Stand des Programms bietet die Darstellung der Rohdaten des Sensor-Arrays sowohl in Vektorfeld- als auch in Farbmatrixdarstellung und eine Demonstration der Vektoranzeige mit Versatz als Vorlage für zukünftige Erweiterungen.

Die im Kapitel 6 festgestellten Probleme sind sehr einfach zu umgehen und beeinflussen die Nutzung des Demonstrators nicht.

Mögliche Verbesserungen sind zum einen die Implementierung von Shadern für die OpenGL, die mit der Hardware des Grafikprozessors des Mikrocomputers kompatibel sind und so auf diesem ausgeführt werden können, um den Zentralprozessor zu entlasten. Zum anderen ist insbesondere die Nutzbarmachung der im Kapitel 6.2.2 ermittelten Zeit zwischen dem Senden eines Befehls durch den Master und dessen Reaktion auf den Empfang der Bestätigung des Slaves von großem Vorteil. Der Mikrocomputer ist in dieser Zeit untätig, die aktuell den größten Teil der Zykluszeit einnimmt und für die Signalverarbeitung genutzt werden könnte.

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungen

ADC Analog-to-Digital-Converter.

ASCII American Standard Code for Information-Interchange.

CTS Clear To Send.

DPS Data/Parity/Stop.

FIFO First In First Out.

GCC GNU Compiler-Collection.

GPIO General-Purpose Input / Output.

GTK GNU Image-Manipulation-Program-Tool-Kit.

GUI Graphical User Interface.

HAW Hochschule für Angewandte Wissenschaften.

HDMI High-Definition Multimedia Interface.

IC Integrated Circuit.

ISR Interrupt-Service-Routine.

LSB Least Significant Bit.

OpenGL Open Graphics Library.

RAM Random Access Memory.

RTS Ready To Send.

RxD Receive-Data.

SS Sample-Sequence.

TMR Tunnel Magnetoresistance.

TxD Transmit-Data.

UART Universal Asynchronous Receive-Transmit.

USB Universal Serial Bus.

VAO Vertex-Array-Object.

VBO Vertex-Buffer-Object.

Symbolverzeichnis

$\frac{b}{s}$ $\frac{Bit}{Sekunde}$ - Einheit der Übertragungsrate in Bit pro Sekunde.

Hz Hertz - Einheit der Frequenz als Kehrwert der Zeit in Sekunden.

m Meter - Einheit der Distanz.

s Sekunde - Einheit der Zeit.

V Volt - Einheit der elektrischen Spannung.

Literaturverzeichnis

[1] Begic, Abraham: *Tunnel-Magnetoresistives Sensor-Array - Controllersteuerung, Platinen-Layout und Prüfstands-Erprobung.* Bachelorarbeit an der Hochschule für Angewandte Wissenschaften Hamburg, 2018. – URL https://edoc.sub.uni-hamburg.de/haw/volltexte/2018/4381/

[2] Clasen, Matthias: *Quelltext für die Generierung und Initialisierung der Shader der OpenGL.* – URL https://gitlab.gnome.org/GNOME/gtk/blob/master/demos/gtk-demo/glarea.c. – Zugriffsdatum: 2020-06-19

[3] CodeNerd (Profilname; echter Name unbekannt): *Anleitung für GTK 3.* – URL https://prognotes.net/2016/03/gtk-3-c-code-hello-world-tutorial-using-glade-3/. – Zugriffsdatum: 2020-06-18

[4] Dreyer, Jennifer; Jörck, Julie; Rahe, Mareike: *Demonstrator für ein magnetisches Sensor-Array.* Bachelorprojekt an der Hochschule für Angewandte Wissenschaften Hamburg, 2019. – URL nichtverfügbar

[5] Hollingworth, Matthew: *GitHub Projekt rpirtsrtc - Programm zum Ein- und Ausschalten der Hardware-Flusskontroll-Pins von Raspberry Pi Mikrocomputern.* – URL https://github.com/mholling/rpirtscts. – Zugriffsdatum: 2020-06-06

[6] Mehm, Thorbjörn: *Schaltungsentwurf und Mikrocontrollersteuerung für ein Tunnel-Magnetoresistives Sensor-Array.* Bachelorarbeit an der Hochschule für Angewandte Wissenschaften Hamburg, 2019. – URL https://edoc.sub.uni-hamburg.de/haw/volltexte/2019/4906/

[7] Prolific Technology Inc.: *Datenblatt für Prolific PL-2303HX USB-zu-Seriell-Adapter.* – URL http://www.prolific.com.tw/UserFiles/files/ds_pl2303HXD_v1_4_4.pdf. – Zugriffsdatum: 2020-06-11

[8] Raspberry Pi Foundation: *Internetseite des Raspberry Pi 3 Model B Mikrocomputers*. – URL https://www.raspberrypi.org/products/raspberry-pi-3-model-b/. – Zugriffsdatum: 2020-06-01

[9] Raspberry PI Foundation: *Anleitung zur Installation des Betriebssystems Raspbian*. 2020. – URL https://www.raspberrypi.org/documentation/installation/installing-images/README.md. – Zugriffsdatum: 2020-05-26

[10] Raspberry PI Foundation: *Internetseite des Betriebssystems Raspbian Buster with desktop*. 2020. – URL https://www.raspberrypi.org/downloads/raspbian/. – Zugriffsdatum: 2020-05-26

[11] Rindelaub, Simon: *Signalverarbeitung für magnetoresistive Sensor-Arrays mit Controller und Einplatinen-Computer*. Bachelorarbeit an der Hochschule für Angewandte Wissenschaften Hamburg, 2018. – URL https://edoc.sub.uni-hamburg.de/haw/volltexte/2019/4516/

[12] Silicon Laboratories, Inc.: *Datenblatt für Silab CP2102 USB-zu-Seriell-Adapter*. – URL https://www.silabs.com/documents/public/data-sheets/CP2102-9.pdf. – Zugriffsdatum: 2020-06-11

[13] Texas Instruments Incorporated: *Internetseite der TM4C1294 Connected LaunchPad™ Entwicklungsplatine*. – URL https://www.ti.com/tool/EK-TM4C1294XL. – Zugriffsdatum: 2020-06-01

[14] Texas Instruments Incorporated: *Programmieranleitung für die TivaWare™ Peripheral Driver Library*. – URL http://software-dl.ti.com/tiva-c/SW-TM4C/latest/exports/SW-TM4C-DRL-UG-2.1.4.178.pdf. – Zugriffsdatum: 2020-06-01

[15] Texas Instruments Incorporated: *Internetseite der Entwicklungsumgebung Code Composer Studio 9.3 für den Mikrocontroller*. 2020. – URL http://software-dl.ti.com/ccs/esd/documents/ccs_downloads.html. – Zugriffsdatum: 2020-05-26

[16] Texas Instruments Incorporated: *Internetseite der TivaWare for C Series Peripheral Treiberbibliothek Version 2.1.4.178 für den Mikrocontroller*. 2020. – URL http://software-dl.ti.com/tiva-c/SW-TM4C/latest/index_FDS.html. – Zugriffsdatum: 2020-05-26

[17] Texas Instruments Incorporated: *Internetseite des Entwicklungspakets der Version 2.1.1.71 für den Mikrocontroller*. 2020. – URL http://software-dl.ti.com/tiva-c/SW-DK-TM4C129X/latest/index_FDS.html. – Zugriffsdatum: 2020-05-26

[18] Schüthe T., Petrak O., Jünemann K., Riemschneider KR.: *"Positionserfassung mittels Sensor-Array aus Tunnel-Magnetoresistiven Vortex-Dots und lernender Signalverarbeitung", Tille T. (eds) Automobil-Sensorik 3*. Springer Vieweg, Berlin, Heidelberg, 2020. – URL https://doi.org/10.1007/978-3-662-61260-6_14. – ISBN 978-3-662-61259-0

[19] Tran, Toan: *Quelltext für die Initialisierung der Pufferstruktur der OpenGL*. – URL https://stackoverflow.com/questions/42231698/how-to-convert-glsl-version-330-core-to-glsl-es-version-100. – Zugriffsdatum: 2020-06-19

[20] Wikipedia: *Wikipedia-Eintrag zu OpenGL*. – URL https://de.wikipedia.org/w/index.php?title=OpenGL&oldid=200220758. – Zugriffsdatum: 2020-06-05

A Quelltexte des Mikrocontrollers

Der Anhang zur Arbeit befindet sich auf CD und kann beim Erstgutachter eingesehen werden.

