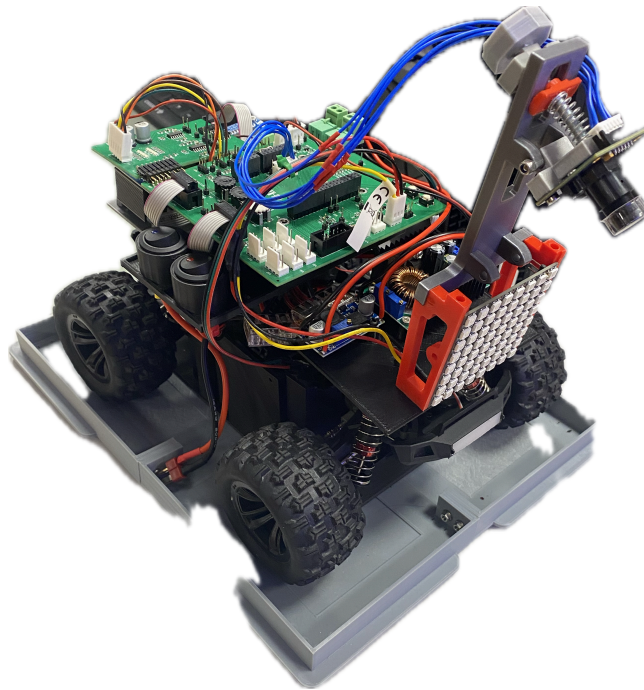


## NXPCAR DOKUMENTATION

---



<b>Subjekt</b>	Microcontroller und Sensorik
<b>Studiengang (Fakultät)</b>	Angewandte Informatik - Studienrichtung Embedded Systems; Internet of Things
<b>Betreuer/Dozent</b>	Prof. Dr.-Ing. Markus Barkowsky
<b>Teammitglieder (Matrikelnr.)</b>	Adham Beshr (22209885)
	Ismail Shah Bin Iman Shah (12306210)
<b>Abgabedatum</b>	02.02.2025
<b>Institution</b>	THD - Technische Hochschule Deggendorf

# Inhaltsverzeichnis

<b>Inhaltsverzeichnis.....</b>	<b>1</b>
<b>Ziel des Projekts.....</b>	<b>2</b>
<b>Hardwareübersicht und Anbindung am Mikrocontroller.....</b>	<b>2</b>
Einleitung.....	2
Hardwarekomponenten.....	2
Verbindung zum Mikrocontroller.....	3
Softwareunterstützung.....	3
<b>Verfahren.....</b>	<b>4</b>
Schritt 1 - WS2812B LED-Matrix.....	4
Schritt 2 - Implementierung der BLDC Motorsteuerung.....	4
Schritt 3 - Implementierung der Servo Motor.....	4
Schritt 4 - Auslesen und Steuerung der Linescan Kamera.....	4
Schritt 5 - Main()-Funktion.....	5
<b>Programmcode.....</b>	<b>5</b>
WS2812B LED-Matrix.....	5
BLDC Motor.....	8
Servo Motor.....	11
TSL1401 Line Scan Camera.....	13
Main()-Funktion.....	18
<b>Probleme und Lösungen.....</b>	<b>19</b>

## Ziel des Projekts

Das Hauptziel dieses Projekts ist die Entwicklung eines autonomen Fahrzeugs, das auf einer unbekannten Strecke innerhalb von 3 Versuchen vom Start zum Ziel fahren kann. Die Fahrstrecke besteht aus einer breiten weißen Fläche mit schwarzen Linien auf beiden Seiten, die als Begrenzung der Strecke dienen. Hier geht es darum, das Auto zu optimieren, damit es effizient und sicher durch die Strecke fahren kann. Ein wichtiger Aspekt, der berücksichtigt werden muss, ist die Integration von Sensoren und Aktuatoren, um das Auto so gut wie möglich zu steuern und zu navigieren.

## Hardwareübersicht und Anbindung am Mikrocontroller

### Einleitung

Im folgenden Abschnitt wird ein detaillierter Überblick über die Hardwarekomponenten des NXP-Cars und deren Integration in das Gesamtsystem, gesteuert durch den Mikrocontroller FRDM-K66F, gegeben.

### Hardwarekomponenten

- **WS2812B LED-Matrix:**
  - Positioniert an der Fahrzeugfront zur Streckenbeleuchtung.
  - Verwendet eine Daisy-Chain-Verbindung für eine effiziente Datenübertragung von der FRDM-K66F-Platine zu jeder einzelnen LED.
- **Linescan-Kamera:**
  - Oberhalb der LED-Matrix positioniert.
  - Liefert visuelle Informationen über die Strecke, um eine intelligente Fahrzeugnavigation zu ermöglichen.
  - Das Design der Kamera basiert auf Entwürfen von THD.
- **BLDC-Motor:**
  - Treibt die Fahrzeugreifen an.
  - Wird über 3 "halbe" H-Brückenmodule gesteuert, um eine präzise Drehzahl- und Drehmomentregelung zu ermöglichen.



Abbildung 1 - WS2812B LED-Matrix



Abbildung 3 - BLDC Motor



Abbildung 3 - BLDC Motor

- **Servomotor:**
  - Steuert die Lenkung des Fahrzeugs.
  - Ermöglicht eine flexible Anpassung der Fahrtrichtung.



Abbildung 4 - Servo Motor

- **FRDM-K66F-Mikrocontroller:**
  - Herzstück des Systems.
  - Verarbeitet alle Steuerungsaufgaben und koordiniert die Interaktion zwischen den einzelnen Komponenten.
  - Bietet eine leistungsstarke Plattform mit einem ARM Cortex M4F-Prozessor, 2 MB Flash-Speicher und zahlreichen ADCs und FTMs.



Abbildung 5 - K66F Mikrocontroller

## Verbindung zum Mikrocontroller

Der FRDM-K66F-Mikrocontroller funktioniert als zentrale Steuerungseinheit und kommuniziert direkt oder indirekt mit allen Hardwarekomponenten. Die Kommunikation erfolgt über spezifische Schnittstellen, wie z.B. SPI für die LED-Matrix oder PWM-Signale für die Motoren.

## Softwareunterstützung

Um die Entwicklung zu erleichtern, wird eine MCUXpresso-Ressourcendatei bereitgestellt. Diese Datei enthält detaillierte Informationen zur Pinbelegung und Konfiguration der einzelnen Komponenten und dient als solide Grundlage für die Softwareentwicklung. Die Datei wurde von Prof. Dr.-Ing. Marcus Barkowsky zur Verfügung gestellt.

Für die Implementierungen des Projekts werden die folgenden Peripherien des K66F Mikrocontrollers verwendet:

- **DMA (Direct Memory Access):** Der DMA ermöglicht die Übertragung von Daten zwischen der Peripherie und dem Speicher, ohne die CPU zu belasten. In diesem Projekt verwenden wir das DMA-Modul in Kombination mit dem SPI, um die WS2812B-LED-Matrix zu steuern. Auf diese Weise ist eine reibungslose und präzise Steuerung der LEDs möglich, während die Belastung der CPU minimiert wird.
- **SPI (Serial Peripheral Interface):** Die SPI wird für eine schnellere serielle Kommunikation mit der Peripherie verwendet. In diesem Projekt wird SPI verwendet, um die vom DMA erhaltenen Daten effizient an die WS2812B LED-Matrix zu übertragen.



- **FTM (Flex Timer Module):** Das FTM-Modul bietet viele flexible Timer-Funktionen, z.B. PWM-Signale und CLK. In diesem Projekt wird es zur Steuerung der Abweichungen des Servomotors, der Steuerung des BLDC-Motors und des CLK der Linescan Kamera verwendet.
- **ADC (Analog-Digital Converter):** Wandelt analoge Signale in digitale Werte um, um Sensordaten zu erfassen und die Systemspannung zu überwachen.
- **GPIO (General Purpose Input/Output):** Vielseitige Pins, die entweder als Eingänge (zum Empfangen von Signalen) oder als Ausgänge (zum Senden von Signalen) konfiguriert werden können, ermöglichen die Interaktion mit externen Komponenten wie Sensoren, Schaltern und Displays.

