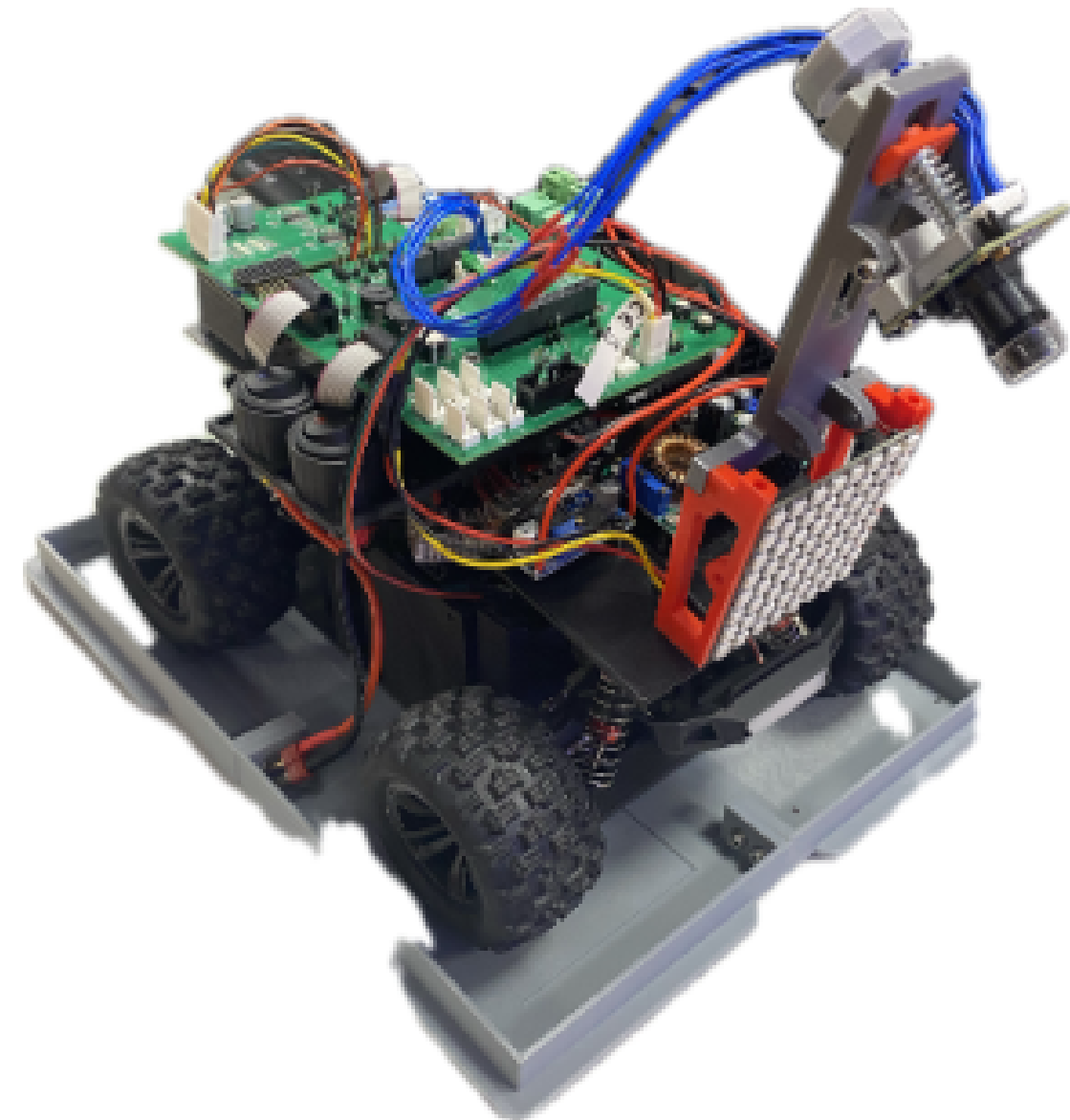


Mikrorechnertechnik

DriveX

- Ismail Shah Bin Iman Shah
- Adham Beshr



Overview

- ▶ Objective
- ▶ Hardware
- ▶ Peripherals
- ▶ Implementation
- ▶ LED Matrix
- ▶ Servo Motor
- ▶ BLDC Motor
- ▶ Line-Scan Camera
- ▶ Challenges



Objective



The objective of this project is to enable the **car to navigate the track autonomously**. By utilizing a line scan camera and integrating a microcontroller, the car will be able to detect the track's boundaries and adjust its movement accordingly. With real-time data processing and precise control of motors and steering mechanisms, the car will follow the track seamlessly, making decisions to stay on course without any human intervention.