main.c

```
/*-------------------------------------------------------------------
 * TMR-Sensor-Array Controller
 *  When extending this code be mindful of already used
 *  hardware. Refer to the header of the setup functions to
 *  avoid overwriting hardware configurations.
 *                    Jacob Ernsting 06/2020
 *-------------------------------------------------------------------
 */

#include <stdbool.h>
#include <stdint.h>
#include "tm4c1294ncpdt.h"
#include "driverlib/sysctl.h"
#include "driverlib/pin_map.h"
#include "driverlib/gpio.h"
#include "driverlib/adc.h"
#include "driverlib/uart.h"
#include "driverlib/interrupt.h"
#include "inc/hw_memmap.h"



// Number of Array Sensors

#define ROWS (8)
//^ in sinus-/y-direction
```

```
#define COLUMNS (8)
//^ in cosinus-/x-direction


// UART Package Definition Macros

#define UART_INSTRUCTION_MASK (7 << 5)
//^ upper 3 Bits of a Byte

#define UART_ARGUMENT_MASK (31 << 0)
//^ lower 5 Bits of a Byte

#define UART_INSTRUCTION_REQUEST_TAKE_SAMPLE (1 << 5)
//^ instruction code for a request to sample the array

#define UART_INSTRUCTION_REQUEST_TAKE_SAMPLE_SEND_ARRAY_DATA \
(2 << 5)
//^ instruction code for a request to sample the array and then
//^ send all data to the master

#define UART_INSTRUCTION_REQUEST_SEND_ARRAY_DATA (4 << 5)
//^ instruction code for a request to send all data to the
//^ master

#define UART_INSTRUCTION_REQUEST_SEND_ROW_DATA (5 << 5)
//^ instruction code for a request to send all data of the
//^ specified row to the master

#define MUX_SETTLE_TIME (0.000001)
//^ required delay time in Seconds for the analog MUX's output
//^ signal to settle



// Global Variables (marked with g_*)
const uint16_t g_rawDataSize[]    {ROWS, COLUMNS * 4};
//^ stores the size of the rawData array

const uint16_t g_diffDataSize[]    {ROWS, COLUMNS * 2};
```

```
//^ stores the size of the diffData array

uint16_t g_rawData[ROWS][COLUMNS * 4];
//^ stores the sensor signals in the form:
//^ [rows][0_sin+, 0_sin-, 0_cos+, 0_cos-, 1_sin+, 1_sin-,
//^ 1_cos+, 1_cos-, ... , 7_sin+, 7_sin-, 7_cos+, 7_cos-]
//^ (where 0 is closest to the 0 axes, i.e. bottom left)

int16_t g_diffData[ROWS][COLUMNS * 2];
//^ stores the difference of the signals per sensor in the form:
//^ [rows][(0_sin+ - 0_sin-), (0_cos+ - 0_cos-), (1_sin+ -
//^ 1_sin-), (1_cos+ - 1_cos-), ..., (7_sin+ - 7_sin-),
//^ (7_cos+ - 7_cos-)]
//^ (where 0 is closest to the 0 axes, i.e. bottom left)

const uint8_t g_lut0[]
{14, 15, 10, 11, 6, 7, 2, 3, 1, 0, 5, 4, 9, 8, 13, 12};
//^ Look up table for muxAddress-to-g_rawData-array association
//^ for SS0 of ADC0 this ensures, that the entries of the SS are
//^ written to the correct entries of the g_rawData array

const uint8_t g_lut1[]
{17, 16, 21, 20, 25, 24, 29, 28, 30, 31, 26, 27, 22, 23, 18, 19
};
//^ Look up table for muxAddress-to-g_rawData-array association
//^ for SS0 of ADC1 this ensures, that the entries of the SS are
//^ written to the correct entries of the g_rawData array

uint32_t g_sysClk;
//^ System clock frequency

uint32_t g_analogMUXdelay;
//^ third of required delay clock cycles for the analog MUX's
//^ output signal to settle

bool g_adc0done, g_adc1done, g_uart0messageReceived;
//^ interrupt flags for program control
```

```
/*
 * Interrupt Handler for SS0 of ADC0
 *  - This clears the interrupt of SS0 of ADC0 and sets the
 *  g_adc0done flag
 *
 * Writes: g_adc0done
 */
void adc0_ss0_interrupt_handler()
{
  ADCIntClear(ADC0_BASE, 0);
  //^ clear interrupt flag of SS0 of ADC0
  //^ "Because there is a write buffer in the Cortex-M
  //^ processor, it may take several clock cycles before the
  //^ interrupt source is actually cleared. Therefore, it is
  //^ recommended that the interrupt source be cleared early
  //^ in the interrupt handler (as opposed to the very last
  //^ action) to avoid returning from the interrupt handler
  //^ before the interrupt source is actually cleared. Failure
  //^ to do so may result in the interrupt handler being
  //^ immediately reentered (because the interrupt controller
  //^ still sees the interrupt source asserted)."
  //^ Documentation of the Texas Instruments "TivaWare for C
  //^ Series Peripheral Driver Library"
  g_adc0done   true; // set program control flag
}



/*
 * Interrupt Handler for SS0 of ADC1
 *  - This clears the interrupt of SS0 of ADC1 and sets the
 *  g_adc1done flag
 *
 * Writes: g_adc1done
 */
void adc1_ss0_interrupt_handler()
{
  ADCIntClear(ADC1_BASE, 0);
  //^ clear interrupt flag of SS0 of ADC1
  //^ "Because there is a write buffer in the Cortex-M
```

```
  //^ processor, it may take several clock cycles before the
  //^ interrupt source is actually cleared. Therefore, it is
  //^ recommended that the interrupt source be cleared early
  //^ in the interrupt handler (as opposed to the very last
  //^ action) to avoid returning from the interrupt handler
  //^ before the interrupt source is actually cleared. Failure
  //^ to do so may result in the interrupt handler being
  //^ immediately reentered (because the interrupt controller
  //^ still sees the interrupt source asserted)."
  //^ Documentation of the Texas Instruments "TivaWare for C
  //^ Series Peripheral Driver Library"
  g_adc1done    true; // set program control flag
}



/*
 * Interrupt Handler for UART0
 *  - This clears all pending interrupts of UART0 and sets the
 *  g_uart0messageReceived flag
 *
 * Uses: -
 * Reads: -
 * Writes: g_uart0messageReceived
 */
void uart0_interrupt_handler()
{
  static uint32_t ui32Status;
  ui32Status    UARTIntStatus(UART0_BASE, true);
  //^ get pending interrupt flags
  UARTIntClear(UART0_BASE, ui32Status);
  //^ clear interrupt flag of SS0 of ADC0
  //^ "Because there is a write buffer in the Cortex-M
  //^ processor, it may take several clock cycles before the
  //^ interrupt source is actually cleared. Therefore, it is
  //^ recommended that the interrupt source be cleared early
  //^ in the interrupt handler (as opposed to the very last
  //^ action) to avoid returning from the interrupt handler
  //^ before the interrupt source is actually cleared. Failure
  //^ to do so may result in the interrupt handler being
```

```
  //^ immediately reentered (because the interrupt controller
  //^ still sees the interrupt source asserted)."
  //^ Documentation of the Texas Instruments "TivaWare for C
  //^ Series Peripheral Driver Library"

  if(ui32Status & UART_MIS_RXMIS)
  {
    // interrupt flag for received package was set
    g_uart0messageReceived    true;
  //^ set program control flag
  }
}



/**
 * System-Initialization
 *  - this sets up the system clock
 *  - this initializes some global variables
 *
 * Uses: -
 * Reads: -
 * Writes: g_sysClk
 */
void setup_system()
{
  // setup system clock source and frequency
  g_sysClk    SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ |
  SYSCTL_OSC_MAIN | SYSCTL_USE_PLL | SYSCTL_CFG_VCO_480),
  120000000);
  //^ external crystal oscillator as main clock PLL source and
  //^  120 MHz actual system clock
}



/**
 * MUX Address Bus GPIO-Port-Initialization (P.753)
 *  - this sets up some GPIO:
 *  - PL3 (84)   MUX-Address-Line ~3 (output)
```

```
 *  - PM0 (78)   MUX-Address-Line 0 (output)
 *  - PM1 (77)   MUX-Address-Line 1 (output)
 *  - PM2 (76)   MUX-Address-Line 2 (output)
 *  - PM3 (75)   MUX-Address-Line 3 (output)
 *
 * Uses: -
 * Reads: g_sysClk
 * Writes: g_analogMUXdelay
 */
void setup_mux_address_bus()
{
  // Enable GPIO Ports L, M
  SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOL);
  SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOM);

  // Calculate Delay Calls
  g_analogMUXdelay    (uint32_t)(g_sysClk * MUX_SETTLE_TIME / 3);
  //^ system_clock[Hz] * delay_time[sec] / 3;
  //^ the devision by three is necessary because SysCtlDelay()
  //^ requires three cycles per one as argument

  // Wait for the GPIO Modules L, M to be ready
  while(!SysCtlPeripheralReady(SYSCTL_PERIPH_GPIOL)
     && !SysCtlPeripheralReady(SYSCTL_PERIPH_GPIOM));

  // Configure GPIO Port L, M
  GPIOPinTypeGPIOOutput(GPIO_PORTL_BASE, GPIO_PIN_3);
  GPIOPinTypeGPIOOutput(GPIO_PORTM_BASE, GPIO_PIN_3 | GPIO_PIN_2
  | GPIO_PIN_1 | GPIO_PIN_0);
}



/*
 * ADC0- & ADC1-Initialization (P.1072)
 *  - this sets up the ADC modules 0, 1 and their respective SS0
 *    with interrupt
 *  - this sets up some GPIO:
 *  - PD0 (1)   AIN15 (analog input)
 *  - PD1 (2)   AIN14 (analog input)
```

```
 *  - PD2 (3)    AIN13 (analog input)
 *  - PD3 (4)    AIN12 (analog input)
 *  - PD4 (125)   AIN7 (analog input)
 *  - PD5 (126)   AIN6 (analog input)
 *  - PD6 (127)   AIN5 (analog input)
 *  - PD7 (128)   AIN4 (analog input)
 *  - PE2 (13)   AIN1 (analog input)
 *  - PE3 (12)   AIN0 (analog input)
 *  - PE4 (123)   AIN9 (analog input)
 *  - PE5 (124)   AIN8 (analog input)
 *  - PK0 (18)   AIN16 (analog input)
 *  - PK1 (19)   AIN17 (analog input)
 *  - PK2 (20)   AIN18 (analog input)
 *  - PK3 (21)   AIN19 (analog input)
 *
 * Uses: adc0_ss0_interrupt_handler(),
 *       adc1_ss0_interrupt_handler()
 * Reads: -
 * Writes: -
 */
void setup_adc(void)
{
  // Enable ADC Modules 0, 1 and GPIO Ports D, E, K
  SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
  SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC1);
  SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
  SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);
  SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOK);

  // Wait for the ADC Modules 0, 1 to be ready
  while(!SysCtlPeripheralReady(SYSCTL_PERIPH_ADC0));
  while(!SysCtlPeripheralReady(SYSCTL_PERIPH_ADC1));

  // set external voltage reference (at pin 9 (X11:34)   VREFA+)
  ADCReferenceSet(ADC0_BASE, ADC_REF_EXT_3V);
  ADCReferenceSet(ADC1_BASE, ADC_REF_EXT_3V);

  // Configure sample sequences
  ADCSequenceConfigure(ADC0_BASE, 0, ADC_TRIGGER_PROCESSOR, 0);
  //^ ADC, SS, trigger, priority
```

```
ADCSequenceConfigure(ADC1_BASE, 0, ADC_TRIGGER_PROCESSOR, 0);

// row numbering starting from bottom of matrix according to
// y-axis-arrow
ADCSequenceStepConfigure(ADC0_BASE, 0, 0, ADC_CTL_CH1);
//^ AIN1 (PE2) (row 7 / IC15 / 18)
ADCSequenceStepConfigure(ADC0_BASE, 0, 1, ADC_CTL_CH0);
//^ AIN0 (PE3) (row 6 / IC13 / 20)
ADCSequenceStepConfigure(ADC0_BASE, 0, 2, ADC_CTL_CH9);
//^ AIN9 (PE4) (row 5 / IC11 / 22)
ADCSequenceStepConfigure(ADC0_BASE, 0, 3, ADC_CTL_CH8);
//^ AIN8 (PE5) (row 4 / IC9 / 24)
ADCSequenceStepConfigure(ADC0_BASE, 0, 4, ADC_CTL_CH16);
//^ AIN16 (PK0) (row 3 / IC7 / 26)
ADCSequenceStepConfigure(ADC0_BASE, 0, 5, ADC_CTL_CH17);
//^ AIN17 (PK1) (row 2 / IC5 / 28)
ADCSequenceStepConfigure(ADC0_BASE, 0, 6, ADC_CTL_CH18);
//^ AIN18 (PK2) (row 1 / IC3 / 30)
ADCSequenceStepConfigure(ADC0_BASE, 0, 7, ADC_CTL_CH19 |
ADC_CTL_IE | ADC_CTL_END);
//^ AIN19 (PK3) (row 0 / IC1 / 32)

ADCSequenceStepConfigure(ADC1_BASE, 0, 0, ADC_CTL_CH13);
//^ AIN13 (PD2) (row 7 / IC16 / 52)
ADCSequenceStepConfigure(ADC1_BASE, 0, 1, ADC_CTL_CH15);
//^ AIN15 (PD0) (row 6 / IC14 / 50)
ADCSequenceStepConfigure(ADC1_BASE, 0, 2, ADC_CTL_CH14);
//^ AIN14 (PD1) (row 5 / IC12 / 48)
ADCSequenceStepConfigure(ADC1_BASE, 0, 3, ADC_CTL_CH12);
//^ AIN12 (PD3) (row 4 / IC10 / 46)
ADCSequenceStepConfigure(ADC1_BASE, 0, 4, ADC_CTL_CH5);
//^ AIN5 (PD6) (row 3 / IC8 / 44)
ADCSequenceStepConfigure(ADC1_BASE, 0, 5, ADC_CTL_CH4);
//^ AIN4 (PD7) (row 2 / IC6 / 42)
ADCSequenceStepConfigure(ADC1_BASE, 0, 6, ADC_CTL_CH7);
//^ AIN7 (PD4) (row 1 / IC4 / 40)
ADCSequenceStepConfigure(ADC1_BASE, 0, 7, ADC_CTL_CH6 |
ADC_CTL_IE | ADC_CTL_END);
//^ AIN6 (PD5) (row 0 / IC2 / 38) (incl. interrupt)
```

```
  // Enable Interrupts
  IntEnable(INT_ADC0SS0); // enable interrupt for ADC0 SS0
  ADCIntEnable(ADC0_BASE, 0);
  ADCIntEnableEx(ADC0_BASE, ADC_INT_SS0);
  ADCIntRegister(ADC0_BASE, 0, (&adc0_ss0_interrupt_handler));
  IntEnable(INT_ADC1SS0); // enable interrupt for ADC1 SS0
  ADCIntEnable(ADC1_BASE, 0);
  ADCIntEnableEx(ADC1_BASE, ADC_INT_SS0);
  ADCIntRegister(ADC1_BASE, 0, (&adc1_ss0_interrupt_handler));

  // Enable SSs
  ADCSequenceEnable(ADC0_BASE, 0);
  ADCSequenceEnable(ADC1_BASE, 0);

  // Configure GPIO Port D
  while(!SysCtlPeripheralReady(SYSCTL_PERIPH_GPIOD));
  GPIOPinTypeADC(GPIO_PORTD_BASE, GPIO_PIN_7 | GPIO_PIN_6 |
  GPIO_PIN_5 | GPIO_PIN_4 | GPIO_PIN_3 | GPIO_PIN_2 | GPIO_PIN_1
  | GPIO_PIN_0);

  // Configure GPIO Port E
  while(!SysCtlPeripheralReady(SYSCTL_PERIPH_GPIOE));
  GPIOPinTypeADC(GPIO_PORTE_BASE, GPIO_PIN_5 | GPIO_PIN_4 |
  GPIO_PIN_3 | GPIO_PIN_2);

  // Configure GPIO Port K
  while(!SysCtlPeripheralReady(SYSCTL_PERIPH_GPIOK));
  GPIOPinTypeADC(GPIO_PORTK_BASE, GPIO_PIN_3 | GPIO_PIN_2 |
  GPIO_PIN_1 | GPIO_PIN_0);
}



/**
 * UART0-Initialization (P.1172)
 *  - this sets up UART0 with 8E1 protocol and hardware flow
 *    control using CTS and RTS generating an
 *  interrupt on reception
 *  - this sets up some GPIO:
 *  - PA0 (33)   U0Rx (input, uart module handles pull up)
```

```
 *  - PA1 (34)    U0Tx (output, push-pull)
 *  - PH0 (29)    U0RTS (output, push-pull)
 *  - PH1 (30)    U0CTS (input, weak pull-up)
 *
 * Uses: uart0_interrupt_handler()
 * Reads: -
 * Writes: -
 */
void setup_uart()
{
  // Enable GPIO Ports A, H and UART 0 Module
  SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
  SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOH);
  SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);

  // Configure GPIO Port A
  while(!SysCtlPeripheralReady(SYSCTL_PERIPH_GPIOA));
  //^ wait for peripheral ready status
  GPIOPinConfigure(GPIO_PA0_U0RX);
  //^ enable alternate pin function (PCTL)
  GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0);
  //^ configure alternate pin function (dir, afsel, o_pc, drive
  //^ strength & slew rate, odr/pur/pdr/den, wake_lvl, wake_pen,
  //^ amsel)
  GPIOPinConfigure(GPIO_PA1_U0TX);
  GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_1);

  // Configure GPIO Port H
  while(!SysCtlPeripheralReady(SYSCTL_PERIPH_GPIOH));
  GPIOPinConfigure(GPIO_PH0_U0RTS);
  GPIOPinTypeUART(GPIO_PORTH_BASE, GPIO_PIN_0);
  GPIOPinConfigure(GPIO_PH1_U0CTS);
  GPIOPinTypeUART(GPIO_PORTH_BASE, GPIO_PIN_1);
  GPIOPadConfigSet(GPIO_PORTH_BASE, GPIO_PIN_1,
  GPIO_STRENGTH_2MA, GPIO_PIN_TYPE_STD_WPU);

  // Configure UART 0 Module
  while(!SysCtlPeripheralReady(SYSCTL_PERIPH_UART0));
  UARTDisable(UART0_BASE);
  //^ wait for end of TX, disable FIFO, disable UART
```

```
  UARTClockSourceSet(UART0_BASE, UART_CLOCK_SYSTEM);
  //^ set system clock as source
  UARTFlowControlSet(UART0_BASE, (UART_FLOWCONTROL_TX |
  UART_FLOWCONTROL_RX));
  //^ enable hardware flow control
  UARTConfigSetExpClk(UART0_BASE, g_sysClk, 460800,
  (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
  UART_CONFIG_PAR_EVEN));
  //^ 8E1 @ 1 Mb/sec (raw) (8E1 data efficiency is 72 percent)
  UARTFIFODisable(UART0_BASE);
  //^ disable the Tx and Rx FIFOs (UART operates with one 8 bit
  //^ buffer instead of 16)

  // Enable UART 0 interrupt
  IntEnable(INT_UART0);
  //^ enable interrupt for UART0
  UARTIntEnable(UART0_BASE, UART_INT_RX);
  //^ enable interrupt when receiving
  UARTFIFOLevelSet(UART0_BASE, UART_IFLS_TX4_8, UART_IFLS_RX1_8
  );
  //^ configure interrupt level to half on transmission
  //^ (default) and one sample on reception
  UARTIntRegister(UART0_BASE, uart0_interrupt_handler);
  //^ associate interrupt handler function
}



/*
 * Sample Sensor Matrix
 *  - this reads all sensor values from the sensor-array
 *  - this uses ADC0 and ADC1 with their respective SS0 as well
*      as the MUX address bus
 *
 * Uses: -
 * Reads: g_analogMUXdelay, g_adc0done, g_adc1done
 * Writes: g_rawData[][], g_adc0done, g_adc1done
 */
void sample_array()
{
```

```
static uint8_t muxAddress;
//^ same as xIndex
static uint8_t yIndex;
//^ 0 is closest to x-axis (bottom)
static uint32_t ui32buffer[8];
//^ buffer for 12 Bit ADC SS0 read-out
static bool adc0read, adc1read;
//^ flags

for(muxAddress    0; muxAddress < 16; muxAddress++)
{
  if(g_uart0messageReceived) break;
      //^ exit ongoing operation when a new message is received

  // set external MUX's-address
  static uint8_t pinMaskL; // Pin 3
  static uint8_t pinMaskM; // Pin 0-3

  pinMaskL    (0x08 & ((muxAddress & 0x08) << 0));
  pinMaskM    (0x0F & ((muxAddress & 0x0F) << 0));
  GPIO_PORTL_DATA_R    (pinMaskL); // set ~3
  GPIO_PORTM_DATA_R    (pinMaskM); // set 0, 1, 2, 3
  //^ the use of direct register access is deliberate and
  //^ minimizes the delay between the ports of the address
  //^ lines

  // wait for analog signal to settle after multiplexing
  SysCtlDelay(g_analogMUXdelay);
  //^ at 120 MHz with 40 cycles this takes about 1175 ns

  // start sample sequences
  ADCProcessorTrigger(ADC0_BASE, 0);
  ADCProcessorTrigger(ADC1_BASE, 0);

  // wait for sample sequences to finish, then store their
      // values
  adc0read    false;
  adc1read    false;

  while(!(adc0read && adc1read))
```

```
    {
      if(g_adc0done)
      {
        g_adc0done    false;
        adc0read    true;
        ADCSequenceDataGet(ADC0_BASE, 0, &(ui32buffer[0]));
        for(yIndex    0; yIndex < 8; yIndex++)
                  g_rawData[yIndex][g_lut0[muxAddress]]
                  (uint16_t)(ui32buffer[yIndex]);
      }
      if(g_adc1done)
      {
        g_adc1done    false;
        adc1read    true;
        ADCSequenceDataGet(ADC1_BASE, 0, &(ui32buffer[0]));
        for(yIndex    0; yIndex < 8; yIndex++)
                  g_rawData[yIndex][g_lut1[muxAddress]]
                  (uint16_t)(ui32buffer[yIndex]);
      }
    }
  }
}



/*
 * Compute Differential of Sensor Data
 *  - this computes the difference of the plus and minus signals
 *    of the sensors
 *
 * Uses: -
 * Reads: g_rawData[][], g_diffDataSize[]
 * Writes: g_diffData[][]
 */
void compute_diff()
{
  static uint8_t yIndex, diffSensorIndex, rawSensorIndex;

  for(yIndex    0; yIndex < g_diffDataSize[0]; yIndex++)
  {
```

```
    if(g_uart0messageReceived) break;
    //^ exit ongoing operation when a new message is received
    for(diffSensorIndex   0; diffSensorIndex < g_diffDataSize[1]
        - 1; diffSensorIndex   diffSensorIndex + 2)
    {
      if(g_uart0messageReceived) break;
      //^ exit ongoing operation when a new message is received
      rawSensorIndex   2 * diffSensorIndex;
      g_diffData[yIndex][diffSensorIndex]
          g_rawData[yIndex][rawSensorIndex] -
          g_rawData[yIndex][rawSensorIndex + 1];
      g_diffData[yIndex][diffSensorIndex + 1]
          g_rawData[yIndex][rawSensorIndex + 2] -
          g_rawData[yIndex][rawSensorIndex + 3];
    }

  }
}



/*
 * Send a Row of Differential Data over UART
 *  - this sends a row (x-direction) of the differential data
 *    over the UART0 interface
 *
 * Uses: -
 * Reads: g_diffData[][], g_diffDataSize[]
 * Writes: UART0
 */
void send_row(uint8_t row)
{
  static uint8_t uart0xIndex;
  if(row < g_diffDataSize[0])
  {
    for(uart0xIndex   0; uart0xIndex < g_diffDataSize[1];
        uart0xIndex++)
    {
      if(g_uart0messageReceived) break;
      //^ exit ongoing operation when a new message is received
```

```
      UARTCharPut(UART0_BASE, (unsigned char)
          (g_diffData[row][uart0xIndex] >> 8)); // upper Byte
      UARTCharPut(UART0_BASE, (unsigned char)
          (g_diffData[row][uart0xIndex] & 0xFF)); // lower Byte
    }
  }
}



/*
 * Send a All Differential Data over UART
 *  - this sends all of the differential data row-by-row
 *    (x-direction) over the UART0 interface
 *
 * Uses: -
 * Reads: g_diffData[][], g_diffDataSize[]
 * Writes: UART0
 */
void send_array()
{
  static uint8_t uart0xIndex, uart0yIndex;
  for(uart0yIndex   0; uart0yIndex < g_diffDataSize[0];
  uart0yIndex++)
  {
    if(g_uart0messageReceived) break;
    //^ exit ongoing operation when a new message is received
    for(uart0xIndex   0; uart0xIndex < g_diffDataSize[1];
        uart0xIndex++)
    {
      if(g_uart0messageReceived) break;
      //^ exit ongoing operation when a new message is received
      UARTCharPut(UART0_BASE, (unsigned char)
          (g_diffData[uart0yIndex][uart0xIndex] >> 8));
    //^ upper Byte
      UARTCharPut(UART0_BASE, (unsigned char)
          (g_diffData[uart0yIndex][uart0xIndex] & 0xFF));
    //^ lower Byte
    }
  }
```

```
}



void main()
{
  setup_system();
  setup_mux_address_bus();
  setup_adc();
  setup_uart();

  while(1)
  {
    while(!g_uart0messageReceived);
    //^ wait here until a message is received

    g_uart0messageReceived    false;
    static int32_t uart0buffer;
    uart0buffer    UARTCharGetNonBlocking(UART0_BASE);
    //^ read out UART0 receive FIFO

    switch(uart0buffer & UART_INSTRUCTION_MASK)
    {
    case UART_INSTRUCTION_REQUEST_TAKE_SAMPLE_SEND_ARRAY_DATA:
      // sample array, then send array data
      // 1. execute requested action
      sample_array();
      compute_diff();
      // 2. confirm request
      UARTCharPut(UART0_BASE, (unsigned char)
          (uart0buffer & 0xFF));
      // 3. send data
      send_array();
      break;
    case UART_INSTRUCTION_REQUEST_TAKE_SAMPLE:
      // sample array
      // 1. execute requested action
      sample_array();
      compute_diff();
      // 2. confirm request
```

```
      UARTCharPut(UART0_BASE, (unsigned char)
          (uart0buffer & 0xFF));
      // 3. no data to send
      break;
    case UART_INSTRUCTION_REQUEST_SEND_ARRAY_DATA:
      // send array data
      // 1. no action to execute
      // 2. confirm request
      UARTCharPut(UART0_BASE, (unsigned char)
          (uart0buffer & 0xFF));
      // 3. send data
      send_array();
      break;
    case UART_INSTRUCTION_REQUEST_SEND_ROW_DATA:
      // send row data
      // 1. no action to execute
      // 2. confirm request
      UARTCharPut(UART0_BASE, (unsigned char)
          (uart0buffer & 0xFF));
      // 3. send data
      send_row((unsigned int)(uart0buffer &
          UART_ARGUMENT_MASK));
      break;
    default:
      // confirm unknown received request
      UARTCharPut(UART0_BASE, (unsigned char)
          (uart0buffer & 0xFF));
      break;
    }
  }
}
```

