Design Document

EN2160 - Electronic Design Realization

Department of Electronic and Telecommunication Engineering
University of Moratuwa

Capacitive Torque Sensor

Group Members:

De Zoysa.A.S.I - 220106D
Dayananthan.T - 220096T
Mathujan.S - 220389U
Pirathishanth.A - 220480P
Jeyasekara.S.P.R - 220257N
Sulojan.R - 220626V
Ananthakumar.T - 220029T
Ahilakumaran.T - 220017F

May 21, 2025

Contents

1 General Introduction

1.1 Overview

Torque measurement is a critical aspect in various mechanical and electromechanical systems, playing a vital role in applications ranging from industrial automation and automotive systems to robotics and biomedical devices. Accurate and reliable torque sensing is essential for performance monitoring, control, and safety in rotating machinery.

Conventional torque sensors such as strain gauge-based sensors and magnetoelastic sensors, while widely used, often suffer from limitations including sensitivity to environmental disturbances, mechanical wear, and complex signal conditioning requirements. In contrast, capacitive sensing offers a promising alternative due to its high sensitivity, low power consumption, compact form factor, and compatibility with modern digital signal processing techniques.

1.2 Functionality

This project focuses on the design and development of a **Capacitive Torque Sensor**, which operates based on the principle that the application of torque to a shaft induces a mechanical twist, resulting in a change in capacitance between specially arranged conductive plates. By accurately detecting and processing these capacitance variations, the system can quantify the applied torque.

The primary objective of this project is to develop a compact, sensitive, and cost-effective capacitive torque sensor that can be integrated into rotating machinery. The proposed system includes a high-resolution capacitance-to-digital converter (PCAP04), innovative plate designs to enhance sensitivity, and a digital telemetry system to wirelessly transmit data from the rotating shaft to a stationary receiver.

This report outlines the background theory, concept development, system design, component selection, and the overall implementation strategy of the capacitive torque sensing solution. The project aims to bridge the gap between theoretical torque measurement principles and practical, deployable sensor systems suitable for real-world engineering applications.

2 User Need Analysis

After observing videos of torque sensors used by individuals, we can get a sense of what they expect from a product and how they interact with existing products. It is also possible to see where current solutions cause inconveniences and frustrations to users and give us an idea about how to improve what is already available.

- **Light Weight**: First thing that was observed was the fact that individuals in videos handled the sensors often with one hand. This shows that the designed sensor has to be lightweight enough for users to handle comfortably. Also there were footages of torque sensors being attached to robot joints. Making the sensors heavy would put additional strain on limbs of the robot.
- **Accuracy and Precision**: There were videos of torque sensors being used in robots which did pick and placing of objects as well as medical robots which performed surgeries. Accuracy and precision of the measurements are critical in these scenarios since that data is fed into the control system for the robot.
- **Structural Rigidity**: Since there are deforming parts inside the sensor, it must be made sure that no part will be permanently deformed. Otherwise, it will require frequent sensor replacement. Proper material analysis and finite element analysis is required for this.
- **Easy to set up**: The sensors we observed had one cable attaching it to computers. The data from the sensor and the power to sensor is supplied through that. If users were able to use the sensor with no(minimal) calibration and set-up procedure, it would improve their experience of using the product.
- **Stability over time and Environmental Condition**: The sensor readings can change over time due to internal components being worn out(Sensor Drift). Environmental factors such as humidity and temperature can also affect capacitance. Users will require sensors that compensate for these issues.

3 Stimulate Ideas

To explore innovative approaches for measuring torque using capacitive sensing, we engaged in brainstorming sessions and concept generation techniques. Our goal was to identify methods that not only leverage the capacitive principle but also overcome practical limitations in real-world applications.

5.1 Basic Principle

The core idea behind capacitive torque sensing is based on the measurement of changes in capacitance caused by the deformation of a structure under torque. When a torque is applied to a shaft, it induces a slight twist or displacement. If capacitive plates are positioned in a way that their relative alignment changes with this twist, the resulting change in capacitance can be correlated directly to the applied torque.

5.2 Initial Concept: Variable Overlap Capacitive Plates

Our starting concept used interdigitated electrodes—alternating conductive fingers mounted on both rotating and stationary parts of the shaft. As the shaft twists under torque, the overlap area between these electrodes changes, altering the capacitance. This variation can be measured and processed to derive the torque.

To increase sensitivity and resolution, we explored the use of **high-permittivity dielectric materials** placed between the electrode pairs. This enhancement significantly amplifies the measurable capacitance change, making the sensor more responsive to even small torque variations.

5.3 Advanced Concept: Parallel-Plate Design with Radial Displacement

We developed an alternative design where capacitive plates are arranged radially around the shaft. Under torque, one plate set rotates slightly with the shaft, causing radial displacement relative to the stationary plates. This geometry maximizes the effect of twist on capacitance and allows for compact sensor integration.

5.4 Signal Acquisition and Processing

A major challenge with capacitive torque sensors is accurately capturing the small capacitance changes, especially in noisy environments. We selected the **PCAP04 capacitance-to-digital converter (CDC)** for its precision and low noise characteristics. This IC allows real-time digital readout of capacitance variations with high resolution.

To improve performance further, we considered **differential measurement techniques**, using a reference capacitor to eliminate common-mode noise and temperature effects.

4 Our Approach

In the initial phase of our project, we conducted extensive background research on capacitive torque sensing technologies. A key reference was the patent titled **Differential Capacitive Torque Sensor**, which disclosed a multi-disk configuration for torque measurement. Building on this concept, we developed a modified three-disk assembly optimized for sensitivity and manufacturability.

4.1 Mechanical Design

Our sensor consists of three coaxial disks (Figure 1):

- **Rotor 1A and Rotor 1B**: Two outer **metal disks**, fixed to the stationary housing (stator). These serve as the capacitive sensing electrodes.
- **Rotor 2**: A central **dielectric disk**, rigidly attached to the rotating shaft. Its angular displacement under torque modulates the overlapping area between the metal disks.

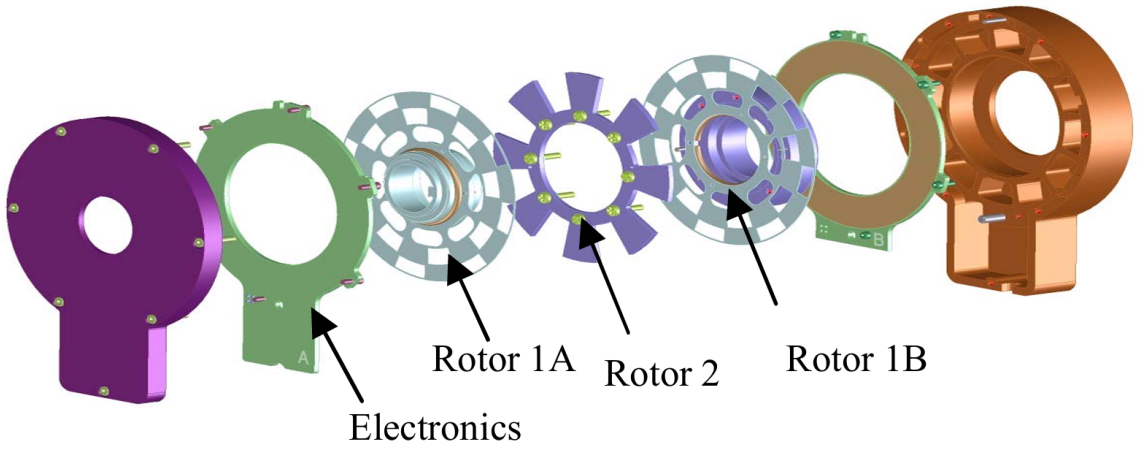


Figure 1: Exploded view of the sensor assembly. The central dielectric disk (Rotor 2) rotates with the shaft, while the outer metal disks (Rotor 1A/B, gray) remain stationary.

4.2 Working Principle

When torque is applied:

1. The shaft twists, causing the dielectric disk (Rotor 2) to rotate relative to the fixed metal disks.
2. This rotation changes the effective overlapping area between the dielectric and metal disks, creating a **differential capacitance** (Figure 2).
3. The capacitance variation is measured by electronics connected to the metal disks, proportional to the applied torque.

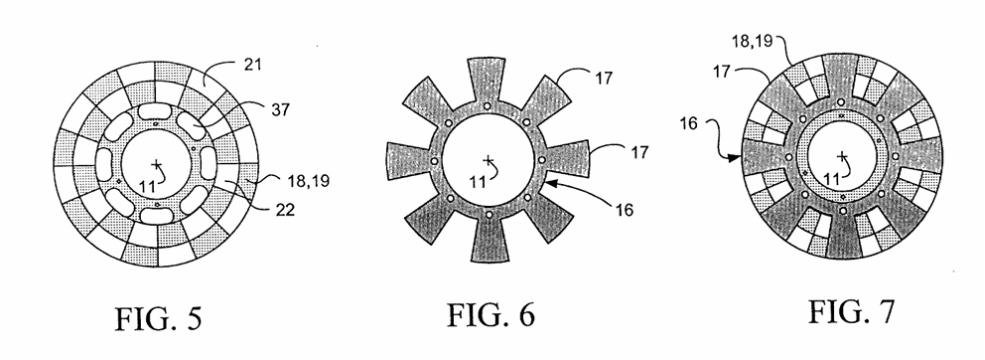


FIG. 5 FIG. 6 FIG. 7

Figure 2: Top-down views of disk geometries: (a) Metal disks (Rotor 1A/1B) with symmetric electrode patterns; (b) Dielectric disk (Rotor 2) with alternating permittivity regions.

4.3 Advantages of the Three-Disk Configuration

- **Enhanced Sensitivity**: The dielectric disk amplifies capacitance changes compared to air gaps.
- **Reduced Crosstalk**: Fixed metal disks minimize parasitic capacitances from shaft movement.
- **Linear Response**: The symmetric design ensures a linear relationship between torque and capacitance.

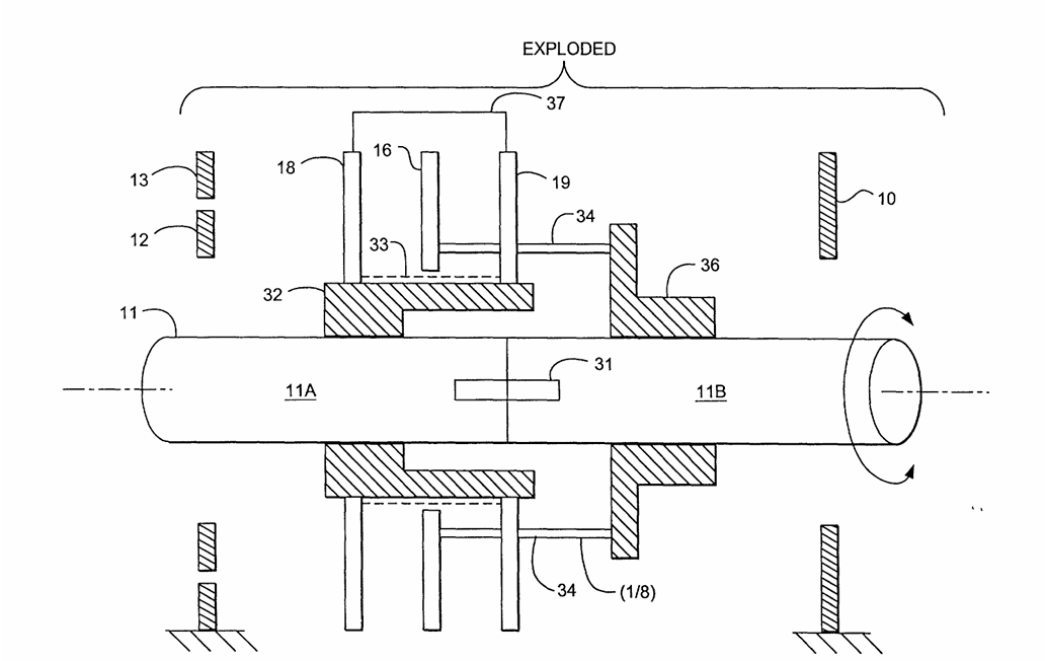


Figure 3: Cross-sectional schematic showing torque transfer (blue arrows) and capacitive coupling (red fields) between disks. The dielectric disk's rotation modulates the electric field between the metal disks.

This design, inspired by the patent but refined for our application, ensures reliable torque measurement while simplifying assembly and calibration.

5 Design and Development Timeline

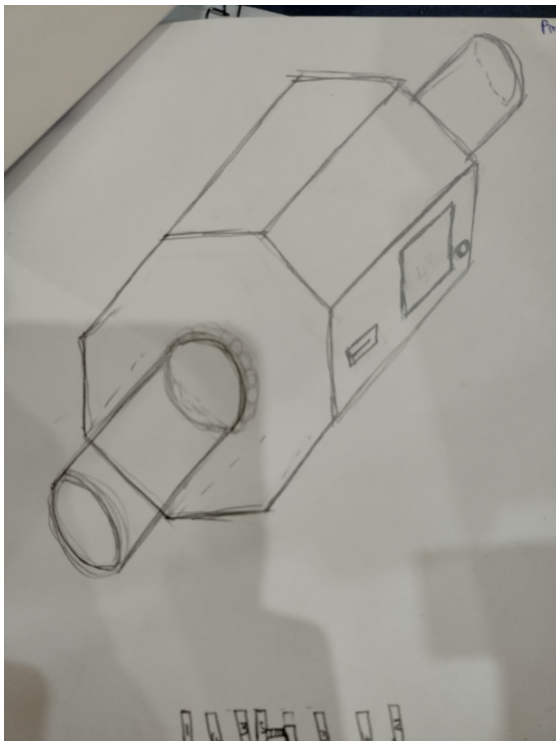
The product development followed a systematic approach with the following chronological stages:

1. **Research About Existing Products - (Week 05)**
    - Investigated similar products in the market
    - Analyzed about patents or Research papers about product
    - Identified gaps and improvement of existing product
2. **Conceptual Design Modeling - (Week 06)**
    - Developed initial concept sketches
    - Created basic 3D models of key components
    - Evaluated different design approaches
3. **SolidWorks Modeling Iteration 01 - (Week 07)**
    - First complete 3D CAD model of the assembly
    - Focused on overall dimensions and form factor
    - Identified major mechanical interfaces
4. **FEA Analysis for Shaft - (Week 08)**
    - Performed stress analysis on critical shaft components
    - Evaluated deformation under various Torque values
    - Optimized material selection based on results
5. **SolidWorks Modeling Iteration 02 - (Week 08)**
    - Incorporated FEA feedback into design
    - Refined mechanical interfaces
    - Improved ergonomics and manufacturability
6. **Calculations for Components and Component Selection - (Week 09)**
    - Performed engineering calculations for Capaciatance, CDC and MCU
    - Selected appropriate bearings, products, and materials
    - Verified component specifications against requirements
7. **SolidWorks Modeling Iteration 03 - (Week 10)**
    - Finalized 3D model with all components and Sir's advice and Guidance
    - Changed the overall enclosure design alligned with our reference product
    - Prepared technical drawings for production
8. **Schematics Design - (Week 11)**
    - Developed electronic circuit diagrams
    - Selected appropriate sensors and ICs
    - Designed power distribution network

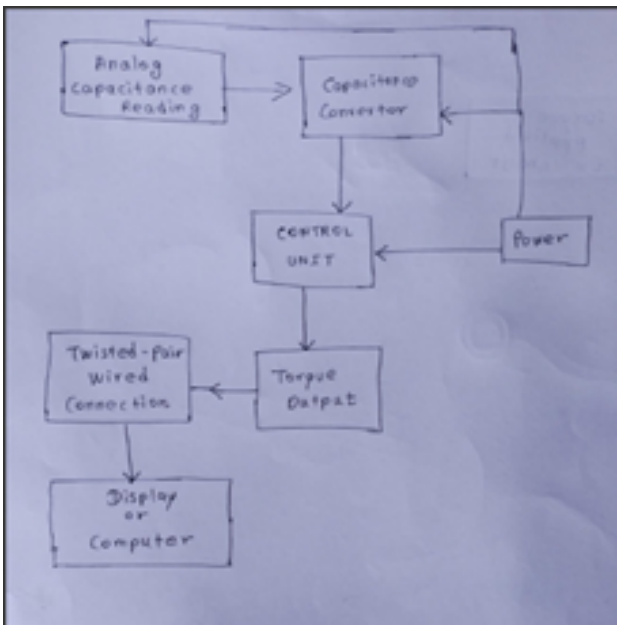
9. **PCB Design - (Week 12)**
    - Created 4-layer board layout in Altium Designer
    - Optimized for signal integrity and EMI
    - Incorporated capacitive sensing functionality
10. **Metal Cutting and Making Enclosures - (Week 13)**
    - Manufactured disks and mechanical components
    - Fabricated custom enclosures
    - Machined shaft to precise specifications
11. **Assembling Mechanical Parts - (Week 14)**
    - Verified fit and function of all components
    - Performed alignment and calibration
    - Prepared final assembly for testing

6 Conceptual Designs

6.1 Conceptual Design 1



(a) Enclosure and Shaft Design



(b) Block Diagram

Enclosure Design

The hexagonal enclosure with cylindrical shafts is compact and stable for housing capacitive electrodes. A non-conductive material is recommended to avoid interference.

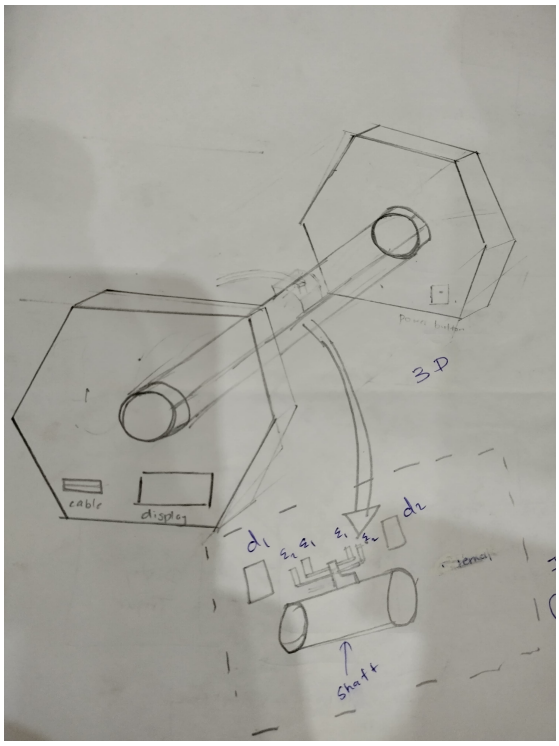
Shaft Design

The cylindrical shafts enable torque transmission and deformation for capacitance measurement. Adding a high-permittivity dielectric, as suggested in prior methodologies, could enhance sensitivity.

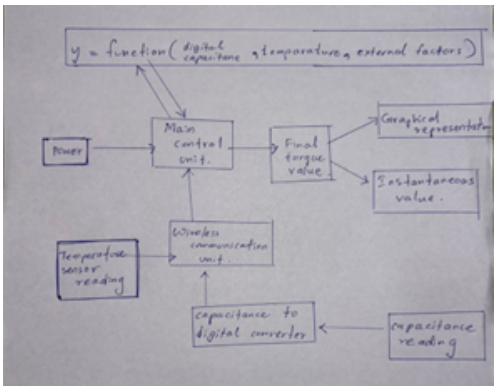
Block Diagram

The system converts analog capacitance readings into torque output via a control unit, using a twisted-pair wired connection for reliable data transfer. Wireless telemetry could improve flexibility.

6.2 Conceptual Design 2



(a) Enclosure and Shaft Design



(b) Block Diagram

Figure 5: Conceptual Design 2

Enclosure Design

The dual hexagonal enclosures with cylindrical shafts provide structural stability for housing capacitive electrodes. The compact design suits space-limited applications, but a non-conductive material is essential to prevent interference.

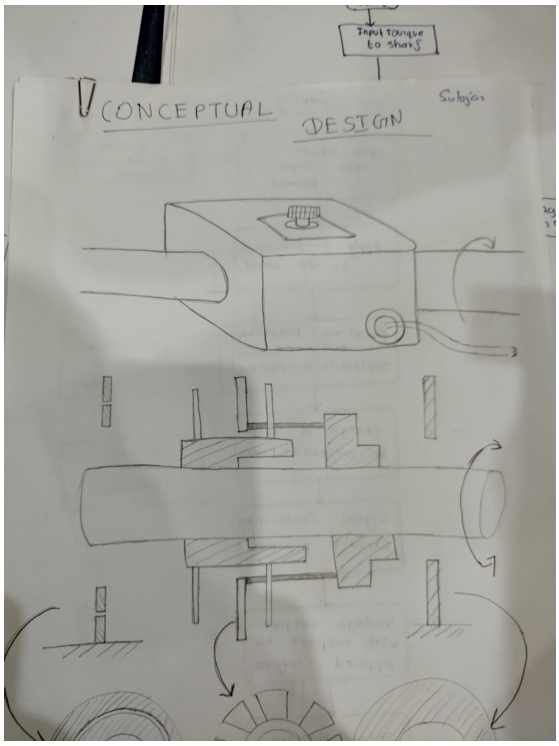
Shaft Design

The cylindrical shaft connecting the enclosures transmits torque, enabling deformation for capacitance measurement. Dimensions (d1, d2, 3D) suggest a focus on precision, but a high-permittivity dielectric could enhance sensitivity.

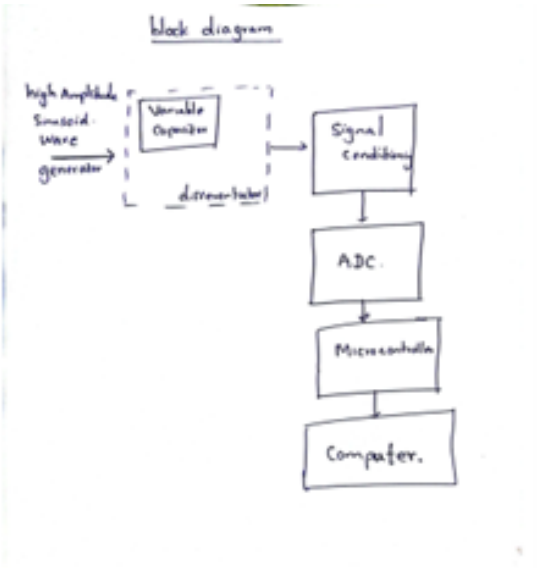
Block Diagram

The system uses a capacitance-to-digital converter to process readings, feeding into a main control unit. Wireless communication transmits the final torque value to a graphical representation on a display or PC. Temperature sensor readings account for external factors, improving accuracy.