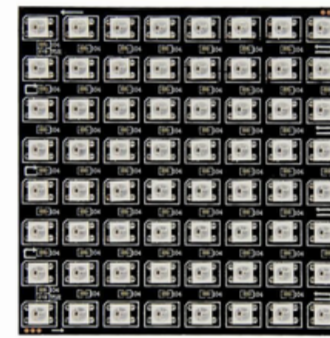
Hardware used

1) K66F NXP Microcontroller



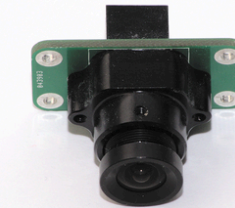
- **Processor:** ARM Cortex-M4, 180 MHz
- **Memory:** 1 MB Flash, 256 KB RAM
- **I/O:** Multiple GPIO, PWM, ADC, UART, I2C, SPI
- **Power:** 3.3V, low power consumption
- **Real-Time Processing:** Ideal for sensor and motor control
- **Sensor Support:** Compatible with line scan cameras and infrared sensors
- **Software:** MCUXpresso IDE support

2) WS2812B LED-Matrix



- **RGB LEDs:** Individually addressable with full RGB color control.
- **Single Data Line:** Can be controlled with just one data line.
- **Daisy Chain:** Multiple WS2812B matrices can be connected in a **chain for larger displays**.
- **Bright Display:** Ideal for lighting up the track with vibrant colors.

3) TSL1401R-LF Linescan Camera



- **Single Line Capture:** Captures one line of an image at a time based on light intensity.
- **Edge Detection:** High value differences in intensity indicate edges or changes in the environment.
- **Real-Time Processing:** Ideal for tracking lines or detecting boundaries in applications like track-following cars.

Hardware used

4) BLDC Motor



- **Uses 6-Step commutation** to control wheel movement efficiently.
- **Detects Back EMF** to enable sensorless operation and free running.
- **Higher efficiency** due to reduced energy loss (no brushes).
- **Low maintenance** since there are no brushes to wear out.
- **Longer lifespan** compared to brushed DC motors.

5) H Bridge



- Controls motor direction by **switching current flow**.
- **Enables commutation** in the BLDC motor.
- Works with PWM for speed and torque control.
- Allows braking and **freewheeling**.
- **Helps detect back EMF** for rotor position sensing.

6) Servo Motor



- **PWM Control:** Uses PWM signals to control the degree of deflection (position).
- **Steering:** Steers the direction of the car by adjusting the servo's angle.
- **Precise Movement:** Allows for precise adjustments in the car's orientation.
- **Efficient Control:** Provides smooth and accurate control for steering mechanisms.

Peripherals used

GPIO

- Controls digital inputs and outputs.
- Crucial for controlling BLDC commutation.
- Allow image capturing.

FTM

- Provides flexible timer functionalities.
- Used to generate PWM.

ADC

- Used for capturing analog signals from sensors like the Line Scan Camera.
- Crucial for allowing free running of BLDC.

DMA

- Efficiently transfers data between peripherals and memory.
- Minimizes CPU load during data transfer

SPI

- Enables fast serial communication with peripherals.
- Used in conjunction with DMA