B Quelltexte des Mikrocomputers

makefile

```
# Source: https://prognotes.net/2015/07/gtk-3-glade-c-programming-template/
EXE ISAR_Viewer

# compiler
CC gcc
# debug
DEBUG -g
# optimisation
OPT -O0
# warnings
WARN -Wall

PTHREAD -pthread

CCFLAGS $(DEBUG) $(OPT) $(WARN) $(PTHREAD) -pipe -DGL_GLEXT_PROTOTYPES

GTKLIB `pkg-config --cflags --libs gtk+-3.0 gl`

# linker
LD gcc
LDFLAGS $(PTHREAD) $(GTKLIB) -lm -lrt -lGL -export-dynamic

OBJS    main.o

all: $(OBJS)
	$(LD) -o $(EXE) $(OBJS) $(LDFLAGS)

main.o: src/main.c
	$(CC) -c $(CCFLAGS) src/main.c $(GTKLIB) -o main.o
```

```
clean:
	rm -f *.o $(EXE)
```

src/auxiliary.c

```
#include <stdio.h>
#include <time.h>

#include "global.h"


/**
 * @brief  read_configuration_file - Read Text File for
 *         Configuration Parameters
 *
 * @param  (const char *)location - File path of the
 *         configuration file to read.
 *         Pointer to character array. Argument is read.
 *
 * Uses: global.h, stdio.h
 * Reads: g_sensorArraySize
 * Writes: g_pixelsSize, g_sensorArraySize, g_serialInterface
 */
int read_configuration_file(const char *location)
{
  FILE *fp;

  fp   fopen(location, "r");

  // check file availabilty and set file position indicator to
  // beginning
  if(fp     NULL || fseek(fp, 0, SEEK_SET)     -1)
  {
    fclose(fp);
    printf("Error in read_configuration_file: Inaccessible \
    configuration file at path %s.\n", location);
    return -1;
  }

  // read serial interface path
  if(fscanf(fp, "%s\n", g_serialInterface)     EOF)
  {
    fclose(fp);
    printf("Error in read_configuration_file: Incomplete \
```

```
    configuration file. Missing serial interface device path.\n"
    );
    return -1;
  }

  // read number of sensors in x-direction
  if(fscanf(fp, "%u\n", &g_sensorArraySize[0])     EOF)
  {
    fclose(fp);
    printf("Error in read_configuration_file: Incomplete \
    configuration file. Missing number of sensors of the array \
    in x-direction.\n");
    return -1;
  }

  if(g_sensorArraySize[0] > MAX_X_NUMBER_OF_SENSORS)
  {
    printf("Warning in read_configuration_file: Requested \
    sensor array size in x direction of %u is too high and will \
    be set to MAX_X_NUMBER_OF_SENSORS.\n", g_sensorArraySize[0]);
    g_sensorArraySize[0]    MAX_X_NUMBER_OF_SENSORS;
  }

  // read number of sensors in y-direction
  if(fscanf(fp, "%u\n", &g_sensorArraySize[1])     EOF)
  {
    fclose(fp);
    printf("Error in read_configuration_file: Incomplete \
    configuration file. Missing number of sensors of the array \
    in y-direction.\n");
    return -1;
  }

  if(g_sensorArraySize[1] > MAX_Y_NUMBER_OF_SENSORS)
  {
    printf("Warning in read_configuration_file: Requested \
    sensor array size in y direction of %u is too high and will \
    be set to MAX_X_NUMBER_OF_SENSORS.\n", g_sensorArraySize[0]);
    g_sensorArraySize[1]    MAX_Y_NUMBER_OF_SENSORS;
  }
```

```
  // read number of pixels in x-direction
  if(fscanf(fp, "%u\n", &g_pixelsSize[0])     EOF)
  {
    fclose(fp);
    printf("Error␣in␣read_configuration_file:␣Incomplete␣\
␣␣␣␣configuration␣file.␣Missing␣number␣of␣pixels␣of␣the␣\
␣␣␣␣animation␣area␣in␣x-direction.\n");
    return -1;
  }

  // read number of pixels in y-direction
  if(fscanf(fp, "%u\n", &g_pixelsSize[1])     EOF)
  {
    fclose(fp);
    printf("Error␣in␣read_configuration_file:␣Incomplete␣\
␣␣␣␣configuration␣file.␣Missing␣number␣of␣pixels␣of␣the␣\
␣␣␣␣animation␣area␣in␣y-direction.\n");
    return -1;
  }

  fclose(fp);
  return 0;
}


/**
 * @brief  read_coefficient_file - Read Text File for
 *         Coefficients for Signal Processing
 *
 * @param  (const char *)location - File path of the coefficients
 *         file to read.
 *         Pointer to character array. Argument is read.
 *
 * Uses: global.h, stdio.h
 * Reads: g_sensorArraySize
 * Writes: g_filterCoefficients
 */
int read_coefficient_file(const char *location)
{
```

```
  FILE *fp;
  int i;

  fp = fopen(location, "r");

  if(fp == NULL || fseek(fp, 0, SEEK_SET) == -1)
  {
    fclose(fp);
    printf("Error in read_coefficient_file: Inaccessible \
    coefficients file at path %s.\n", location);
    return -1;
  }

  // read filter coefficients
  for (i = 0; i < g_sensorArraySize[0] * g_sensorArraySize[1];
  i++)
  {
    // check if file ends prematurely
    if(fscanf(fp, "%f\n", &g_filterCoefficients[i]) == EOF)
    {
      fclose(fp);
      printf("Error in read_configuration_file: Incomplete \
      coefficients file. Expected %d floating point values. The \
      values must be separated by a \"\\n\" character.\n",
      g_sensorArraySize[0] * g_sensorArraySize[1]);
      return -1;
    }
  }

  fclose(fp);
  return 0;
}
```

src/auxiliary.c

```
#include <stdio.h>
#include <time.h>

#include "global.h"


/**
 * @brief  read_configuration_file - Read Text File for
 *         Configuration Parameters
 *
 * @param  (const char *)location - File path of the
 *         configuration file to read.
 *         Pointer to character array. Argument is read.
 *
 * Uses: global.h, stdio.h
 * Reads: g_sensorArraySize
 * Writes: g_pixelsSize, g_sensorArraySize, g_serialInterface
 */
int read_configuration_file(const char *location)
{
  FILE *fp;

  fp   fopen(location, "r");

  // check file availabilty and set file position indicator to
  // beginning
  if(fp     NULL || fseek(fp, 0, SEEK_SET)     -1)
  {
    fclose(fp);
    printf("Error in read_configuration_file: Inaccessible \
    configuration file at path %s.\n", location);
    return -1;
  }

  // read serial interface path
  if(fscanf(fp, "%s\n", g_serialInterface)     EOF)
  {
    fclose(fp);
    printf("Error in read_configuration_file: Incomplete \
```

```
    configuration file. Missing serial interface device path.\n"
    );
    return -1;
  }

  // read number of sensors in x-direction
  if(fscanf(fp, "%u\n", &g_sensorArraySize[0])     EOF)
  {
    fclose(fp);
    printf("Error in read_configuration_file: Incomplete \
    configuration file. Missing number of sensors of the array \
    in x-direction.\n");
    return -1;
  }

  if(g_sensorArraySize[0] > MAX_X_NUMBER_OF_SENSORS)
  {
    printf("Warning in read_configuration_file: Requested \
    sensor array size in x direction of %u is too high and will \
    be set to MAX_X_NUMBER_OF_SENSORS.\n", g_sensorArraySize[0]);
    g_sensorArraySize[0]    MAX_X_NUMBER_OF_SENSORS;
  }

  // read number of sensors in y-direction
  if(fscanf(fp, "%u\n", &g_sensorArraySize[1])     EOF)
  {
    fclose(fp);
    printf("Error in read_configuration_file: Incomplete \
    configuration file. Missing number of sensors of the array \
    in y-direction.\n");
    return -1;
  }

  if(g_sensorArraySize[1] > MAX_Y_NUMBER_OF_SENSORS)
  {
    printf("Warning in read_configuration_file: Requested \
    sensor array size in y direction of %u is too high and will \
    be set to MAX_X_NUMBER_OF_SENSORS.\n", g_sensorArraySize[0]);
    g_sensorArraySize[1]    MAX_Y_NUMBER_OF_SENSORS;
  }
```

```
  // read number of pixels in x-direction
  if(fscanf(fp, "%u\n", &g_pixelsSize[0])     EOF)
  {
    fclose(fp);
    printf("Error␣in␣read_configuration_file:␣Incomplete␣\
␣␣␣␣configuration␣file.␣Missing␣number␣of␣pixels␣of␣the␣\
␣␣␣␣animation␣area␣in␣x-direction.\n");
    return -1;
  }

  // read number of pixels in y-direction
  if(fscanf(fp, "%u\n", &g_pixelsSize[1])     EOF)
  {
    fclose(fp);
    printf("Error␣in␣read_configuration_file:␣Incomplete␣\
␣␣␣␣configuration␣file.␣Missing␣number␣of␣pixels␣of␣the␣\
␣␣␣␣animation␣area␣in␣y-direction.\n");
    return -1;
  }

  fclose(fp);
  return 0;
}


/**
 * @brief  read_coefficient_file - Read Text File for
 *         Coefficients for Signal Processing
 *
 * @param  (const char *)location - File path of the coefficients
 *         file to read.
 *         Pointer to character array. Argument is read.
 *
 * Uses: global.h, stdio.h
 * Reads: g_sensorArraySize
 * Writes: g_filterCoefficients
 */
int read_coefficient_file(const char *location)
{
```

```
  FILE *fp;
  int i;

  fp = fopen(location, "r");

  if(fp == NULL || fseek(fp, 0, SEEK_SET) == -1)
  {
    fclose(fp);
    printf("Error in read_coefficient_file: Inaccessible \
    coefficients file at path %s.\n", location);
    return -1;
  }

  // read filter coefficients
  for (i = 0; i < g_sensorArraySize[0] * g_sensorArraySize[1];
  i++)
  {
    // check if file ends prematurely
    if(fscanf(fp, "%f\n", &g_filterCoefficients[i]) == EOF)
    {
      fclose(fp);
      printf("Error in read_configuration_file: Incomplete \
      coefficients file. Expected %d floating point values. The \
      values must be separated by a \"\\n\" character.\n",
      g_sensorArraySize[0] * g_sensorArraySize[1]);
      return -1;
    }
  }

  fclose(fp);
  return 0;
}
```