## Verfahren

Das Projekt wird in mehreren Schritten von unterschiedlicher Komplexität durchgeführt.

### Schritt 1 - WS2812B LED-Matrix

- DMA- und SPI- Peripherien werden konfiguriert, um die WS2812B-LED-Matrix zu steuern, um die Strecke zu beleuchten und ihren Turn zu signalisieren.
- Implementierung von Funktionen, um die LED-Matrix bei Geradeausfahrt neutral weiß, bei Linksdrehung grün und bei Rechtsdrehung grün zu beleuchten.
- Implementierung einer Funktion zum Ausschalten/Rücksetzen aller LEDs.

### Schritt 2 - Implementierung der BLDC Motorsteuerung

- GPIO, FTM und ADC sind für die Steuerung der BLDC-Motoren konfiguriert.
- Der Motor durchläuft 6 Phasen, wodurch verschiedene „halbe“ H-Brücken aktiviert werden, um die Kommutierung zu ermöglichen.
- Interrupts sind implementiert, um einen Free-Running zu ermöglichen.

### Schritt 3 - Implementierung der Servo Motor

- FTM-Peripherie wird konfiguriert, um die Servo Motor zu steuern. Der Servomotor verwendet eine Periodenlänge von 20ms. Ein PWM-Signal im Bereich von 1ms bis 2ms wird zur Steuerung des Servomotorenausschlags verwendet.
- Implementierung von Funktionen um rechts abzubiegen, links abzubiegen und geradeaus zu fahren

### Schritt 4 - Auslesen und Steuerung der Linescan Kamera

- Konfiguration der ADC-Peripherie (Analog-Digital-Wandler) zum Auslesen der Pixelintensitäten der Kamera.
- Die Kamera scannt die Umgebung zeilenweise und liefert Intensitätswerte.

- Eine Funktion wird implementiert, um die schwarze Linie durch Intensitätsunterschiede zu erkennen.
- Die Kanten der Linie werden durch den höchsten Kontrast bestimmt.
- Die Abweichung vom Idealpfad wird basierend auf den Kantenpositionen berechnet.
- Die Abweichungsdaten werden an die Steuerlogik übergeben, um den Lenkwinkel des Servomotors anzupassen.

## Schritt 5 - Main()-Funktion

- Integration der einzelnen Teile des Programms für das Fahrzeug, einschließlich LEDs, Servo, BLDC und Kamera.
- Sicherstellung des korrekten Zusammenspiels zwischen den verschiedenen Programmteilen durch Zusammenfassung aller Funktionen im Hauptprogramm (Hauptfunktion) und entsprechende Tests.
- Diese Strukturierung hilft, die Entwicklung des Projekts methodisch voranzutreiben und sicherzustellen, dass alle Komponenten des Fahrzeugs reibungslos zusammenarbeiten.

## Programmcode

### WS2812B LED-Matrix

Die LED-Matrix dient in erster Linie der Beleuchtung der Strecke. Daher haben wir eine Funktion, die alle LEDs in der Matrix mit einer neutralen weißen Farbe aufleuchten lässt, wenn das Auto geradeaus fährt. Gleichzeitig verwenden wir die LEDs auch als visuelle Rückmeldung, die uns den Bewegungsstatus des Fahrzeugs anzeigt, d.h. die rechte Seite der LED-Matrix wird grün, wenn das Fahrzeug nach rechts abbiegt, und die linke Seite wird grün, wenn das Fahrzeug nach links abbiegt.

SPI0 wird verwendet, um die seriellen Daten schnell an die WS2812B Led-Matrix zu senden, während eDMA verwendet wird, um direkt auf die Daten aus dem Speicher zuzugreifen, was die Belastung der CPU reduziert. Der eDMA-Controller hat die Aufgabe, die Daten automatisch und kontinuierlich zu übertragen, und der SPI überträgt die Daten präzise und mit hohem Durchsatz an die LEDs. Dadurch wird sichergestellt, dass das strenge Timing der WS2812B-LEDs eingehalten wird und diese korrekt leuchten.

```

8 #ifndef LED_H
9 #define LED_H
10
11 #include <stdint.h>
12 #include <stdbool.h>
13 #include "fsl_dspi.h"
14
15 // Define the buffer size
16 #define BUFF_LENGTH (3*3*8*8)
17
18 // Public variables
19 extern volatile bool g_Transfer_Done;
20 extern volatile bool isTransferCompleted;
21
22 // Public function prototypes
23 void sendLEDData(uint8_t *srcAddr);
24 void whiteLight(void);
25 void rightLight(void);
26 void leftLight(void);
27 void resetLight(void);
28
29 #endif /* LED_H */
30

```

Abbildung 6 - led.h file

```

/*
 * led.c
 *
 * Created on: 12 Jan 2025
 * Author: user
 */
#include "led.h"
#include "board.h"
#include "peripherals.h"
#include "pin_mux.h"
#include "clock_config.h"
#include "MK66F18.h"
#include "fsl_debug_console.h"

// Buffer for storing LED data
AT_NONCACHEABLE_SECTION(uint8_t srcAddr[BUFF_LENGTH]);

// Public variables
volatile bool g_Transfer_Done = false;
volatile bool isTransferCompleted = false;

```

Abbildung 7 - ein Teil von led.c file

Die obige Abbildungen enthält die Definition von Variablen und Funktionen zur Steuerung der WS2812B-LEDs, die später in der Quelldatei **led.c** verwendet werden.

## Variablen:

**#define BUFF\_LENGTH (3\*3\*8\*8)** definiert die Länge des Datenpuffers, der für die Verarbeitung der LEDs verwendet wird. Hier verwenden wir eine 8x8-Matrix aus WS2812B-LEDs, also insgesamt 64 LEDs. Eine logische 1 wird hier aus den Bits „110“ und eine logische 0 aus den Bits „100“ gebildet. Für jede Farbe werden 8 logische Nullen benötigt, die aus  $(3 \times 8) = 24$  Bits = 3 Bytes bestehen. Jede LED hat 3 Farben. Daraus ergibt sich eine Pufferlänge von  $3(\text{Bytes pro Farbe}) \times 3(\text{Farben pro LED}) \times 64(\text{Anzahl der LEDs}) = 576$  Bytes an Daten, die durch **BUFF\_LENGTH** dargestellt werden

Das Puffer-Array (**srcAddr**) wird zum Speichern und Steuern der Farbinformationen für jede LED in der Matrix verwendet. **AT\_NONCACHEABLE\_SECTION** stellt sicher, dass sich das **srcAddr**-Array im nicht-cachefähigen Speicherbereich befindet, damit die Daten für die Steuerung der WS2812B-LEDs über SPI konsistent bleiben.

**g\_Transfer\_Done** und **isTransferCompleted** sind Flags, die den Status der Datenübertragung kennzeichnen. Für beide Variablen wird **volatile** verwendet, da sich die Werte unerwartet ändern können, z. B. aufgrund von Interrupts oder der DMA-Übertragung.

## Funktionen:

```

23 // Callback for SPI transfer completion
24 void ws2812b_spi0_edma_handler(SPI_Type *base, dspi_master_edma_handle_t *handle, status_t status, void *userData) {
25     if (status == kStatus_Success) {
26         PRINTF("This is DSPI master EDMA callback.\r\n");
27     }
28     isTransferCompleted = true;
29 }

```

Abbildung 8 - SPI0 handler

Die Funktion in Abbildung 8 wird aufgerufen, wenn ein **eDMA-Transfer** abgeschlossen ist. Sie setzt **isTransferCompleted** auf true und kann für Fehlerbehandlungen oder Debugging-Informationen genutzt werden.

```

32● void sendLEDDData(uint8_t *srcAddr) {
33     dspi_transfer_t masterXfer;
34
35     isTransferCompleted = false;
36     masterXfer.txData = srcAddr;
37     masterXfer.rxData = NULL;
38     masterXfer.dataSize = BUFF_LENGTH;
39     masterXfer.configFlags = kDSPI_MasterCtar0 | kDSPI_MasterPcs0 | kDSPI_MasterPcsContinuous;
40
41     if (kStatus_Success != DSPI_MasterTransferEDMA(SPI0_PERIPHERAL, &ws2812b_spi0_edma_handle, &masterXfer)) {
42         PRINTF("Error: Failed to start SPI transfer.\r\n");
43     }
44
45     while (!isTransferCompleted) {
46     }
47 }