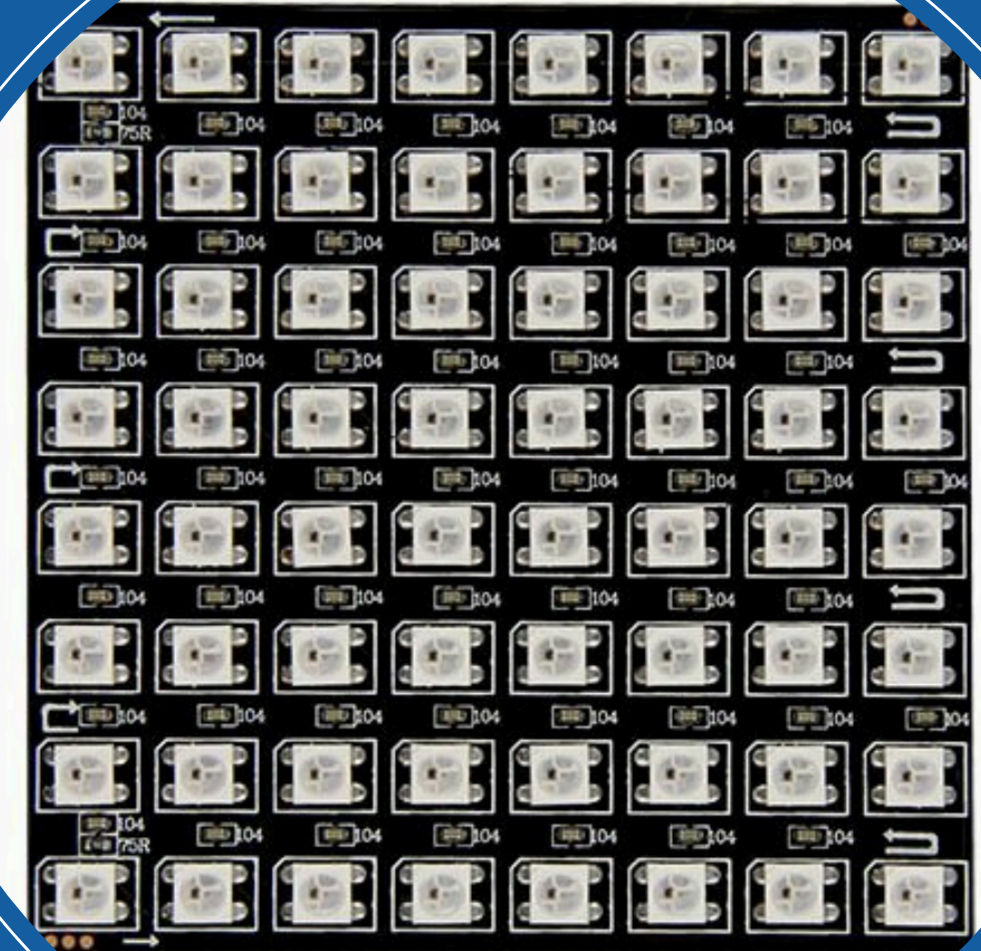
WS2812B LED-Matrix

Peripheral Utilised :

DMA – Obtain data from the memory directly

SPI – Transfer data from the microcontroller to the LED-Matrix

- Buffer of 3*3*8*8
- Functions:
 - whiteLight()
 - resetLight() } every 3 byte
 - rightLight()
 - leftLight() } every 9 byte
 - sendLEDData()
 - ws2812b_spi0_edma_handler()
- Bit Pattern:
 - ~ 0xD26924 = 1101 0010 0110 1001 0010 0100
 - ~ 0x924924 = 1001 0010 0100 1001 0010 0100



LED Code

```
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     sendLEDData(srcAddr);
58 }
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     sendLEDData(srcAddr);
70 }
```

```
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     sendLEDData(srcAddr);
89 }
90
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     sendLEDData(srcAddr);
108 }
```

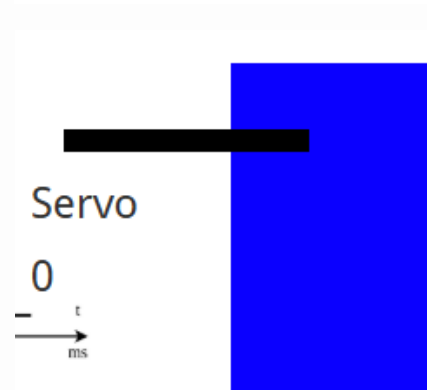
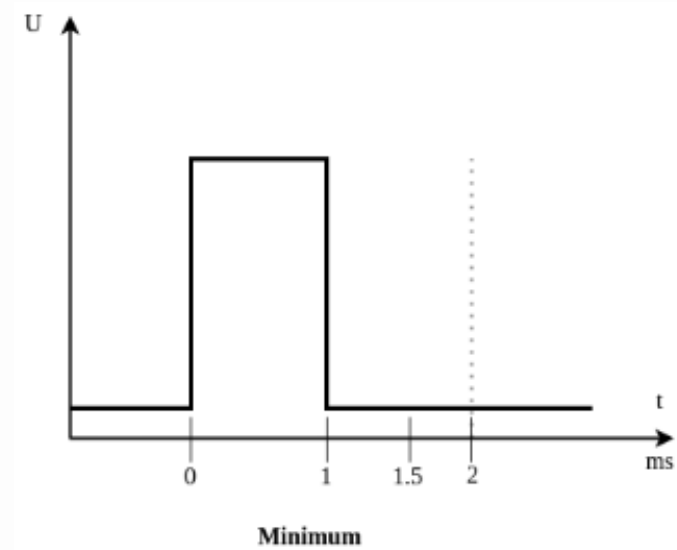