src/main.c

```
#include <stdio.h>
#include <pthread.h>
#include <termios.h>

#include "global.h"
#include "auxiliary.c"
#include "sensor_array.c"
#include "signal_processing.c"
#include "ui.c"


int main(int argc, char *argv[])
{
  // Read Configuration Data
  const char *config   CONFIGURATION_FILE_PATH;
  const char *coeff    COEFFICIENTS_FILE_PATH;
  if(read_configuration_file(config) < 0) return -1;
  if(read_coefficient_file(coeff) < 0) return -1;

  // Open And Set Up UART Interface for Sensor Array
  int uartFd;
  struct termios originalUARTsettings;
  //^ settings of UART serial interface before this program
  //^ changes them
  if(set_up_uart(&uartFd, g_serialInterface,
  &originalUARTsettings) <0)
    return -1;

  int failedRequestAttempts   0;

  // Test Sensor Array
  if(sensor_array_send_request(&uartFd,
  SENSOR_ARRAY_INSTRUCTION_SAMPLE_ARRAY, 0) < 0)
  {
    // test failed
    set_down_uart(&uartFd, &originalUARTsettings);
    return -1;
  }
```

```
  // Set Up Program Control UI Thread
  pthread_t uiThread;
  uiData_t uiData;
  uiData.programControl    0;
  uiData.animationControl    0;
  uiData.animationMode    0;
  uiData.pixelsSize[0]    g_pixelsSize[0];
  uiData.pixelsSize[1]    g_pixelsSize[1];
  pthread_create(&uiThread, NULL, ui_thread, &uiData);


  //###################### Time Measurement ######################
  //~ clock_t startTime, stopTime;
  //~ FILE *sample_and_send_array    fopen("/home/pi/Desktop/\
  sample_and_send_array","w");
  //~ FILE *collect_array  fopen("/home/pi/Desktop/\
  collect_array","w");
  //~ FILE *main  fopen("/home/pi/Desktop/main","w");
  //~ uiData.animationControl |
  //~ UI_ANIMATION_SELECTION_DATA_DIRECT <<
  //~ UI_ANIMATION_CONTROL_VIEW_SELECTION_OFFSET;
  //~ uiData.animationControl |  1 <<
  //~ UI_ANIMATION_CONTROL_INTERPOLATION_OFFSET;
  //#########################################################################

  // Data Aquisition Loop
  while(!(uiData.programControl & UI_PROGRAM_CONTROL_EXIT_MASK))
  {
    //###################### Time Measurement ######################
    //~ startTime  clock();
    //#########################################################################

    // optional: wait for the request of the ui
    //~ while(!(uiData.programControl &
    //~ UI_PROGRAM_CONTROL_REFRESH_MASK))
    //~ {
      //~ if((uiData.programControl &
      //~ UI_PROGRAM_CONTROL_EXIT_MASK))
      //~ {
        //~ // ui has requested program exit
```

```
        //~set_down_uart(&uartFd,&originalUARTsettings);
        //~pthread_cancel(uiThread);
        //~return-1;
    //~}
//~}

//#######################Time Measurement#######################
//~startTime =clock();
//##############################################################

// request sensor array sample
if(sensor_array_send_request(&uartFd,
SENSOR_ARRAY_INSTRUCTION_SAMPLE_AND_SEND_ARRAY,0)<0)
    failedRequestAttempts+=2;

//#######################Time Measurement#######################
//~stopTime =clock();
//~fprintf(sample_and_send_array,"%f\n",(float)(stopTime-
//~startTime)/CLOCKS_PER_SEC);
//##############################################################

// Process UI Data
if(((uiData.programControl&
UI_PROGRAM_CONTROL_LOAD_CONFIGURATION_MASK)>>
UI_PROGRAM_CONTROL_LOAD_CONFIGURATION_OFFSET)  =1)
{
    uiData.programControl&=!(
    UI_PROGRAM_CONTROL_LOAD_CONFIGURATION_MASK);//reset flag

    // restore serial interface
    set_down_uart(&uartFd,&originalUARTsettings);

    // load configuration paramters from file
    if(read_configuration_file(config)<0)
    {
        // failed
        set_down_uart(&uartFd,&originalUARTsettings);
        pthread_cancel(uiThread);
        return-1;
    }
```

```
      //␣reconfigure␣and␣test␣serial␣interface
      if(set_up_uart(&uartFd,␣g_serialInterface,
      &originalUARTsettings)␣<␣0␣||␣sensor_array_send_request(
      &uartFd,␣SENSOR_ARRAY_INSTRUCTION_SEND_ARRAY,␣0)␣<␣0)
      {
        //␣reconfiguration␣or␣test␣failed
        set_down_uart(&uartFd,␣&originalUARTsettings);
        pthread_cancel(uiThread);
        return␣-1;
      }

      //set␣animation␣size
      uiData.pixelsSize[0]␣ ␣g_pixelsSize[0];
      uiData.pixelsSize[1]␣ ␣g_pixelsSize[1];
    }

    if(((uiData.programControl␣&
    UI_PROGRAM_CONTROL_LOAD_COEFFICIENTS_MASK)␣>>
    UI_PROGRAM_CONTROL_LOAD_COEFFICIENTS_OFFSET)␣  ␣1)
    {
      uiData.programControl␣& ␣!(
      UI_PROGRAM_CONTROL_LOAD_COEFFICIENTS_MASK);␣//␣reset␣flag

      //␣load␣coefficients␣from␣file
      if(read_coefficient_file(coeff)␣<0)
      {
        //␣failed
        set_down_uart(&uartFd,␣&originalUARTsettings);
        pthread_cancel(uiThread);
        return␣-1;
      }
    }

    //#####################␣Time␣Measurement␣#####################
    //~␣startTime␣ ␣clock();
    //#############################################################

    //␣collect␣sensor␣array␣data
    if(sensor_array_collect_array_data(&uartFd,
```

```
␣␣␣␣g_sensorArrayXvalues,␣g_sensorArrayYvalues,␣g_sensorArraySize
␣␣␣␣)␣<␣0)
␣␣␣␣␣␣failedRequestAttempts␣+␣2;

␣␣␣␣//####################␣Time␣Measurement␣####################
␣␣␣␣//~␣stopTime␣␣␣clock();
␣␣␣␣//~␣fprintf(collect_array,␣"%f\n",␣(float)(stopTime␣-
␣␣␣␣//~␣startTime)␣/␣CLOCKS_PER_SEC);
␣␣␣␣//#############################################################

␣␣␣␣//␣check␣request␣failure␣count
␣␣␣␣if(failedRequestAttempts␣>␣10)
␣␣␣␣{
␣␣␣␣␣␣//␣over␣threshold
␣␣␣␣␣␣set_down_uart(&uartFd,␣&originalUARTsettings);
␣␣␣␣␣␣pthread_cancel(uiThread);
␣␣␣␣␣␣return␣-1;
␣␣␣␣}
␣␣␣␣else␣if(failedRequestAttempts␣>␣0)
␣␣␣␣{
␣␣␣␣␣␣failedRequestAttempts--;
␣␣␣␣}

␣␣␣␣switch((uiData.animationControl␣&
␣␣␣␣UI_ANIMATION_CONTROL_VIEW_SELECTION_MASK)␣>>
␣␣␣␣UI_ANIMATION_CONTROL_VIEW_SELECTION_OFFSET)
␣␣␣␣{
␣␣␣␣␣␣case␣UI_ANIMATION_SELECTION_DATA_DIRECT:
␣␣␣␣␣␣//␣display␣the␣unprocessed␣sensor␣array␣data␣as␣a␣vector
␣␣␣␣␣␣//␣field
␣␣␣␣␣␣␣␣uiData.animationMode␣␣UI_ANIMATION_MODE_VECTOR_FIELD;
␣␣␣␣␣␣␣␣//^␣set␣animation␣mode

␣␣␣␣␣␣␣␣if(((uiData.animationControl␣&
␣␣␣␣␣␣␣␣UI_ANIMATION_CONTROL_INTERPOLATION_MASK)␣>>
␣␣␣␣␣␣␣␣UI_ANIMATION_CONTROL_INTERPOLATION_OFFSET)␣␣␣1)
␣␣␣␣␣␣␣␣{
␣␣␣␣␣␣␣␣␣␣//␣interpolate␣data
␣␣␣␣␣␣␣␣␣␣interpolate_data(g_sensorArrayXvalues,
␣␣␣␣␣␣␣␣␣␣g_sensorArrayYvalues,␣g_sensorArraySize,
```

```
␣␣␣␣␣␣␣␣␣␣uiData.xValues,␣uiData.yValues,␣uiData.valuesSize);
␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣else
␣␣␣␣␣␣␣␣{
␣␣␣␣␣␣␣␣␣␣//␣pass␣through␣data
␣␣␣␣␣␣␣␣␣␣uiData.valuesSize[0]␣␣g_sensorArraySize[0];
␣␣␣␣␣␣␣␣␣␣uiData.valuesSize[1]␣␣g_sensorArraySize[1];

␣␣␣␣␣␣␣␣␣␣for(int␣i␣␣0;␣i␣<␣g_sensorArraySize[0]␣*
␣␣␣␣␣␣␣␣␣␣g_sensorArraySize[1];␣i++)
␣␣␣␣␣␣␣␣␣␣{
␣␣␣␣␣␣␣␣␣␣␣␣uiData.xValues[i]␣␣g_sensorArrayXvalues[i];
␣␣␣␣␣␣␣␣␣␣␣␣uiData.yValues[i]␣␣g_sensorArrayYvalues[i];
␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣break;
␣␣␣␣␣␣//␣-------------------------------------------------------------
␣␣␣␣␣␣case␣UI_ANIMATION_SELECTION_DATA_DIRECT_X:
␣␣␣␣␣␣//␣display␣the␣unprocessed␣sensor␣array␣data␣of␣the␣x-
␣␣␣␣␣␣//␣direction␣as␣a␣heatmap
␣␣␣␣␣␣␣␣uiData.animationMode␣␣UI_ANIMATION_MODE_HEATMAP;
␣␣␣␣␣␣␣␣//^␣set␣animation␣mode

␣␣␣␣␣␣␣␣if(((uiData.animationControl␣&
␣␣␣␣␣␣␣␣UI_ANIMATION_CONTROL_INTERPOLATION_MASK)␣>>
␣␣␣␣␣␣␣␣UI_ANIMATION_CONTROL_INTERPOLATION_OFFSET)␣␣␣1)
␣␣␣␣␣␣␣␣{
␣␣␣␣␣␣␣␣␣␣//␣interpolate␣data
␣␣␣␣␣␣␣␣␣␣interpolate_data(g_sensorArrayXvalues,
␣␣␣␣␣␣␣␣␣␣g_sensorArrayYvalues,␣g_sensorArraySize,
␣␣␣␣␣␣␣␣␣␣uiData.xValues,␣uiData.yValues,␣uiData.valuesSize);
␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣else
␣␣␣␣␣␣␣␣{
␣␣␣␣␣␣␣␣␣␣//␣pass␣through␣data
␣␣␣␣␣␣␣␣␣␣uiData.valuesSize[0]␣␣g_sensorArraySize[0];
␣␣␣␣␣␣␣␣␣␣uiData.valuesSize[1]␣␣g_sensorArraySize[1];

␣␣␣␣␣␣␣␣␣␣for(int␣i␣␣0;␣i␣<␣g_sensorArraySize[0]␣*
␣␣␣␣␣␣␣␣␣␣g_sensorArraySize[1];␣i++)
```

```
          {
              uiData.xValues[i]  g_sensorArrayXvalues[i];
          }
      }
      break;
    //-------------------------------------------------------------------
    case UI_ANIMATION_SELECTION_DATA_DIRECT_Y:
    //display the unprocessed sensor array data of the y-
    //direction as a heatmap
      uiData.animationMode  UI_ANIMATION_MODE_HEATMAP;
      //^ set animation mode

      if (((uiData.animationControl &
      UI_ANIMATION_CONTROL_INTERPOLATION_MASK) >>
      UI_ANIMATION_CONTROL_INTERPOLATION_OFFSET)    1)
      {
        //interpolate data
        interpolate_data(g_sensorArrayXvalues,
        g_sensorArrayYvalues, g_sensorArraySize,
        uiData.yValues, uiData.xValues, uiData.valuesSize);
        //^ note that the uiData for heatmap mode is always
        //^ xValues. Therefore, it is necessary to swap x and y
        //^ when interpolating.
      }
      else
      {
        //pass through data
        uiData.valuesSize[0]  g_sensorArraySize[0];
        uiData.valuesSize[1]  g_sensorArraySize[1];

        for(int i  0;i<g_sensorArraySize[0] *
        g_sensorArraySize[1];i++)
        {
          uiData.xValues[i]  g_sensorArrayYvalues[i];
          //^ note that the uiData for heatmap mode is always
          //^ xValues
        }
      }
      break;
    //-------------------------------------------------------------------
```

```
      case UI_ANIMATION_SELECTION_ANGLE:
      //  display a demo for angle view
        uiData.animationMode  UI_ANIMATION_MODE_VECTOR;
        //^ set animation mode

        static float g_angle;
        g_angle + 0.03;
        if(g_angle > 2 * M_PI) g_angle  0;

        uiData.xValues[0]  -0.14 * g_angle; // x-Position
        uiData.xValues[1]  0.14 * g_angle; // y-Position
        uiData.xValues[2]  g_angle; // angle (rad)

        break;
      //-------------------------------------------------------------------
      // EXTENTION:
      //~ case UI_ANIMATION_SELECTION_MY_MACRO:

        // HOWTO:  1. first process your signals
        //          2. then set the animation mode (shown below)
        //          3. last copy your data to the uiData directly
        //             or using the interpolation function (shown
        //             below)

        //~ uiData.animationMode  UI_ANIMATION_MODE_*;
        //~ //^ set animation mode

        //~ if(((uiData.animationControl &
        //~ UI_ANIMATION_CONTROL_INTERPOLATION_MASK) >>
        //~ UI_ANIMATION_CONTROL_INTERPOLATION_OFFSET)    1)
        //~ {
          //~ // interpolate data
          //~ interpolate_data(myXvalues, myYvalues,
          //~ myValuesSize, uiData.xValues, uiData.yValues,
          //~ uiData.valuesSize);
        //~ }
        //~ else
        //~ {
          //~ // pass through data
          //~ uiData.valuesSize[0]  myValuesSize[0];
```

```
          //~ uiData.valuesSize[1]  myValuesSize[1];

          //~ for(int i  0;i<myValuesSize[0]*
          //~ myValuesSize[1];i++)
          //~ {
          //~ uiData.xValues[i]  myXvalues[i];
          //~ uiData.yValues[i]  myYvalues[i];
          //~ }
          //~ }
          //~ break;
      //  ------------------------------------------------------------
      default:
          break;
    }
  //######################## Time Measurement ########################
  //~ stopTime  clock();
  //~ fprintf(main,"%f\n",(float)(stopTime-startTime)/
  //~ CLOCKS_PER_SEC);
  //########################################################################
  }

  // restore the serial interface
  set_down_uart(&uartFd,&originalUARTsettings);

  // wait for ui thread to exit
  pthread_join(uiThread,NULL);

  return 0;
}
```

src/sensor_array.c

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>
#include <termios.h>
#include <time.h>

#include "global.h"


//                                                                      //
//----------------------Communication Functions-------------------------//
//                                                                      //

/**
 * @brief  set_up_uart - Configure Serial Interface in Non-
 *         Canonical Mode
 *
 * @param  (int *)uartFd - File descriptor of the serial
 *         interface to configure.
 *         Pointer to integer. Argument is written.
 * @param  (char *)device - Device path of the serial interdace
 *         to configure.
 *         Pointer to character array. Argument is read.
 * @param  (struct termios *)originalSettings - Variable to save
 *         the existing configuration to when this function is
 *         called.
 *         Pointer to termios struct. Argument is written.
 *
 * Uses: fcntl.h, stdio.h, termios.h
 * Reads: -
 * Writes: -
 */
int set_up_uart(int *uartFd, char *device, struct termios
*originalSettings)
{
  *uartFd   open(device, O_RDWR | O_NOCTTY | O_NDELAY);
  //^ open in non-blocking read/write mode
```

```
  if (*uartFd < 0)
  {
    perror("Error in set_up_uart: Unable to open UART");
    return -1;
  }

  // initialize new settings
  struct termios settings;
  if(tcgetattr(*uartFd, &settings) < 0)
  {
    perror("Error in set_up_uart: Couldn't get UART interface \
    configuration");
    return -1;
  }

  // store initial settings
  if(originalSettings != NULL) *originalSettings = settings;

  // set control options
  settings.c_cflag &= ~(CBAUD // clear baud rate
  | CSIZE // clear word size
  | CSTOPB // disable second stop bit
  | PARODD); // disable odd parity / enable even parity
  settings.c_cflag |= (B460800 // set baud rate (old way)
  | CS8 // set word size to 8
  | CREAD // enable reception
  | PARENB // enable parity
  | CRTSCTS); // enable RTS/CTS hardware flow control

  // set the baud rate (new way)
  //cfsetispeed(&settings, B460800);
  //cfsetospeed(&settings, B460800);

  // set line options for non-canonical (raw) mode
  settings.c_lflag &= ~(ISIG | ICANON | XCASE | ECHO | ECHOE |
  ECHOK | ECHONL | NOFLSH | TOSTOP | ECHOCTL | ECHOPRT | ECHOKE |
  FLUSHO | PENDIN | IEXTEN | EXTPROC);
  //^ don't post process input data
  settings.c_lflag |= (0);
```

```
  // set input options
  settings.c_iflag &  ~(IGNBRK // don't ignore break condition
  | PARMRK | ISTRIP // don't mark or strip the parity bits
  | IGNCR // don't discard CR on input
  | INLCR | ICRNL // don't interpret '|r' or '|n' characters
  | IUCLC // don't translate upercase to lowercase
  | IXON | IXANY | IXOFF // disable software flow control
  | IMAXBEL // don't send special character when buffer is full
  | IUTF8); // don't interpret input as UTF-8 encoded
  settings.c_iflag |  (BRKINT
  //^ send a SIGINT when a break condition is detected
  | INPCK // enable input parity checking
  | IGNPAR); // discard receptions with parity- or framing-error