```

Abbildung 9 - sendLEDDData() Funktion

Die Funktion in Bild 9 sendet die in **srcAddr** vorbereiteten Daten mittels eDMA-Transfer über SPI an die LEDs. Anschließend wartet sie mit einer while-Schleife auf den Abschluss der Übertragung, bevor sie weiterarbeitet.

```

48
49● // Function to clear the values in the buffer
50 void resetLight(void) {
51     for (int i = 0; i < BUFF_LENGTH; i += 3) {
52         // 0b100100100100100100100100 = 0x924924
53         srcAddr[i + 0] = 0x92;
54         srcAddr[i + 1] = 0x49;
55         srcAddr[i + 2] = 0x24;
56     }
57     sendLEDDData(srcAddr);
58 }
59

```

Abbildung 10 - resetLight() Funktion

```

59
60● // Function to set all LEDs to white
61 void whiteLight(void) {
62     int i;
63     for (i = 0; i < BUFF_LENGTH; i += 3) {
64         // 0b110100100110100100100100 = 0xD26924
65         srcAddr[i + 0] = 0xD2;
66         srcAddr[i + 1] = 0x69;
67         srcAddr[i + 2] = 0x24;
68     }
69     sendLEDDData(srcAddr);
70 }
71

```

Abbildung 11 - whiteLight() Funktion

Die Funktion in Abb. 10 bereitet den Puffer mit logischen 0en unter Verwendung der entsprechenden Kodierung vor, bevor die Daten übertragen werden. Sie schaltet alle LEDs aus, indem sie den Wert für alle Farben aller LEDs effektiv auf 0 setzt und den Puffer zurücksetzt, um eine Überschneidung der Daten mit dem zuvor verwendeten Puffer zu vermeiden. Die Funktion in Abb. 11 füllt den Puffer mit logischen 1en, um ein neutrales weißes Licht zu erzeugen.

```

72 // Function to set the first half of the LED Matrix green
73● void rightLight(void) {
74     resetLight(); // clear the value in the buffer
75     int i;
76     for (i = 0; i < BUFF_LENGTH; i += 3 * 3) {
77         // 0b110110100100100100100100 = 0xD26924
78         if (i < BUFF_LENGTH/2) {
79             srcAddr[i + 0] = 0xD2;
80             srcAddr[i + 1] = 0x69;
81             srcAddr[i + 2] = 0x24;
82         } else { // 0b100100100100100100100100 = 0x924924
83             srcAddr[i + 0] = 0x92;
84             srcAddr[i + 1] = 0x49;
85             srcAddr[i + 2] = 0x24;
86         }
87     }
88     sendLEDDData(srcAddr);
89 }

```

Abbildung 12 - rightLight() Funktion

```

91 // Function to set the last half of the LED Matrix green
92● void leftLight(void) {
93     resetLight();
94     int i;
95     for (i = 0; i < BUFF_LENGTH; i += 3 * 3) {
96         // 0b110110100100100100100100 = 0xD26924
97         if (i > BUFF_LENGTH/2) {
98             srcAddr[i + 0] = 0xD2;
99             srcAddr[i + 1] = 0x69;
100             srcAddr[i + 2] = 0x24;
101         } else { // 0b100100100100100100100100 = 0x924924
102             srcAddr[i + 0] = 0x92;
103             srcAddr[i + 1] = 0x49;
104             srcAddr[i + 2] = 0x24;
105         }
106     }
107     sendLEDDData(srcAddr);
108 }

```

Abbildung 13 - leftLight() Funktion

Diese Funktionen setzen beide den Puffer zurück, bevor sie den Puffer mit den gewünschten Werten füllen. Die Funktion füllt alle 9 Bytes 3 Bytes an Daten mit Hilfe einer for-Schleife. Dadurch wird effektiv nur der Wert für das grüne Licht der RGB-LEDs mit 1en gesetzt. Die Funktion `rightLight()` setzt den Wert für die erste Hälfte des Puffers (die ersten 32 LEDs) und lässt den Rest mit 0s stehen, während `leftLight()` das Gegenteil bewirkt.

## BLDC Motor

```

8  #ifndef BLDC_H
9  #define BLDC_H
10
11 #include <stdint.h>
12 #include <stdbool.h>
13 #include <stdio.h>
14
15 // Constants
16 #define START_DUTY_CYCLE 10
17
18 // Global Variables
19 extern volatile bool free_running;
20 extern volatile int wait_counter;
21 extern volatile bool waiting_for_falling_edge;
22 extern volatile int threshold;
23
24 extern volatile uint8_t current_phase;
25 extern volatile uint8_t current_duty_cycle;
26
27 // Function Prototypes
28 void bldc_motor_init(void);
29 void bldc_motor_set_speed(int percentage);
30 void bldc_motor_stop(void);
31 void bldc_force_commute_start(void);
32 void commute(void);
33 int configure_off(void);
34 uint8_t reconfigure_for_phase(uint8_t phase, uint8_t duty_cycle);
35 void motor_bemf_irq_handler(void);
36
37 #endif /* BLDC_H */

```

Abbildung 14 - `bldc.h` file

Die obige Header-Datei `bldc.h` enthält die Definition von Variablen und Funktionen zur Steuerung der BLDC-motor, die später in der Quelldatei `bldc.c` verwendet werden.

## Funktionen:

```

// Lookup Tables
// GPIO pin masks for enabling motor phases
uint32_t motor_enable_gpioa_mask[] = {
    BOARD_MOTOR_EN_A_PTA6_GPIO_PIN_MASK | BOARD_MOTOR_EN_C_PTA1_GPIO_PIN_MASK,
    BOARD_MOTOR_EN_A_PTA6_GPIO_PIN_MASK | BOARD_MOTOR_EN_B_PTA7_GPIO_PIN_MASK,
    BOARD_MOTOR_EN_C_PTA1_GPIO_PIN_MASK | BOARD_MOTOR_EN_B_PTA7_GPIO_PIN_MASK,
    BOARD_MOTOR_EN_C_PTA1_GPIO_PIN_MASK | BOARD_MOTOR_EN_A_PTA6_GPIO_PIN_MASK,
    BOARD_MOTOR_EN_B_PTA7_GPIO_PIN_MASK | BOARD_MOTOR_EN_A_PTA6_GPIO_PIN_MASK,
    BOARD_MOTOR_EN_B_PTA7_GPIO_PIN_MASK | BOARD_MOTOR_EN_C_PTA1_GPIO_PIN_MASK
};

```

```

// FTM (FlexTimer Module) channels for each motor phase
uint32_t motor_direction_ftm_channel[] = {
    FTM0_MOTOR_A_FTM0_CH0_CHANNEL,
    FTM0_MOTOR_A_FTM0_CH0_CHANNEL,
    FTM0_MOTOR_C_FTM0_CH2_CHANNEL,
    FTM0_MOTOR_C_FTM0_CH2_CHANNEL,
    FTM0_MOTOR_B_FTM0_CH1_CHANNEL,
    FTM0_MOTOR_B_FTM0_CH1_CHANNEL
};

// ADC channels for sensing back-EMF on the inactive phase
uint8_t back_emf_adc_channel_number[] = {
    ADC0_MOTOR_BEMF10_B_IRQ_CHANNEL,
    ADC0_MOTOR_BEMF10_C_IRQ_CHANNEL,
    ADC0_MOTOR_BEMF10_A_IRQ_CHANNEL,
    ADC0_MOTOR_BEMF10_B_IRQ_CHANNEL,
    ADC0_MOTOR_BEMF10_C_IRQ_CHANNEL,
    ADC0_MOTOR_BEMF10_A_IRQ_CHANNEL
};