6.3 Conceptual Design 3



(a) Enclosure and Shaft Design



(b) Block Diagram

Figure 6: Conceptual Design 3

Enclosure Design

The rectangular enclosure with cylindrical shafts provides a stable housing for the capacitive sensor. Its design supports integration into a rotary system, but using a non-conductive material is critical to avoid interference.

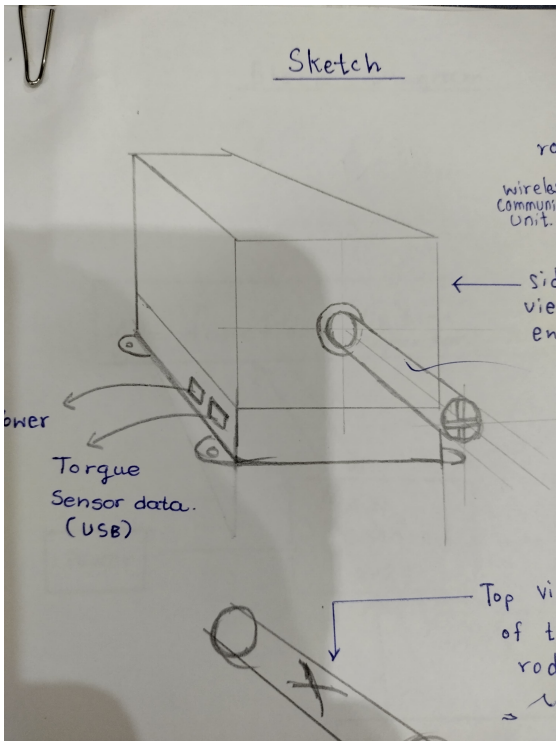
Shaft Design

The cylindrical shaft, with a gear-like component, transmits torque, causing deformation for capacitance measurement. The gear may aid in precise torque application, but a high-permittivity dielectric could improve sensitivity.

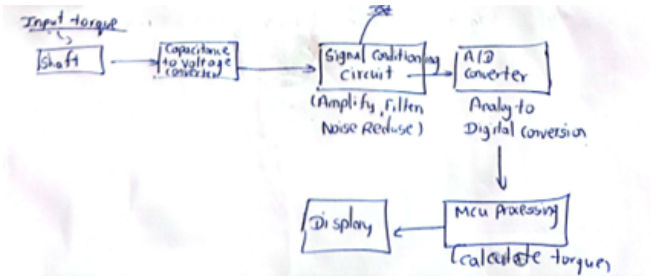
Block Diagram

A high-amplitude sinusoidal wave generator powers a variable capacitor (distometer), feeding into signal conditioning, an ADC, and a microcontroller, with final output to a computer. This setup converts capacitance changes into digital torque data effectively.

6.4 Conceptual Design 4



(a) Enclosure and Shaft Design



(b) Block Diagram

Figure 7: Conceptual Design 4

Enclosure Design

The rectangular enclosure with a cylindrical shaft is sturdy for housing the sensor. Feet on the base suggest stable mounting, but a non-conductive material is necessary to avoid interference.

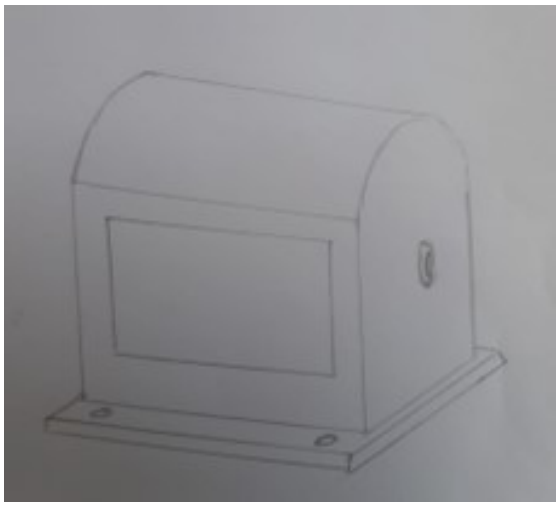
Shaft Design

The cylindrical shaft transmits torque, enabling deformation for capacitance measurement. The top view shows a rod for torque input, but a high-permittivity dielectric could improve sensitivity.

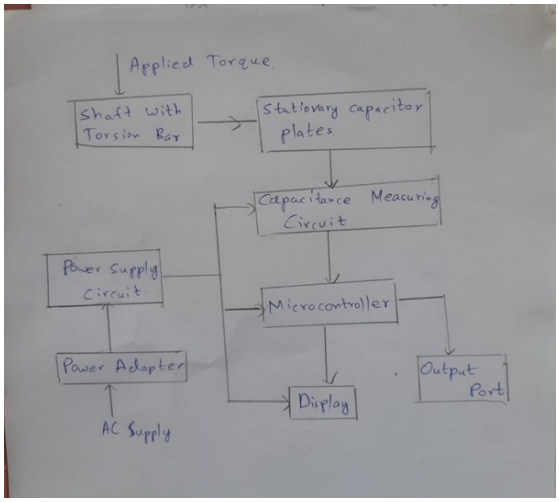
Block Diagram

The system converts capacitance to voltage, processes it through a signal conditioning circuit (amplify, filter, noise reduction), and uses an A/D converter and MCU for torque calculation, displayed on a screen. USB torque data output and wireless communication enhance flexibility.

6.5 Selected Design previously



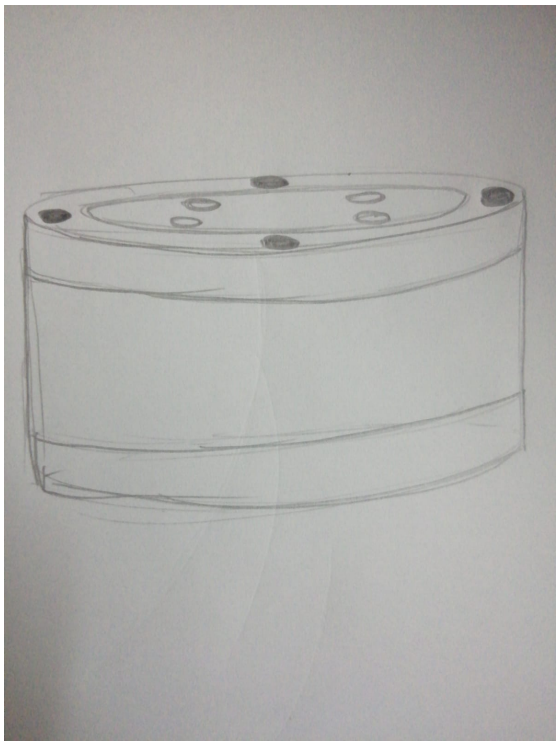
(a) Enclosure and Shaft Design



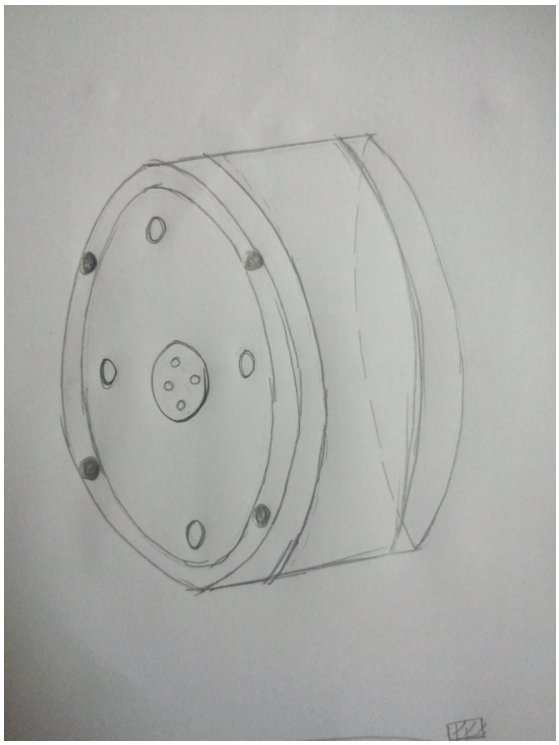
(b) Block Diagram

Figure 8: Final Selected Design

6.6 Selected Design Final



(a) View 01



(b) View 02

Figure 9: Final Selected Design

Enclosure Design

The semi-cylindrical enclosure with a flat base and mounting feet ensures stability and easy integration. The compact design is practical, but a non-conductive material is essential to prevent interference with capacitance measurement.

Shaft Design

The shaft with a torsion bar, as shown in the block diagram, transmits torque, causing deformation between stationary capacitor plates. Incorporating a high-permittivity dielectric could enhance sensitivity.

Block Diagram

Torque applied to the shaft alters capacitance, measured by a circuit, processed by a microcontroller, and displayed. A power supply circuit with an adapter ensures reliable operation, and an output port allows data transfer. The setup is straightforward and effective.

7 Evaluation of the Designs

Design	User Need	Design 01	Design 02	Design 03	Design 04
Enclosure	Ergonomics	6	8	8	6
	Durability	6	6	6	9
	Size & Weight	7	8	5	7
	Manufacturability	8	7	8	6
	Robustness	6	6	7	6
	Cost Effectiveness	8	9	6	8
Block Diagram	Accuracy	7	7	7	8
	Signal Integrity	6	7	8	5
	Cost Effectiveness	7	8	5	7
	Ease of Use	8	7	7	8
	Power Efficiency	9	7	6	7
	Scalability	7	7	8	6
Overall		85	87	81	83

Table 1: Evaluation Table for Design Aspects

8 Capacitance and CDC Requirements Evaluations

8.1 Capacitance Calculation

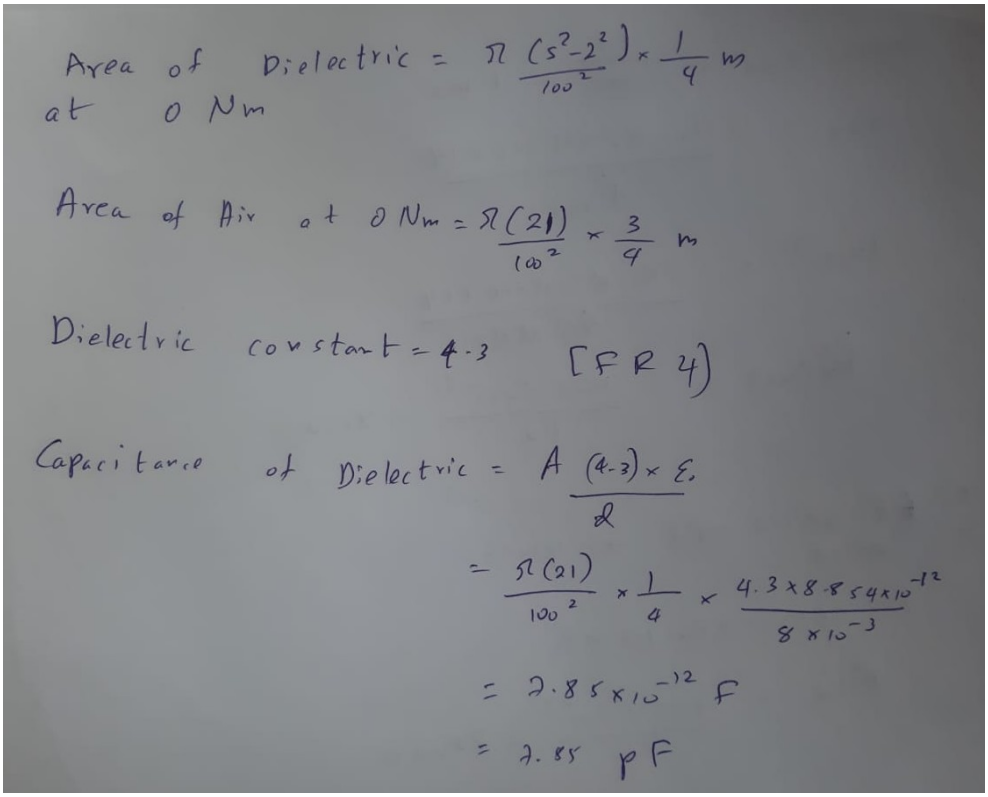


Figure 10: Area and Capacitance Calculation

8.2 Change of Capacitance

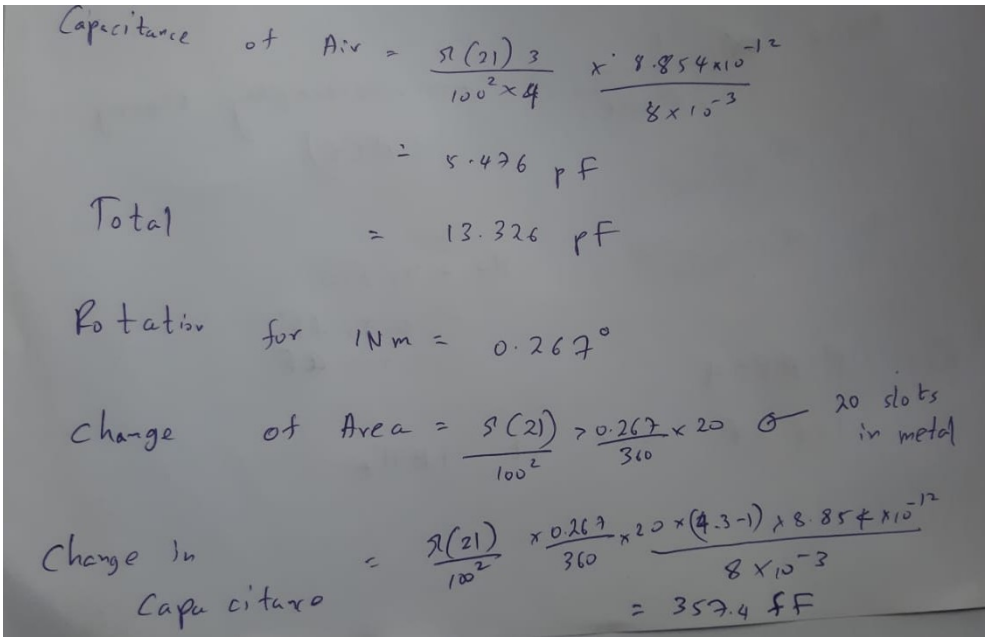


Figure 11: Capacitance calculation

8.3 CDC Calculations

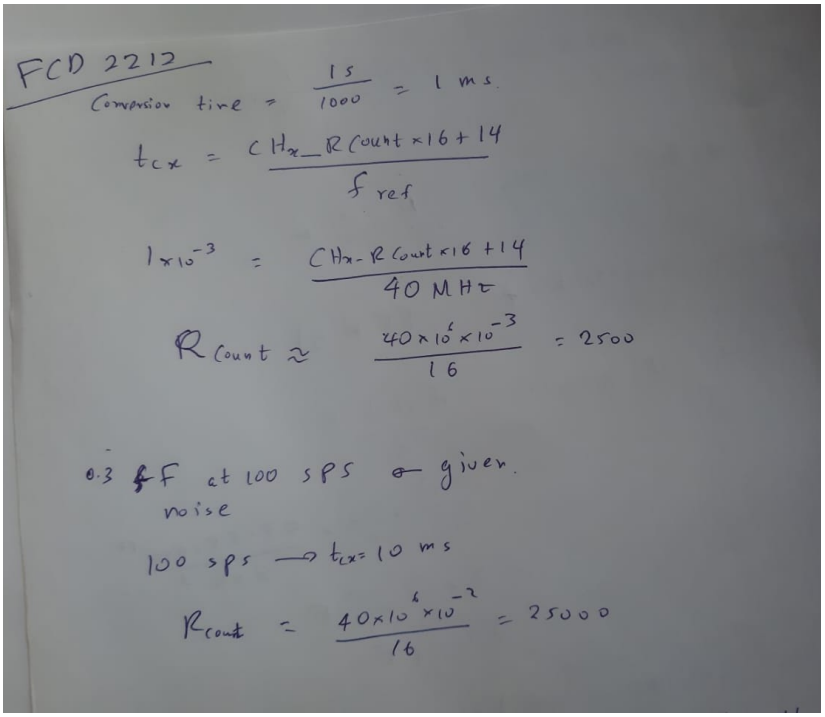


Figure 12: CDC calculation(1)

9 CDC Evaluation

The requirements for choosing a capacitance-to-digital converter (CDC) were:

- Resolution – Able to measure few femto Farads of capacitance change.
- Sampling frequency – 1000 samples/second
- Range – ±360 fF

	AD7746	AD7150	FDC1004	PCAP04	FDC2114
Frequency	2	4	6	10	10
Resolution	9	6	7	8	6
Range	7	8	9	9	9
Environmental Compensation	9	6	7	8	8
Price	6	8	9	7	7
Availability	10	7	9	8	8
Total	43	39	47	50	48

Figure 13: CDC Evaluation table

AD7746, AD7150, and FDC1004 could not support the 1000 Samples/second requirement and were therefore eliminated. Although the AD7746 had 4aF resolution, its sampling frequency was only 10Hz, which was well below the requirements. The final selection was between PCAP04 and FDC2114. Both satisfied the required sampling rate and capacitance range, but PCAP04 had better resolution at 1kHz.

PCAP Qualities

- 156aF resolution at 1kHz
- ±50pF range
- Temperature compensation

Figure 14: PACP04 CDC

10 MCU Calculation

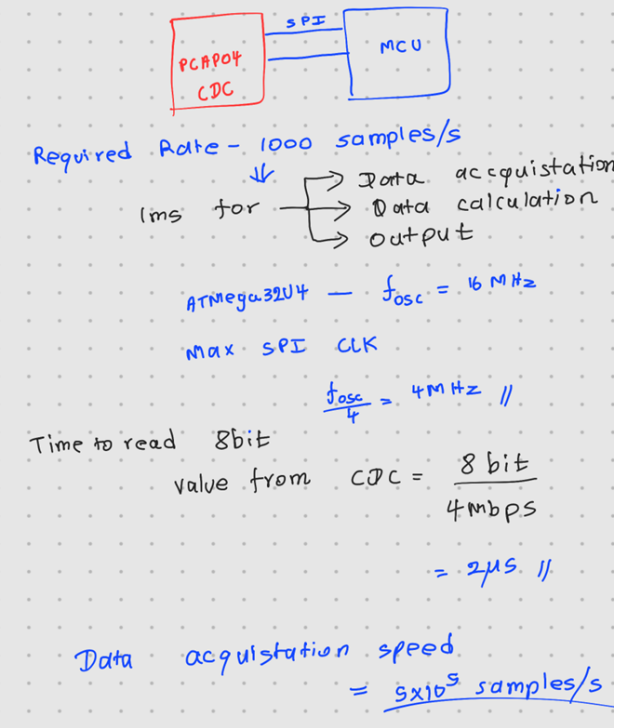


Figure 15: MCU calculation(1)

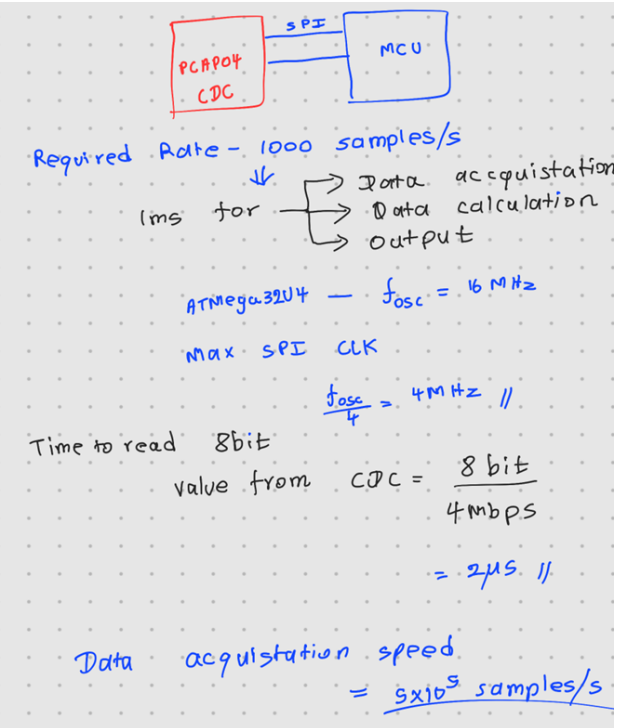


Figure 16: MCU calculation(2)

11 MCU Evaluation

Comparison Table

MCU	ATmega32U4	ATmega32U2	ATmega64M1	PIC18F46J50	EFM8UB20
Performance	8	7	10	10	9
Size (Compactness)	8	9	9	8	9
USB Capability	10	10	10	10	10
EMC/EMI Robustness	7	7	8	7	9
Ease of Development	10	9	8	7	7
Documentation	9	8	7	8	7
Availability	10	10	8	9	7
Cost	7	8	6	7	9
Total	69	68	66	66	67

Selection Criteria

- Native USB support
- Enough performance for 1kHz sample processing
- Ease of development
- EMC/EMI resilience

Chosen MCU – ATmega32U4

Specifications:

- 16MHz / 32KB Flash / 2.5KB SRAM
- Compact – 44 pins
- Microchip Studio free IDE / beginner friendly
- Internal crystal oscillator / brownout protection

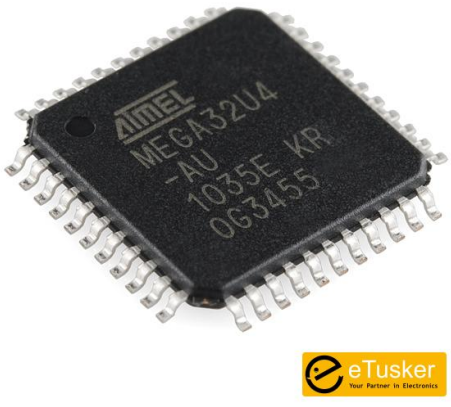


Figure 17: ATMega32U4

12 Dielectric Material Evaluation

Desired Specifications

- Higher dielectric constant
- Good strength
- Ease of machining
- Lower cost

	PTEF (Teflon)	Polyimide (Kapton)	Alumina Ceramic (Al_2O_3)	FR4 (Glass Epoxy)	Polycarbonate
Dielectric Constant	7	9	10	8	6
Flexural Strength	4	7	9	10	5
Ease of Machining	8	6	3	7	9
Stability	9	8	10	8	7
Cost	6	4	3	9	10
Total	34	34	35	42	37

Figure 18: Evaluation Table

Al_2O_3 was initially considered due to its high dielectric constant and excellent stability. However, it is brittle and requires diamond cutting/laser sintering, making it unsuitable for the project.

Chosen Material – FR4 (Glass Epoxy)

Properties:

- Dielectric constant – Around 4.5
- Shear Strength – Approx. 25,000 psi
- Flexural Strength – Approx. 60,000 psi
- Machinability – CNC milling, laser cutting, water jet cutting
- Stability – 140°C / Water Absorption 0.1%

Figure 19: FR4 - Epoxy Glass

13 Shaft Design

The shaft for mounting metal/dielectric disks was designed and simulated in SolidWorks. Several shaft designs were explored with different geometries and materials.