Servo Motor

Peripheral Utilised :

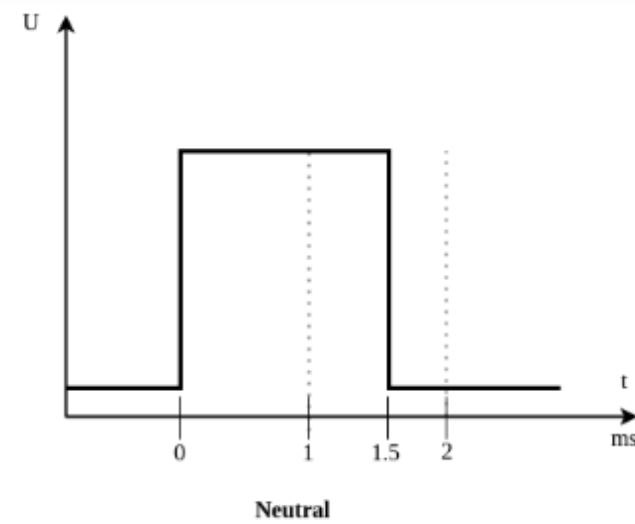
FTM – PWM Signal to control the deflection

- Period – 20ms
- Duty Cycle = $(\text{Pulse Width} / \text{Period Length}) \times 100$
- Functions:
 - `void setServoPosition(float duty_cycle)`

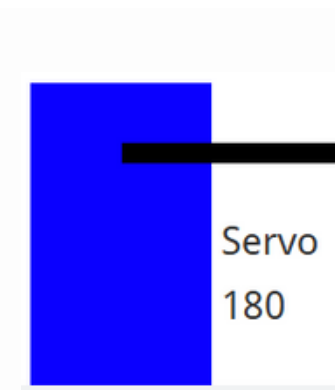
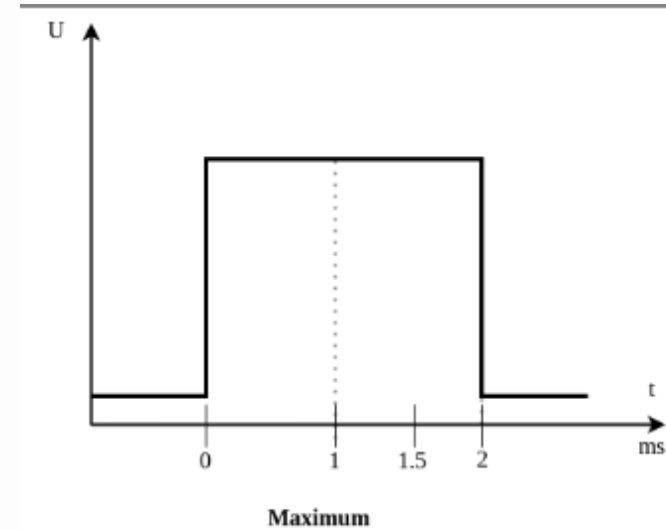
`turnLeft()` ~ 5.0



`centerServo()` ~ 7.5



`turnRight()` ~ 10.0



Servo Motor Code

```
#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
```

```
// Function to set the servo position based on the desired duty cycle
void setServoPosition(float duty_cycle) {
    // Update PWM duty cycle for SERVO0 (FTM3 CH6)
    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);
}
```