```

Abbildung 15 - Arrays für phase config in `bldc.c` file

Diese Arrays werden in `bldc.c` verwendet, um die Kommutierung der Motorphasen zu verwalten. `motor_enable_gpia_mask[]` steuert, welche GPIO-Pins für jeden der sechs Kommutierungsschritte aktiviert werden. `motor_direction_ftm_channel[]` ordnet jede Phase dem richtigen PWM-Kanal zu und stellt so sicher, dass der Motor die entsprechenden Leistungssignale erhält. `back_emf_adc_channel_number[]` weist die inaktive Phase in jedem Schritt einem ADC-Kanal für die Erfassung der Back-EMF zu, die für den sensorlosen Betrieb entscheidend ist. Diese Arrays arbeiten zusammen, um eine reibungslose Motorsteuerung zu ermöglichen, indem sie die Phasen richtig umschalten, die PWM anpassen und Nulldurchgänge erkennen.

```

53 // Disables all motor phases and sets PWM duty cycle to 0%
54 int configure_off(void) {
55     uint8_t success = 0;
56
57     GPIO_PortClear(BEARD_MOTOR_EN_A_PTA6_GPIO, BOARD_MOTOR_EN_A_PTA6_GPIO_PIN_MASK);
58     GPIO_PortClear(BEARD_MOTOR_EN_B_PTA7_GPIO, BOARD_MOTOR_EN_B_PTA7_GPIO_PIN_MASK);
59     GPIO_PortClear(BEARD_MOTOR_EN_C_PTA1_GPIO, BOARD_MOTOR_EN_C_PTA1_GPIO_PIN_MASK);
60
61     if (FTM_UpdatePwmDutyCycle(FTM0_PERIPHERAL, FTM0_MOTOR_A_FTM0_CH0_CHANNEL,
62                               kFTM_EdgeAlignedPwm, 0) == kStatus_Success &&
63         FTM_UpdatePwmDutyCycle(FTM0_PERIPHERAL, FTM0_MOTOR_B_FTM0_CH1_CHANNEL,
64                               kFTM_EdgeAlignedPwm, 0) == kStatus_Success &&
65         FTM_UpdatePwmDutyCycle(FTM0_PERIPHERAL, FTM0_MOTOR_C_FTM0_CH2_CHANNEL,
66                               kFTM_EdgeAlignedPwm, 0) == kStatus_Success) {
67         success = 1;
68     }
69
70     return success;
71 }

```

```

73 // Configures motor phase and PWM duty cycle
74 uint8_t reconfigure_for_phase(uint8_t phase, uint8_t duty_cycle) {
75     uint8_t success = 0;
76
77     if (configure_off()) {
78         if (FTM_UpdatePwmDutyCycle(FTM0_PERIPHERAL, motor_direction_ftm_channel[phase],
79                                   kFTM_EdgeAlignedPwm,
80                                   duty_cycle) == kStatus_Success) {
81             FTM_SetSoftwareTrigger(FTM0_PERIPHERAL, true);
82             GPIO_PortSet(GPIOA, motor_enable_gpia_mask[phase]);
83             success = 1;
84         } else {
85             printf("Error: Failed to configure duty cycle for phase %d\n", phase);
86         }
87     }
88
89     return success;
90 }
91
92 // Advances to the next commutation phase
93 void commute(void) {
94     current_phase = (current_phase + 1) % 6;
95     reconfigure_for_phase(current_phase, current_duty_cycle);
96 }

```

Abbildung 16 & 17 - Motor Funktionen(1) in `bldc.c` file

`configure_off()` löscht alle GPIOs und setzt das Tastverhältnis auf 0. `reconfigure_for_phase()` setzt alle GPIOs und FTM mit `configure_off()` zurück, erzeugt dann das entsprechende PWM-Signal für den entsprechenden Kanal und aktiviert den entsprechenden GPIO. Schließlich durchläuft `commute()` alle 6 Phasen und ruft `reconfigure_for_phase()` mit den richtigen Phasen auf. Die Funktion `commute()` ist die primäre Funktion, die immer dann aufgerufen wird, wenn sich der Motor drehen soll.

```

98 // Interrupt handler for back-EMF detection and commutation control
99 void motor_bemf_irq_handler(void) {
100     uint32_t result_values[2] = {0};
101
102     for (int i = 0; i < 2; i++) {
103         uint32_t status = ADC16_GetChannelStatusFlags(ADC0_PERIPHERAL, i);
104         if (status == kADC16_ChannelConversionDoneFlag) {
105             result_values[i] = ADC16_GetChannelConversionValue(ADC0_PERIPHERAL, i);
106         }
107     }
108
109     if (free_running) {
110         if (wait_counter < 20) {
111             waiting_for_falling_edge = (result_values[0] >= threshold);
112             wait_counter++;
113         } else {
114             if ((waiting_for_falling_edge && result_values[0] < threshold) ||
115                 (!waiting_for_falling_edge && result_values[0] > threshold)) {
116                 commute();
117                 wait_counter = 0;
118                 waiting_for_falling_edge = !waiting_for_falling_edge;
119             }
120         }
121     }
122
123     ADC16_SetChannelConfig(ADC0_PERIPHERAL, ADC0_CH0_CONTROL_GROUP,
124                           &ADC0_channelsConfig[back_emf_adc_channel_number[current_phase]]);
125
126 #if defined _CORTEX_M && (_CORTEX_M == 4U)
127     __DSB();
128 #endif
129 }

```

Abbildung 18 - ADC0 interrupt Handler for motor

Der Interrupt `motor_bemf_irq_handler()` ist für die Erfassung der Back-EMF und die Auslösung der Kommutierung in einem sensorlosen BLDC-Motorsteuerungssystem zuständig. Er wird jedes Mal ausgeführt, wenn eine ADC-Wandlung abgeschlossen ist, und prüft die BEMF-Spannung der inaktiven Motorphase, um ein Zero-Crossing zu erkennen.

Die Variable `wait_counter` dient als Entprellungsmechanismus, um falsche Erkennungen aufgrund von Rauschen oder geringfügigen Schwankungen im BEMF-Signal zu verhindern. Zu Beginn wartet das System 20 Zyklen, bevor es reagiert, um sicherzustellen, dass das Signal stabil ist.

Läuft der Motor im Free-Running mode, überwacht das System kontinuierlich das BEMF-Signal und wartet darauf, dass es den `threshold` überschreitet. Wenn der Übergang (steigende oder fallende Flanke) eintritt, wechselt der Motor in die nächste Phase, aktualisiert `current_phase` und setzt `wait_counter` zurück, um ein vorzeitiges Umschalten zu verhindern. Im Free-Running Mode wiederholt sich dieser Vorgang unbegrenzt, so dass der Motor eine kontinuierliche Drehung ohne externen Eingriff beibehalten kann.

```

131 // Initializes the motor system
132 void bldc_motor_init(void) {
133     current_duty_cycle = START_DUTY_CYCLE;
134     printf("Motor initialized\n");
135 }
136
137 // Sets motor speed based on percentage input
138 void bldc_motor_set_speed(int percentage) {
139     if (percentage < 1) {
140         bldc_motor_stop();
141     } else {
142         current_duty_cycle = 12 + (50 * (percentage / 100));
143     }
144 }
145
146 // Stops the motor operation
147 void bldc_motor_stop(void) {
148     free_running = false;
149     configure_off();
150 }

```

Abbildung 19 - Motor Funktionen(2)

```

// Forces initial commutations to start motor before BEMF control takes over
void bldc_force_commute_start(void) {
    bldc_motor_init();
    commute();
    int t = 90000;
    for (int j = 10; j <= 35; j += 5) {
        for (int i = 0; i < 30; i++) {
            commute();
            for (int k = 0; k < t; k++) {}
        }
        t -= 10000;
        bldc_motor_set_speed(j);
    }

    free_running = true;
    ADC16_SetChannelConfig(ADC0_PERIPHERAL, ADC0_CH0_CONTROL_GROUP,
        &ADC0_channelsConfig[back_emf_adc_channel_number[current_phase]]);
}

```

Abbildung 20 - `bldc_force_commute_start` Funktion

Die Funktion `bldc_motor_init()` initialisiert den Motor, indem sie den aktuellen Betriebszyklus (`current_duty_cycle`) auf seinen Standardwert (`START_DUTY_CYCLE`) setzt und eine Meldung ausgibt, dass der Motor initialisiert ist. Sie stellt sicher, dass der Motor mit einer bekannten Konfiguration startet, bevor ein Betrieb beginnt. Die Funktion `bldc_motor_set_speed(int percentage)` passt die Motordrehzahl anhand eines bestimmten Prozentsatzes an. Liegt der Prozentsatz unter 1, wird der Motor angehalten, um eine unbeabsichtigte Bewegung zu verhindern. Andernfalls wird das Tastverhältnis in einem Bereich von 10-80 % aktualisiert, um die an den Motor abgegebene Leistung zu steuern und so seine Geschwindigkeit zu beeinflussen. Diese Funktion ermöglicht eine dynamische Geschwindigkeitsregelung und gewährleistet gleichzeitig einen sicheren Betrieb.