  // set output options for non-canonical (raw) mode
  settings.c_oflag &  ~(OPOST | OLCUC | ONLCR | OCRNL | ONOCR |
 ONLRET | OFILL | OFDEL | NLDLY | CRDLY | TABDLY | XTABS |
 BSDLY | VTDLY | FFDLY); // don't post process output data
  settings.c_oflag |  (0);

  settings.c_cc[VMIN]   1;
  //^ at least one byte must be in the read buffer in order for
  //^ read(3) to return successfully
  settings.c_cc[VTIME]   0;
  //^ read(3) can read the buffer immediately after function call

  // apply new settings
  if(tcsetattr(*uartFd, TCSAFLUSH, &settings) < 0)
  {
    perror("Error in set_up_uart: Couldn't update UART \
    interface configuration");
    return -1;
  }

  return 0;
}


/**
 * @brief  set_down_uart - Configure Serial Interface in Given
```

```
 *          Mode
 *
 * @param  (int *)uartFd - File descriptor of the serial
 *          interface to configure.
 *          Pointer to integer. Argument is read.
 * @param  (struct termios *)originalSettings - Variable that
 *          stores the configuration.
 *          Pointer to termios struct. Argument is read.
 *
 * Uses: stdio.h, termios.h, unistd.h
 * Reads: -
 * Writes: -
 */
int set_down_uart(int *uartFd, struct termios *originalSettings)
{
  if(tcsetattr(*uartFd, TCSAFLUSH, originalSettings) < 0)
  {
    perror("Error in set_down_uart: Couldn't restore UART \
    interface configuration");
    return -1;
  }

  close(*uartFd);

  return 0;
}



//                                                              //
//-------------------Sensor Array Functions---------------------//
//                                                              //

/**
 * @brief  sensor_array_send_request - Send a request to the
 *          sensor-array via the serial interface and check for
 *          confirmation answer
 *
 * @param  (int *)uartFd - File descriptor of the serial
 *          interface to configure.
```

```
 *          Pointer to integer. Argument is written.
 * @param   (unsigned char)requestInstruction - Instruction code
 *          to send to the sensor-array.
 *          The valid range is zero (0) to seven (7).
 * @param   (unsigned char)requestArgument - Argument to send to
 *          the sensor-array.
 *          The valid range is zero (0) to thirty-one (31).
 *
 * Uses: global.h, stdio.h, time.h
 * Reads: -
 * Writes: -
 */
int sensor_array_send_request(int *uartFd, unsigned char
requestInstruction, unsigned char requestArgument)
{
  const float timeoutTime = 0.5f;

  // empty read buffer
  unsigned char reception;
  clock_t startTime, currentTime;
  if(read(*uartFd, &reception, 1) != 0)
  {
    printf("Warning in sensor_array_send_request: Unexpected \
    data in receive buffer. Clearing buffer ... ");
    startTime = clock();
    while(1)
    {
      if(read(*uartFd, &reception, 1) != 0)
      {
        // read buffer is not empty
        startTime = clock(); // reset start time for next attempt
      }

      currentTime = clock();

      if((float)(currentTime - startTime) / CLOCKS_PER_SEC >
      timeoutTime)
      {
        // read attempt has timed out and read buffer is empty
        printf("done.\n");
```

```
        break;
      }
    }
  }

  // send instruction
  unsigned char transmission   (requestInstruction <<
  SENSOR_ARRAY_INSTRUCTION_OFFSET) | (requestArgument &
  SENSOR_ARRAY_ARGUMENT_MASK);

  if(write(*uartFd, &transmission, 1) < 0) // failed transmission
  {
    perror("Error in sensor_array_send_request: Could not send \
    instruction");
    return -1;
  }

  // wait for first response
  startTime   clock();

  while(read(*uartFd, &reception, 1) < 0)
  {
    currentTime   clock();

    if((float)(currentTime - startTime) / CLOCKS_PER_SEC >
    timeoutTime)
    {
      // timeout
      printf("Error in sensor_array_send_request: Request was \
      not answered.\n");
      return -1;
    }
  }

  // check response
  if(reception !  transmission)
  {
    // response differs from request
    printf("Error in sensor_array_send_request: Erroneous \
    answer received. Should be %2x not %2x\n", transmission,
```

```
    reception);
    return -1;
  }

  return 0;
}


/**
 * @brief  sensor_array_collect_array_data - Read out the serial
 *         interface buffer for received sensor-array data of all
 *         sensors
 *
 * @param  (int *)uartFd - File descriptor of the serial
 *         interface to configure.
 *         |tPointer to integer. Argument is written.
 * @param  (float *)xValues - Values of x-direction of sensor-
 *         array.
 *         Pointer to 1d array of (valuesSize[0] *
 *         valuesSize[1]). Entries are written.
 * @param  (float *)yValues - Values of y-direction of sensor-
 *         array.
 *         Pointer to 1d array of (valuesSize[0] *
 *         valuesSize[1]). Entries are written.
 * @param  (unsigned int *)valuesSize - Size of *Values arrays.
 *         Pointer to 1d array of size two, with first element
 *         being x-size and second being y-size of the *Values
 *         arrays. Entries are read.
 *
 * Uses: global.h, stdio.h, stdlib.h, time.h
 * Reads: -
 * Writes: -
 */
int sensor_array_collect_array_data(int *uartFd, float *xValues,
float *yValues, unsigned int *valuesSize)
{
  const float timeoutTime   0.5f;

  // collect returned data
  unsigned char reception[4 * MAX_X_NUMBER_OF_SENSORS *
```

```
  MAX_Y_NUMBER_OF_SENSORS]    {};
  clock_t startTime, currentTime;
  unsigned int helper, print;

  for(helper   0; helper < 4 * valuesSize[0] * valuesSize[1];
  helper++)
  {
    // attempt a read
    startTime   clock();

    while(read(*uartFd, &reception[helper], 1) < 0)
    {
      currentTime   clock();

      if((float)(currentTime - startTime) / CLOCKS_PER_SEC >
      timeoutTime)
      {
        // timeout
        printf("Error in sensor_array_collect_array_data: \
        Timeout while receiving data.\n");
        return -1;
      }
    }
  }

  // store data
  print   0;
  for(helper   0; helper < valuesSize[0] * valuesSize[1];
  helper++)
  {
    // read succeeded
    *(yValues + helper)   (float)((int16_t)((reception[4 *
    helper + 0] << 8) | reception[4 * helper + 1])) /
    MAX_ABS_SENSOR_VALUE;
    *(xValues + helper)   (float)((int16_t)((reception[4 *
    helper + 2] << 8) | reception[4 * helper + 3])) /
    MAX_ABS_SENSOR_VALUE;

    if(xValues[helper] > 1.0f)
    {
```

```
      xValues[helper]  = 1.0f;
      print++;
    }
    else if(xValues[helper] < -1.0f)
    {
      xValues[helper]  = -1.0f;
      print++;
    }
    if(yValues[helper] > 1.0f)
    {
      yValues[helper]  = 1.0f;
      print++;
    }
    else if(yValues[helper] < -1.0f)
    {
      yValues[helper]  = -1.0f;
      print++;
    }
  }
  if(print != 0)
    printf("Warning in sensor_array_collect_array_data: Data \
    was out of range %d times.\n", print);

  return 0;
}
```

src/signal_processing.c

```
#include <stdio.h>
#include <math.h>
#include "global.h"



void interpolate_data(float* initialXvalues, float*
initialYvalues, unsigned int* initialSize, float*
interpolatedXvalues, float* interpolatedYvalues, unsigned int
*outSize)
{
  // Original by Simon Rindelaub
  // Modified to take separate x- & y-arrays of any size

  unsigned int xIdx  0, yIdx  0, index;
  unsigned int interpolatedSize[2]  {2 * initialSize[0] - 1, 2 *
       initialSize[1] - 1};
  outSize[0]  interpolatedSize[0];
  outSize[1]  interpolatedSize[1];

  // copy existing vectors to interpolated array
  for(yIdx  0; yIdx < initialSize[1]; yIdx++)
  {
    for(xIdx  0; xIdx < initialSize[0]; xIdx++)
    {
      index  yIdx * initialSize[0] + xIdx;

      interpolatedXvalues[2 * (yIdx * interpolatedSize[1] +
      xIdx)]  initialXvalues[index];
      interpolatedYvalues[2 * (yIdx * interpolatedSize[1] +
      xIdx)]  initialYvalues[index];
    }
  }

  // caclulate interpolated vectors
  for(yIdx  0; yIdx < interpolatedSize[1]; yIdx++)
  {
    for(xIdx  0; xIdx < interpolatedSize[0]; xIdx++)
    {
```

```
index   yIdx * interpolatedSize[0] + xIdx;

if(yIdx % 2    0) // even row
{
  if(xIdx % 2    0) // even column
  {

  }
  else
  {
    // calculate average of x-neighbouring vectors
    interpolatedXvalues[index]   0.5 * (
    interpolatedXvalues[index - 1] +
    interpolatedXvalues[index + 1]);
    interpolatedYvalues[index]   0.5 * (
    interpolatedYvalues[index - 1] +
    interpolatedYvalues[index + 1]);
  }
}
else // odd row
{
  if(xIdx % 2    0) // even column
  {
    // calculate average of y-neighbouring vectors
    interpolatedXvalues[index]   0.5 * (
    interpolatedXvalues[(yIdx - 1) * interpolatedSize[1] +
    xIdx] + interpolatedXvalues[(yIdx + 1) *
    interpolatedSize[1] + xIdx]);
    interpolatedYvalues[index]   0.5 * (
    interpolatedYvalues[(yIdx - 1) * interpolatedSize[1] +
    xIdx] + interpolatedYvalues[(yIdx + 1) *
    interpolatedSize[1] + xIdx]);
  }
  else
  {
    // calculate average of cross-neighbouring vectors
    interpolatedXvalues[index]   0.5 *
    interpolatedXvalues[index - 1] + 0.25 * (
    interpolatedXvalues[(yIdx - 1) * interpolatedSize[1] +
    xIdx + 1] + interpolatedXvalues[(yIdx + 1) *
```

```
            interpolatedSize[1] + xIdx + 1]);
            interpolatedYvalues[index]    0.5 *
            interpolatedYvalues[index - 1] + 0.25 *
            (interpolatedYvalues[(yIdx - 1) * interpolatedSize[1] +
            xIdx + 1] + interpolatedYvalues[(yIdx + 1) *
            interpolatedSize[1] + xIdx + 1]);
          }
        }
      }
    }
}


//#########################################################################
//##                                                                     ###
//##    The Code Below Is Mostly Untested. Use With Caution!             ###
//##                                                                     ###
//#########################################################################


//~ //                                                                    //
//~ //--------------------Filter Functions--------------------//
//~ //                                                                    //

//~ /**
 //~ * @brief calculate_twiddle - Calculate Twiddle-Matrix of 2D
 //~ *        Matrix
 //~ *
 //~ * @param twiddleXvalues - Output values of x-direction of 2D
 //~ *        array. Pointer to 1d array of (size[0] * size[1]).
 //~ *        Entries are written.
 //~ * @param twiddleYvalues - Output values of y-direction of 2D
 //~ *        array. Pointer to 1d array of (size[0] * size[1]).
 //~ *        Entries are written.
 //~ * @param twiddleSize - Size of input and output arrays.
 //~ *        Pointer to 1d array of size two, with first element
 //~ *        being x-size and second being y-size of the input
 //~ *        and output arrays. Entries are read.
 //~ * @param inverse - Calculate inverse twiddle values if
 //~ *        inverse > 0.
```

```
 //~ */
//~ void calculate_twiddle(float* twiddleXvalues, float*
//~ twiddleYvalues, unsigned int* twiddleSize, unsigned int
//~ inverse)
//~ {
  //~ // Original by Simon Rindelaub (twiddle, InvTwiddle,
  //~ // twiddle15, InvTwiddle15)
  //~ // Modified to take separate x- & y-arrays of any size

  //~ unsigned int xIdx, yIdx, index;
  //~ float helper;

  //~ for(yIdx   0; yIdx < twiddleSize[1]; yIdx++)
  //~ {
    //~ for(xIdx   0; xIdx < twiddleSize[0]; xIdx++)
    //~ {
      //~ index   yIdx * twiddleSize[0] + xIdx;
      //~ helper   - PI_2_1 * (float)(xIdx * yIdx);

      //~ if(inverse > 0) helper   - helper;

      //~ twiddleXvalues[index]   cosf(helper / twiddleSize[0]);
      //~ twiddleYvalues[index]   sinf(helper / twiddleSize[1]);
    //~ }
  //~ }
//~ }



//~ /**
 //~ * @brief bidirectional_twiddle_fft - Calculate Fourier
 //~ *        Transformation of 2D Matrix with Twiddle-Matrix
 //~ *        (Output   (W * Input) * W)
 //~ *
 //~ * @param inputXvalues - Input values of x-direction of 2D
 //~ *        array. Pointer to 1d array of (size[0] * size[1]).
 //~ *        Entries are read.
 //~ * @param inputYvalues - Input values of y-direction of 2D
 //~ *        array. Pointer to 1d array of (size[0] * size[1]).
 //~ *        Entries are read.
```

```
//~ * @param twiddleXvalues - Input values of x-direction of 2D
//~ *          array. Pointer to 1d array of (size[0] * size[1]).
//~ *          Entries are read.
//~ * @param twiddleYvalues - Input values of y-direction of 2D
//~ *          array. Pointer to 1d array of (size[0] * size[1]).
//~ *          Entries are read.
//~ * @param outputXvalues - Output values of x-direction of 2D
//~ *          array. Pointer to 1d array of (size[0] * size[1]).
//~ *          Entries are written.
//~ * @param outputYvalues - Output values of y-direction of 2D
//~ *          array. Pointer to 1d array of (size[0] * size[1]).
//~ *          Entries are written.
//~ * @param size - Size of input and output arrays. Pointer to
//~ *          1d array of size two, with first element being x-
//~ *          size and second being y-size of the input and
//~ *          output arrays. Entries are read.
//~ */
//~ void bidirectional_twiddle_fft(float* inputXvalues, float*
//~ inputYvalues, float* twiddleXvalues, float* twiddleYvalues,
//~ float* outputXvalues, float* outputYvalues, unsigned int*
//~ size)
//~ {
  //~ // Original by Simon Rindelaub (fft, iffft, fft15, ifft15)
  //~ // Modified to take separate x- & y-arrays of any size

  //~ unsigned int c, d, k, inputIndex, twiddleIndex,
  //~ outputIndex;
  //~ // Buffer for the first step of the matrix-multiplication
  //~ float xBuffer[MAX_X_NUMBER_OF_VIRTUAL_SENSORS][
  //~ MAX_Y_NUMBER_OF_VIRTUAL_SENSORS];
  //~ float yBuffer[MAX_X_NUMBER_OF_VIRTUAL_SENSORS][
  //~ MAX_Y_NUMBER_OF_VIRTUAL_SENSORS];

  //~ // first loop g1   W * input
  //~ for (c   0; c < size[0]; c++)
  //~ {
    //~ for (d   0; d < size[1]; d++)
    //~ {
      //~ xBuffer[c][d]   0;
      //~ yBuffer[c][d]   0;
```

```
      //~ for (k    0; k < size[0]; k++)
      //~ {
        //~ inputIndex    d * size[0] + k;
        //~ twiddleIndex    k * size[0] + c;

        //~ xBuffer[c][d] +  twiddleXvalues[twiddleIndex] *
        //~ inputXvalues[inputIndex] +
        //~ twiddleYvalues[twiddleIndex] *
        //~ inputYvalues[inputIndex];
        //~ yBuffer[c][d] +  twiddleXvalues[twiddleIndex] *
        //~ inputYvalues[inputIndex] +
        //~ twiddleYvalues[twiddleIndex] *
        //~ inputXvalues[inputIndex];
      //~ }
    //~ }
  //~ }

  //~ // second loop out    g1 * W
  //~ for (c    0; c < size[0]; c++)
  //~ {
    //~ for (d    0; d < size[1]; d++)
    //~ {
      //~ outputIndex    d * size[0] + c;