BLDC Motor

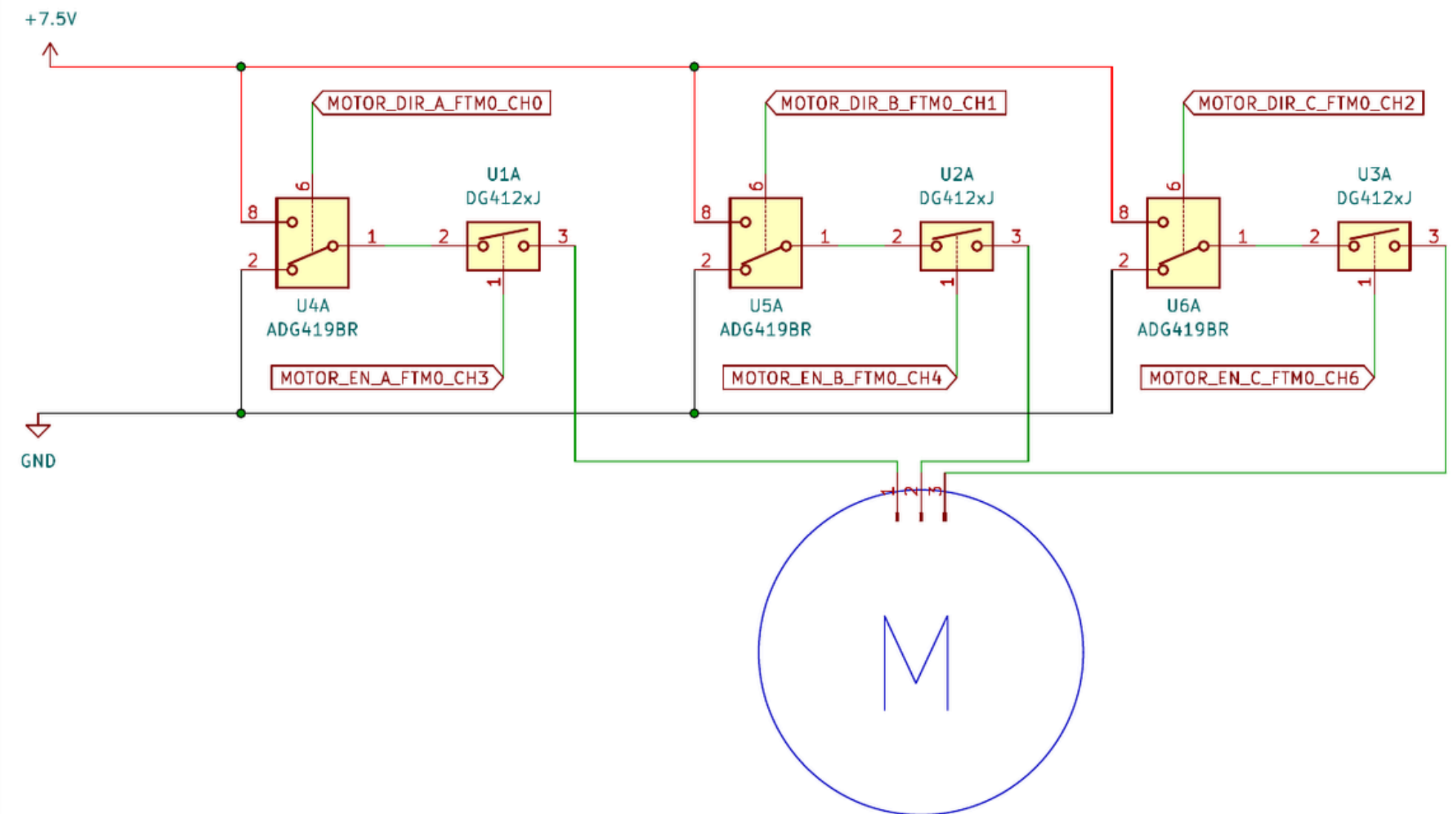
Peripheral Utilised :

ADC – Used to read the Back-EMF

FTM – Used to generate PWM signals for motor commutation.