Die Funktion `bldc_force_commute_start()` ist für den Kickstart der Motordrehung zuständig, bevor die sensorlose Steuerung über die BEMF-Erkennung übernimmt. Da ein BLDC-Motor eine anfängliche Bewegung benötigt, um eine BEMF zu erzeugen, erzwingt diese Funktion eine Reihe von Kommutierungen mit einer abnehmenden Verzögerung zwischen den einzelnen Schritten und ahmt so eine "Ramp-up" Sequenz nach. Sie beginnt mit einer niedrigen Drehzahl



und erhöht allmählich das Tastverhältnis, während die Verzögerung verringert wird, so dass der Motor sanft in Free-Running mode übergehen kann. Sobald die Sequenz abgeschlossen ist, wird der Freilaufmodus aktiviert und die BEMF-Erfassung übernimmt, um den Motorbetrieb aufrechtzuerhalten.

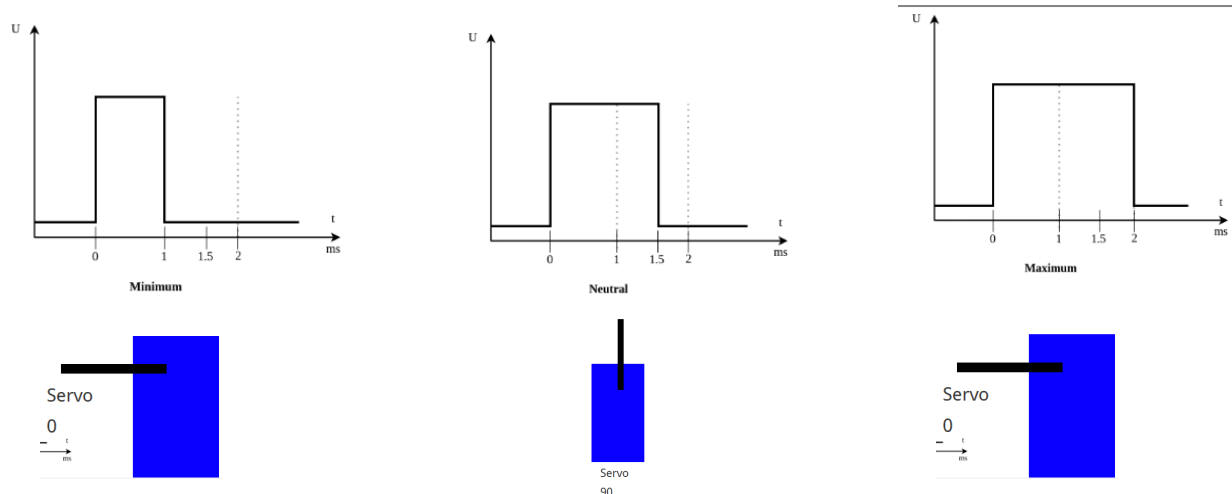
---

## Servo Motor

Der Servomotor ist für die Steuerung der Räder verantwortlich und richtet sich dabei nach den Eingaben der Line Scan Camera. Das System berechnet die Abweichung vom idealen Kurs und passt die Position des Servomotors entsprechend an, um den Lenkwinkel zu korrigieren.

Zur Ansteuerung des Servomotors wird das **FlexTimer Module (FTM)** verwendet, um ein **PWM-Signal (Pulse Width Modulation)** zu erzeugen. Der **Duty Cycle** des PWM-Signals bestimmt die Position des Motors und ermöglicht präzise Winkelanpassungen. Durch Variieren des Duty Cycles kann das System die Räder dynamisch steuern und so eine sanfte sowie reaktionsschnelle Navigation gewährleisten.

$$\text{Duty Cycle} = \left( \frac{\text{Pulse Width}}{\text{Period Length}} \right) \times 100$$





```

#ifndef SERVO_H // Prevents multiple inclusions of this header file
#define SERVO_H

#include <stdint.h> // Includes standard integer types (e.g., uint8_t, uint32_t)
#include "fsl_dspi.h" // Includes DSPI (Serial Peripheral Interface) driver for communication

void setServoPosition(float duty_cycle); // Function to set the servo motor position using a specified duty cycle
void turnLeft(); // Function to turn the servo motor fully to the left
void turnRight(); // Function to turn the servo motor fully to the right
void centerServo(); // Function to center the servo motor

#endif // SERVO_H // End of header file guard

```

Abbildung 21 - Servo.h file

Die **Servo.h**-Datei ist eine Header-Datei für die Steuerung des Servomotors. Sie enthält Funktionsdeklarationen, die zur Anpassung der Servoposition genutzt werden. Diese Datei stellt sicher, dass der Servomotor korrekt gesteuert werden kann, um die gewünschte Lenkbewegung auszuführen.

### Funktionsdeklarationen :

- **void setServoPosition(float duty\_cycle);**
  - Stellt die Position des Servomotors ein, indem ein bestimmter **Duty Cycle** (Tastverhältnis) verwendet wird.
- **void turnLeft();**
  - Dreht den Servomotor vollständig nach links.
- **void turnRight();**
  - Dreht den Servomotor vollständig nach rechts.
- **void centerServo();**
  - Setzt den Servomotor in die neutrale Mittelstellung.

```

// Function to set the servo position based on the desired duty cycle
void setServoPosition(float duty_cycle) {
    // Update PWM duty cycle for servo FTM3 channel
    FTM_UpdatePwmDutyCycle(FTM3_PERIPHERAL, FTM3_FTM_SERVO_CHANNEL, kFTM_EdgeAlignedPwm, duty_cycle);
    FTM_SetSoftwareTrigger(FTM3_PERIPHERAL, true); // Apply the new duty cycle
}

// Function to turn the car left
void turnLeft() {
    float duty_cycle = 5.0; // 1ms pulse width -- 5% duty cycle
    setServoPosition(duty_cycle);
}

// Function to turn the car right
void turnRight() {
    float duty_cycle = 10.0; // 2ms pulse width -- 10% duty cycle
    setServoPosition(duty_cycle);
}

// Function to center the servo (straight position)
void centerServo() {
    float duty_cycle = 7.5; // 1.5ms pulse width -- 7.5% duty cycle
    setServoPosition(duty_cycle);
}

```

Abbildung 22 - Servo.c file

Die Datei **Servo.c** enthält die Implementierung der Steuerungsfunktionen für den Servomotor, der das Lenken des Fahrzeugs übernimmt. Der Servomotor wird über **PWM (Pulse Width Modulation)** gesteuert, wobei der **Duty Cycle** des PWM-Signals die Position des Motors bestimmt. Die zentrale Funktion **setServoPosition(float duty\_cycle)** setzt die gewünschte Position, indem sie das PWM-Signal für den Servomotor auf **FTM3, Kanal 6** aktualisiert und die Änderung durch einen Software-Trigger übernimmt. Drei weitere Funktionen ermöglichen die gezielte Steuerung des Servos: **turnLeft()** setzt den Duty Cycle auf **5%** (entspricht einer **1 ms**

**Impulsbreite)** für eine maximale Linksdrehung, `turnRight()` verwendet **10%** (entspricht **1,8 ms**) für eine maximale Rechtsdrehung und `centerServo()` stellt den Servo mit **7,5%** (entspricht **1,5 ms**) in die Mittelposition. Diese Steuerung sorgt für eine präzise Lenkung des Fahrzeugs, indem der Winkel kontinuierlich angepasst werden kann.