Design Requirements

- High deformation at 1Nm – measurable capacitance difference
- Must not exceed the shear stress of material at 5Nm

Design Iterations

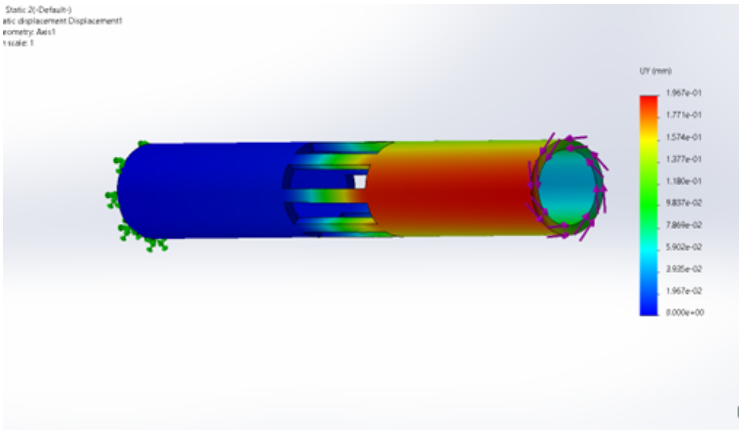


Figure 20: Short shaft – similar to the reference design

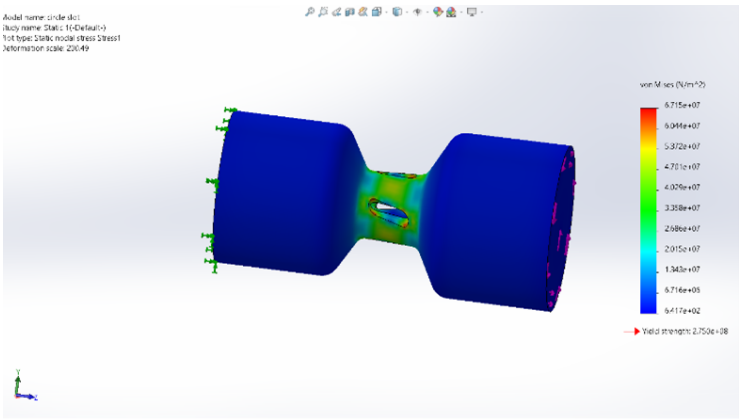


Figure 21: Shaft with pill-shaped cutouts

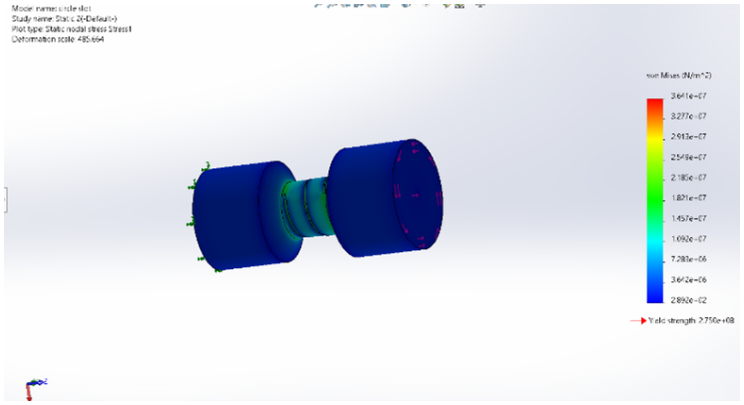


Figure 22: Shaft with helical groove

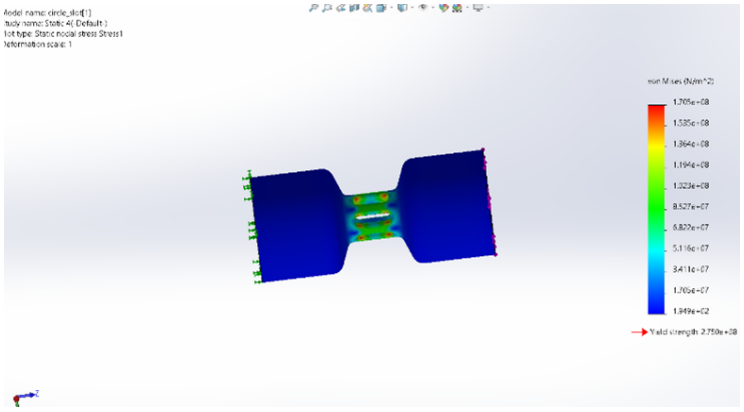


Figure 23: Pill-shaped cutouts + fillets

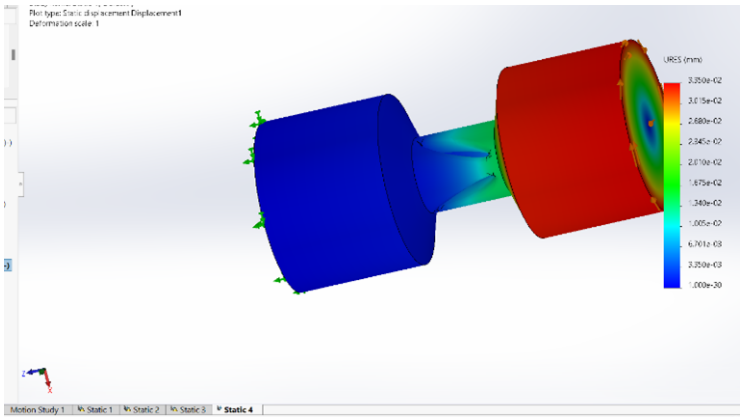


Figure 24: Angled pill-shaped cutouts + fillets

All versions with cutouts exceeded the shear stress limit at 5Nm. The final design was a shaft with no cutouts and a reduced diameter.

Final Shaft Design

- Diameter at ends – 1.7 cm
- Diameter at middle – 0.6 cm
- Thickness – 2 mm
- Deformable region length – 1.4 cm
- Material – Aluminum alloy 6061-T6

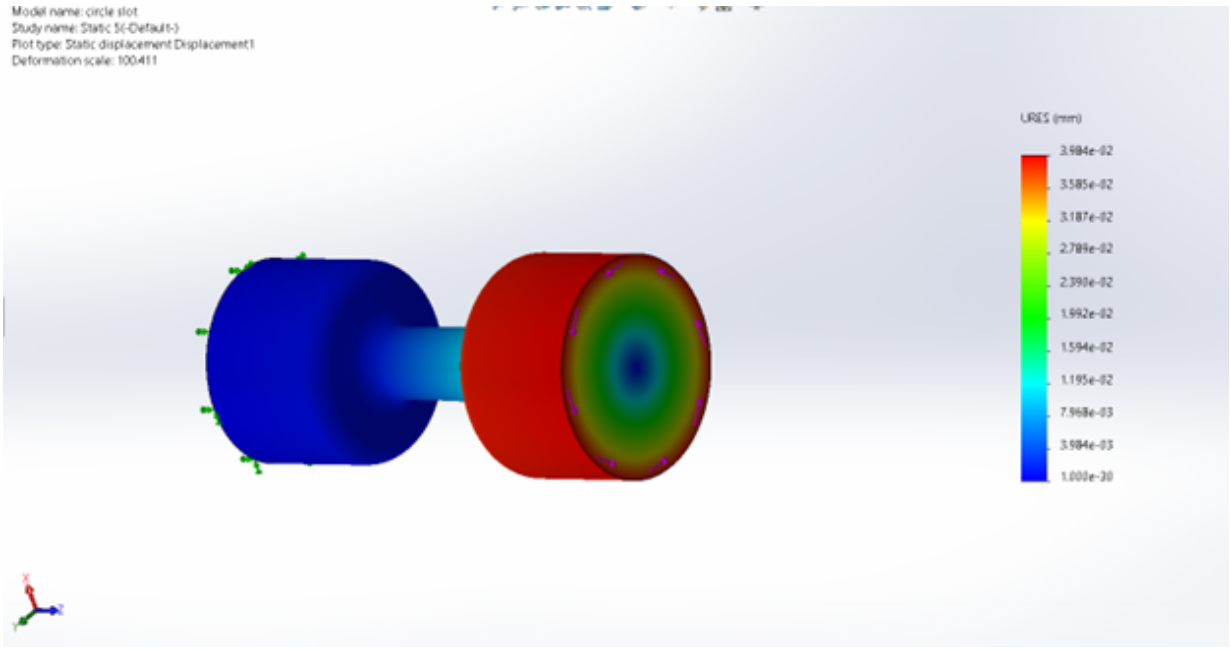


Figure 25: Final Shaft

Performance:

- Deformation at 1Nm – 3.984×10^{-4} mm
- Angle of rotation – 0.2680 degrees
- Change in capacitance – 357.4 fF (measurable with 156aF resolution of PCAP04)

14 Schematic Circuit Design

14.1 MCU Circuit

- **Core Components**:
  - ATmega32U4 microcontroller
  - Key functions:
    - Reads capacitance via SPI
    - Computes torque using calibration equation
    - Manages USB communication
- **Power Management**:
  - 5x V_{CC} pins with decoupling capacitors
  - AVCC pin filtered via ferrite bead
- **Clock**: ECS-8FMX-160-TR 16MHz oscillator for USB timing
- **UI**: Reset button, 3x LEDs (USB/SPI/status)

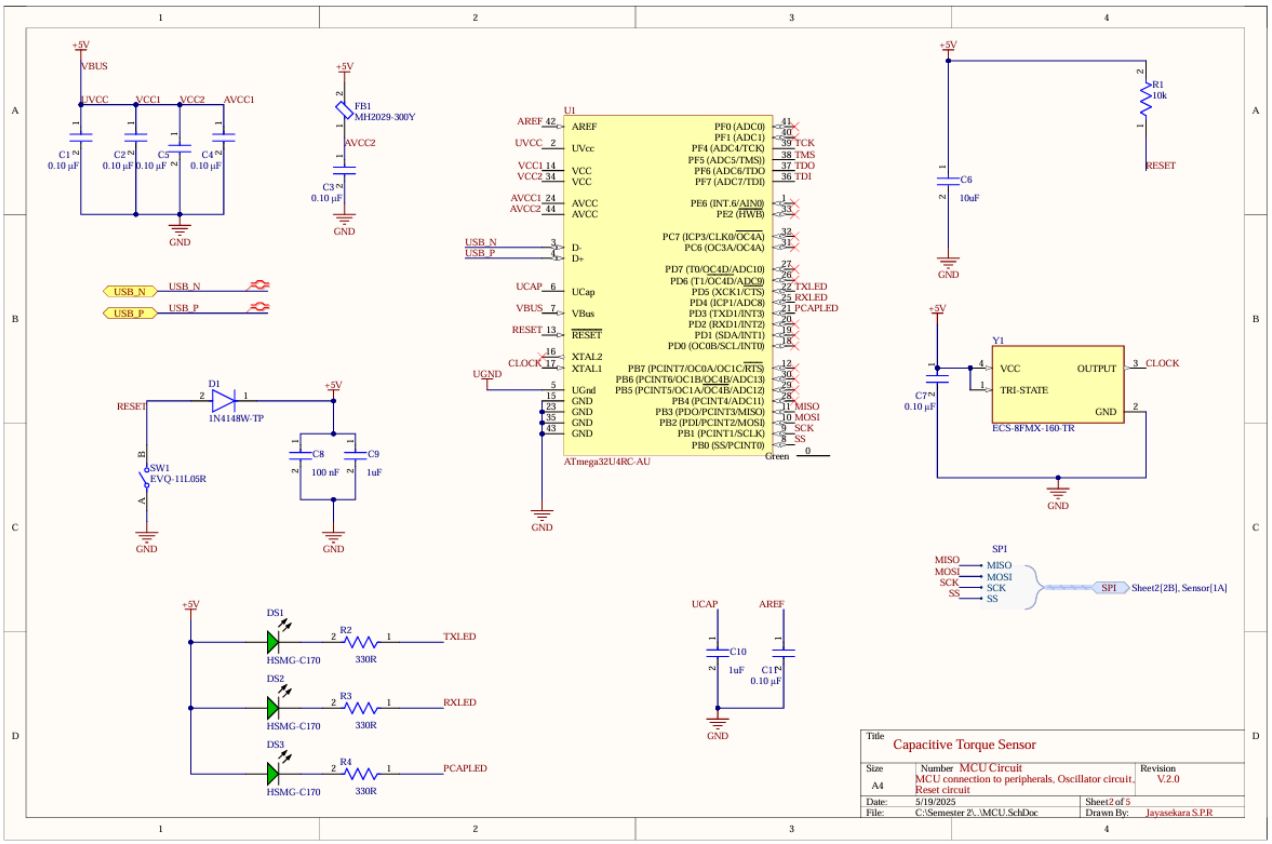


Figure 26: MCU Schematic

14.2 USB Circuit

- Provides **data transmission** and **power supply** (+5V via VBUS).
- **ESD Protection**:
  - +D/-D lines connected to ground through **varistors**.
- **Grounding**:
  - USB ground (UGND) and main ground linked via **ferrite bead** for:
    - High-frequency noise filtering
    - Prevention of ground loops
- **Power Line Protection**:
  - Resettable fuse for overcurrent protection
  - Capacitor C12 for ripple smoothing
  - LED DS4 as power indicator

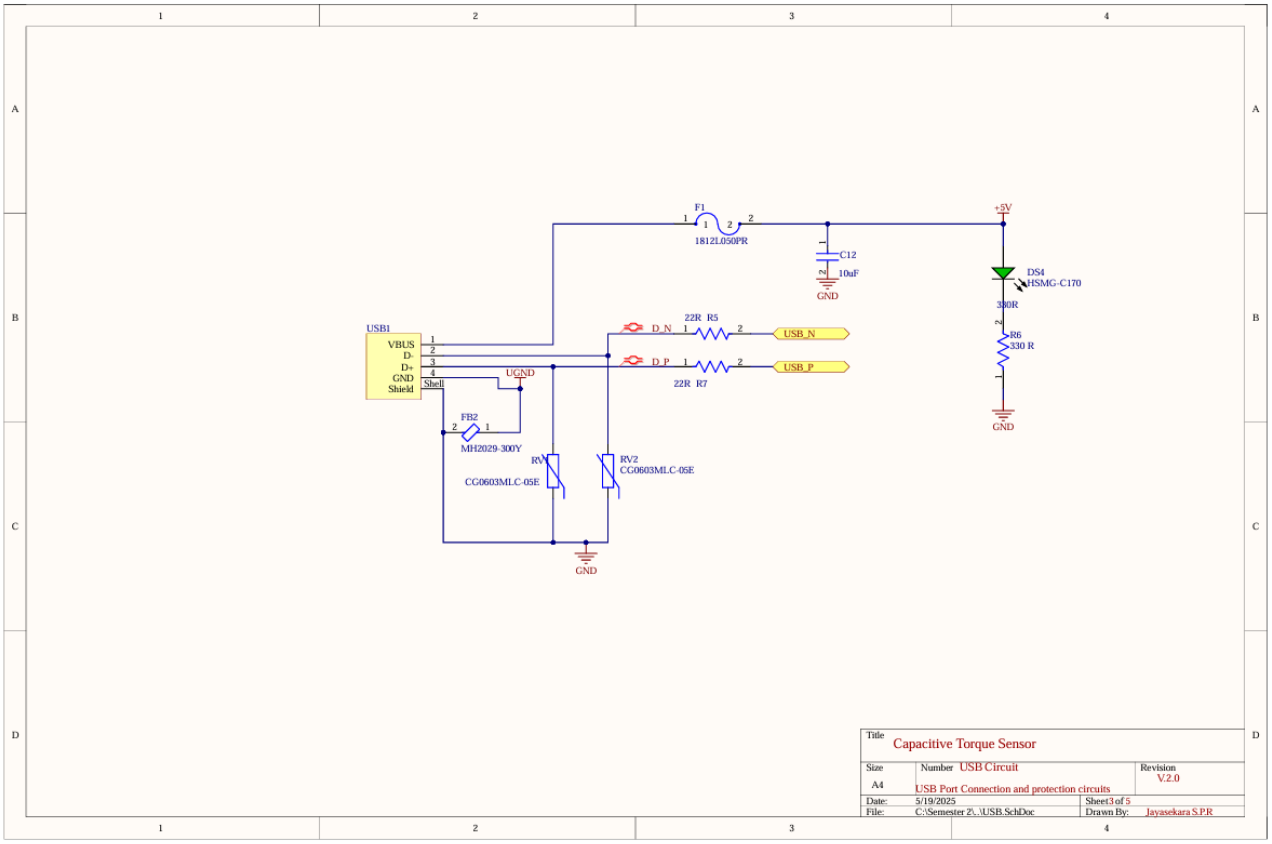


Figure 27: USB Schematic

14.3 Power Circuit

- Generates regulated voltages:
  - **+3.3V** and **+1.8V** for PCAP04 CDC
- **Design Features**:
  - Enable pins tied to V_{CC} for continuous operation
  - Input/output capacitors for:
    - High-frequency noise filtering
    - Ripple reduction

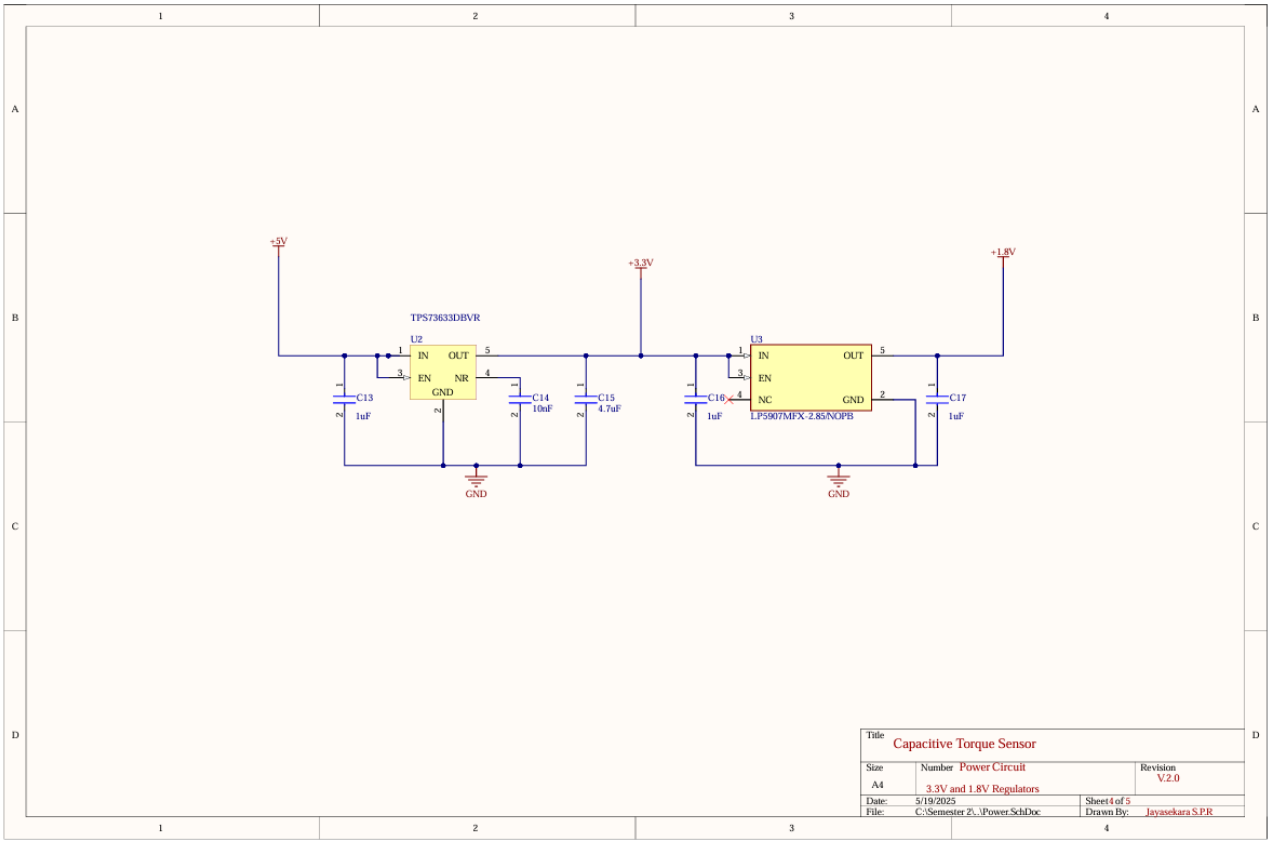


Figure 28: Power Supply Schematic

14.4 Sensor (CDC) Circuit

- **Capacitance-to-Digital Converter (PCAP04)**:
  - Powered by +3.3V (digital) and +1.8V (analog)
  - PC1 pin connected to sensor plates via 2-wire JST connector
  - Operates in **differential capacitance mode**
  - SPI communication with MCU
- **MOSFET Level Shifters**:
  - Converts MCU's +5V signals to +3.3V for CDC
  - Replaced TXB0104PWR IC due to low drive current
  - Used for SPI lines (SCK, MOSI, MISO, CS)