      //~ outputXvalues[outputIndex]    0;
      //~ outputYvalues[outputIndex]    0;

      //~ for (k    0; k < size[0]; k++)
      //~ {
        //~ // Multiplication of two complex numbers - Imag:
        //~ // (j*a1*b2)+(j*a2*b1) Real: (a1*b1)-(a2*b2)
        //~ twiddleIndex    d * size[0] + k;

        //~ outputXvalues[outputIndex] +  xBuffer[c][k] *
        //~ twiddleXvalues[twiddleIndex] - yBuffer[c][k] *
        //~ twiddleYvalues[twiddleIndex];
        //~ outputYvalues[outputIndex] +  yBuffer[c][k] *
        //~ twiddleXvalues[twiddleIndex] - xBuffer[c][k] *
        //~ twiddleYvalues[twiddleIndex];
```

```
      //~ }
    //~ }
  //~ }
//~ }



//~ /**
 //~ * @brief convolute_filter - Apply filter to 2D-Matrix in
 //~ *         Frequency Domain
 //~ *
 //~ * @param inputXvalues - Input values of x-direction of 2D
 //~ *         array. Pointer to 1d array of (size[0] * size[1]).
 //~ *         Entries are read.
 //~ * @param inputYvalues - Input values of y-direction of 2D
 //~ *         array. Pointer to 1d array of (size[0] * size[1]).
 //~ *         Entries are read.
 //~ * @param filterValues - Input values of x-direction of 2D
 //~ *         array. Pointer to 1d array of (size[0] * size[1]).
 //~ *         Entries are read.
 //~ * @param outputXvalues - Output values of x-direction of 2D
 //~ *         array. Pointer to 1d array of (size[0] * size[1]).
 //~ *         Entries are written.
 //~ * @param outputYvalues - Output values of y-direction of 2D
 //~ *         array. Pointer to 1d array of (size[0] * size[1]).
 //~ *         Entries are written.
 //~ * @param size - Size of input and output arrays. Pointer to
 //~ *         1d array of size two, with first element being x-
 //~ *         size and second being y-size of the input and
 //~ *         output arrays. Entries are read.
 //~ */
//~ void convolute_filter(float* inputXvalues, float*
//~ inputYvalues, float* filterValues, float* outputXvalues,
//~ float* outputYvalues, unsigned int* size)
//~ {
  //~ // Original by Simon Rindelaub (filter8 & multiply8,
  //~ filter15 & multiply15)
  //~ // Modified to take separate x- & y-arrays of any size

  //~ unsigned int index;
```

```
  //~ for (index    0; index < size[0] * size[1]; index++)
  //~ {
    //~ outputXvalues[index]    inputXvalues[index] *
    //~ filterValues[index];
    //~ outputYvalues[index]    inputYvalues[index] *
    //~ filterValues[index];
  //~ }
//~ }



//~ //                                                                    //
//~ //--------------------Math Functions--------------------//
//~ //                                                                    //

//~ /**
 //~ * @brief calculate_angle - Calculate Counter Clockwise Angle
 //~ *        of Vector to positive x-Axis of Matrix of 2D-
 //~ *        Vectors in 0 to 360 degrees
 //~ *
 //~ * @param inputXvalues - Input values of x-direction of 2D
 //~ *        array. Pointer to 1d array of (size[0] * size[1]).
 //~ *        Entries are read.
 //~ * @param inputYvalues - Input values of y-direction of 2D
 //~ *        array. Pointer to 1d array of (size[0] * size[1]).
 //~ *        Entries are read.
 //~ * @param outputAngles - Output values of angle of 2D
 //~ *        vectors. Pointer to 1d array of (size[0] *
 //~ *        size[1]). Entries are written.
 //~ * @param valuesSize - Size of input and output arrays.
 //~ *        Pointer to 1d array of size two, with first element
 //~ *        being x-size and second being y-size of the input
 //~ *        and output arrays. Entries are read.
 //~ */
//~ void calculate_angle(float* inputXvalues, float*
//~ inputYvalues, float* outputAngles, unsigned int* valuesSize)
//~ {
  //~ unsigned int index;
```

```
  //~ for(index    0; index < valuesSize[0] * valuesSize[1];
  //~ index++)
  //~ {
    //~ outputAngles[index]    atanf(inputYvalues[index] /
    //~ inputXvalues[index]) * (180 / PI);

    //~ if(inputXvalues[index] < 0) //Q2 (-,+) or Q3 (-,-)
    //~ {
      //~ outputAngles[index] +  180.0;
    //~ }
    //~ else // Q1 (+,+) or Q4 (+,-)
    //~ {
      //~ if(inputYvalues[index] < 0) // Q4 (+,-)
      //~ {
        //~ outputAngles[index] +  360.0;
      //~ }
      //~ // no operation required for first quadrant
    //~ }
  //~ }
//~ }



//~ /**
 //~ * @brief calculate_average - Calculate Average of Values
 //~ *
 //~ * @param inputValues - Input values. Pointer to 1d array.
 //~ *         Entries are read.
 //~ * @param size - Size of input array.
 //~ * @return Average of input values.
 //~ */
//~ float calculate_average(float* inputValues, unsigned int
//~ size)
//~ {
  //~ unsigned int i;
  //~ float avg    0;

  //~ for(i    0; i < size; i++) avg +  inputValues[i];
  //~ avg /  size;
```

```
  //~ return avg;
//~ }



//~ void adjungieren(float input[3][3], float output[3][3])
//~ {
  //~ // By Simon Rindelaub

  //~ output[0][0]    (input[1][1]*input[2][2]) -
  //~ (input[1][2]*input[2][1]);   //A(2,2)*A(3,3)-A(2,3)*A(3,2)
  //~ output[0][1]    (input[0][2]*input[2][1]) -
  //~ (input[0][1]*input[2][2]);   //A(1,3)*A(3,2)-A(1,2)*A(3,3)
  //~ output[0][2]    (input[0][1]*input[1][2]) -
  //~ (input[0][2]*input[1][1]);   //A(1,2)*A(2,3)-A(1,3)*A(2,2)

  //~ output[1][0]    (input[1][2]*input[2][0]) -
  //~ (input[1][0]*input[2][2]);   //A(2,3)*A(3,1)-A(2,1)*A(3,3)
  //~ output[1][1]    (input[0][0]*input[2][2]) -
  //~ (input[0][2]*input[2][0]);   //A(1,1)*A(3,3)-A(1,3)*A(3,1)
  //~ output[1][2]    (input[0][2]*input[1][0]) -
  //~ (input[0][0]*input[1][2]);   //A(1,3)*A(2,1)-A(1,1)*A(2,3)

  //~ output[2][0]    (input[1][0]*input[2][1]) -
  //~ (input[1][1]*input[2][0]);   //A(2,1)*A(3,2)-A(2,2)*A(3,1)
  //~ output[2][1]    (input[0][1]*input[2][0]) -
  //~ (input[0][0]*input[2][1]);   //A(1,2)*A(3,1)-A(1,1)*A(3,2)
  //~ output[2][2]    (input[0][0]*input[1][1]) -
  //~ (input[0][1]*input[1][0]);   //A(1,1)*A(2,2)-A(1,2)*A(2,1)
//~ }



//~ float det3x3(float input[3][3])
//~ {
  //~ // By Simon Rindelaub

  //~ float det;
  //~ det    (input[0][0]*input[1][1]*input[2][2]) +
  //~ (input[0][1]*input[1][2]*input[2][0]) +
```

```
  //~ (input[0][2]*input[1][0]*input[2][1]) -
  //~ (input[2][0]*input[1][1]*input[0][2]) -
  //~ (input[2][1]*input[1][2]*input[0][0]) -
  //~ (input[2][2]*input[1][0]*input[0][1]);

  //~ return det;
//~ }



//~ /**
 //~ * @brief calculate_offset_8
 //~ * @param input
 //~ * Berechnung des Offsetvektors: c    (A^T * A)^T * A^T * b
 //~ * Yc    -(C2/2)      <- Sinusoffset
 //~ * Xc    -(c0/2)      <- Cosinusoffset
 //~ * r    sqrt((C1^2 + C2^2) / 4) - C3)     <- Kreisradius
 //~ */

//~ /**
 //~ * @brief calculate_axis_position - TODO
 //~ *
 //~ * @param inputXvalues - Input values of x-direction of 2D
 //~ *         array. Pointer to 1d array of (size[0] *
 //~ *         size[1]). Entries are read.
 //~ * @param inputYvalues - Input values of y-direction of 2D
 //~ *         array. Pointer to 1d array of (size[0] *
 //~ *         size[1]). Entries are read.
 //~ * @param size - Size of input arrays. Pointer to 1d array
 //~ *         of size two, with first element being x-size and
 //~ *         second being y-size of the input and output arrays.
 //~ *         Entries are read.
 //~ * @param outputValues - Output values containing axis
 //~ *         radius, x- & y-offset. Pointer to 1d array of
 //~ *         length 3. Entries are written.
 //~ */
//~ int calculate_axis_position(float* inputXvalues, float*
//~ inputYvalues, unsigned int* size, float* outputValues)
//~ {
  //~ // Original by Simon Rindelaub (calculate_offset_8,
```

```
//~ // calculate_offset_15)
//~ // Modified to take separate x- & y-arrays of any size and
//~ // optimized

//~ unsigned int k, i, j, index;
//~ float det_B   0;
//~ float A[MAX_X_NUMBER_OF_SENSORS * MAX_Y_NUMBER_OF_SENSORS][
//~ 3];
//~ float B[3][3];
//~ float B_inv[3][3];
//~ float C[3][MAX_X_NUMBER_OF_SENSORS *
//~ MAX_Y_NUMBER_OF_SENSORS];

//~ // Input in die 64x3 Matrix uebertragen (A   (x, y, 1))
//~ k   0;
//~ for(i   0; i < size[0]; i++)
//~ {
  //~ for(j   0; j < size[1]; j++)
  //~ {
    //~ index   j * size[0] + i;

    //~ A[k][0]   inputXvalues[index];
    //~ A[k][1]   inputYvalues[index];
    //~ A[k][2]   1;

    //~ k++;
  //~ }
//~ }

//~ // B   A^T * A
//~ for(i   0; i < 3; i++)
//~ {
  //~ for(j   0; j < 3; j++)
  //~ {
    //~ B[i][j]   0;

    //~ for(k   0; k < 64; k++)
    //~ {
      //~ B[i][j] +  A[k][j] * A[k][i];
    //~ }
```

```
    //~ }
  //~ }

  //~ det_B    det3x3(B);

  //~ if(det_B !   0)
  //~ {
    //~ // B_inv    B^-1
    //~ adjungieren(B, B_inv);

    //~ for (i    0; i < 3; i++)
    //~ {
      //~ for (j    0; j < 3; j++)
      //~ {
        //~ B_inv[i][j] /   det_B;
      //~ }
    //~ }

    //~ // C    B_inv * A^T
    //~ for(i    0; i < 3; i++)
    //~ {
      //~ for(j    0; j < 64; j++)
      //~ {
        //~ C[i][j]    0;

        //~ for(k    0; k < 3; k++)
        //~ {
          //~ C[i][j] +   B_inv[i][k] * A[j][k];
        //~ }
      //~ }
    //~ }

    //~ // c    C * b
    //~ for(k    0; k < 3; k++)
    //~ {
      //~ outputValues[k]    0;

      //~ for(j    0; j < 64; j++)
      //~ {
        //~ outputValues[k] +   C[k][j] * (- pow(A[j][0], 2) -
```

```
        //~ pow(A[j][1], 2));
      //~ }
    //~ }

    //~ outputValues[2]    sqrtf((powf(outputValues[0], 2) + powf(
    //~ outputValues[1], 2)) / 4 - outputValues[2]);
    //~ //^ value radius
    //~ outputValues[0] /   outputValues[0]; // axis x-offset
    //~ outputValues[1] /   outputValues[1]; // axis y-offset

    //~ return 0;
  //~ }
  //~ else
  //~ {
    //~ return -1;
  //~ }
//~ }
```

src/ui.c

```
#include <gtk/gtk.h>
#include <GL/gl.h>
#include <math.h>

#include "global.h"


//                                                                    //
//-------------------------OpenGL Functions---------------------------//
//                                                                    //

GLuint g_shaderProgramVectorField;
GLuint g_vao;
//^ vertex array object: stores vertex attribute calls of a VBO
GLuint g_vbo;
//^ vertex buffer object: buffer to store verticles and transfer
//^ them to the GPU
GLfloat g_vertices[12 * MAX_X_NUMBER_OF_VIRTUAL_SENSORS *
MAX_Y_NUMBER_OF_VIRTUAL_SENSORS];
//^ number of entries   nPointsPerVector * verticesPerPoint *
//^ nXsensors * nYsensors

// vertex shader source code
static const char *g_vertexShaderSource
"#version 330\n"
"in vec3 position;\n"
"in vec3 color;\n"
"out vec4 vColor;\n"
"out float vSize;\n"
"void main()\n"
"{\n"
"  gl_Position  vec4(position, 1.0);\n" //gl_Position is predef.
"  vColor  vec4(color, 1.0);\n"
"}\n";

// fragment shader source code
static const char *g_fragmentShaderSource
"#version 330\n"
"in vec4 vColor;\n"
```

```
"void main() {\n"
"  gl_FragColor  vColor;\n" //gl_FragColor is predefined
"}\n";



/**
 * @brief  gl_init_buffers - generate and initialize vertex
 *         attribute array and vertex buffer object and define buffer structure.
 *
 * @param  (GLuint *)vao - Vertex Array Object.
 *         Pointer to variable. Variable is written.
 * @param  (GLuint *)vbo - Vertex Buffer Object.
 *         Pointer to variable. Variable is written.
 * @param  (float *)vertices - Vertices array.
 *         Pointer to 1D array of size 8 * verticesSize. Entries
 *         are read.
 * @param  (int)verticesSize - Size of vertices array in Bytes.
 *         Variable to indicate size of vertices array in Bytes.
 *         Variable is read.
 * @param  (GLuint *)shaderProgram - Shader program to use for
 *         drawing.
 *         Pointer to variable. Variable is read.
 *
 * Uses: GL/gl.h
 * Reads: -
 * Writes: -
 */
static void gl_init_buffers(GLuint *vao, GLuint *vbo, float
*vertices, int verticesSize, GLuint *shaderProgram)
{
  //Modified from Source:
  //https://stackoverflow.com/questions/42231698 - May 2020
  //Original Author: Toan Tran

  glGenVertexArrays(1, vao);
  //^ generate vertex array object to store vertex buffer
  //^ attributes
  glGenBuffers(1, vbo); // generate vertex buffer object
  glBindVertexArray(*vao); // bind vao for openGL to use
```

```
  glBindBuffer(GL_ARRAY_BUFFER, *vbo);
  //^ bind vbo for openGL to use
  glBufferData(GL_ARRAY_BUFFER, verticesSize, vertices,
 GL_DYNAMIC_DRAW); // copy vertices to GPU

  // set the vertex attributes pointers and store in vao
  GLuint attribLocation;
  // position attribute
  attribLocation   glGetAttribLocation(*shaderProgram,
  "position\0");
  if(attribLocation    -1) g_warning("gl_init_buffers_es_-_\
__\"position\"_is_not_an_active_attribute");
  glVertexAttribPointer(attribLocation, 3, GL_FLOAT, GL_FALSE,
  6 * sizeof(GLfloat), (GLvoid*)0);
  glEnableVertexAttribArray(attribLocation);
  // color attribute
    attribLocation   glGetAttribLocation(*shaderProgram,
    "color\0");
  if(attribLocation    -1) g_warning("gl_init_buffers_es_-_\
__\"color\"_is_not_an_active_attribute");
    glVertexAttribPointer(attribLocation, 3, GL_FLOAT, GL_FALSE,
    6 * sizeof(GLfloat), (GLvoid*)(3 * sizeof(GLfloat)));
    glEnableVertexAttribArray(attribLocation);
}