## TSL1401 Line Scan Camera

Die **Line Scan Camera** ist eine spezialisierte Kamera, die Bilder **zeilenweise** aufnimmt, anstatt ein vollständiges Bild in einer einzigen Belichtung zu erfassen. Sie besteht aus einem **linearen Array von Fotodioden**, die typischerweise in einer einzigen Reihe angeordnet sind. Diese Fotodioden erfassen die **Lichtintensität** und wandeln sie in elektrische Signale um. Die Signale werden **sequenziell verarbeitet und ausgelesen**, sodass die Kamera eine Szene oder ein Objekt zeilenweise erfassen und ein vollständiges Bild rekonstruieren kann.

Die **Line Scan Camera** wird zur **Erkennung einer schwarzen Linie** verwendet, indem sie **Intensitätsunterschiede der Pixel analysiert**. Die erfassten Pixelwerte werden mit einem **ADC (Analog-Digital-Wandler)** verarbeitet, und das System erkennt die **linken und rechten Kanten der Linie**, indem es die Punkte mit dem höchsten Kontrast identifiziert. Diese Daten werden anschließend genutzt, um **Abweichungen zu berechnen und Bewegungen entsprechend zu steuern**.

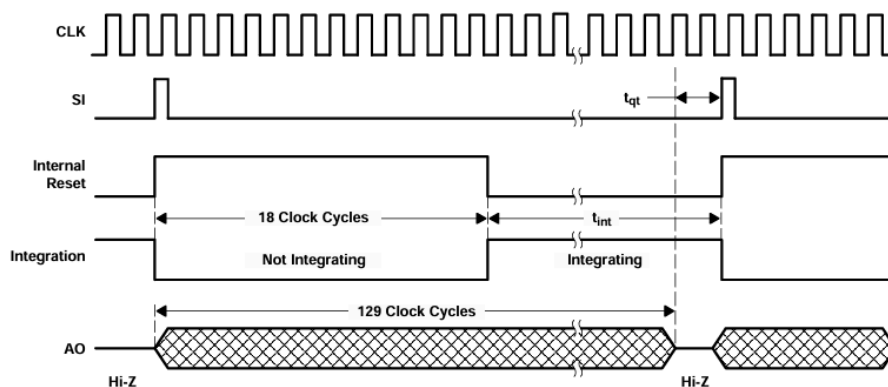


Figure 1. Timing Waveforms

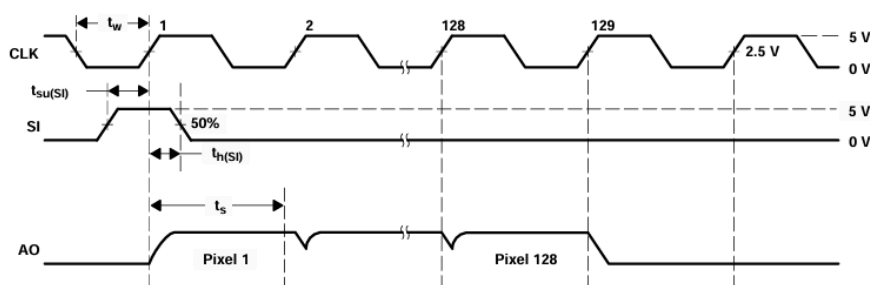


Figure 2. Operational Waveforms

```

#ifndef CAMERA_H
#define CAMERA_H

#include <stdint.h>
#include "fsl_adc16.h"
#include "fsl_ftm.h"
#include "fsl_gpio.h"

// Define constants
#define IMAGE_SIZE 128 // Number of pixels in the line scan camera
#define THRESHOLD 20 // Threshold to detect black line edges
#define SI_CYCLE_LIMIT 200 // Limit for SI pulse cycle

// Global variables (extern for use in the .c file)
extern uint16_t ImageData[IMAGE_SIZE]; // Stores pixel intensity values
extern volatile uint16_t currentPixelNumber; // Current pixel being processed
extern int ImageDataDifference[IMAGE_SIZE]; // Stores pixel intensity differences
extern int CompareValue; // Stores temporary comparison value
extern int BlackLineLeft; // Position of the left black line
extern int BlackLineRight; // Position of the right black line
extern int RoadMiddle; // Midpoint of detected track
extern int Deviation; // Difference from ideal center

// Function declarations
void calculateDeviation(void);
void findBlackLineRight(void);
void findBlackLineLeft(void);
void ADC1_IRQHANDLER(void);
void FTM1_IRQHANDLER(void);

#endif // CAMERA_PROCESSING_H

```

Abbildung 23 -Camera.h file

Die **camera.h** Datei definiert **Konstanten, globale Variablen und Funktionsprototypen**, die für die Verarbeitung der **Line Scan Camera**-Daten erforderlich sind. Sie dient als **Schnittstelle** zwischen dem Hauptprogramm und dem Kameraverarbeitungsmodul.

### Definierte Konstanten :

- **IMAGE\_SIZE (128)**: Definiert die **Anzahl der Pixel pro Zeile** der Kamera.
- **THRESHOLD (20)**: Legt den **Minimalen Intensitätsunterschied** fest, um die Kanten der schwarzen Linie zu erkennen.
- **SI\_CYCLE\_LIMIT (200)**: Begrenzung der **SI-Pulszyklen**, um eine Übersteuerung zu vermeiden.

### Globale Variablen :

- **ImageData[IMAGE\_SIZE]**: Enthält die **gemessenen Intensitätswerte** für jedes Pixel.
- **currentPixelNumber**: Speichert das **gerade verarbeitete Pixel**.
- **ImageDataDifference[IMAGE\_SIZE]**: Speichert die **Differenzen zwischen aufeinanderfolgenden Pixeln**, um Intensitätsänderungen zu erkennen.
- **CompareValue**: Temporäre Variable für Vergleiche während der Verarbeitung.
- **BlackLineLeft & BlackLineRight**: Speichern die **Positionen der linken und rechten Kante** der schwarzen Linie.

- **RoadMiddle**: Enthält den **Mittelpunkt** der erkannten Linie.
- **Deviation**: Zeigt die **Abweichung vom idealen Zentrum** der Kamera an.

## Funktionsdeklarationen :

- **calculateDeviation(void)**
  - Berechnet die **Position der schwarzen Linie** und bestimmt die **Abweichung von der Mitte**.
- **findBlackLineRight(void)**
  - Durchsucht die **Pixel 64 bis 127**, um die **rechte Kante** der schwarzen Linie anhand von Intensitätsänderungen zu finden.
- **findBlackLineLeft(void)**
  - Durchsucht die **Pixel 65 bis 0**, um die **linke Kante** der schwarzen Linie zu bestimmen.
- **ADC1\_IRQHANDLER(void)**
  - **Interrupt-Handler** für den **Analog-Digital-Wandler (ADC)**.
  - Liest die **Pixelintensitäten** der **Line Scan Camera** aus und speichert sie in **ImageData[]**.
- **FTM1\_IRQHANDLER(void)**
  - **Interrupt-Handler** für das **FlexTimer-Modul (FTM)**.
  - Generiert **Timing-Signale** für die Kamera-Synchronisation und steuert die **SI- und CLK-Pulse**.

---

```

/* ADC1 Interrupt Handler
- Reads ADC conversion values from the camera sensor
- Stores the pixel intensity into ImageData[]
*/
void ADC1_IRQHANDLER(void) {
    uint32_t result_values[2] = {0}; // Store ADC conversion results

    for (int i = 0; i < 2; i++) {
        uint32_t status = ADC16_GetChannelStatusFlags(ADC1_PERIPHERAL, i);
        if (status == KADC16_ChannelConversionDoneFlag) {
            result_values[i] = ADC16_GetChannelConversionValue(ADC1_PERIPHERAL, i);
        }
    }

    // Store the pixel intensity value
    ImageData[currentPixelNumber] = result_values[0];
    currentPixelNumber++; // Move to the next pixel

#ifdef _CORTEX_M_44_ (_CORTEX_M == 4U)
    __DSB(); // Data synchronization barrier for Cortex-M4
#endif
}