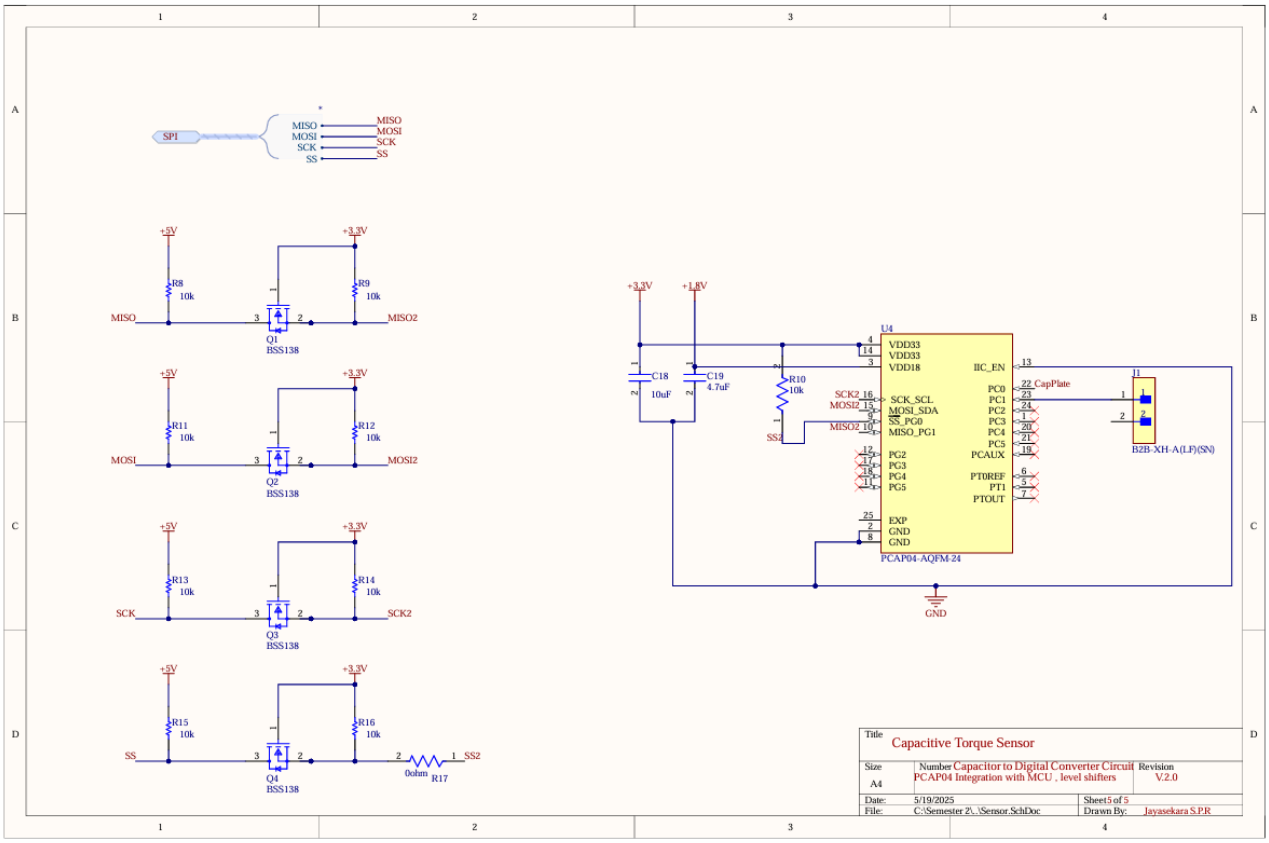


Figure 29: CDC Sensor Schematic

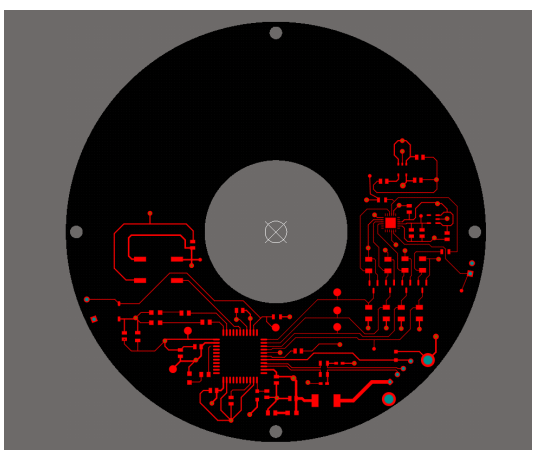
15 PCB Design

The two PCBs were designed using Altium Designer. The PCB serves two purposes in our product:

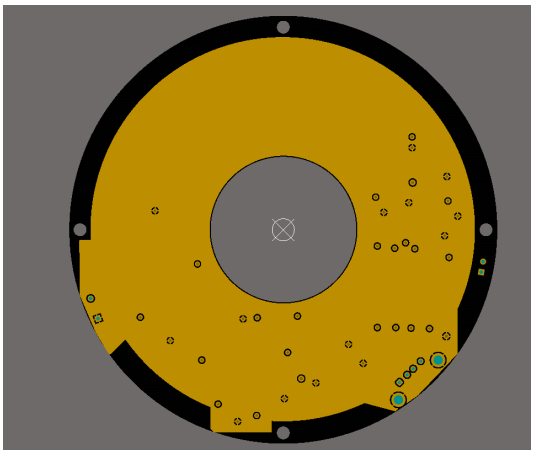
- Holds and routes components
- Acts as a capacitor plate for measuring capacitance

15.1 PCB Specifications

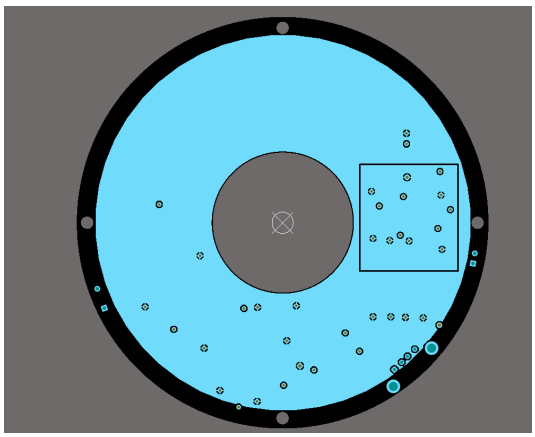
- Circular shape with a hole in middle for the shaft and disk holders
- Outer diameter: 10 cm and Inner diameter: 3.4 cm
- 4-layer stack-up:
  1. Signal Layer for component placement and routing
  2. Solid ground layer
  3. Power layer (+5V and +3.3V)
  4. Solid copper pour which acts as a capacitor plate



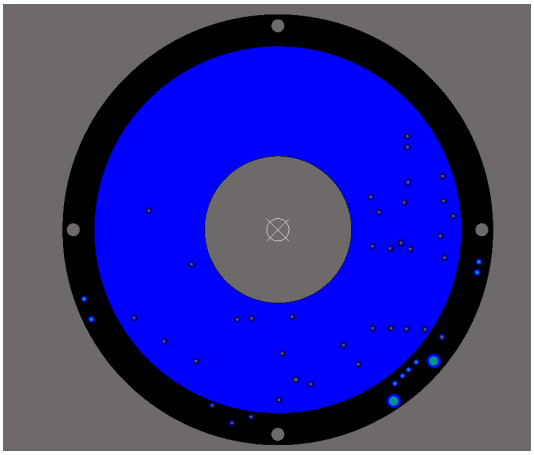
(a) Top Signal Layer



(b) Ground Layer



(c) Power Layer



(d) Capacitive Plate layer

Figure 30: PCB Layers view

15.2 Noise Immunity Features

The PCB incorporates ground and power planes to ensure noise immunity:

- Continuous ground plane acts as a shield, minimizing radiated emissions and reducing susceptibility to external noise
- Provides low impedance return path for signals (crucial for high-speed signals like USB)
- Power layer (+5V and +3.3V) creates natural parallel-plate capacitor with ground layer, providing distributed decoupling capacitance
- Ground plane (layer 2) provides isolation between routing layer and capacitive sensing plate, preventing noise coupling

15.3 Routing Considerations

- Components tightly arranged to avoid long noise-prone traces
- Through-hole components placed near board edges to minimize effect on capacitance plate
- USB traces routed as differential pair (0.3mm width, 0.254mm gap) for 90 impedance matching

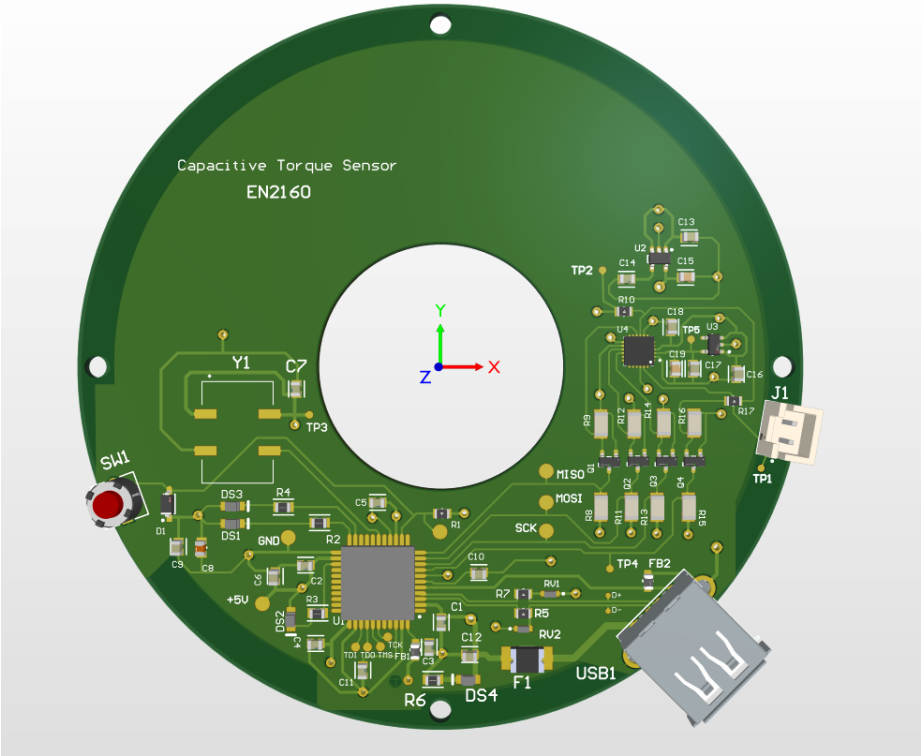


Figure 31: Front View

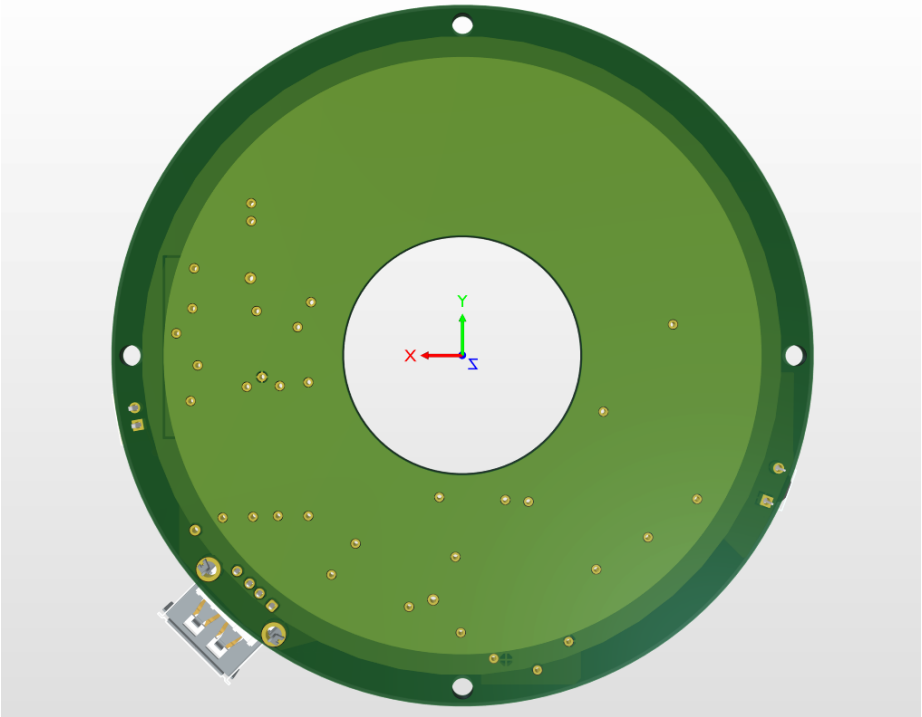


Figure 32: Backside View

16 Testing Procedure

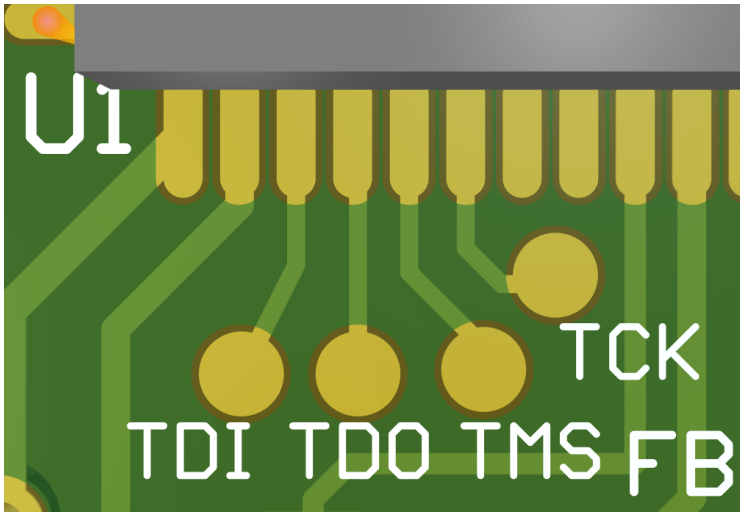


Figure 33: Test Points Implementation on PCB

After learning about proper test point implementation in our coursework, we enhanced our PCB design to include comprehensive testing capabilities. The evolution of our test point implementation occurred in two phases:

16.1 Initial Testing Implementation

The first PCB revision included basic test points primarily for power verification:

- Voltage test pads for regulator output validation
- Bootloader programming headers that doubled as general test points

16.2 Enhanced Testing Capabilities

Through further study of PCB testing methodologies, we added advanced test features:

- USB differential pair test points for signal integrity verification
- Full JTAG interface for the Atmega32U4 microcontroller

The JTAG test points (TDI, TDO, TMS, TCK) enable several critical testing procedures:

- **Interconnect Testing**: Verifying proper connections between components
- **Boundary Scan**: Testing PCB traces and solder joints
- **Programming/Debugging**: Allowing in-circuit firmware updates and debugging

This implementation allows us to perform comprehensive board validation, from basic power-on tests to advanced boundary scan testing, significantly improving our testing capabilities compared to the initial design.

17 Bill of Materials (BOM)

Comment	Designator	Quantity	Unit Price (USD)	Total Price(USD)
C0805C104J8RACTU	, C2, C3, C4, C5, C7, C	7	0.127	0.889
C0805C106K8PACTU	C10, C12, C13, C16, C	8	0.556	4.448
CC0805KRX7R7BB104	C8	1	0.167	0.167
10nF	C14	1	0.165	0.165
4.7uF	C15, C19	2	0.55	1.1
1N4148W-TP	D1	1	0.03	0.03
HSMG-C170	DS1, DS2, DS3, DS4	4	0.04	0.16
1812L050PR	F1	1	0.4	0.4
MH2029-300Y	FB1, FB2	2	0.1	0.2
B2B-XH-A(LF)(SN)	J1	1	0.3	0.3
BSS138	Q1, Q2, Q3, Q4	4	0.03	0.12
CRCW080522R0FKEA	R1, R5, R7, R10, R17	5	0.104	0.52
330 R	R2, R3, R4, R6	4	0.14	0.56
10k	, R11, R12, R13, R14, R1	8	0.027	0.216
CG0603MLC-05E	RV1, RV2	2	0.15	0.3
EVQ-11L05R	SW1	1	0.07	0.07
ATmega32U4RC-AU	U1	1	5.3	5.3
TPS73633DBVR	U2	1	2.15	2.15
LP5907MFX-2.85/NOPB	U3	1	0.49	0.49
PCAP04-AQFM-24	U4	1	7.66	7.66
61400416021	USB1	1	2.02	2.02
ECS-8FMX-160-TR	Y1	1	4.15	4.15
			Total	31.415

Figure 34: Complete Bill of Materials for the Capacitive Torque Sensor

The complete Bill of Materials shown in Figure 34 contains all components required for the assembly of one sensor unit, categorized by:

- **Electronics Components**:
  - Active components (MCU, regulators, ICs)
  - Passive components (resistors, capacitors)
  - Connectors and interfaces
- **Mechanical Parts**:
  - Precision-machined components (shaft, disks)
  - Enclosure and structural elements
  - Fasteners and mounting hardware
- **PCB Details**:
  - Board specifications (4-layer, 1.6mm thickness)
  - Special materials (FR4, copper weight)

Key Features of the BOM:

- Each component includes manufacturer part numbers for precise sourcing
- Quantities are specified for single-unit production
- Critical components are highlighted with tolerance requirements
- Alternate part numbers are provided for key components

Note: The BOM represents the final production version after all design iterations. Component availability and prices may vary based on market conditions.

18 RTL Code for PCB (ATmega32U4)

18.1 Software Architecture

The firmware for the capacitive torque sensor consists of three main components:

- Capacitance measurement by CDC and send it to MCU via SPI
- MCU Calaculation convert it into respective Torque value
- MCU send the Torque values via USB serial Commun

18.2 main.c

Listing 1: Main C Code

```
#ifndef F_CPU
#define F_CPU 16000000UL // 16 MHz
#endif

#include <avr/io.h>
#include <util/delay.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include "Src/m_usb.h"
#include <math.h>

#define SS_PIN    PB0
#define MOSI_PIN  PB2
#define MISO_PIN  PB3
#define SCK_PIN   PB1

// Define polynomial coefficients for torque calculation
#define A3 0.0002
#define A2 -0.015
#define A1 3.2
#define A0 -150

// ========== SPI ==========
void SPI_init(void) {
    DDRB |= (1 << MOSI_PIN) | (1 << SCK_PIN) | (1 << SS_PIN); // Set MOSI,
        SCK, SS as output
    DDRB &= ~(1 << MISO_PIN); // Set MISO as input
    SPCR = (1 << SPE) | (1 << MSTR) | (1 << SPR0); // Enable SPI, Master,
       Fosc/16(SCK frequency)
    SPSR = 0;
}

uint8_t SPI_transfer(uint8_t data) {
    SPDR = data;
    while (!(SPSR & (1 << SPIF)));
    return SPDR;
}

```

```
38 // ========== PCAP04 SPI Interface ==========
39 void PCAP04_select(void) {
40     PORTB &= ~(1 << PB0); // CS LOW
41 }
42
43 void PCAP04_deselect(void) {
44     PORTB |= (1 << PB0); // CS HIGH
45 }
46
47 void PCAP04_write(uint8_t addr, uint8_t data) {
48     PCAP04_select();
49     SPI_transfer(addr & 0x7F); // Write = MSB 0
50     SPI_transfer(data);
51     PCAP04_deselect();
52 }
53
54 uint8_t PCAP04_read(uint8_t addr) {
55     PCAP04_select();
56     SPI_transfer(addr | 0x80); // Read = MSB 1
57     uint8_t result = SPI_transfer(0x00);
58     PCAP04_deselect();
59     return result;
60 }
61
62 // ========== PCAP04 Config ==========
63 void PCAP04_init(void) {
64     pcap_write_reg(0x00, 0x01); // Set Oscillator Frequency to 50 kHz
65     pcap_write_reg(0x04, 0b10010000); // CFG4: grounded sensor  +
        C_REF_INT
66     pcap_write_reg(0x06, 0x01); // CFG6: Enable PC0
67     pcap_write_reg(0x07, 0x08); // CFG7: averaging = 8
68     pcap_write_reg(0x08, 0x00); // CFG8: high byte of averaging
69     pcap_write_reg(0x09, 0x19); //  C F G 9 11 : CONV_TIME for 1 kHz
70     pcap_write_reg(0x0A, 0x00);
71     pcap_write_reg(0x0B, 0x00);
72     uint8_t reg17 = pcap_read_reg(0x11); // Read existing value
73     reg17 &= 0x03;                        // Clear bits 6:2 (keep bits 1:0)
74     reg17 |= (7 << 2);                   // Set C_REF_SEL = 7 (approx. 10
        pF internal reference)
75     pcap_write_reg(0x11, reg17);         // Write back updated value
76     pcap_write_reg(0x2F, 0x01); // CFG47: RUNBIT = 1
77 }
78
79
80 uint32_t PCAP04_read_result(void) {
81     uint32_t val = 0;
82     val |= ((uint32_t)PCAP04_read(0x03) << 24);
83     val |= ((uint32_t)PCAP04_read(0x02) << 16);
84     val |= ((uint32_t)PCAP04_read(0x01) << 8);
85     val |= ((uint32_t)PCAP04_read(0x00));
86     return val;
87 }
88
89 float PCAP04_to_pF(uint32_t raw, float Cref) {
```

```
    float ratio = (float)raw / (1UL << 27); // 5.27 fixed-point
    return ratio * Cref;
}

float calculateTorque(float x) {
    return A3 * pow(x, 3) + A2 * pow(x, 2) + A1 * x + A0;
}

int main(void)
{
    SPI_init();
    m_usb_init();
    _delay_ms(10);
    PORTB |= (1 << PB0); // CS HIGH idle

    PCAP04_init();
    _delay_ms(100);

    while (!m_usb_isconnected()) { }

    while (1) {
        uint32_t raw = PCAP04_read_result();
        float cap_pf = PCAP04_to_pF(raw, 10.0f); // Assuming Cref = 10.0pF
        float torque = calculateTorque(cap_pf);
        m_usb_tx_long(torque);
        m_usb_tx_push();       // Ensure transmission
        _delay_ms(1);// 1 kHz loop        // Prevent buffer overflow
    }
}
```