/**
 * @brief  gl_create_shader - Create and compile shader of given
 *         type with given source code.
 *
 * @param  (int)type - Shader Type
 *         enum type of shader [GL_VERTEX_SHADER,
 *         GL_FRAGMENT_SHADER]. Variable is read.
 * @param  (const char *)source - GLSL source code
 *         Pointer to 1D array. Entries are read.
 *
 * @return shader - The compiled shader.
 *
 * Uses: GL/gl.h
```

```
 * Reads: -
 * Writes: -
 */
static GLuint gl_create_shader(int type, const char *source)
{
  //Source:
  //https://gitlab.gnome.org/GNOME/gtk/blob/master/demos/gtk-demo
  ///glarea.c - May 2020
  //Original Author: Matthias Clasen

  GLuint shader;
  int status;

  shader = glCreateShader(type);
  glShaderSource(shader, 1, &source, NULL);
  glCompileShader(shader);

  glGetShaderiv(shader, GL_COMPILE_STATUS, &status);
  if(status == GL_FALSE)
  {
    int log_len;
    char *buffer;

    glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &log_len);

    buffer = g_malloc(log_len + 1);
    glGetShaderInfoLog(shader, log_len, NULL, buffer);

    g_warning("gl_create_shader - Compile failure in %s shader: \
\n    %s", type == GL_VERTEX_SHADER ? "vertex" : "fragment", buffer
    );

    g_free(buffer);

    glDeleteShader(shader);

    return 0;
  }

  return shader;
```

```
}



/**
 * @brief  gl_init_shaders - Create and compile given shaders and
 *         create and link shaders to given shader program.
 *
 * @param  (const char *)vertexShaderSource - GLSL source code of
 *         vertex shader
 *         Pointer to 1D array. Entries are read.
 * @param  (const char *)fragmentShaderSource - GLSL source code
 *         of fragment shader
 *         Pointer to 1D array. Entries are read.
 * @param  (GLuint *)shaderProgram - Shader program.
 *         Pointer to variable. Variable is written.
 *
 * Uses: GL/gl.h
 * Reads: -
 * Writes: -
 */
static void gl_init_shaders(const char *vertexShaderSource,
const char *fragmentShaderSource, GLuint *shaderProgram)
{
  //Source:
  //https://gitlab.gnome.org/GNOME/gtk/blob/master/demos/gtk-demo
  ///glarea.c - May 2020
  //Original Author: Matthias Clasen

  GLuint vertex, fragment;
  int status;

  vertex   gl_create_shader(GL_VERTEX_SHADER, vertexShaderSource
  );
  if(vertex    0)
  {
    g_warning("gl_init_shaders - Couldn't create vertex shader");
    *shaderProgram   0;
    return;
  }
```

```
fragment = gl_create_shader(GL_FRAGMENT_SHADER,
fragmentShaderSource);
if(fragment == 0)
{
  g_warning("gl_init_shaders - Couldn't create fragment shader"
  );
  glDeleteShader(vertex);
  glDeleteShader(fragment);
  *shaderProgram = 0;
  return;
}

*shaderProgram = glCreateProgram();
glAttachShader(*shaderProgram, vertex);
glAttachShader(*shaderProgram, fragment);
glLinkProgram(*shaderProgram);

glGetProgramiv(*shaderProgram, GL_LINK_STATUS, &status);
if(status == GL_FALSE) // linking failed
{
  int log_len;
  char *buffer;
  glGetProgramiv(*shaderProgram, GL_INFO_LOG_LENGTH, &log_len);
  buffer = g_malloc(log_len + 1);
  glGetProgramInfoLog(*shaderProgram, log_len, NULL, buffer);
  g_warning("gl_init_shaders - Linking failure: %s", buffer);
  g_free(buffer);

  glDeleteProgram(*shaderProgram);
  shaderProgram = 0;
}
else // linking succeeded
{
  glDetachShader(*shaderProgram, vertex);
  glDetachShader(*shaderProgram, fragment);
}

glDeleteShader(vertex);
glDeleteShader(fragment);
```

```
}



/**
 * @brief   gl_draw_vector_field - Draw a Vector Field
 *
 * @param   (GLuint *)vao - Vertex Array Object.
 *          Pointer to variable. Variable is read.
 * @param   (GLuint *)vbo - Vertex Buffer Object.
 *          Pointer to variable. Variable is read.
 * @param   (GLuint *)shaderProgram - Shader program.
 *          Pointer to variable. Variable is read.
 * @param   (unsigned int *)pixelsSize - Size of drawing area in
 *          Pixels
 *          Pointer to 1D array of size two, with first element
 *          being x-size and second being y-size of the drawing
 *          area. Entries are read.
 * @param   (float *)vertices - Vertices array.
 *          Pointer to 1D array of at least size 12 *
 *          (valuesSize[0] * valuesSize[1]). Entries are written.
 * @param   (int)verticesSize - Size of vertices array in Bytes.
 *          Variable to indicate size of vertices array in Bytes.
 *          Variable is read.
 * @param   (float *)xValues - Values of x-direction of 2D array.
 *          Pointer to 1D array of size (valuesSize[0] *
 *          valuesSize[1]) with entries normalized to 1. Entries
 *          are read.
 * @param   (float *)yValues - Values of y-direction of 2D array.
 *          Pointer to 1D array of size (valuesSize[0] *
 *          valuesSize[1]) with entries normalized to 1. Entries
 *          are read.
 * @param   (unsigned int *)valuesSize - Size of *Values arrays.
 *          Pointer to 1D array of size two, with first element
 *          being x-size and second being y-size of the input and
 *          output arrays. Entries are read.
 *
 * Uses: GL/gl.h, math.h
 * Reads: -
 * Writes: -
```

```
 */
static void gl_draw_vector_field(GLuint *vao, GLuint *vbo, GLuint
*shaderProgram, unsigned int *pixelsSize, float *vertices, int
verticesSize, float *xValues, float *yValues, unsigned int
*valuesSize)
{
  if(vao     NULL
  || vbo     NULL
  || shaderProgram     NULL
  || pixelsSize     NULL
  || vertices     NULL
  || xValues     NULL
  || yValues     NULL
  || valuesSize     NULL) return;
  if(verticesSize / sizeof(vertices[0]) < 12 * valuesSize[0] *
  valuesSize[1]) return;

  unsigned int xIdx, yIdx, vertexIndex, valueIndex, indexHelper;
  float length;
  float vertexColor[3]; //rgb
  float xDiff    2.0f / (float)(valuesSize[0] + 1);
  //^ vector spacing in x-direction
  float yDiff    2.0f / (float)(valuesSize[1] + 1);
  //^ vector spacing in y-direction

  // Calculate Vector Vertices
  for(yIdx    0; yIdx < valuesSize[1]; yIdx++)
  {
    indexHelper    yIdx * valuesSize[0];
    for(xIdx    0; xIdx < valuesSize[0]; xIdx++)
    {
      valueIndex    indexHelper + xIdx; // vector value index
      vertexIndex    12 * valueIndex;
      //^ vertex index; there are 12 vertices per vector

      // calculate vector color
      length    sqrtf(powf(xValues[valueIndex], 2) + powf(
      yValues[valueIndex], 2));

      vertexColor[0]    2.0f * length; // red
```

```
    vertexColor[1]   0.0f; // green
    vertexColor[2]   2.0f * (1.0f - length); // blue

    if(vertexColor[0] > 1.0f) vertexColor[0]   1.0f;
    if(vertexColor[2] > 1.0f) vertexColor[2]   1.0f;

    // set start vertices of vector
    vertices[vertexIndex + 0]   xDiff * (float)(xIdx + 1) -
    1.0f; // x
    vertices[vertexIndex + 1]   yDiff * (float)(yIdx + 1) -
    1.0f; // y
    vertices[vertexIndex + 2]   0.0f; // z
    vertices[vertexIndex + 3]   vertexColor[0]; // red
    vertices[vertexIndex + 4]   vertexColor[1]; // green
    vertices[vertexIndex + 5]   vertexColor[2]; // blue
    // set end vertices of vector
    vertices[vertexIndex + 6]   vertices[vertexIndex + 0] +
    xDiff * xValues[valueIndex]; // x
    vertices[vertexIndex + 7]   vertices[vertexIndex + 1] +
    yDiff * yValues[valueIndex]; // y
    vertices[vertexIndex + 8]   0.0f; // z;
    vertices[vertexIndex + 9]   vertexColor[0]; // red
    vertices[vertexIndex + 10]   vertexColor[1]; // green
    vertices[vertexIndex + 11]   vertexColor[2]; // blue
  }
}

// set required line width
if(pixelsSize[0] > pixelsSize[1])
{
  glLineWidth((float)pixelsSize[0] / 350.0f + 1.0f);
}
else
{
  glLineWidth((float)pixelsSize[1] / 350.0f + 1.0f);
}

glBindBuffer(GL_ARRAY_BUFFER, *vbo);
//^ bind VBO to GL_ARRAY_BUFFER target
glBufferData(GL_ARRAY_BUFFER, verticesSize, vertices,
```

```
  GL_DYNAMIC_DRAW); // copy vertex data to dynamic GPU buffer

  glUseProgram(*shaderProgram);
  //^ select shader program for use in shader and draw calls
  glBindVertexArray(*vao);
  //^ bind VAO to use VBO's buffer structure for drawing

  glDrawArrays(GL_LINES, 0, 2 * valuesSize[0] * valuesSize[1]);
  //^ draw from GPU buffer
}



/**
 * @brief  gl_draw_heatmap - Draw a Heatmap
 *
 * @param  (GLuint *)vao - Vertex Array Object.
 *         Pointer to variable. Variable is read.
 * @param  (GLuint *)vbo - Vertex Buffer Object.
 *         Pointer to variable. Variable is read.
 * @param  (GLuint *)shaderProgram - Shader program.
 *         Pointer to variable. Variable is read.
 * @param  (unsigned int *)pixelsSize - Size of drawing area in
 *         Pixels
 *         Pointer to 1D array of size two, with first element
 *         being x-size and second being y-size of the drawing
 *         area. Entries are read.
 * @param  (float *)vertices - Vertices array.
 *         Pointer to 1D array of at least size 12 *
 *         (valuesSize[0] * valuesSize[1]). Entries are written.
 * @param  (int)verticesSize - Size of vertices array in Bytes.
 *         Variable to indicate size of vertices array in Bytes.
 *         Variable is read.
 * @param  (float *)values - Values of of 2D array.
 *         Pointer to 1D array of size (valuesSize[0] *
 *         valuesSize[1]) with entries normalized to 1. Entries
 *         are read.
 * @param  (unsigned int *)valuesSize - Size of values array.
 *         Pointer to 1D array of size two, with first element
 *         being x-size and second being y-size of the input and
```

```
 *          output arrays. Entries are read.
 *
 * Uses: GL/gl.h, math.h
 * Reads: -
 * Writes: -
 */
static void gl_draw_heatmap(GLuint *vao, GLuint *vbo, GLuint
*shaderProgram, unsigned int *pixelsSize, float *vertices, int
verticesSize, float *values, unsigned int *valuesSize)
{
  if(vao     NULL
  || vbo     NULL
  || shaderProgram     NULL
  || pixelsSize     NULL
  || vertices     NULL
  || values     NULL
  || valuesSize     NULL) return;
  if(verticesSize / sizeof(vertices[0]) < 6 * valuesSize[0] *
  valuesSize[1]) return;

  unsigned int xIdx, yIdx, vertexIndex, valueIndex, indexHelper;
  float vertexColor[3]; //rgb
  float xDiff    2.0f / (float)valuesSize[0];
  //^ vector spacing in x-direction
  float yDiff    2.0f / (float)valuesSize[1];
  //^ vector spacing in y-direction

  // Calculate Vector Vertices
  for(yIdx    0; yIdx < valuesSize[1]; yIdx++)
  {
    indexHelper    yIdx * valuesSize[0];
    for(xIdx    0; xIdx < valuesSize[0]; xIdx++)
    {
      valueIndex    indexHelper + xIdx; // point value index
      vertexIndex    6 * valueIndex;
      //^ point index; there are 6 vertices per point

      // calculate point color
      vertexColor[0]    1.0f + values[valueIndex]; // red
      vertexColor[1]    1.0f - fabs(values[valueIndex]); // green
```

```
      vertexColor[2]    1.0f - values[valueIndex]; // blue

      if(vertexColor[0] > 1.0f) vertexColor[0]    1.0f;
      if(vertexColor[2] > 1.0f) vertexColor[2]    1.0f;

      // set point vertices
      vertices[vertexIndex + 0]    xDiff * ((float)xIdx + 0.5f) -
      1.0f; // x
      vertices[vertexIndex + 1]    yDiff * ((float)yIdx + 0.5f) -
      1.0f; // y
      vertices[vertexIndex + 2]    0.0f; // z
      vertices[vertexIndex + 3]    vertexColor[0]; // red
      vertices[vertexIndex + 4]    vertexColor[1]; // green
      vertices[vertexIndex + 5]    vertexColor[2]; // blue
    }
  }

  // set required point size
  if(pixelsSize[0] > pixelsSize[1])
  {
    if(valuesSize[0] > valuesSize[1])
      glPointSize((float)pixelsSize[0] / (float)valuesSize[1] +
      1.0f);
    else
      glPointSize((float)pixelsSize[0] / (float)valuesSize[0] +
      1.0f);
  }
  else
  {
    if(valuesSize[0] > valuesSize[1])
      glPointSize((float)pixelsSize[1] / (float)valuesSize[1] +
      1.0f);
    else
    {
      glPointSize((float)pixelsSize[1] / (float)valuesSize[0] +
      1.0f);
    }
  }

  glBindBuffer(GL_ARRAY_BUFFER, *vbo);
```

```
  //^ bind VBO to GL_ARRAY_BUFFER target
  glBufferData(GL_ARRAY_BUFFER, verticesSize, vertices,
 GL_DYNAMIC_DRAW); // copy vertex data to dynamic GPU buffer

  glUseProgram(*shaderProgram);
  //^ select shader program for use in shader and draw calls
  glBindVertexArray(*vao);
  //^ bind VAO to use VBO's buffer structure for drawing

  glDrawArrays(GL_POINTS, 0, valuesSize[0] * valuesSize[1]);
  //^ draw from GPU buffer
}



/**
 * @brief  gl_draw_vector - Draw a Vector
 *
 * @param  (GLuint *)vao - Vertex Array Object.
 *         Pointer to variable. Variable is read.
 * @param  (GLuint *)vbo - Vertex Buffer Object.
 *         Pointer to variable. Variable is read.
 * @param  (GLuint *)shaderProgram - Shader program.
 *         Pointer to variable. Variable is read.
 * @param  (unsigned int *)pixelsSize - Size of drawing area in
 *         Pixels.
 *         Pointer to 1D array of size two, with first element
 *         being x-size and second being y-size of the drawing
 *         area. Entries are read.
 * @param  (float *)vertices - Vertices array.
 *         Pointer to 1D array of at least size 6 *
 *         (valuesSize[0] * valuesSize[1]). Entries are written.
 * @param  (int)verticesSize - Size of vertices array in Bytes.
 *         Variable to indicate size of vertices array in Bytes.
 *         Variable is read.
 * @param  (float *)values - Values of vector.
 *         Pointer to 1D array of size 3 containing in order x- &
 *         y-starting position in the range of [-1, 1] and angle
 *         in degrees. Entries are read.
 *
```

```
 * Uses: GL/gl.h, math.h
 * Reads: -
 * Writes: -
 */
static void gl_draw_vector(GLuint *vao, GLuint *vbo, GLuint
*shaderProgram, unsigned int *pixelsSize, float *vertices, int
verticesSize, float *values) // xPos, yPos, angle [rad]
{
  if(vao     NULL
  || vbo     NULL
  || shaderProgram     NULL
  || pixelsSize     NULL
  || vertices     NULL
  || values     NULL) return;
  if(verticesSize / sizeof(vertices[0]) < 12) return;

  float xDelta, yDelta, offset, angle    values[2];
  float vertexColor[3]; //rgb

  // calculate vector color
  offset    sqrtf(powf(values[0], 2) + powf(values[1], 2));

  vertexColor[0]    2.0f * offset; // red
  vertexColor[1]    0.0f; // green
  vertexColor[2]    2.0f * (1.0f - offset); // blue

  if(vertexColor[0] > 1.0f) vertexColor[0]    1.0f;
  if(vertexColor[2] > 1.0f) vertexColor[2]    1.0f;