```

Abbildung 24 -Camera.c file / ADC Interrupt Handler Funktion

Die Funktion **ADC1\_IRQHANDLER** ist eine Interrupt-Service-Routine (ISR), die ADC (Analog-Digital-Wandler)-Konvertierungen für einen Kamerasensor verarbeitet. Sobald eine ADC-Konvertierung abgeschlossen ist, wird der Interrupt ausgelöst, und die Funktion liest die Konvertierungsergebnisse von ADC1 aus. Zunächst wird ein Array zur Speicherung von zwei Konvertierungswerten initialisiert. Anschließend wird über zwei ADC-Kanäle iteriert, wobei mithilfe von **ADC16\_GetChannelStatusFlags** überprüft wird, ob die Konvertierung

abgeschlossen ist. Falls dies der Fall ist, wird der entsprechende Wert mit `ADC16_GetChannelConversionValue` abgerufen und im Array gespeichert. Der primäre Pixel-Intensitätswert des ersten ADC-Kanals wird dann im `ImageData`-Array an der durch `currentPixelNumber` angegebenen Stelle gespeichert. Danach wird `currentPixelNumber` inkrementiert, um das nächste Pixel vorzubereiten. Schließlich wird auf einem Cortex-M4-Prozessor ein **Data Synchronization Barrier** (`__DSB()`) ausgeführt, um sicherzustellen, dass alle Speicheroperationen vor der nächsten Instruktion abgeschlossen sind. Diese Funktion ist entscheidend für die Erfassung und Verarbeitung von Bilddaten des Kamerasensors.

```

// FTM1 Interrupt Handler
// - Generates clock cycles for the line scan camera
// - Controls the SI signal and ADC sampling process
//
void FTM1_IRQHANDLER(void) {
    uint32_t intStatus;
    static int16_t parallax1_si_count = 0; // SI cycle counter

    // Read and clear interrupt flags
    intStatus = FTM_GetStatusFlags(FTM1_PERIPHERAL);
    FTM_ClearStatusFlags(FTM1_PERIPHERAL, intStatus);

    // Increment SI cycle count (counts clock pulses)
    parallax1_si_count++;

    // First clock cycle: Reset SI signal and start pixel capturing
    if(parallax1_si_count == 1) {
        GPIO_PortClear(BOARD_PARALLAX1_SI_PTA9_GPIO, BOARD_PARALLAX1_SI_PTA9_GPIO_PIN_MASK);
        currentPixelNumber = 0; // Reset pixel counter
    }

    // ADC conversion is triggered between 1st and 128th cycle
    if((parallax1_si_count >= 1) && (parallax1_si_count <= 128)) {
        ADC16_SetChannelConfig(ADC1_PERIPHERAL, ADC1_CH0_CONTROL_GROUP, &ADC1_channelsConfig[0]);
    }

    // Reset SI cycle counter after SI_CYCLE_LIMIT, restarting the frame capture
    if(parallax1_si_count > SI_CYCLE_LIMIT) {
        parallax1_si_count = 0;
        GPIO_PortSet(BOARD_PARALLAX1_SI_PTA9_GPIO, BOARD_PARALLAX1_SI_PTA9_GPIO_PIN_MASK);
    }

    #if defined _CORTEX_M && (_CORTEX_M == 4U)
        __DSB(); // Ensure proper synchronization
    #endif
}

```

Abbildung 25 -Camera.c file / FTM Interrupt Handler Funktion

Die Funktion `FTM1_IRQHANDLER` ist ein Interrupt-Handler, der für die Erzeugung von Taktsignalen für eine Zeilenkamera verantwortlich ist und gleichzeitig das SI-Signal (Start Integration) sowie die ADC-Abtastung (Analog-Digital-Wandlung) steuert. Zu Beginn werden die Interrupt-Flags des FlexTimer-Moduls (FTM1) ausgelesen und zurückgesetzt, um die Unterbrechung zu bestätigen. Ein statischer Zähler (`parallax1_si_count`) verfolgt die Clock Cycles und sorgt für die korrekte Steuerung des SI-Signals sowie der ADC-Wandlungen. Beim ersten Clock Cycle wird das SI-Signal zurückgesetzt, um den Beginn einer neuen Bildaufnahme zu markieren, und der Pixelzähler wird zurückgesetzt. Zwischen dem ersten und dem 128. Clock Cycle werden ADC-Wandlungen ausgelöst, um die Intensitätswerte der einzelnen Pixel zu erfassen. Sobald das Zykluslimit erreicht ist, wird das SI-Signal auf HIGH gesetzt, wodurch die Bildaufnahme abgeschlossen wird, und der Zähler wird für den nächsten Zyklus zurückgesetzt. Zusätzlich wird auf Cortex-M4-Prozessoren eine Data Synchronization Barrier (`__DSB()`) verwendet, um eine korrekte Speicher- und Befehlssynchronisation sicherzustellen. Diese Funktion stellt eine präzise Taktung sicher und ermöglicht die exakte Erfassung von Bilddaten durch den Kamerasensor.

```

/* Function: findBlackLineLeft
- Identifies the left edge of the black line
- Uses pixel intensity differences to detect the largest change
*/
void findBlackLineLeft() {
    for (int i = 65; i > 0; i--) {
        ImageDataDifference[i] = ImageData[i] - ImageData[i - 1];
    }

    CompareValue = THRESHOLD; // Set initial threshold
    for (int i = 65; i > 0; i--) {
        if (ImageDataDifference[i] > CompareValue) {
            CompareValue = ImageDataDifference[i];
            BlackLineLeft = i; // Store position of detected edge
        }
    }
}

```

```

/* Function: findBlackLineRight
- Identifies the right edge of the black line
- Uses pixel intensity differences to detect the largest change
*/
void findBlackLineRight() {
    for (int i = 64; i < IMAGE_SIZE - 1; i++) {
        ImageDataDifference[i] = ImageData[i] - ImageData[i + 1];
    }

    CompareValue = THRESHOLD; // Set initial threshold
    for (int i = 64; i < IMAGE_SIZE - 1; i++) {
        if (ImageDataDifference[i] > CompareValue) {
            CompareValue = ImageDataDifference[i];
            BlackLineRight = i; // Store position of detected edge
        }
    }
}

```

Abbildung 26 -Camera.c file / Black Lines Edges Funktionen

Die Funktion `findBlackLineRight()` dient dazu, die rechte Kante einer schwarzen Linie im erfassten Bild zu identifizieren. Dazu wird die Intensitätsdifferenz zwischen benachbarten Pixeln im Bild berechnet. Der Algorithmus beginnt ab Pixel 64 und läuft bis zum vorletzten Pixel des Bildes, wobei die Differenz zwischen jedem Pixel und dem darauffolgenden Pixel in einem separaten Array (`ImageDataDifference`) gespeichert wird. Anschließend wird die größte Intensitätsänderung gesucht, indem jedes Differenzwert mit einem vordefinierten Schwellenwert (`THRESHOLD`) verglichen wird. Wenn ein größerer Differenzwert gefunden wird, wird dieser als neuer Vergleichswert gespeichert, und die Position (`BlackLineRight`) der größten Intensitätsänderung wird festgehalten. Dies ermöglicht die präzise Erkennung der rechten Kante der schwarzen Linie.

Die Funktion `findBlackLineLeft()` arbeitet nach demselben Prinzip, jedoch mit umgekehrter Laufrichtung zur Erkennung der linken Kante der schwarzen Linie. Der Algorithmus beginnt bei Pixel 65 und arbeitet sich rückwärts bis zum ersten Pixel vor. Hierbei wird ebenfalls die Intensitätsdifferenz zwischen benachbarten Pixeln berechnet und in `ImageDataDifference` gespeichert. Anschließend wird die größte Intensitätsänderung gesucht, indem jeder Differenzwert mit dem Schwellenwert (`THRESHOLD`) verglichen wird. Wenn eine größere Differenz erkannt wird, wird die Position (`BlackLineLeft`) der größten Intensitätsänderung gespeichert. Durch diese Methode können sowohl die linke als auch die rechte Begrenzung der schwarzen Linie im Bild zuverlässig erkannt werden.

```

/* Function: calculateDeviation
- Finds the left and right black lines
- Calculates deviation from the center
*/
void calculateDeviation() {
    findBlackLineRight();
    findBlackLineLeft();
    RoadMiddle = (BlackLineRight + BlackLineLeft) / 2;
    Deviation = RoadMiddle - (IMAGE_SIZE / 2);
}

```

Abbildung 27 - Camera.c file / Calculate the Deviation Funktionen

Die Funktion `calculateDeviation()` dient dazu, die Abweichung der erkannten schwarzen Linie von der Bildmitte zu berechnen. Zunächst werden die Funktionen `findBlackLineRight()` und

`findBlackLineLeft()` aufgerufen, um die Positionen der rechten und linken Begrenzung der schwarzen Linie im Bild zu bestimmen. Anschließend wird die Mitte der Fahrbahn (`RoadMiddle`) berechnet, indem der Durchschnitt der erkannten linken und rechten Kanten gebildet wird. Schließlich wird die Abweichung (`Deviation`) ermittelt, indem die Differenz zwischen der berechneten Fahrbahnmitte und der tatsächlichen Bildmitte (`IMAGE_SIZE / 2`) bestimmt wird. Diese Abweichung gibt an, um wie viele Pixel die schwarze Linie von der idealen mittigen Fahrtrichtung abweicht, was für Steuerungsanpassungen genutzt werden kann.