18.3 USB.h

Listing 2: USB header file

```
//
   ------------------------------------------------------------------------



#ifndef m_usb__
#define m_usb__

#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/pgmspace.h>
#include <stdlib.h>


//
   ------------------------------------------------------------------------------

// Public functions:
//
   ------------------------------------------------------------------------------
```

```
15
16 // INITIALIZATION:
       ------------------------------------------------------------
17
18 void m_usb_init(void);
19 // initialize the USB subsystem
20
21 char m_usb_isconnected(void);
22 // confirm that the USB port is connected to a PC
23
24
25 // RECEIVE:
       ---------------------------------------------------------------------
26
27 unsigned char m_usb_rx_available(void);
28 // returns the number of bytes (up to 255) waiting in the receive FIFO
       buffer
29
30 char m_usb_rx_char(void);
31 // retrieve a oldest byte from the receive FIFO buffer (-1 if timeout/
       error)
32
33 void m_usb_rx_flush(void);
34 // discard all data in the receive buffer
35
36
37
38 // TRANSMIT:
       ---------------------------------------------------------------------
39
40 char m_usb_tx_char(unsigned char c);
41 // add a single 8-bit unsigned char to the transmit buffer, return -1 if
       error
42
43 void m_usb_tx_hexchar(unsigned char i);
44 // add an unsigned char to the transmit buffer, send as two hex-value
       characters
45
46 void m_usb_tx_hex(unsigned int i);
47 // add an unsigned int to the transmit buffer, send as four hex-value
       characters
48
49 void m_usb_tx_int(int i);
50 // add a signed int to the transmit buffer, send as a sign character then
       5 decimal-value characters
51
52 void m_usb_tx_uint(unsigned int i);
53 // add an unsigned int to the transmit buffer, send as 5 decimal-value
       characters
54
55 void m_usb_tx_long(long i);
56 // add a signed long to the transmit buffer, send as a sign character then
        5 decimal-value characters
57
```

```
58 void m_usb_tx_ulong(unsigned long i);
59 // add an unsigned long to the transmit buffer, send as 5 decimal-value
      characters
60
61 #define m_usb_tx_string(s) print_P(PSTR(s))
62 // add a string to the transmit buffer
63
64
65
66 //
      ------------------------------------------------------------------------
67 //
      ------------------------------------------------------------------------
68 //
      ------------------------------------------------------------------------
69
70 // ---- OVERLOADS FOR M1 BACK COMPATIBILITY ----
71 #define usb_init()          m_usb_init()
72 #define usb_configured()    m_usb_isconnected()
73
74 #define usb_rx_available()  m_usb_rx_available()
75 #define usb_rx_flush()      m_usb_rx_flush()
76 #define usb_rx_char()       m_usb_rx_char()
77
78 #define usb_tx_char(val)    m_usb_tx_char(val)
79 #define usb_tx_hex(val)     m_usb_tx_hex(val)
80 #define usb_tx_decimal(val) m_usb_tx_uint(val)
81 #define usb_tx_string(val)  m_usb_tx_string(val)
82 #define usb_tx_push()       m_usb_tx_push()
83
84 #define m_usb_rx_ascii()    m_usb_rx_char()
85 #define m_usb_tx_ascii(val) m_usb_tx_char(val)
86
87
88 // EVERYTHING ELSE
      ************************************************************
89
90 // setup
91
92 int8_t usb_serial_putchar(uint8_t c);   // transmit a character
93 int8_t usb_serial_putchar_nowait(uint8_t c);  // transmit a character, do
      not wait
94 int8_t usb_serial_write(const uint8_t *buffer, uint16_t size); // transmit
       a buffer
95 void print_P(const char *s);
96 void phex(unsigned char c);
97 void phex16(unsigned int i);
98 void m_usb_tx_hex8(unsigned char i);
99 void m_usb_tx_push(void);
100
101
```

```
102 // serial parameters
103 uint32_t usb_serial_get_baud(void); // get the baud rate
104 uint8_t usb_serial_get_stopbits(void);  // get the number of stop bits
105 uint8_t usb_serial_get_paritytype(void);// get the parity type
106 uint8_t usb_serial_get_numbits(void);   // get the number of data bits
107 uint8_t usb_serial_get_control(void);   // get the RTS and DTR signal
       state
108 int8_t usb_serial_set_control(uint8_t signals); // set DSR, DCD, RI, etc
109
110 // constants corresponding to the various serial parameters
111 #define USB_SERIAL_DTR          0x01
112 #define USB_SERIAL_RTS          0x02
113 #define USB_SERIAL_1_STOP       0
114 #define USB_SERIAL_1_5_STOP     1
115 #define USB_SERIAL_2_STOP       2
116 #define USB_SERIAL_PARITY_NONE      0
117 #define USB_SERIAL_PARITY_ODD       1
118 #define USB_SERIAL_PARITY_EVEN      2
119 #define USB_SERIAL_PARITY_MARK      3
120 #define USB_SERIAL_PARITY_SPACE     4
121 #define USB_SERIAL_DCD          0x01
122 #define USB_SERIAL_DSR          0x02
123 #define USB_SERIAL_BREAK        0x04
124 #define USB_SERIAL_RI           0x08
125 #define USB_SERIAL_FRAME_ERR        0x10
126 #define USB_SERIAL_PARITY_ERR       0x20
127 #define USB_SERIAL_OVERRUN_ERR      0x40
128
129 // This file does not include the HID debug functions, so these empty
130 // macros replace them with nothing, so users can compile code that
131 // has calls to these functions.
132 #define usb_debug_putchar(c)
133 #define usb_debug_flush_output()
134
135 #define EP_TYPE_CONTROL         0x00
136 #define EP_TYPE_BULK_IN         0x81
137 #define EP_TYPE_BULK_OUT        0x80
138 #define EP_TYPE_INTERRUPT_IN        0xC1
139 #define EP_TYPE_INTERRUPT_OUT       0xC0
140 #define EP_TYPE_ISOCHRONOUS_IN      0x41
141 #define EP_TYPE_ISOCHRONOUS_OUT     0x40
142 #define EP_SINGLE_BUFFER        0x02
143 #define EP_DOUBLE_BUFFER        0x06
144 #define EP_SIZE(s)  ((s) == 64 ? 0x30 : \
145 ((s) == 32 ? 0x20 : \
146 ((s) == 16 ? 0x10 : \
147 0x00)))
148
149 #define MAX_ENDPOINT        4
150
151 #define LSB(n) (n & 255)
152 #define MSB(n) ((n >> 8) & 255)
153
154 #define HW_CONFIG() (UHWCON = 0x01)
```

```
155
156 #ifdef M1
157 #define PLL_CONFIG() (PLLCSR = 0x02) // fixed to 8MHz clock
158 #else
159 #define PLL_CONFIG() (PLLCSR = 0x12) // 0001 0010 For a 16MHz clock
160 #endif
161
162 #define USB_CONFIG() (USBCON = ((1<<USBE)|(1<<OTGPADE)))
163 #define USB_FREEZE() (USBCON = ((1<<USBE)|(1<<FRZCLK)))
164
165 // standard control endpoint request types
166 #define GET_STATUS          0
167 #define CLEAR_FEATURE           1
168 #define SET_FEATURE         3
169 #define SET_ADDRESS         5
170 #define GET_DESCRIPTOR          6
171 #define GET_CONFIGURATION       8
172 #define SET_CONFIGURATION       9
173 #define GET_INTERFACE           10
174 #define SET_INTERFACE           11
175 // HID (human interface device)
176 #define HID_GET_REPORT          1
177 #define HID_GET_PROTOCOL        3
178 #define HID_SET_REPORT          9
179 #define HID_SET_IDLE            10
180 #define HID_SET_PROTOCOL        11
181 // CDC (communication class device)
182 #define CDC_SET_LINE_CODING     0x20
183 #define CDC_GET_LINE_CODING     0x21
184 #define CDC_SET_CONTROL_LINE_STATE  0x22
185
186 #endif
```