  // calculate vector
  if(angle < PI_1_4) // PI/4 / 45 degree
  {
    // Q1
    // 0 <  tan(angle) < 1
    xDelta    2.0f;
    yDelta    2.0f * tanf(angle);
  }
  else if(angle < PI_1_2) // PI/2 / 90 degree
  {
    // Q1
```

```
    // 1 <  tan(angle) < inf
    xDelta   2.0f / tanf(angle);
    yDelta   2.0f;
  }
  else if(angle < PI_3_4) // 3PI/4 / 135 degree
  {
    // Q2
    // -inf < tan(angle) < -1
    xDelta   2.0f / tanf(angle);
    yDelta   2.0f;
  }
  else if(angle < PI) // PI / 180 degree
  {
    // Q2
    // -1 <  tan(angle) < 0
    xDelta   - 2.0f;
    yDelta   - 2.0f * tanf(angle);
  }
  else if(angle < PI_5_4) // 5PI/4 / 225 degree
  {
    // Q3
    // 0 <  tan(angle) < 1
    xDelta   - 2.0f;
    yDelta   - 2.0f * tanf(angle);
  }
  else if(angle < PI_3_2) // 3PI/2 / 270 degree
  {
    // Q3
    // 1 <  tan(angle) < inf
    xDelta   - 2.0f / tanf(angle);
    yDelta   - 2.0f;
  }
  else if(angle < PI_7_4) // 7PI/4 / 315 degree
  {
    // Q4
    // -inf < tan(angle) < -1
    xDelta   - 2.0f / tanf(angle);
    yDelta   - 2.0f;
  }
  else // 2PI / 360 degree
```

```
{
  // Q4
  // -1 < tan(angle) < 0
  xDelta  2.0f;
  yDelta  2.0f * tanf(angle);
}

// set start vertices of vector
vertices[0]   values[0]; // x
vertices[1]   values[1]; // y
vertices[2]   0.0f; // z
vertices[3]   vertexColor[0]; // r
vertices[4]   vertexColor[1]; // g
vertices[5]   vertexColor[2]; // b
// set end vertices of vector
vertices[6]   vertices[0] + xDelta; // x
vertices[7]   vertices[1] + yDelta; // y
vertices[8]   0.0f; // z
vertices[9]   vertexColor[0]; // r
vertices[10]   vertexColor[1]; // g
vertices[11]   vertexColor[2]; // b

// set required line width
if(pixelsSize[0] > pixelsSize[1])
{
  glLineWidth((float)pixelsSize[0] / 300.0f + 1.0f);
}
else
{
  glLineWidth((float)pixelsSize[1] / 300.0f + 1.0f);
}

glBindBuffer(GL_ARRAY_BUFFER, *vbo);
//^ bind VBO to GL_ARRAY_BUFFER target
glBufferData(GL_ARRAY_BUFFER, verticesSize, vertices,
GL_DYNAMIC_DRAW); // copy vertex data to dynamic GPU buffer

glUseProgram(*shaderProgram);
//^ select shader program for use in shader and draw calls
glBindVertexArray(*vao);
```

```
  //^ bind VAO to use VBO's buffer structure for drawing

  glDrawArrays(GL_LINES, 0, 2); // draw from GPU buffer
}



//                                                                    //
//-------------------------GLArea Functions---------------------------//
//                                                                    //

/**
 * @brief   on_render - On GTK render signal draw selected
 *          animation
 *
 * @param   (GtkGLArea *)area - OpenGL drawing area of the GUI.
 *          Pointer to variable. Variable is read.
 * @param   (GdkGLContext *)context - GTK context of the drawing
 *          area.
 *          Pointer to variable. Variable is read.
 * @param   (callback_t *)userData->(uiData_t *)flags->(unsigned
 *          int *)pixelsSize - Size of drawing area in Pixels.
 *          Pointer to 1D array of size two, with first element
 *          being x-size and second being y-size of the drawing
 *          area. Entries are read.
 * @param   (callback_t *)userData->(uiData_t *)flags->(unsigned
 *          int)animationMode - Animation mode.
 *          Variable determines the animation mode the OpenGL area
 *          displays. Variable is read.
 * @param   (callback_t *)userData->(uiData_t *)flags->(unsigned
 *          int *)valuesSize - Size of *Values arrays.
 *          Pointer to 1D array of size two, with first element
 *          being x-size and second being y-size of the input and
 *          output arrays. Entries are read.
 * @param   (callback_t *)userData->(uiData_t *)flags->(float *)
 *          xValues - Depending on the animation mode this array
 *          contains either the values of the x-direction of 2D
 *          array in range [-1, 1] or in order x- & y-starting
 *          position in the range of [-1, 1] and angle in degrees
 *          in range [0, 2*PI]. Pointer to 1D array of size
```

```
 *          (valuesSize[0] * valuesSize[1]). Entries are read.
 * @param   (callback_t *)userData->(uiData_t *)flags->(float *)
 *          yValues - Values of y-direction of 2D array.
 *          Pointer to 1D array of size (valuesSize[0] *
 *          valuesSize[1]) with entries normalized to 1. Entries
 *          are read.
 *
 * Uses: GL/gl.h, global.h, gtk/gtk.h
 * Reads: -
 * Writes: -
 */
static gboolean on_render(GtkGLArea *area, GdkGLContext *context,
callback_t *userData)
{
  if (gtk_gl_area_get_error(area) != NULL) return FALSE;

  gtk_widget_set_size_request(GTK_WIDGET(area), userData->flags->
  pixelsSize[0], userData->flags->pixelsSize[1]);

  glClearColor(1.0, 1.0, 1.0, 0.0); // set color
  glClear(GL_COLOR_BUFFER_BIT);
  //^ clear area with previously selected color

  // Draw
  switch(userData->flags->animationMode)
  {
    case UI_ANIMATION_MODE_VECTOR_FIELD:
      gl_draw_vector_field(&g_vao, &g_vbo,
      &g_shaderProgramVectorField, userData->flags->pixelsSize,
      g_vertices, sizeof(g_vertices), userData->flags->xValues,
      userData->flags->yValues, userData->flags->valuesSize);
      break;
    case UI_ANIMATION_MODE_HEATMAP:
      gl_draw_heatmap(&g_vao, &g_vbo,
      &g_shaderProgramVectorField, userData->flags->pixelsSize,
      g_vertices, sizeof(g_vertices), userData->flags->xValues,
      userData->flags->valuesSize);
      break;
    case UI_ANIMATION_MODE_VECTOR:
      gl_draw_vector(&g_vao, &g_vbo, &g_shaderProgramVectorField,
```

```
        userData->flags->pixelsSize, g_vertices,
        sizeof(g_vertices), userData->flags->xValues);
        break;
      default:
        break;
    }

    userData->flags->programControl |= 1 <<
    UI_PROGRAM_CONTROL_REFRESH_OFFSET;

    return TRUE;
}


/**
 * @brief  on_realize - On GTK realize signal configure the
 *         animation area and setup OpenGL resources
 *
 * @param  (GtkGLArea *)area - OpenGL drawing area of the GUI.
 *         Pointer to variable. Variable is read.
 * @param  (callback_t *)userData is unused.
 *
 * Uses: GL/gl.h, gtk/gtk.h
 * Reads: -
 * Writes: -
 */
static void on_realize(GtkGLArea *area, callback_t *userData)
{
    gtk_gl_area_make_current(area);
    if(gtk_gl_area_get_error(area) != NULL) return;

    gl_init_shaders(g_vertexShaderSource, g_fragmentShaderSource,
    &g_shaderProgramVectorField);
    gl_init_buffers(&g_vao, &g_vbo, g_vertices, sizeof(g_vertices),
    &g_shaderProgramVectorField);

    // Print version info:
    const GLubyte* renderer = glGetString(GL_RENDERER);
    const GLubyte* version = glGetString(GL_VERSION);
    printf("Renderer: %s\n", renderer);
```

```
  printf("OpenGL version supported %s\n", version);

  // set update signal source of render area
  GdkGLContext *glcontext = gtk_gl_area_get_context(area);
  GdkWindow *glwindow = gdk_gl_context_get_window(glcontext);
  GdkFrameClock *frame_clock = gdk_window_get_frame_clock(
  glwindow);
  g_signal_connect_swapped(frame_clock, "update", G_CALLBACK(
  gtk_gl_area_queue_render), area);
  gdk_frame_clock_begin_updating(frame_clock);
}



/**
 * @brief  on_unrealize - On GTK unrealize free OpenGL resources
 *
 * @param  (GtkGLArea *)area - OpenGL drawing area of the GUI.
 *         Pointer to variable. Variable is read.
 * @param  (callback_t *)userData is unused.
 *
 * Uses: GL/gl.h, gtk/gtk.h
 * Reads: -
 * Writes: -
 */
static void on_unrealize(GtkGLArea *area, callback_t *userData)
{
  glDeleteBuffers(1, &g_vao);
  glDeleteBuffers(1, &g_vbo);
}



//                                                              //
//-------------------------Control Functions--------------------//
//                                                              //

static void on_window_destroy(GtkWindow *window, callback_t
*userData)
{
  userData->flags->programControl |= 1 <<
```

```
  UI_PROGRAM_CONTROL_EXIT_OFFSET;
  gtk_main_quit();
}



static void on_buttonLoadConfiguration_clicked(GtkWidget *widget,
callback_t *userData)
{
  userData->flags->programControl |= 1 <<
  UI_PROGRAM_CONTROL_LOAD_CONFIGURATION_OFFSET;
}



static void on_buttonLoadCoefficients_clicked(GtkWidget *widget,
callback_t *userData)
{
  userData->flags->programControl |= 1 <<
  UI_PROGRAM_CONTROL_LOAD_COEFFICIENTS_OFFSET;
}



static void on_comboBoxTextViewSelection_changed(GtkWidget
*widget, callback_t *userData)
{
  userData->flags->animationControl
    (userData->flags->animationControl &
    ~UI_ANIMATION_CONTROL_VIEW_SELECTION_MASK) |
    //^ clear field
    ((gtk_combo_box_get_active(GTK_COMBO_BOX(widget))
    //^ get value ...
    & (UI_ANIMATION_CONTROL_VIEW_SELECTION_MASK >>
    UI_ANIMATION_CONTROL_VIEW_SELECTION_OFFSET))
    //^ ... trim to size ...
    << UI_ANIMATION_CONTROL_VIEW_SELECTION_OFFSET);
    //^ ... and shift to position
}
```

```
static void on_toggleButtonInterpolation_clicked(GtkWidget
*widget, callback_t *userData)
{
  userData->flags->animationControl
    (userData->flags->animationControl &
    ~UI_ANIMATION_CONTROL_INTERPOLATION_MASK) |
    //^ clear field
    (gtk_toggle_button_get_active(GTK_TOGGLE_BUTTON(widget))
    //^ get value ...
    << UI_ANIMATION_CONTROL_INTERPOLATION_OFFSET);
    //^ ... and shift to position
}



//                                                              //
//-------------------------------GTK Main-------------------------------//
//                                                              //

void *ui_thread(void *flags)
{
  // initialize GTK
  if (!gtk_init_check(NULL, NULL))
  {
    ((uiData_t *)flags)->programControl |  1 <<
    UI_PROGRAM_CONTROL_EXIT_OFFSET;
    printf("Error in ui_thread: Could not initialize GTK.\n");
    return NULL;
  }

  callback_t *userData   g_slice_new(callback_t);
  userData->flags    (uiData_t *)flags;

  // configure objects
  GtkWidget *window    gtk_window_new(GTK_WINDOW_TOPLEVEL);
  gtk_window_set_title(GTK_WINDOW(window), "ISAR Viewer");
  gtk_window_set_resizable(GTK_WINDOW(window), FALSE);
  GtkWidget *boxMain    gtk_box_new(GTK_ORIENTATION_HORIZONTAL, 0
```

```
);
GtkWidget *boxDiv    gtk_box_new(GTK_ORIENTATION_VERTICAL, 20);

GtkWidget *glarea    gtk_gl_area_new();
gtk_widget_set_size_request(glarea, userData->flags->
pixelsSize[0], userData->flags->pixelsSize[1]);

GtkWidget *boxParameter    gtk_box_new(GTK_ORIENTATION_VERTICAL,
5);
GtkWidget *labelParameter    gtk_label_new("Parameter");
GtkWidget *buttonLoadConfiguration    gtk_button_new_with_label(
"Lade Konfiguration");
GtkWidget *buttonLoadCoefficients    gtk_button_new_with_label(
"Lade Koeffizienten");

GtkWidget *boxAnimation    gtk_box_new(GTK_ORIENTATION_VERTICAL,
5);
GtkWidget *labelAnimation    gtk_label_new("Animation");
GtkWidget *comboBoxTextViewSelection    gtk_combo_box_text_new(
);
gtk_combo_box_text_append(GTK_COMBO_BOX_TEXT(
comboBoxTextViewSelection),
"UI_ANIMATION_SELECTION_DATA_DIRECT", "Rohdaten");
gtk_combo_box_text_append(GTK_COMBO_BOX_TEXT(
comboBoxTextViewSelection),
"UI_ANIMATION_SELECTION_DATA_DIRECT_X", "Rohdaten X");
gtk_combo_box_text_append(GTK_COMBO_BOX_TEXT(
comboBoxTextViewSelection),
"UI_ANIMATION_SELECTION_DATA_DIRECT_Y", "Rohdaten Y");
gtk_combo_box_text_append(GTK_COMBO_BOX_TEXT(
comboBoxTextViewSelection), "UI_ANIMATION_SELECTION_ANGLE",
"Winkel Demo");
// EXTENTION: gtk_combo_box_text_append(
//             GTK_COMBO_BOX_TEXT(comboBoxTextViewSelection),
//             "UI_ANIMATION_SELECTION_MY_MACRO", "My View");
GtkWidget *toggleButtonInterpolation
gtk_check_button_new_with_label("Interpolation");

// assemble window
gtk_container_add(GTK_CONTAINER(window), boxMain);
```

```
    gtk_box_pack_start(GTK_BOX(boxMain), glarea, 0, 0, 0);
    gtk_box_pack_start(GTK_BOX(boxMain), boxDiv, 1, 1, 20);

  gtk_container_add(GTK_CONTAINER(boxDiv), boxParameter);
  gtk_container_add(GTK_CONTAINER(boxParameter), labelParameter);
  gtk_container_add(GTK_CONTAINER(boxParameter),
  buttonLoadConfiguration);
  gtk_container_add(GTK_CONTAINER(boxParameter),
  buttonLoadCoefficients);

  gtk_container_add(GTK_CONTAINER(boxDiv), boxAnimation);
  gtk_container_add(GTK_CONTAINER(boxAnimation), labelAnimation);
  gtk_container_add(GTK_CONTAINER(boxAnimation),
  comboBoxTextViewSelection);
  gtk_container_add(GTK_CONTAINER(boxAnimation),
  toggleButtonInterpolation);

  // connect signals to callback functions
  g_signal_connect(window, "destroy", G_CALLBACK(
  on_window_destroy), userData);
  g_signal_connect(window, "unrealize", G_CALLBACK(on_unrealize),
  userData);
  g_signal_connect(glarea, "realize", G_CALLBACK(on_realize),
  userData);
  g_signal_connect(glarea, "render", G_CALLBACK(on_render),
  userData);
  g_signal_connect(buttonLoadConfiguration, "clicked",
  G_CALLBACK(on_buttonLoadConfiguration_clicked), userData);
  g_signal_connect(buttonLoadCoefficients, "clicked",
  G_CALLBACK(on_buttonLoadCoefficients_clicked), userData);
  g_signal_connect(comboBoxTextViewSelection, "changed",
  G_CALLBACK(on_comboBoxTextViewSelection_changed), userData);
  g_signal_connect(toggleButtonInterpolation, "clicked",
  G_CALLBACK(on_toggleButtonInterpolation_clicked), userData);

  // show ui
  gtk_widget_show_all(window);

  // enter ui event loop
```

```
    gtk_main();

    return NULL;
}
```

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort Datum Unterschrift im Original