GPIO – Controls the motor phase switching

- Utilises 3 “half” H-Brücke Module to induce commutation
- 6 Phases (2 Active and 1 Inactive per phase)
- Functions:
 - `configure_off()`
 - `reconfigure_for_phase()`
 - `commute()`
 - `bldc_motor_set_speed()`
 - `bldc_force_commute_start()`
 - `motor_bemf_irq_handler()`



BLDC Code

```
// Initializes the motor system
• void bldc_motor_init(void) {
    current_duty_cycle = START_DUTY_CYCLE;
    printf("Motor Initialized\n");
}

// Sets motor speed based on percentage input
• void bldc_motor_set_speed(int percentage) {
    if (percentage < 1) {
        bldc_motor_stop();
    } else {
        current_duty_cycle = 12 + (50 * (percentage / 100));
    }
}

// Stops the motor operation
• void bldc_motor_stop(void) {
    free_running = false;
    configure_off();
}
```

BLDC Code

```
// 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]]);
}
```

Linescan Camera



Peripheral Utilised :

ADC – To convert the Analogue signals to digital values

FTM – Sets an internal clock

GPIO – Signals the camera to start taking readings

- Image Size – 128
- SI to start Clock Cycle to tell the sensor to start reading the light values.
- 1 pixel for each clock cycle, one at a time.
- Functions:
 - ADC Interrupt handler
 - FTM Interrupt handler
 - findBlackLineRight()
 - findBlackLineLeft()
 - calculateDeviation()