18.4 USB.c

Listing 3: USB Main C file

```
1 //
     ------------------------------------------------------------------------

2 #define USB_SERIAL_PRIVATE_INCLUDE
3 #include "m_usb.h"
4 // ---- OVERLOADS FOR M1 BACK COMPATIBILITY ----
5 #define usb_init()          m_usb_init()
6 #define usb_configured()    m_usb_isconnected()
7
8 #define usb_rx_available()  m_usb_rx_available()
9 #define usb_rx_flush()      m_usb_rx_flush()
10 #define usb_rx_char()       m_usb_rx_char()
11
12 #define usb_tx_char(val)    m_usb_tx_char(val)
13 #define usb_tx_hex(val)     m_usb_tx_hex(val)
14 #define usb_tx_decimal(val) m_usb_tx_uint(val)
15 #define usb_tx_string(val)  m_usb_tx_string(val)
```

```
16 #define usb_tx_push()       m_usb_tx_push()
17
18 #define m_usb_rx_ascii()    m_usb_rx_char()
19 #define m_usb_tx_ascii(val) m_usb_tx_char(val)
20
21
22 // EVERYTHING ELSE
   ************************************************************
23
24 // setup
25
26 int8_t usb_serial_putchar(uint8_t c);   // transmit a character
27 int8_t usb_serial_putchar_nowait(uint8_t c);  // transmit a character, do
   not wait
28 int8_t usb_serial_write(const uint8_t *buffer, uint16_t size); // transmit
    a buffer
29 void print_P(const char *s);
30 void phex(unsigned char c);
31 void phex16(unsigned int i);
32 void m_usb_tx_hex8(unsigned char i);
33 void m_usb_tx_push(void);
34
35
36 // serial parameters
37 uint32_t usb_serial_get_baud(void); // get the baud rate
38 uint8_t usb_serial_get_stopbits(void);  // get the number of stop bits
39 uint8_t usb_serial_get_paritytype(void);// get the parity type
40 uint8_t usb_serial_get_numbits(void);   // get the number of data bits
41 uint8_t usb_serial_get_control(void);   // get the RTS and DTR signal
   state
42 int8_t usb_serial_set_control(uint8_t signals); // set DSR, DCD, RI, etc
43
44 // constants corresponding to the various serial parameters
45 #define USB_SERIAL_DTR          0x01
46 #define USB_SERIAL_RTS          0x02
47 #define USB_SERIAL_1_STOP       0
48 #define USB_SERIAL_1_5_STOP     1
49 #define USB_SERIAL_2_STOP       2
50 #define USB_SERIAL_PARITY_NONE      0
51 #define USB_SERIAL_PARITY_ODD       1
52 #define USB_SERIAL_PARITY_EVEN      2
53 #define USB_SERIAL_PARITY_MARK      3
54 #define USB_SERIAL_PARITY_SPACE     4
55 #define USB_SERIAL_DCD          0x01
56 #define USB_SERIAL_DSR          0x02
57 #define USB_SERIAL_BREAK        0x04
58 #define USB_SERIAL_RI           0x08
59 #define USB_SERIAL_FRAME_ERR        0x10
60 #define USB_SERIAL_PARITY_ERR       0x20
61 #define USB_SERIAL_OVERRUN_ERR      0x40
62
63 // This file does not include the HID debug functions, so these empty
64 // macros replace them with nothing, so users can compile code that
65 // has calls to these functions.
```

```
66 #define usb_debug_putchar(c)
67 #define usb_debug_flush_output()
68
69 #define EP_TYPE_CONTROL         0x00
70 #define EP_TYPE_BULK_IN         0x81
71 #define EP_TYPE_BULK_OUT        0x80
72 #define EP_TYPE_INTERRUPT_IN        0xC1
73 #define EP_TYPE_INTERRUPT_OUT       0xC0
74 #define EP_TYPE_ISOCHRONOUS_IN      0x41
75 #define EP_TYPE_ISOCHRONOUS_OUT     0x40
76 #define EP_SINGLE_BUFFER        0x02
77 #define EP_DOUBLE_BUFFER        0x06
78 #define EP_SIZE(s)  ((s) == 64 ? 0x30 : \
79 ((s) == 32 ? 0x20 : \
80 ((s) == 16 ? 0x10 : \
81 0x00)))
82
83 #define MAX_ENDPOINT        4
84
85 #define LSB(n) (n & 255)
86 #define MSB(n) ((n >> 8) & 255)
87
88 #define HW_CONFIG() (UHWCON = 0x01)
89
90 #ifdef M1
91 #define PLL_CONFIG() (PLLCSR = 0x02) // fixed to 8MHz clock
92 #else
93 #define PLL_CONFIG() (PLLCSR = 0x12) // 0001 0010 For a 16MHz clock
94 #endif
95
96 #define USB_CONFIG() (USBCON = ((1<<USBE)|(1<<OTGPADE)))
97 #define USB_FREEZE() (USBCON = ((1<<USBE)|(1<<FRZCLK)))
98
99 // standard control endpoint request types
100 #define GET_STATUS          0
101 #define CLEAR_FEATURE           1
102 #define SET_FEATURE         3
103 #define SET_ADDRESS         5
104 #define GET_DESCRIPTOR          6
105 #define GET_CONFIGURATION       8
106 #define SET_CONFIGURATION       9
107 #define GET_INTERFACE           10
108 #define SET_INTERFACE           11
109 // HID (human interface device)
110 #define HID_GET_REPORT          1
111 #define HID_GET_PROTOCOL        3
112 #define HID_SET_REPORT          9
113 #define HID_SET_IDLE            10
114 #define HID_SET_PROTOCOL        11
115 // CDC (communication class device)
116 #define CDC_SET_LINE_CODING     0x20
117 #define CDC_GET_LINE_CODING     0x21
118 #define CDC_SET_CONTROL_LINE_STATE  0x22
119
```

```
120
121 /*
    ************************************************************************
122  *
123  *  Configurable Options
124  *
125  ************************************************************************
     */
126
127 #define STR_MANUFACTURER    L"J. Fiene"
128 #define STR_PRODUCT     L"M2"
129 #define STR_SERIAL_NUMBER   L"410"
130 #define VENDOR_ID       0x16C0 // must match INF file in Windows
131 #define PRODUCT_ID      0x047A // must match INF file in Windows
132 #define TRANSMIT_FLUSH_TIMEOUT  5   /* in milliseconds */
133 #define TRANSMIT_TIMEOUT    25   /* in milliseconds */
134 #define SUPPORT_ENDPOINT_HALT // can save 116 bytes by removing, makes
    fully USB compliant
135
136 /*
    ************************************************************************
137  *
138  *  Endpoint Buffer Configuration
139  *
140  ************************************************************************
     */
141
142 #define ENDPOINT0_SIZE      16
143 #define CDC_ACM_ENDPOINT    2
144 #define CDC_RX_ENDPOINT     3
145 #define CDC_TX_ENDPOINT     4
146 #define CDC_ACM_SIZE        16
147 #define CDC_ACM_BUFFER      EP_SINGLE_BUFFER
148 #define CDC_RX_SIZE     64
149 #define CDC_RX_BUFFER       EP_DOUBLE_BUFFER
150 #define CDC_TX_SIZE     64
151 #define CDC_TX_BUFFER       EP_DOUBLE_BUFFER
152
153 static const uint8_t PROGMEM endpoint_config_table[] = {
154     0,
155     1, EP_TYPE_INTERRUPT_IN,  EP_SIZE(CDC_ACM_SIZE) | CDC_ACM_BUFFER,
156     1, EP_TYPE_BULK_OUT,      EP_SIZE(CDC_RX_SIZE) | CDC_RX_BUFFER,
157     1, EP_TYPE_BULK_IN,       EP_SIZE(CDC_TX_SIZE) | CDC_TX_BUFFER
158 };
159
160
161 /*
    ************************************************************************
162  *
163  *  Descriptor Data
164  *
```

```
165  ****************************************************************************
     */
166
167  static const uint8_t PROGMEM device_descriptor[] = {
168      18,                 // bLength
169      1,                  // bDescriptorType
170      0x00, 0x02,             // bcdUSB
171      2,                  // bDeviceClass
172      0,                  // bDeviceSubClass
173      0,                  // bDeviceProtocol
174      ENDPOINT0_SIZE,             // bMaxPacketSize0
175      LSB(VENDOR_ID), MSB(VENDOR_ID),     // idVendor
176      LSB(PRODUCT_ID), MSB(PRODUCT_ID),   // idProduct
177      0x00, 0x01,             // bcdDevice
178      1,                  // iManufacturer
179      2,                  // iProduct
180      3,                  // iSerialNumber
181      1                   // bNumConfigurations
182  };
183
184  #define CONFIG1_DESC_SIZE (9+9+5+5+4+5+7+9+7+7)
185  static const uint8_t PROGMEM config1_descriptor[CONFIG1_DESC_SIZE] = {
186      // configuration descriptor, USB spec 9.6.3, page 264-266, Table 9-10
187      9,                  // bLength;
188      2,                  // bDescriptorType;
189      LSB(CONFIG1_DESC_SIZE),         // wTotalLength
190      MSB(CONFIG1_DESC_SIZE),
191      2,                  // bNumInterfaces
192      1,                  // bConfigurationValue
193      0,                  // iConfiguration
194      0xC0,                   // bmAttributes
195      50,                 // bMaxPower
196      // interface descriptor, USB spec 9.6.5, page 267-269, Table 9-12
197      9,                  // bLength
198      4,                  // bDescriptorType
199      0,                  // bInterfaceNumber
200      0,                  // bAlternateSetting
201      1,                  // bNumEndpoints
202      0x02,                   // bInterfaceClass
203      0x02,                   // bInterfaceSubClass
204      0x01,                   // bInterfaceProtocol
205      0,                  // iInterface
206      // CDC Header Functional Descriptor, CDC Spec 5.2.3.1, Table 26
207      5,                  // bFunctionLength
208      0x24,                   // bDescriptorType
209      0x00,                   // bDescriptorSubtype
210      0x10, 0x01,             // bcdCDC
211      // Call Management Functional Descriptor, CDC Spec 5.2.3.2, Table 27
212      5,                  // bFunctionLength
213      0x24,                   // bDescriptorType
214      0x01,                   // bDescriptorSubtype
215      0x01,                   // bmCapabilities
216      1,                  // bDataInterface
```

```
    // Abstract Control Management Functional Descriptor, CDC Spec
       5.2.3.3, Table 28
    4,                  // bFunctionLength
    0x24,                   // bDescriptorType
    0x02,                   // bDescriptorSubtype
    0x06,                   // bmCapabilities
    // Union Functional Descriptor, CDC Spec 5.2.3.8, Table 33
    5,                  // bFunctionLength
    0x24,                   // bDescriptorType
    0x06,                   // bDescriptorSubtype
    0,                  // bMasterInterface
    1,                  // bSlaveInterface0
    // endpoint descriptor, USB spec 9.6.6, page 269-271, Table 9-13
    7,                  // bLength
    5,                  // bDescriptorType
    CDC_ACM_ENDPOINT | 0x80,        // bEndpointAddress
    0x03,                   // bmAttributes (0x03=intr)
    CDC_ACM_SIZE, 0,            // wMaxPacketSize
    64,                 // bInterval
    // interface descriptor, USB spec 9.6.5, page 267-269, Table 9-12
    9,                  // bLength
    4,                  // bDescriptorType
    1,                  // bInterfaceNumber
    0,                  // bAlternateSetting
    2,                  // bNumEndpoints
    0x0A,                   // bInterfaceClass
    0x00,                   // bInterfaceSubClass
    0x00,                   // bInterfaceProtocol
    0,                  // iInterface
    // endpoint descriptor, USB spec 9.6.6, page 269-271, Table 9-13
    7,                  // bLength
    5,                  // bDescriptorType
    CDC_RX_ENDPOINT,            // bEndpointAddress
    0x02,                   // bmAttributes (0x02=bulk)
    CDC_RX_SIZE, 0,             // wMaxPacketSize
    0,                  // bInterval
    // endpoint descriptor, USB spec 9.6.6, page 269-271, Table 9-13
    7,                  // bLength
    5,                  // bDescriptorType
    CDC_TX_ENDPOINT | 0x80,         // bEndpointAddress
    0x02,                   // bmAttributes (0x02=bulk)
    CDC_TX_SIZE, 0,             // wMaxPacketSize
    0                   // bInterval
};

// If you're desperate for a little extra code memory, these strings
// can be completely removed if iManufacturer, iProduct, iSerialNumber
// in the device desciptor are changed to zeros.
struct usb_string_descriptor_struct {
    uint8_t bLength;
    uint8_t bDescriptorType;
    int16_t wString[];
};
static const struct usb_string_descriptor_struct PROGMEM string0 = {
```

```
    4,
    3,
    {0x0409}
};
static const struct usb_string_descriptor_struct PROGMEM string1 = {
    sizeof(STR_MANUFACTURER),
    3,
    STR_MANUFACTURER
};
static const struct usb_string_descriptor_struct PROGMEM string2 = {
    sizeof(STR_PRODUCT),
    3,
    STR_PRODUCT
};
static const struct usb_string_descriptor_struct PROGMEM string3 = {
    sizeof(STR_SERIAL_NUMBER),
    3,
    STR_SERIAL_NUMBER
};

// This table defines which descriptor data is sent for each specific
// request from the host (in wValue and wIndex).
static const struct descriptor_list_struct {
    uint16_t    wValue;
    uint16_t    wIndex;
    const uint8_t   *addr;
    uint8_t     length;
} PROGMEM descriptor_list[] = {
    {0x0100, 0x0000, device_descriptor, sizeof(device_descriptor)},
    {0x0200, 0x0000, config1_descriptor, sizeof(config1_descriptor)},
    {0x0300, 0x0000, (const uint8_t *)&string0, 4},
    {0x0301, 0x0409, (const uint8_t *)&string1, sizeof(STR_MANUFACTURER)},
    {0x0302, 0x0409, (const uint8_t *)&string2, sizeof(STR_PRODUCT)},
    {0x0303, 0x0409, (const uint8_t *)&string3, sizeof(STR_SERIAL_NUMBER)}
};
#define NUM_DESC_LIST (sizeof(descriptor_list)/sizeof(struct
   descriptor_list_struct))


/*
   **********************************************************************

 *
 *  Variables - these are the only non-stack RAM usage
 *
 **********************************************************************
   */

// zero when we are not configured, non-zero when enumerated
static volatile uint8_t usb_configuration=0;

// the time remaining before we transmit any partially full
// packet, or send a zero length packet.
static volatile uint8_t transmit_flush_timer=0;
```

```
320 static uint8_t transmit_previous_timeout=0;
321
322 // serial port settings (baud rate, control signals, etc) set
323 // by the PC.  These are ignored, but kept in RAM.
324 static uint8_t cdc_line_coding[7]={0x00, 0xE1, 0x00, 0x00, 0x00, 0x00, 0
       x08};
325 static uint8_t cdc_line_rtsdtr=0;
326
327
328 /*
       ************************************************************************

329  *
330  *  Public Functions - these are the API intended for the user
331  *
332  ************************************************************************
       */
333
334 // initialize USB serial
335 void m_usb_init(void)
336 {
337     HW_CONFIG();
338         USB_FREEZE();                // enable USB
339         PLL_CONFIG();                // config PLL, 16 MHz xtal
340         while (!(PLLCSR & (1<<PLOCK))) ;    // wait for PLL lock
341         USB_CONFIG();                // start USB clock
342         UDCON = 0;               // enable attach resistor
343     usb_configuration = 0;
344     cdc_line_rtsdtr = 0;
345         UDIEN = (1<<EORSTE)|(1<<SOFE);
346     sei();
347 }
348
349 // return 0 if the USB is not configured, or the configuration
350 // number selected by the HOST
351 char m_usb_isconnected(void)
352 {
353     return (char)usb_configuration;
354 }
355
356 // get the next character, or -1 if nothing received
357 char m_usb_rx_char(void)
358 {
359     uint8_t c, intr_state;
360
361     // interrupts are disabled so these functions can be
362     // used from the main program or interrupt context,
363     // even both in the same program!
364     intr_state = SREG;
365     cli();
366     if (!usb_configuration) {
367         SREG = intr_state;
368         return -1;
369     }
```

```
370     UENUM = CDC_RX_ENDPOINT;
371     if (!(UEINTX & (1<<RWAL))) {
372         // no data in buffer
373         SREG = intr_state;
374         return -1;
375     }
376     // take one byte out of the buffer
377     c = UEDATX;
378     // if buffer completely used, release it
379     if (!(UEINTX & (1<<RWAL))) UEINTX = 0x6B;
380     SREG = intr_state;
381     return (char)c;
382 }
383 
384 // number of bytes available in the receive buffer
385 unsigned char m_usb_rx_available(void)
386 {
387     uint8_t n=0, intr_state;
388 
389     intr_state = SREG;
390     cli();
391     if (usb_configuration) {
392         UENUM = CDC_RX_ENDPOINT;
393         n = UEBCLX;
394     }
395     SREG = intr_state;
396     return (unsigned char)n;
397 }
398 
399 // discard any buffered input
400 void m_usb_rx_flush(void)
401 {
402     uint8_t intr_state;
403 
404     if (usb_configuration) {
405         intr_state = SREG;
406         cli();
407         UENUM = CDC_RX_ENDPOINT;
408         while ((UEINTX & (1<<RWAL))) {
409             UEINTX = 0x6B;
410         }
411         SREG = intr_state;
412     }
413 }
414 
415 // transmit a character.  0 returned on success, -1 on error
416 char m_usb_tx_char(unsigned char c)
417 {
418     uint8_t timeout, intr_state;
419 
420     // if we're not online (enumerated and configured), error
421     if (!usb_configuration) return -1;
422     // interrupts are disabled so these functions can be
423     // used from the main program or interrupt context,
```

```
424     // even both in the same program!
425     intr_state = SREG;
426     cli();
427     UENUM = CDC_TX_ENDPOINT;
428     // if we gave up due to timeout before, don't wait again
429     if (transmit_previous_timeout) {
430         if (!(UEINTX & (1<<RWAL))) {
431             SREG = intr_state;
432             return -1;
433         }
434         transmit_previous_timeout = 0;
435     }
436     // wait for the FIFO to be ready to accept data
437     timeout = UDFNUML + TRANSMIT_TIMEOUT;
438     while (1) {
439         // are we ready to transmit?
440         if (UEINTX & (1<<RWAL)) break;
441         SREG = intr_state;
442         // have we waited too long?  This happens if the user
443         // is not running an application that is listening
444         if (UDFNUML == timeout) {
445             transmit_previous_timeout = 1;
446             return -1;
447         }
448         // has the USB gone offline?
449         if (!usb_configuration) return -1;
450         // get ready to try checking again
451         intr_state = SREG;
452         cli();
453         UENUM = CDC_TX_ENDPOINT;
454     }
455     // actually write the byte into the FIFO
456     UEDATX = (uint8_t)c;
457     // if this completed a packet, transmit it now!
458     if (!(UEINTX & (1<<RWAL))) UEINTX = 0x3A;
459     transmit_flush_timer = TRANSMIT_FLUSH_TIMEOUT;
460     SREG = intr_state;
461     return 0;
462 }
463
464
465 // transmit a character, but do not wait if the buffer is full,
466 //   0 returned on success, -1 on buffer full or error
467 int8_t usb_serial_putchar_nowait(uint8_t c)
468 {
469     uint8_t intr_state;
470
471     if (!usb_configuration) return -1;
472     intr_state = SREG;
473     cli();
474     UENUM = CDC_TX_ENDPOINT;
475     if (!(UEINTX & (1<<RWAL))) {
476         // buffer is full
477         SREG = intr_state;
```

```
478         return -1;
479     }
480     // actually write the byte into the FIFO
481     UEDATX = c;
482         // if this completed a packet, transmit it now!
483     if (!(UEINTX & (1<<RWAL))) UEINTX = 0x3A;
484     transmit_flush_timer = TRANSMIT_FLUSH_TIMEOUT;
485     SREG = intr_state;
486     return 0;
487 }
488
489 // transmit a buffer.
490 //  0 returned on success, -1 on error
491 // This function is optimized for speed!  Each call takes approx 6.1 us
    overhead
492 // plus 0.25 us per byte.  12 Mbit/sec USB has 8.67 us per-packet overhead
     and
493 // takes 0.67 us per byte.  If called with 64 byte packet-size blocks,
    this function
494 // can transmit at full USB speed using 43% CPU time.  The maximum
    theoretical speed
495 // is 19 packets per USB frame, or 1216 kbytes/sec.  However, bulk
    endpoints have the
496 // lowest priority, so any other USB devices will likely reduce the speed.
      Speed
497 // can also be limited by how quickly the PC-based software reads data, as
     the host
498 // controller in the PC will not allocate bandwitdh without a pending read
     request.
499 // (thanks to Victor Suarez for testing and feedback and initial code)
500
501 int8_t usb_serial_write(const uint8_t *buffer, uint16_t size)
502 {
503     uint8_t timeout, intr_state, write_size;
504
505     // if we're not online (enumerated and configured), error
506     if (!usb_configuration) return -1;
507     // interrupts are disabled so these functions can be
508     // used from the main program or interrupt context,
509     // even both in the same program!
510     intr_state = SREG;
511     cli();
512     UENUM = CDC_TX_ENDPOINT;
513     // if we gave up due to timeout before, don't wait again
514     if (transmit_previous_timeout) {
515         if (!(UEINTX & (1<<RWAL))) {
516             SREG = intr_state;
517             return -1;
518         }
519         transmit_previous_timeout = 0;
520     }
521     // each iteration of this loop transmits a packet
522     while (size) {
523         // wait for the FIFO to be ready to accept data
```

```
524         timeout = UDFNUML + TRANSMIT_TIMEOUT;
525         while (1) {
526             // are we ready to transmit?
527             if (UEINTX & (1<<RWAL)) break;
528             SREG = intr_state;
529             // have we waited too long?  This happens if the user
530             // is not running an application that is listening
531             if (UDFNUML == timeout) {
532                 transmit_previous_timeout = 1;
533                 return -1;
534             }
535             // has the USB gone offline?
536             if (!usb_configuration) return -1;
537             // get ready to try checking again
538             intr_state = SREG;
539             cli();
540             UENUM = CDC_TX_ENDPOINT;
541         }
542 
543         // compute how many bytes will fit into the next packet
544         write_size = CDC_TX_SIZE - UEBCLX;
545         if (write_size > size) write_size = size;
546         size -= write_size;
547 
548         // write the packet
549         switch (write_size) {
550             #if (CDC_TX_SIZE == 64)
551             case 64: UEDATX = *buffer++;
552             case 63: UEDATX = *buffer++;
553             case 62: UEDATX = *buffer++;
554             case 61: UEDATX = *buffer++;
555             case 60: UEDATX = *buffer++;
556             case 59: UEDATX = *buffer++;
557             case 58: UEDATX = *buffer++;
558             case 57: UEDATX = *buffer++;
559             case 56: UEDATX = *buffer++;
560             case 55: UEDATX = *buffer++;
561             case 54: UEDATX = *buffer++;
562             case 53: UEDATX = *buffer++;
563             case 52: UEDATX = *buffer++;
564             case 51: UEDATX = *buffer++;
565             case 50: UEDATX = *buffer++;
566             case 49: UEDATX = *buffer++;
567             case 48: UEDATX = *buffer++;
568             case 47: UEDATX = *buffer++;
569             case 46: UEDATX = *buffer++;
570             case 45: UEDATX = *buffer++;
571             case 44: UEDATX = *buffer++;
572             case 43: UEDATX = *buffer++;
573             case 42: UEDATX = *buffer++;
574             case 41: UEDATX = *buffer++;
575             case 40: UEDATX = *buffer++;
576             case 39: UEDATX = *buffer++;
577             case 38: UEDATX = *buffer++;
```

```
578             case 37: UEDATX = *buffer++;
579             case 36: UEDATX = *buffer++;
580             case 35: UEDATX = *buffer++;
581             case 34: UEDATX = *buffer++;
582             case 33: UEDATX = *buffer++;
583             #endif
584             #if (CDC_TX_SIZE >= 32)
585             case 32: UEDATX = *buffer++;
586             case 31: UEDATX = *buffer++;
587             case 30: UEDATX = *buffer++;
588             case 29: UEDATX = *buffer++;
589             case 28: UEDATX = *buffer++;
590             case 27: UEDATX = *buffer++;
591             case 26: UEDATX = *buffer++;
592             case 25: UEDATX = *buffer++;
593             case 24: UEDATX = *buffer++;
594             case 23: UEDATX = *buffer++;
595             case 22: UEDATX = *buffer++;
596             case 21: UEDATX = *buffer++;
597             case 20: UEDATX = *buffer++;
598             case 19: UEDATX = *buffer++;
599             case 18: UEDATX = *buffer++;
600             case 17: UEDATX = *buffer++;
601             #endif
602             #if (CDC_TX_SIZE >= 16)
603             case 16: UEDATX = *buffer++;
604             case 15: UEDATX = *buffer++;
605             case 14: UEDATX = *buffer++;
606             case 13: UEDATX = *buffer++;
607             case 12: UEDATX = *buffer++;
608             case 11: UEDATX = *buffer++;
609             case 10: UEDATX = *buffer++;
610             case  9: UEDATX = *buffer++;
611             #endif
612             case  8: UEDATX = *buffer++;
613             case  7: UEDATX = *buffer++;
614             case  6: UEDATX = *buffer++;
615             case  5: UEDATX = *buffer++;
616             case  4: UEDATX = *buffer++;
617             case  3: UEDATX = *buffer++;
618             case  2: UEDATX = *buffer++;
619             default:
620             case  1: UEDATX = *buffer++;
621             case  0: break;
622         }
623         // if this completed a packet, transmit it now!
624         if (!(UEINTX & (1<<RWAL))) UEINTX = 0x3A;
625         transmit_flush_timer = TRANSMIT_FLUSH_TIMEOUT;
626     }
627     SREG = intr_state;
628     return 0;
629 }
630
631
```

```
632 // immediately transmit any buffered output.
633 // This doesn't actually transmit the data - that is impossible!
634 // USB devices only transmit when the host allows, so the best
635 // we can do is release the FIFO buffer for when the host wants it
636 void m_usb_tx_push(void)
637 {
638     uint8_t intr_state;
639 
640     intr_state = SREG;
641     cli();
642     if (transmit_flush_timer) {
643         UENUM = CDC_TX_ENDPOINT;
644         UEINTX = 0x3A;
645         transmit_flush_timer = 0;
646     }
647     SREG = intr_state;
648 }
649 
650 // functions to read the various async serial settings.  These
651 // aren't actually used by USB at all (communication is always
652 // at full USB speed), but they are set by the host so we can
653 // set them properly if we're converting the USB to a real serial
654 // communication
655 
656 //uint32_t usb_serial_get_baud(void)
657 //{
658 //  return *(uint32_t *)cdc_line_coding;
659 //}
660 uint8_t usb_serial_get_stopbits(void)
661 {
662     return cdc_line_coding[4];
663 }
664 uint8_t usb_serial_get_paritytype(void)
665 {
666     return cdc_line_coding[5];
667 }
668 uint8_t usb_serial_get_numbits(void)
669 {
670     return cdc_line_coding[6];
671 }
672 uint8_t usb_serial_get_control(void)
673 {
674     return cdc_line_rtsdtr;
675 }
676 // write the control signals, DCD, DSR, RI, etc
677 // There is no CTS signal.  If software on the host has transmitted
678 // data to you but you haven't been calling the getchar function,
679 // it remains buffered (either here or on the host) and can not be
680 // lost because you weren't listening at the right time, like it
681 // would in real serial communication.
682 // TODO: this function is untested.  Does it work?  Please email
683 // paul@pjrc.com if you have tried it....
684 int8_t usb_serial_set_control(uint8_t signals)
685 {
```

```
686     uint8_t intr_state;
687
688     intr_state = SREG;
689     cli();
690     if (!usb_configuration) {
691         // we're not enumerated/configured
692         SREG = intr_state;
693         return -1;
694     }
695
696     UENUM = CDC_ACM_ENDPOINT;
697     if (!(UEINTX & (1<<RWAL))) {
698         // unable to write
699         // TODO; should this try to abort the previously
700         // buffered message??
701         SREG = intr_state;
702         return -1;
703     }
704     UEDATX = 0xA1;
705     UEDATX = 0x20;
706     UEDATX = 0;
707     UEDATX = 0;
708     UEDATX = 0; // TODO: should this be 1 or 0 ???
709     UEDATX = 0;
710     UEDATX = 2;
711     UEDATX = 0;
712     UEDATX = signals;
713     UEDATX = 0;
714     UEINTX = 0x3A;
715     SREG = intr_state;
716     return 0;
717 }
718
719
720
721 /*
    ************************************************************************

722  *
723  *  Private Functions - not intended for general user consumption....
724  *
725  ************************************************************************
    */
726
727
728 // USB Device Interrupt - handle all device-level events
729 // the transmit buffer flushing is triggered by the start of frame
730 //
731 ISR(USB_GEN_vect)
732 {
733     uint8_t intbits, t;
734
735         intbits = UDINT;
736         UDINT = 0;
```

```
737         if (intbits & (1<<EORSTI)) {
738         UENUM = 0;
739         UECONX = 1;
740         UECFG0X = EP_TYPE_CONTROL;
741         UECFG1X = EP_SIZE(ENDPOINT0_SIZE) | EP_SINGLE_BUFFER;
742         UEIENX = (1<<RXSTPE);
743         usb_configuration = 0;
744         cdc_line_rtsdtr = 0;
745         }
746     if (intbits & (1<<SOFI)) {
747         if (usb_configuration) {
748             t = transmit_flush_timer;
749             if (t) {
750                 transmit_flush_timer = --t;
751                 if (!t) {
752                     UENUM = CDC_TX_ENDPOINT;
753                     UEINTX = 0x3A;
754                 }
755             }
756         }
757     }
758 }
759
760
761 // Misc functions to wait for ready and send/receive packets
762 static inline void usb_wait_in_ready(void)
763 {
764     while (!(UEINTX & (1<<TXINI))) ;
765 }
766 static inline void usb_send_in(void)
767 {
768     UEINTX = ~(1<<TXINI);
769 }
770 static inline void usb_wait_receive_out(void)
771 {
772     while (!(UEINTX & (1<<RXOUTI))) ;
773 }
774 static inline void usb_ack_out(void)
775 {
776     UEINTX = ~(1<<RXOUTI);
777 }
778
779
780
781 // USB Endpoint Interrupt - endpoint 0 is handled here.  The
782 // other endpoints are manipulated by the user-callable
783 // functions, and the start-of-frame interrupt.
784 //
785 ISR(USB_COM_vect)
786 {
787         uint8_t intbits;
788     const uint8_t *list;
789         const uint8_t *cfg;
790     uint8_t i, n, len, en;
```

```
791     uint8_t *p;
792     uint8_t bmRequestType;
793     uint8_t bRequest;
794     uint16_t wValue;
795     uint16_t wIndex;
796     uint16_t wLength;
797     uint16_t desc_val;
798     const uint8_t *desc_addr;
799     uint8_t desc_length;
800 
801         UENUM = 0;
802         intbits = UEINTX;
803         if (intbits & (1<<RXSTPI)) {
804                 bmRequestType = UEDATX;
805                 bRequest = UEDATX;
806                 wValue = UEDATX;
807                 wValue |= (UEDATX << 8);
808                 wIndex = UEDATX;
809                 wIndex |= (UEDATX << 8);
810                 wLength = UEDATX;
811                 wLength |= (UEDATX << 8);
812                 UEINTX = ~((1<<RXSTPI) | (1<<RXOUTI) | (1<<TXINI));
813                 if (bRequest == GET_DESCRIPTOR) {
814             list = (const uint8_t *)descriptor_list;
815             for (i=0; ; i++) {
816                 if (i >= NUM_DESC_LIST) {
817                     UECONX = (1<<STALLRQ)|(1<<EPEN);  //stall
818                     return;
819                 }
820                 desc_val = pgm_read_word(list);
821                 if (desc_val != wValue) {
822                     list += sizeof(struct descriptor_list_struct);
823                     continue;
824                 }
825                 list += 2;
826                 desc_val = pgm_read_word(list);
827                 if (desc_val != wIndex) {
828                     list += sizeof(struct descriptor_list_struct)-2;
829                     continue;
830                 }
831                 list += 2;
832                 desc_addr = (const uint8_t *)pgm_read_word(list);
833                 list += 2;
834                 desc_length = pgm_read_byte(list);
835                 break;
836             }
837             len = (wLength < 256) ? wLength : 255;
838             if (len > desc_length) len = desc_length;
839             do {
840                 // wait for host ready for IN packet
841                 do {
842                     i = UEINTX;
843                 } while (!(i & ((1<<TXINI)|(1<<RXOUTI))));
844                 if (i & (1<<RXOUTI)) return;    // abort
```

```
845                 // send IN packet
846                 n = len < ENDPOINT0_SIZE ? len : ENDPOINT0_SIZE;
847                 for (i = n; i; i--) {
848                     UEDATX = pgm_read_byte(desc_addr++);
849                 }
850                 len -= n;
851                 usb_send_in();
852             } while (len || n == ENDPOINT0_SIZE);
853             return;
854                 }
855         if (bRequest == SET_ADDRESS) {
856             usb_send_in();
857             usb_wait_in_ready();
858             UDADDR = wValue | (1<<ADDEN);
859             return;
860         }
861         if (bRequest == SET_CONFIGURATION && bmRequestType == 0) {
862             usb_configuration = wValue;
863             cdc_line_rtsdtr = 0;
864             transmit_flush_timer = 0;
865             usb_send_in();
866             cfg = endpoint_config_table;
867             for (i=1; i<5; i++) {
868                 UENUM = i;
869                 en = pgm_read_byte(cfg++);
870                 UECONX = en;
871                 if (en) {
872                     UECFG0X = pgm_read_byte(cfg++);
873                     UECFG1X = pgm_read_byte(cfg++);
874                 }
875             }
876                 UERST = 0x1E;
877                 UERST = 0;
878             return;
879         }
880         if (bRequest == GET_CONFIGURATION && bmRequestType == 0x80) {
881             usb_wait_in_ready();
882             UEDATX = usb_configuration;
883             usb_send_in();
884             return;
885         }
886         if (bRequest == CDC_GET_LINE_CODING && bmRequestType == 0xA1) {
887             usb_wait_in_ready();
888             p = cdc_line_coding;
889             for (i=0; i<7; i++) {
890                 UEDATX = *p++;
891             }
892             usb_send_in();
893             return;
894         }
895         if (bRequest == CDC_SET_LINE_CODING && bmRequestType == 0x21) {
896             usb_wait_receive_out();
897             p = cdc_line_coding;
898             for (i=0; i<7; i++) {
```

```
                *p++ = UEDATX;
            }
            usb_ack_out();
            usb_send_in();
            return;
        }
        if (bRequest == CDC_SET_CONTROL_LINE_STATE && bmRequestType == 0
           x21) {
            cdc_line_rtsdtr = wValue;
            usb_wait_in_ready();
            usb_send_in();
            return;
        }
        if (bRequest == GET_STATUS) {
            usb_wait_in_ready();
            i = 0;
            #ifdef SUPPORT_ENDPOINT_HALT
            if (bmRequestType == 0x82) {
                UENUM = wIndex;
                if (UECONX & (1<<STALLRQ)) i = 1;
                UENUM = 0;
            }
            #endif
            UEDATX = i;
            UEDATX = 0;
            usb_send_in();
            return;
        }
        #ifdef SUPPORT_ENDPOINT_HALT
        if ((bRequest == CLEAR_FEATURE || bRequest == SET_FEATURE)
          && bmRequestType == 0x02 && wValue == 0) {
            i = wIndex & 0x7F;
            if (i >= 1 && i <= MAX_ENDPOINT) {
                usb_send_in();
                UENUM = i;
                if (bRequest == SET_FEATURE) {
                    UECONX = (1<<STALLRQ)|(1<<EPEN);
                } else {
                    UECONX = (1<<STALLRQC)|(1<<RSTDT)|(1<<EPEN);
                    UERST = (1 << i);
                    UERST = 0;
                }
                return;
            }
        }
        #endif
        }
    UECONX = (1<<STALLRQ) | (1<<EPEN);  // stall
}


// BELOW FROM PRINT.C

void print_P(const char *s)
```

```
952 {
953     char c;
954 
955     while (1) {
956         c = pgm_read_byte(s++);
957         if (!c) break;
958         if (c == '\n') usb_tx_char('\r');
959         usb_tx_char(c);
960     }
961 }
962 
963 void phex1(unsigned char c)
964 {
965     usb_tx_char(c + ((c < 10) ? '0' : 'A' - 10));
966 }
967 
968 void phex(unsigned char c)
969 {
970     phex1(c >> 4);
971     phex1(c & 15);
972 }
973 
974 void m_usb_tx_hex(unsigned int i)
975 {
976     phex(i >> 8);
977     phex(i);
978 }
979 
980 void m_usb_tx_hexchar(unsigned char i)
981 {
982     phex(i);
983 }
984 
985 void m_usb_tx_int(int i)
986 {
987     char string[7] = {0,0,0,0,0,0,0};
988     itoa(i,string,10);
989     for(i=0;i<7;i++){
990         if(string[i]){
991             m_usb_tx_char(string[i]);
992         }
993     }
994 }
995 
996 void m_usb_tx_uint(unsigned int i)
997 {
998     char string[6] = {0,0,0,0,0,0};
999     utoa(i,string,10);
1000    for(i=0;i<5;i++){
1001        if(string[i]){
1002            m_usb_tx_char(string[i]);
1003        }
1004    }
1005 }
```

```
1006
1007 void m_usb_tx_long(long i)
1008 {
1009     char string[11] = {0,0,0,0,0,0,0,0,0,0,0};
1010     ltoa(i,string,10);
1011     for(i=0;i<11;i++){
1012         if(string[i]){
1013             m_usb_tx_char(string[i]);
1014         }
1015     }
1016 }
1017
1018 void m_usb_tx_ulong(unsigned long i)
1019 {
1020     char string[11] = {0,0,0,0,0,0,0,0,0,0,0};
1021     ultoa(i,string,10);
1022     for(i=0;i<10;i++){
1023         if(string[i]){
1024             m_usb_tx_char(string[i]);
1025         }
1026     }
1027 }
```

19 Solidworks 3D Design

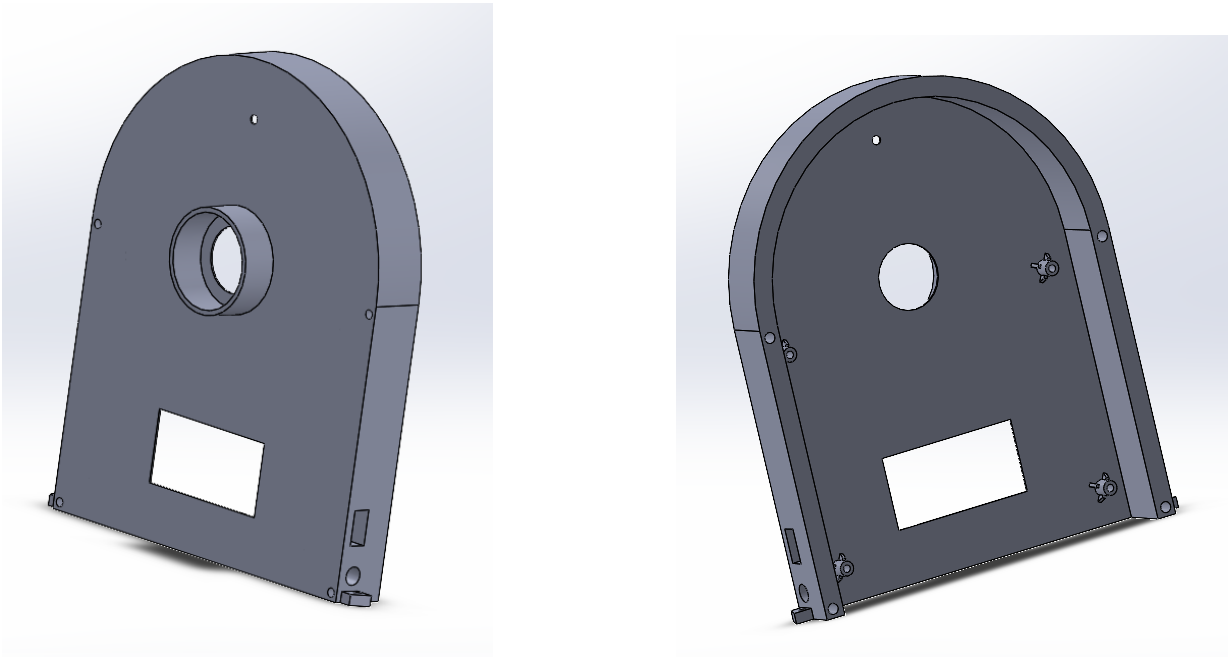
19.1 Initial Design - Version 01

The initial design was developed with these key characteristics:

- **Enclosure Structure**:
  - Two-part post-box shaped aluminum housing (front & backcovers)
  - Included mounting points for PCB and mechanical components
  - Provided space for display and USB Integration
- **User Interface**:
  - Display cutout on top surface
  - USB port for data/power on side panel
- **Internal Components**:
  - Central shaft with bearing supports
  - Stacked disk configuration (3 metal + 1 dielectric)
  - Custom holders for precise disk alignment

19.1.1 Outer Enclosure

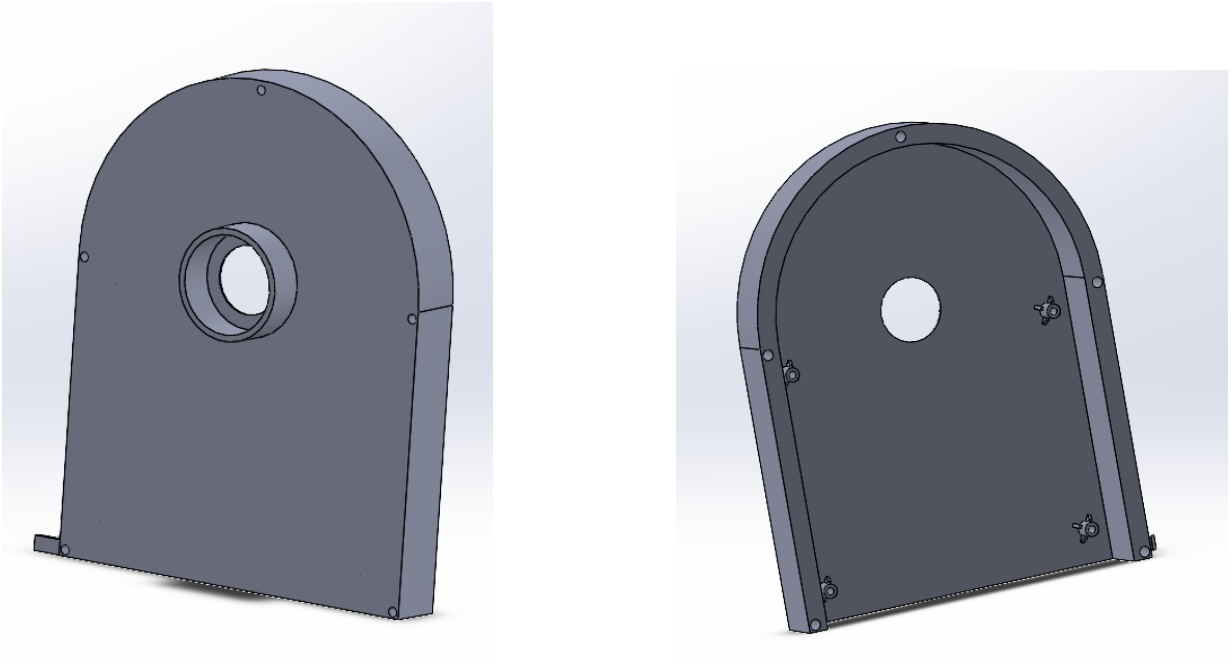
Frontside view



(a) Outer view (b) Inner view

Figure 35: Front Half

Backside view



(a) Outer view

(b) Inner view

Figure 36: Backside Half

Bottom Part

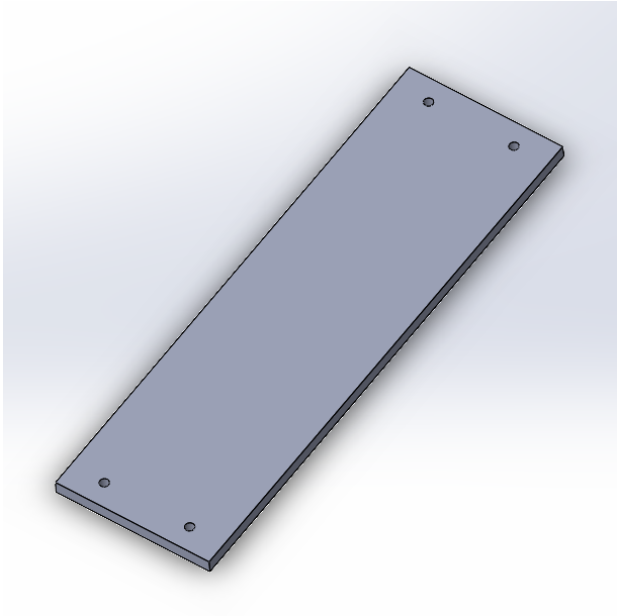


Figure 37: Bottom Part

19.1.2 Inner Parts

Disks

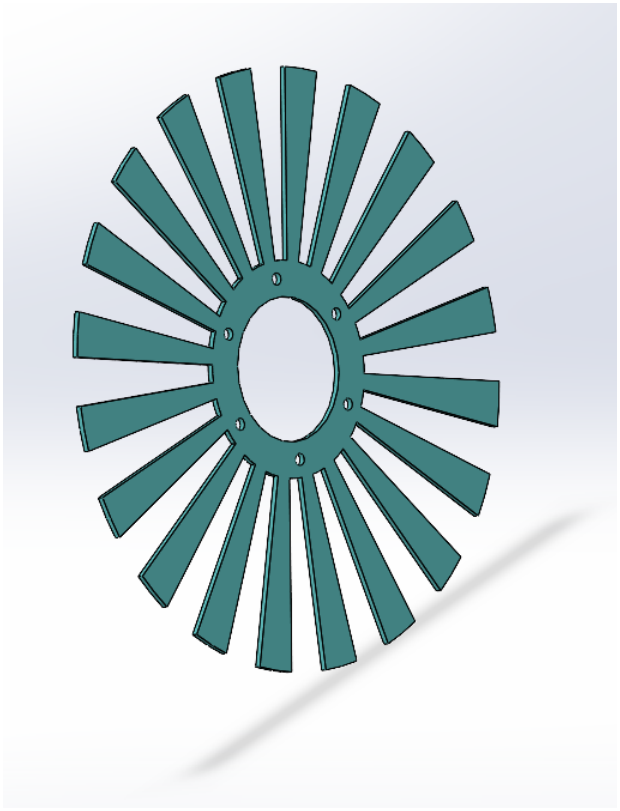


Figure 38: Dielectric Disk

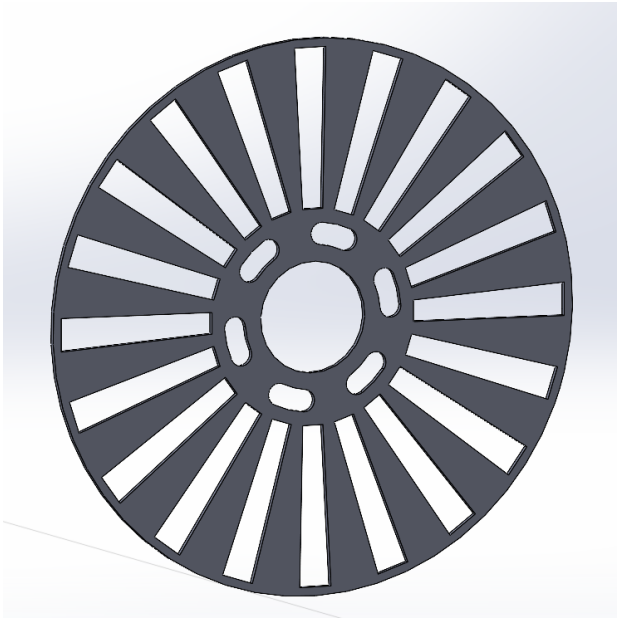
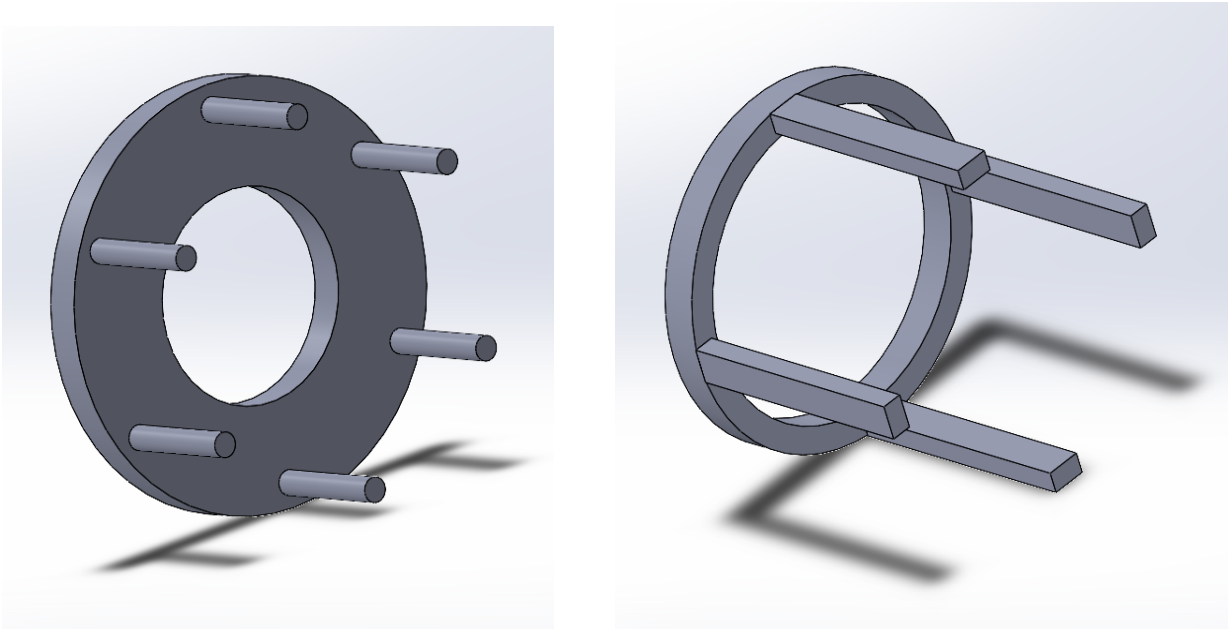


Figure 39: Metal Disk

Connecting Parts



(a) Dielectric Holder (b) Metal Plate Holder

Figure 40: Connecting Holders

Shaft

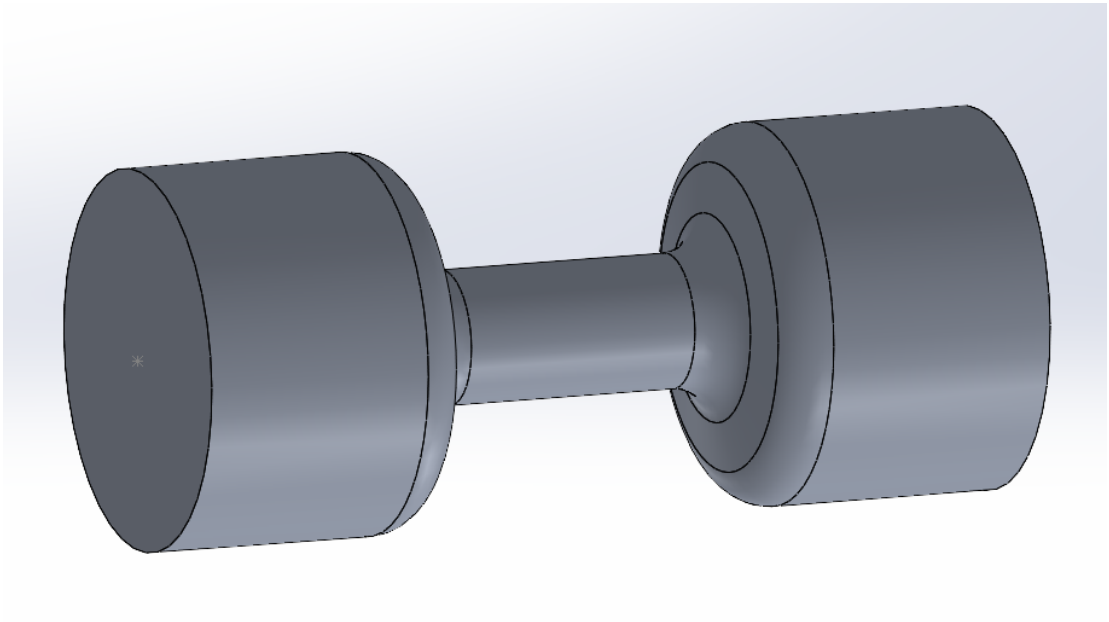


Figure 41: Shaft

19.1.3 Final Assembly

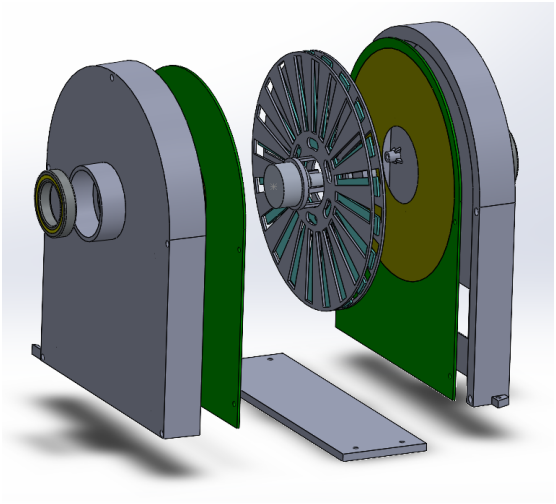


Figure 42: Backside View

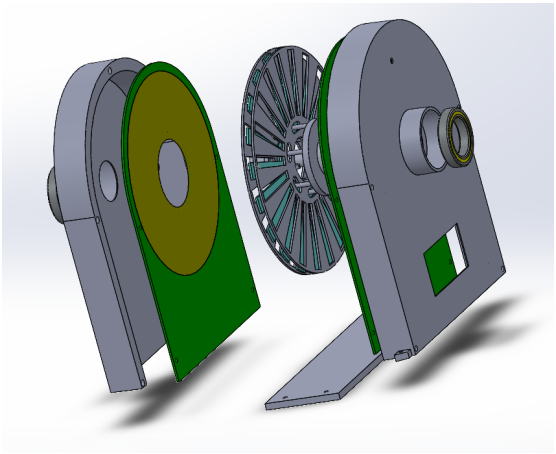


Figure 43: Frontside View

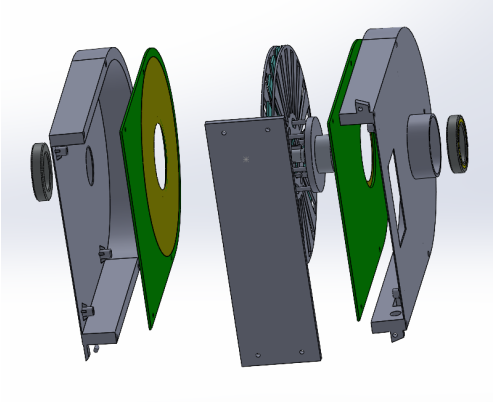


Figure 44: Bottom view

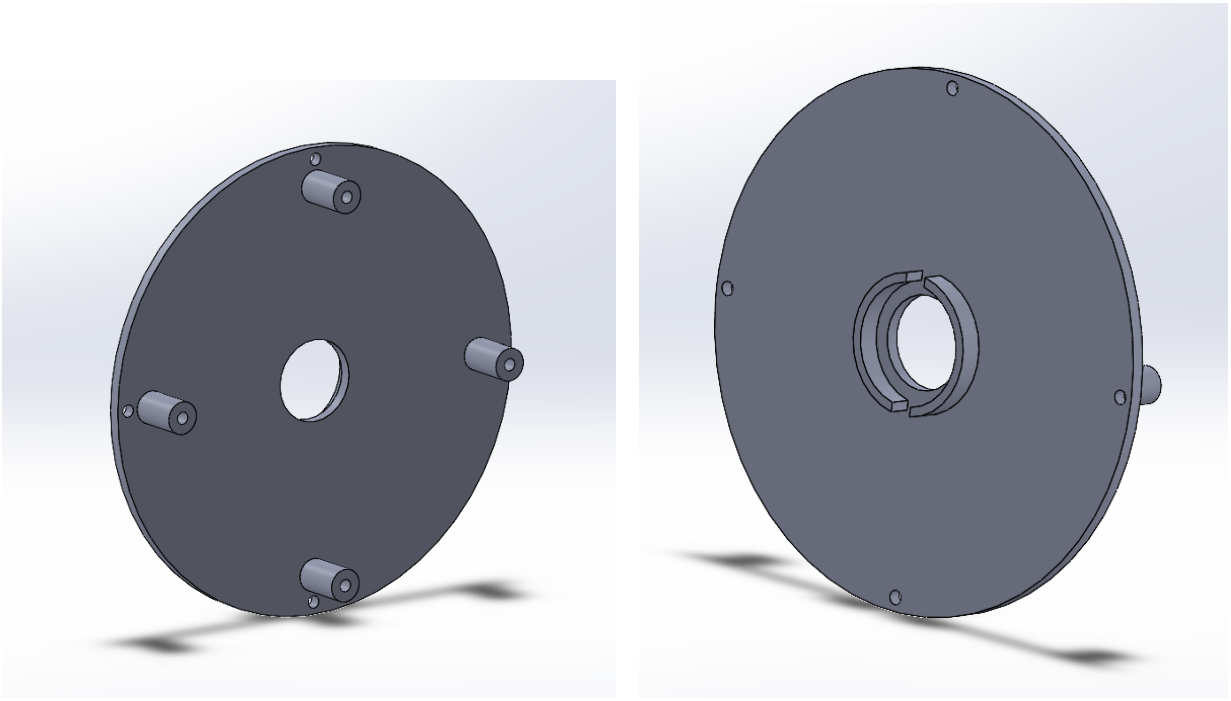
19.2 Final Design - Version 02

After consultation with Sir and benchmarking and **aligning with our refernece Design I-PEX torque sensors**, these modifications were implemented:

- **Form Factor Changes**:
  - **Shifted to cylindrical profile**
  - **Single-piece extruded aluminum body**
  - **Removed all flat surfaces**
  - **Shaft length and diameters are modified**
  - **Put holed in the shaft ends to connect with external body (eg:- Robot arm)**
- **Simplified Interface**:
  - Eliminated display to reducecomplexity
  - Single USB port serial Ouput
- **Improved Mechanical Features**:
  - Unified shaft-bearing-disk assembly
  - Preloaded angular contact bearings for better axial stability
  - Gold-plated contacts for capacitive sensing
- **Common Features Maintained**:
  - Core capacitive sensing principle (inner Disks and Disk Holders Design)
  - Identical disk materials and spacing
  - Same torque measurement range (0-1Nm)
- **Key Improvements**:
  - Better alignment with industry standards
  - 40% reduction in production cost
  - More robust shaft coupling design

19.2.1 Outer Enclosure

Front and back half



(a) Frontside (b) Backside

Figure 45: Enclosure caps

Cylindrical Body

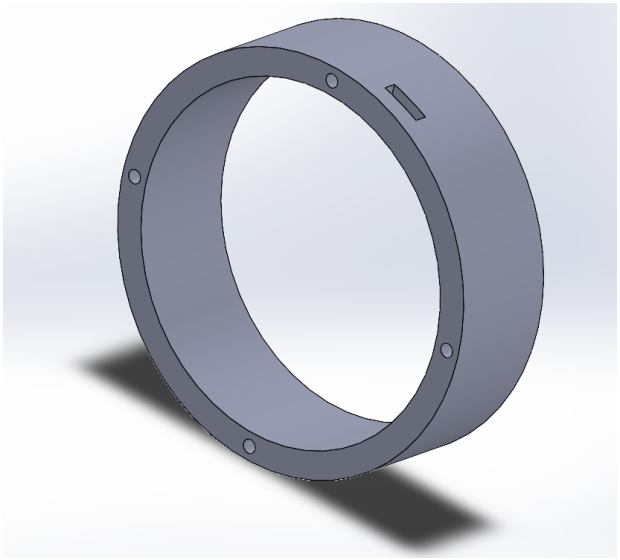


Figure 46: Cylinder Enclosure

19.2.2 Shaft

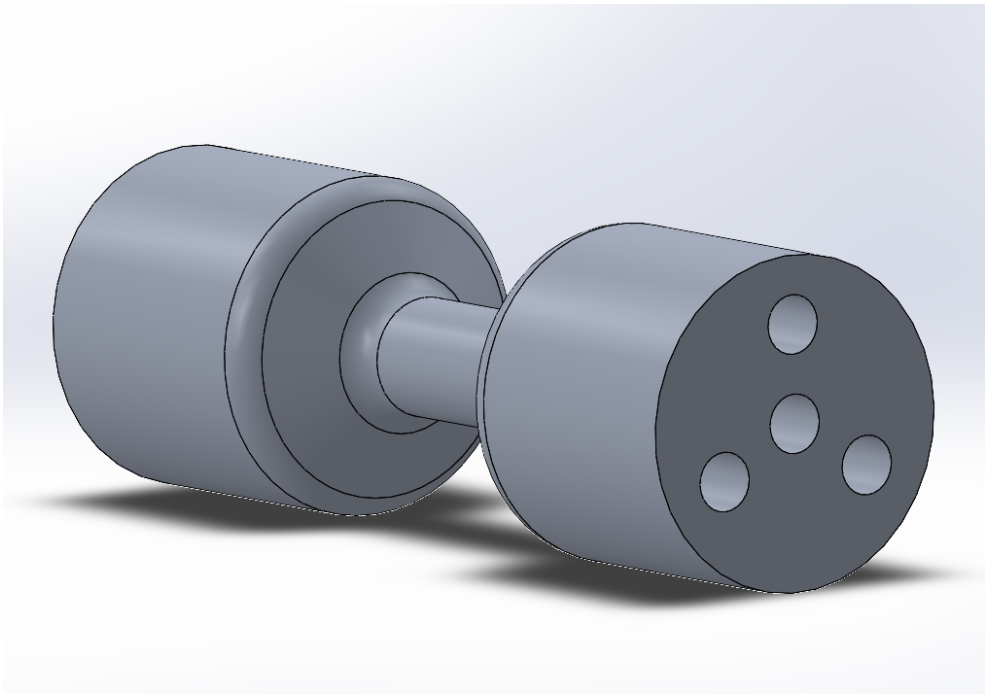
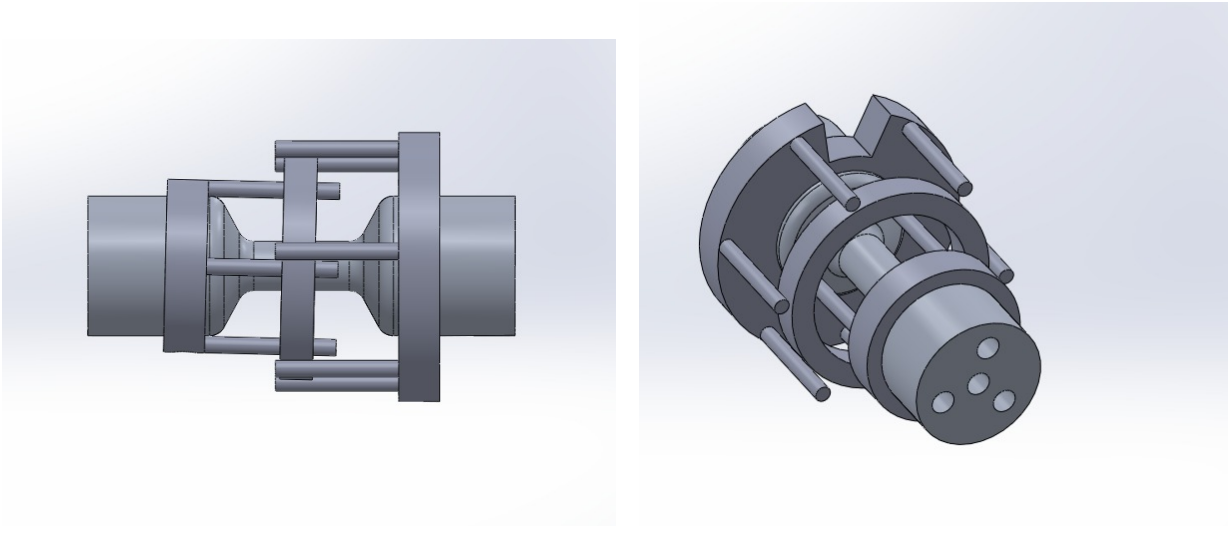