## Main()-Funktion

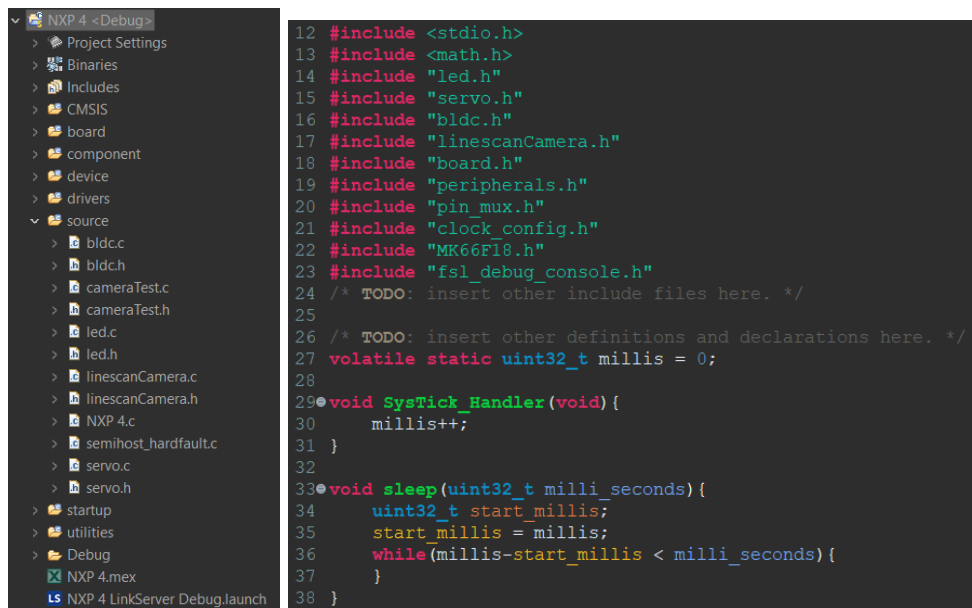


Abbildung 28 - Endgültige Programmstruktur      Abbildung 29 - NXP 4.c file

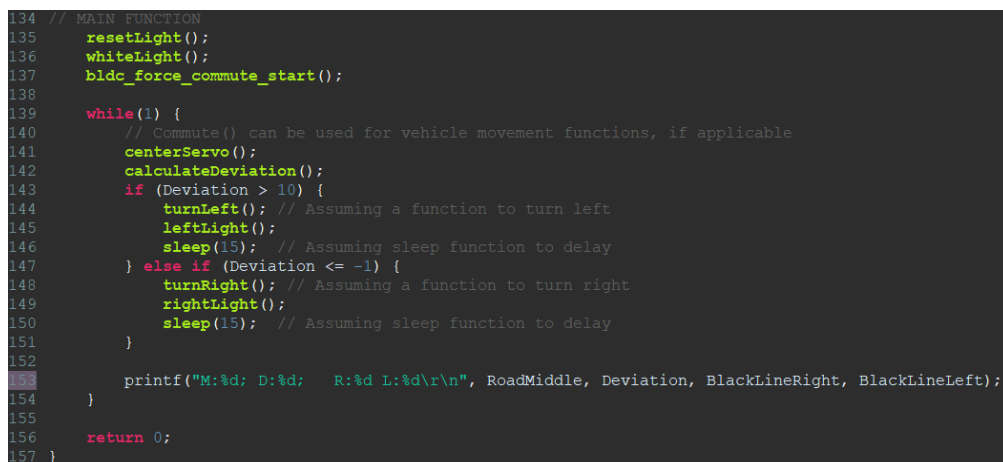


Abbildung 30 - main Funktion

Diese Hauptfunktion ist für die Steuerung des Fahrkorbs verantwortlich, indem sie seine Bewegung auf der Grundlage von Abweichungsberechnungen anpasst. Sie beginnt mit der Initialisierung der Lichter mittels `resetLight()` und `whiteLight()` und startet den BLDC-Motor mit der Funktion `bldcc_force_commute_start()`. Innerhalb der Endlosschleife zentriert es kontinuierlich den Lenkmechanismus, berechnet die Abweichung vom gewünschten Weg und nimmt entsprechende Korrekturen vor. Wenn die Abweichung größer als 10 ist, biegt das System nach links ab, aktiviert das linke Licht und wartet kurz. Ist die Abweichung negativ oder leicht außermittig, biegt es nach rechts ab und aktiviert das rechte Licht, bevor es eine Pause einlegt. Während der gesamten Schleife werden Echtzeitdaten über die Fahrbahnposition und die Abweichung ausgegeben, was bei der Debugging und der Überwachung des Systemverhaltens hilfreich ist.

## Probleme und Lösungen

Problem	Lösungen
<ul style="list-style-type: none"> <li>Die erwartete <b>Deviation</b> auf einer <b>geraden Strecke</b> war <b>0</b>, aber der tatsächliche Wert war <b>24</b>.</li> <li>Eine <b>negative Deviation</b> sollte das Auto <b>nach rechts</b> lenken, eine <b>positive Deviation</b> nach <b>links</b>.</li> <li>Die Funktion <code>turnRight()</code> funktionierte korrekt, aber <code>turnLeft()</code> nicht.</li> <li>Das Problem war, dass sowohl <code>centerServo()</code> (für die Geradeausfahrt) als auch <code>turnLeft()</code> <b>positive Werte</b> benötigten, was zu einem Konflikt führte.</li> </ul>	<ul style="list-style-type: none"> <li>Die Funktion <code>centerServo()</code> wurde <b>entfernt</b>.</li> <li>Das Auto konnte nun <b>korrekt nach rechts und links lenken</b>.</li> <li>Die <b>Vorwärtsbewegung</b> wurde durch die <b>BLDC-Motordrehzahl</b> gesteuert, anstatt durch <code>centerServo()</code>.</li> <li>Die Bewegung war <b>weniger flüssig</b>, aber ausreichend, damit das Auto <b>mit dem Kamerasensor fahren konnte</b>.</li> </ul>
<ul style="list-style-type: none"> <li><b>Das NXP-Car kann sich nicht nur durch Forced Commutation selbst antreiben.</b> Das Timing der Kommutierung wird leicht gestört und die erzeugte Leistung scheint nicht ausreichend zu sein.</li> </ul>	<ul style="list-style-type: none"> <li>Wir verwenden Free-Running. Er verwendet sein eigenes Timing, das auf dem Nulldurchgang basiert. Er kann sich selbst zuverlässiger in Bewegung halten.</li> </ul>
<ul style="list-style-type: none"> <li><b>Das Fahrzeug bewegte sich zu schnell</b>, als dass der Lenkalgorithmus das Fahrzeug präzise hätte steuern können.</li> </ul>	<ul style="list-style-type: none"> <li>Wir verwenden einen niedrigeren Tastgrad.</li> </ul>
<ul style="list-style-type: none"> <li>Der Mikrocontroller neigt dazu, in den Reset-Modus zu gehen, wenn eine hohe Lichtintensität verwendet wird, insbesondere wenn ein weißes Licht erzeugt wird.</li> </ul>	<ul style="list-style-type: none"> <li>Verwenden Sie eine niedrigere Intensität, indem Sie die Daten in <code>scrAddr</code> konfigurieren. Also <code>0xD26924</code> anstelle von <code>0xDB6DB6</code>.</li> </ul>