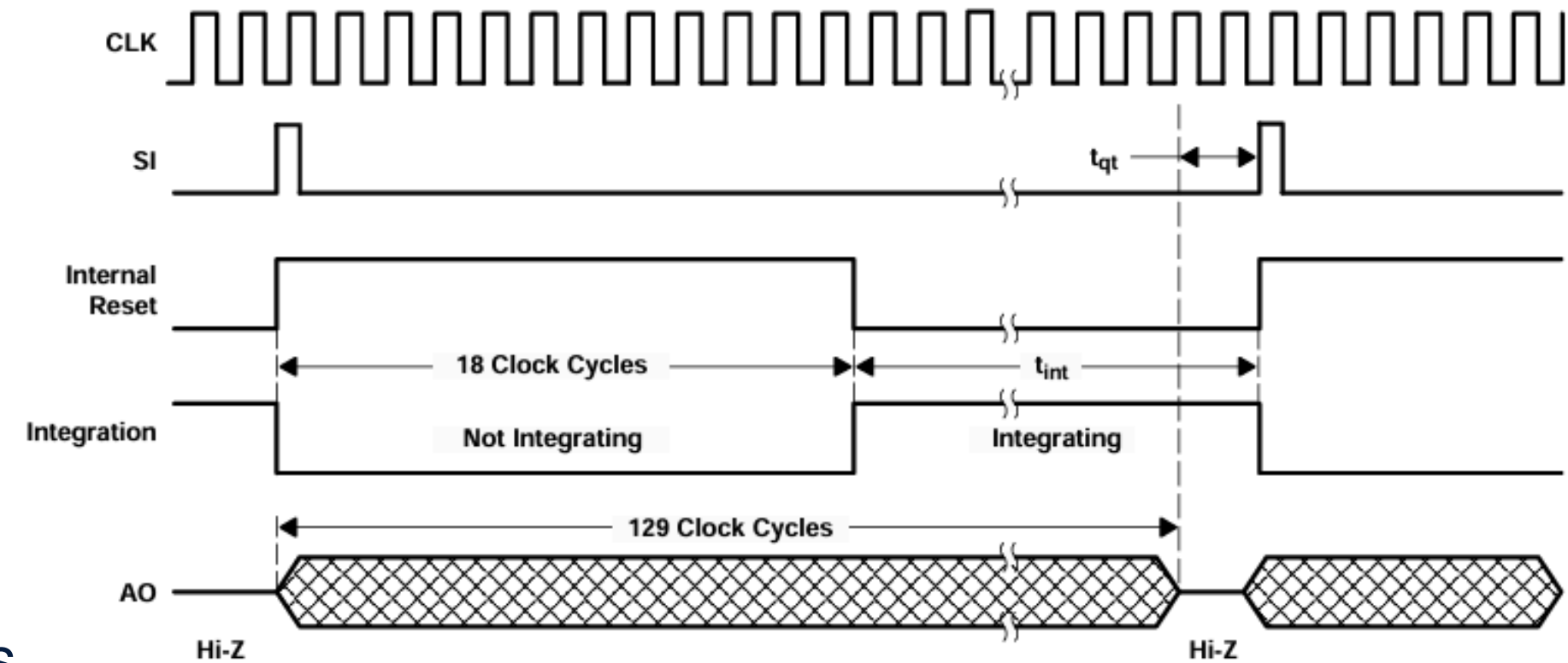


Figure 1. Timing Waveforms

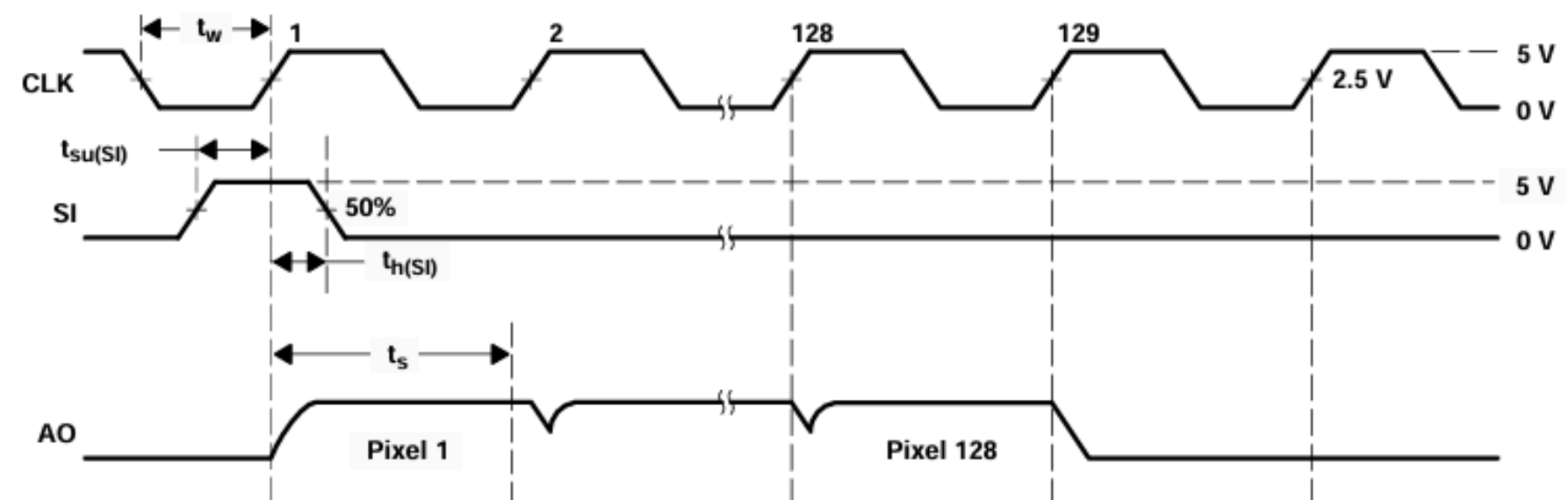


Figure 2. Operational Waveforms

Line Scan Camera Code

```
/* 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_S1_PTA9_GPIO, BOARD_PARALLAX1_S1_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_S1_PTA9_GPIO, BOARD_PARALLAX1_S1_PTA9_GPIO_PIN_MASK);
    }

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

```
/* 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

    #if defined _CORTEX_M && (_CORTEX_M == 4U)
        __DSB(); // Data synchronization barrier for Cortex-M4
    #endif
}
```

Line Scan Camera Code

```
/* 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);
}
```

```
/* 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
        }
    }
}
```

```
/* 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
        }
    }
}
```

Challenges

Problems

Microcontroller tends to lose connection when the intensity is too high across the LED-Matrix

Car is unable to move by itself when using forced commutation. Not enough power to start moving. The timing of the commutation is easily disrupted.

Deviation values were positive for both centerServo() and turnLeft() so there was a problem in detecting left curves .

Solutions

Use a lower intensity when configuring the LED-Matrix

Used a free-running method. It triggers commutation with its own timing based on the Back-EMF.

Implementing with only turnRight() and turnLeft() instead of centerServo()..

THANK YOU!