Figure 47: Final Shaft

19.2.3 Shaft and holders with modified Dimensions



(a) Side View (b) Angle view

Figure 48: Internal assembly of Shaft and Holders

19.2.4 Full assembly

Full Product Overview

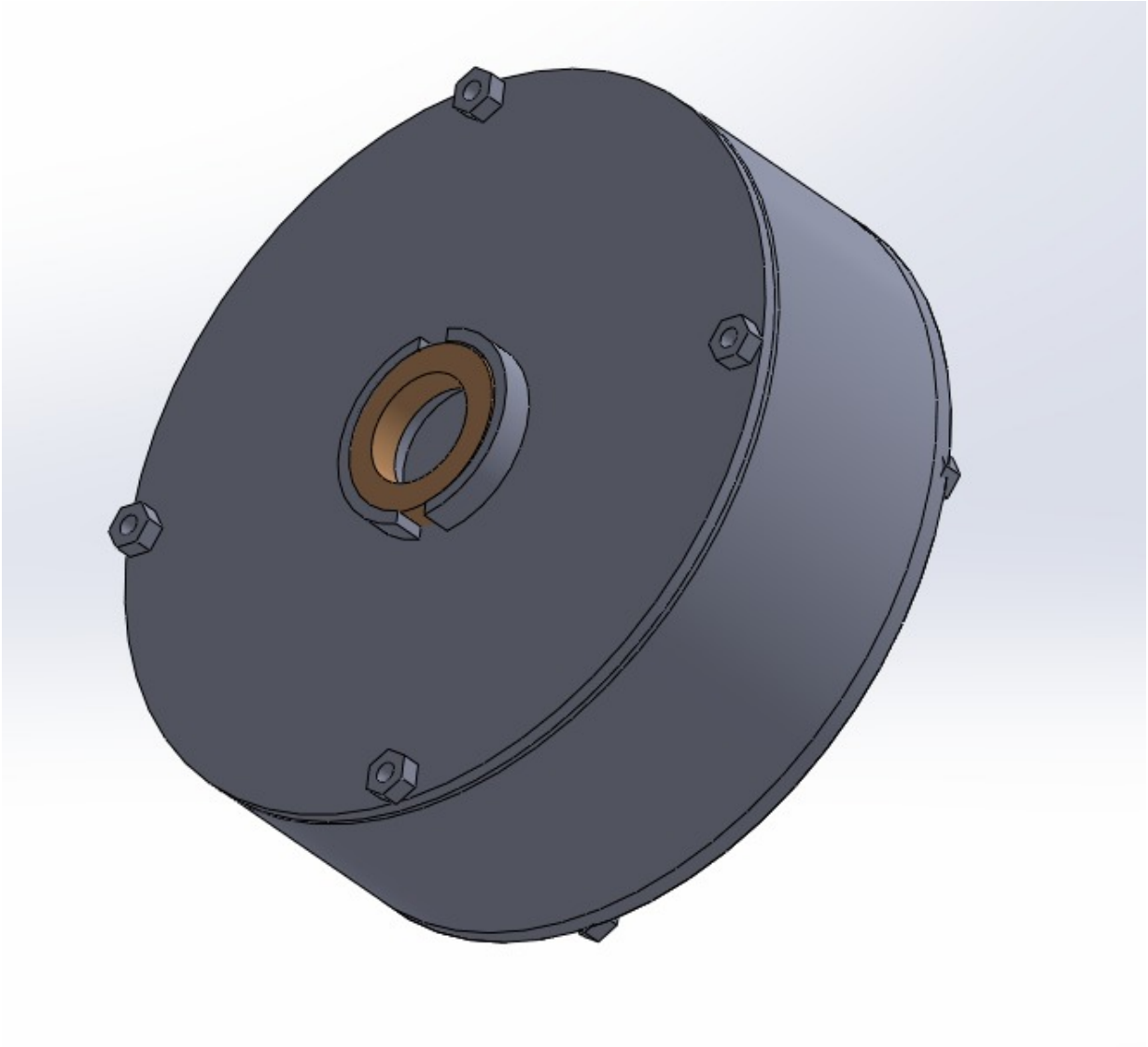


Figure 49: Full Closed view

Exploded view with internal positions

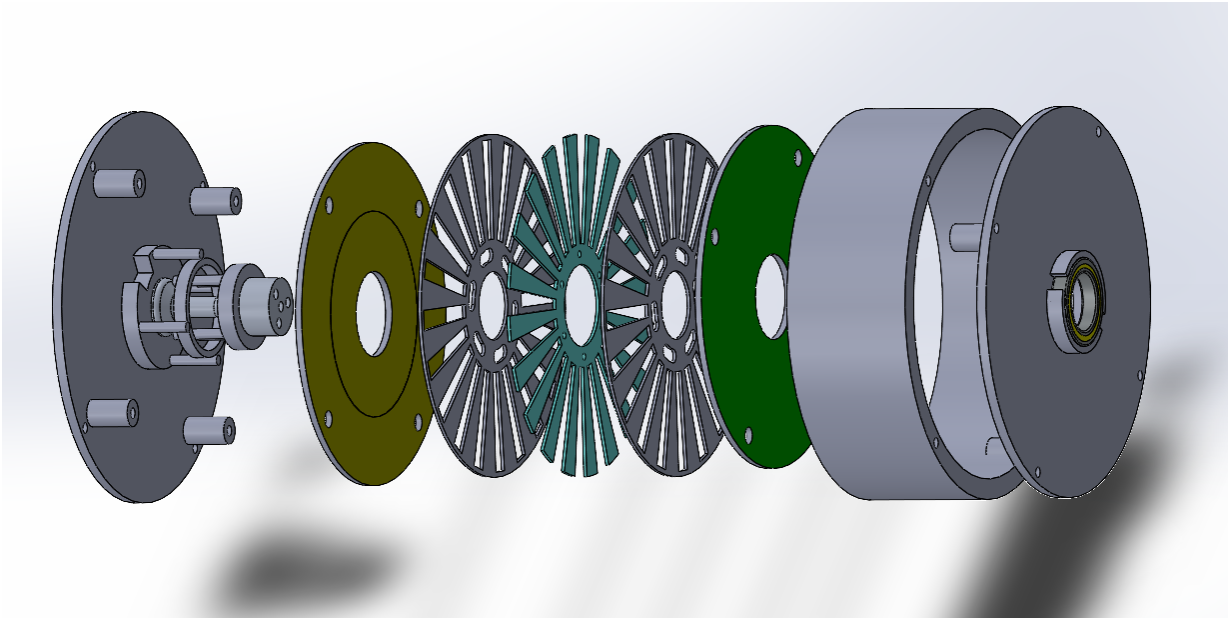


Figure 50: Exploded view 01

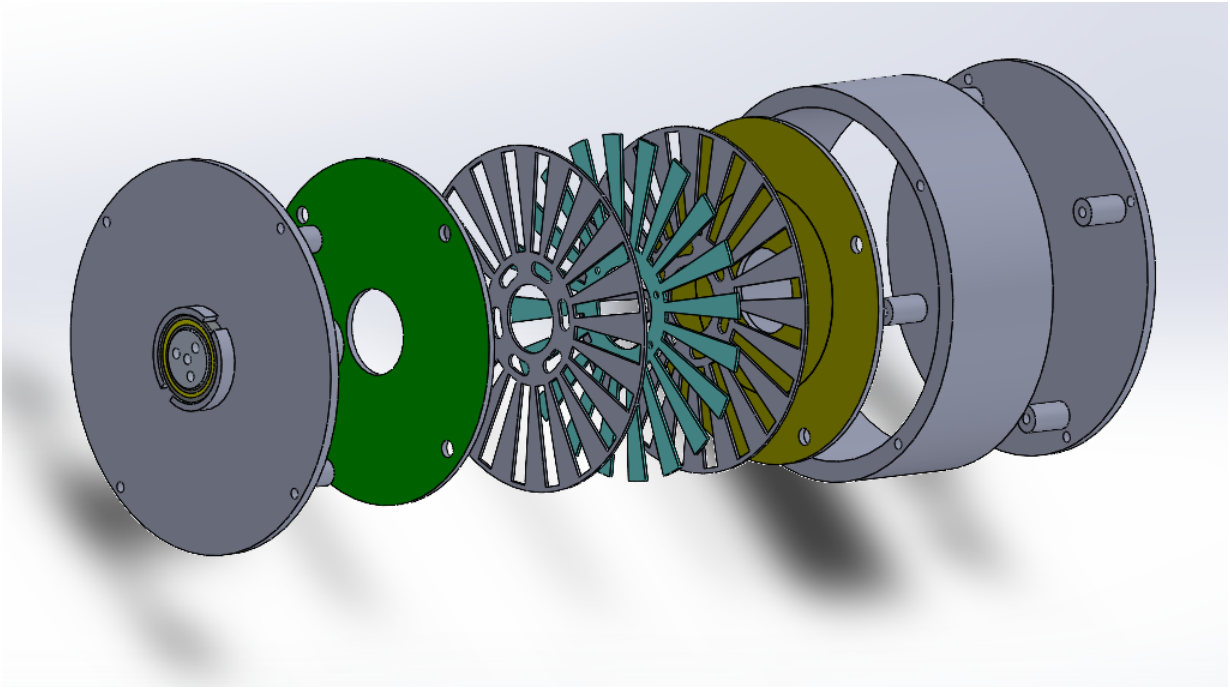
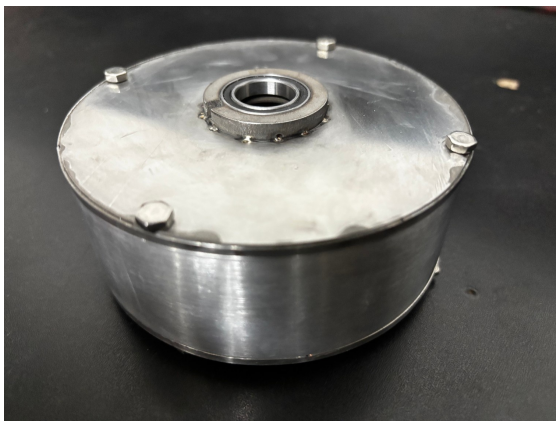
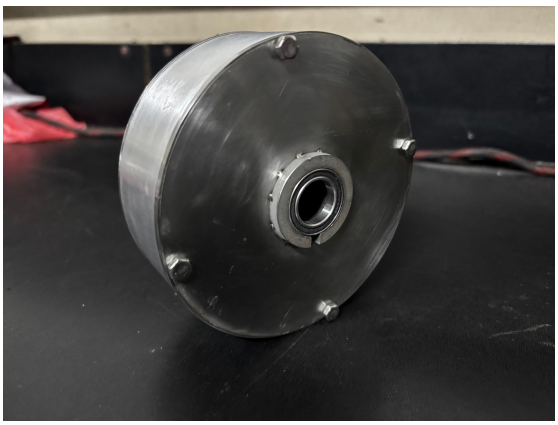


Figure 51: Exploded view 02

20 Actual Product (Physical)

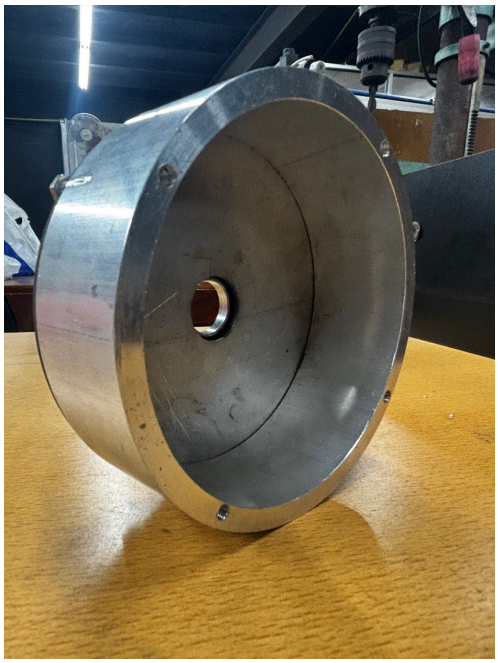


(a) Side View

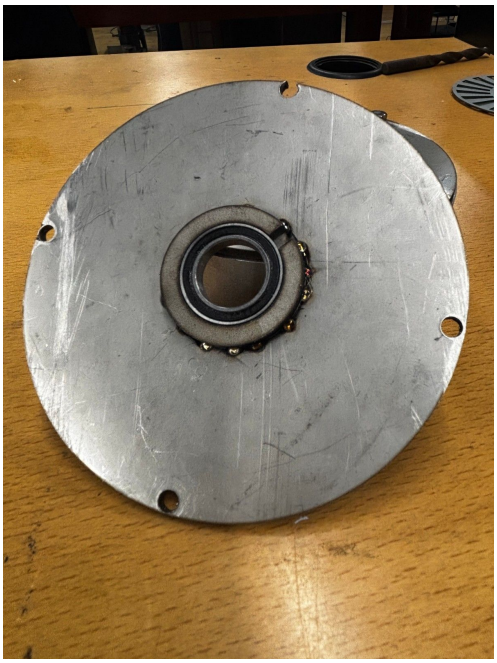


(b) Angle view

Figure 52: Actual Product Outer look



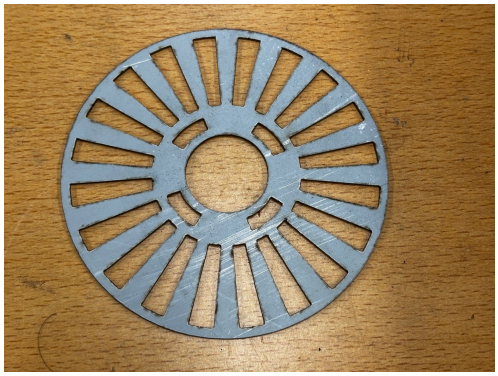
(a) Side View



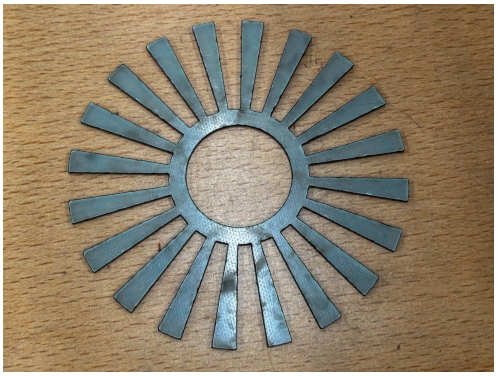
(b) Front cap with Bearings

Figure 53: Enclosure Outer look

Figure 54: Dielectric and Metal holders



(a) Side View



(b) Front cap with Bearings

Figure 55: Inner Disks

(a) Side View

(b) Front cap with Bearings

Figure 56: Shaft

21 PC Interface Software for Torque Visualization

To provide real-time monitoring and analysis of the torque values measured by our capacitive torque sensor, we developed a custom PC application named **CAPTORQ**. This application enables serial communication with the sensor via a USB interface and visually plots the incoming torque data over time.

Features

The main features of the CAPTORQ software include:

- Real-time plotting of torque values against time.
- Adjustable serial port and baud rate selection for flexible connectivity.
- Data snapshot and analysis functions for post-processing.
- Ability to pause, clear, and save data during a measurement session.

The plotting area displays the torque value (Y-axis) as a function of time (X-axis), allowing users to track dynamic changes in torque. The application is built using Python and the PyQt5 library for GUI design, with matplotlib used for plotting the live data stream.

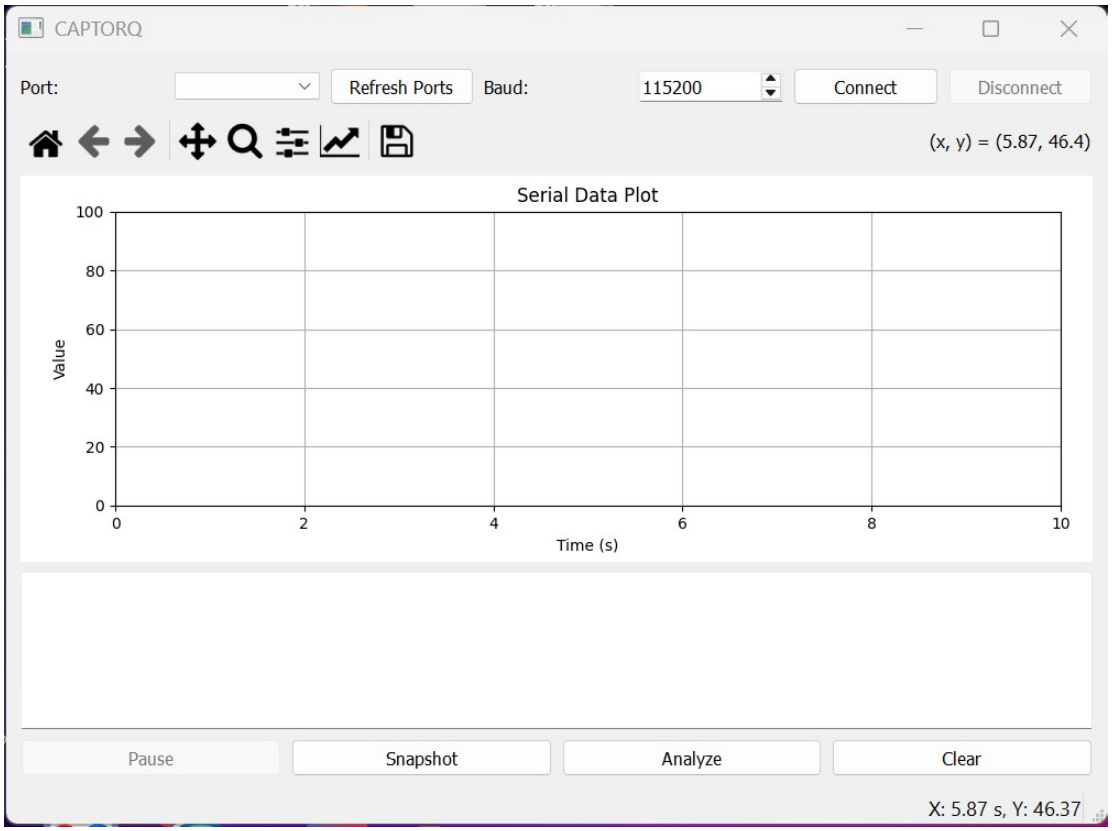


Figure 57: CAPTORQ Software Interface for Visualizing Torque Sensor Output

This tool significantly enhances usability and data interpretability for both debugging and experimental purposes, making the system more practical for laboratory and potential industrial use.

22 Production Cost Analysis for One Product

The estimated cost breakdown for manufacturing one capacitive torque sensor unit is presented in Table 2. All costs are calculated in Sri Lankan Rupees (LKR).

Table 2: Unit Production Cost Breakdown

Component/Process	**Unit Cost (LKR)**	**Percentage**
PCB Manufacturing + Shipping	2,470	9.7%
Electronic Components for PCB	9,300	36.5%
Enclosures, Metal Sheets and Shaft	4,200	16.5%
Cutting Disk and Holders	6,500	25.5%
Assembly and Physical things	3,000	11.8%
Total Production Cost	**25,470**	**100%**

Since we spent around **Rs. 25,000** to make one product when we try to maufacture this in a large scale we will have less spending.

22.1 Cost Breakdown Details

- **PCB Costs**: Includes 4-layer board fabrication, solder mask, silkscreen, and international shipping charges from the manufacturer.
- **Electronic Components**: Covers all SMD and through-hole components including the Atmega32U4 microcontroller, voltage regulators, connectors, and passive components.
- **Mechanical Parts**:
  - Enclosure CNC machining and finishing
  - Precision shaft turning and heat treatment
  - Surface treatment for corrosion resistance
- **Disk Fabrication**:
  - Laser cutting of capacitive plates
  - Precision drilling for mounting holes
  - Surface polishing for consistent capacitance
- **Assembly Costs**:
  - PCB population and soldering
  - Mechanical assembly and alignment
  - Quality control and testing