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.

2.1 Stakeholder Mapping

To develop a successful capacitive torque sensor, it is essential to identify the primary stakeholders who will interact with the product.

Figure 1: Stakeholder map for Capacitive Torque Sensor

- **Industrial Robotics Engineers** – Require precise torque measurement for automation and robotics applications.
- **Automotive Manufacturers** – Need reliable sensors for vehicle powertrain testing and safety applications.
- **Aerospace Engineers** – Use torque sensors in flight control systems and component testing.
- **Medical Device Developers** – Apply torque sensors in robotic surgery and rehabilitation equipment.
- **Research Institutions** – Require torque sensors for mechanical testing and material studies.

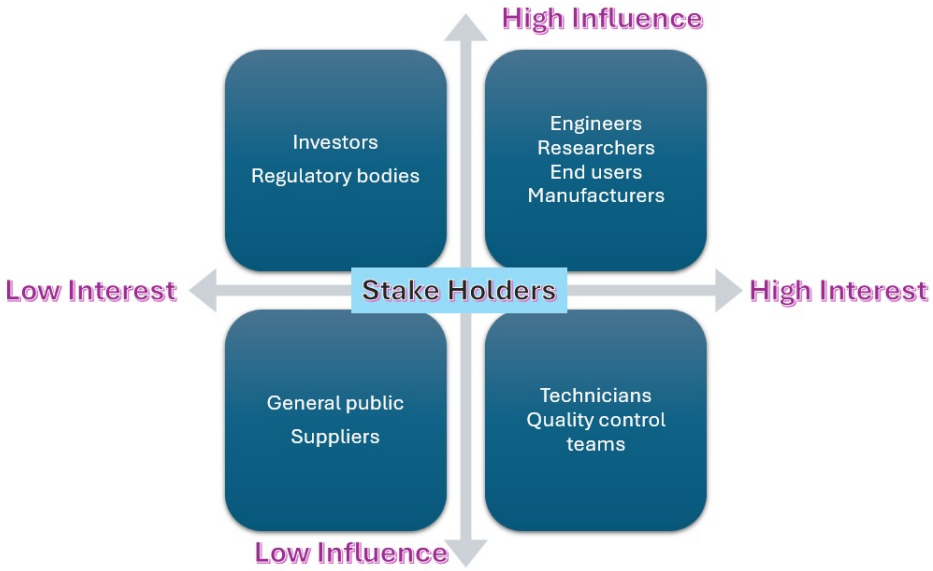


Figure 1: Stake holder Map

2.2 User Analysis: What a User Requires from a Capacitive Torque Sensor

A capacitive torque sensor is a critical component in automation, robotics, and industrial applications where precise force and torque measurements are required. Users of these sensors have specific requirements depending on the application domain. Based on two available industry sensors — **WACOH-TECH Dyn Pick** and the **Robotous RFT-Series** — the following key user requirements have been identified:

High Accuracy and Sensitivity

- Users require sensors with high resolution and sensitivity to detect even minor force variations.
- The RFT-Series provides resolutions as low as 0.08 N (force) and 0.004 Nm (torque) for high-precision applications.

Real-Time Data Acquisition and Communication Interfaces

- High-speed data acquisition is necessary to enable real-time control in robotics and automation.
- **EtherCAT Interface**: Ensures low-latency, high-speed communication.
- **Multi-Protocol Support**: CAN, RS-232, RS-422, and USB connectivity for compatibility.

Durability and Environmental Resistance

- Must withstand overload without losing calibration.
- WACOH-TECH Dyn Pick features an internal stopper mechanism.
- RFT-Series includes temperature compensation and IP65 ingress protection.

Compact and Lightweight Design

- Small size and weight are essential for robotic integration.
- RFT-Series models range from 60 g to 294 g while maintaining high performance.

Ease of Installation and Maintenance

- Plug-and-play mounting, embedded EtherCAT boards, and built-in signal processing reduce setup time.
- Dyn Pick features integrated microcomputers that auto-correct signals.

Cost-Effectiveness and Customization

- Offers a cost-efficient alternative to strain-gauge sensors.
- Custom calibration and mechanical designs for special use cases.

Safety and Overload Protection

- Overload detection and protection features are critical.
- RFT-Series includes overload counters; Dyn Pick includes mechanical stoppers.

Summary of User Requirements

Requirement	**Description**
High Accuracy	Precision as low as 0.08 N (force) and 0.004 Nm (torque) for detailed measurements.
Real-Time Data	EtherCAT, CAN, and RS-232 support for industrial automation and robotics.
Durability	Overload protection, temperature compensation, and IP65 sealing for reliability.
Compact Design	Lightweight models (60 g–294 g) for easy integration into robotic arms and automation setups.
Easy Installation	Plug-and-play EtherCAT and built-in microcomputers for signal correction.
Cost-Effectiveness	Low-cost capacitive sensors compared to strain-gauge alternatives.
Safety	Overload detection counters and mechanical stoppers prevent sensor damage.

Table 1: Summary of Capacitive Torque Sensor User Requirements

2.3 Personas (User Categories)

Based on stakeholder observations, the following user personas were identified:

Persona 1: Robotics Engineer

Needs: High-precision real-time torque feedback, compact design, EtherCAT connectivity.
Challenges: Existing sensors are costly and lack high-frequency response.

Persona 2: Automotive Test Engineer

Needs: Durable sensor for vehicle testing, CAN bus compatibility, high load-bearing capacity.
Challenges: Traditional sensors suffer from mechanical wear and require frequent calibration.

Persona 3: Research Scientist

Needs: High-resolution measurements, low-noise data acquisition, digital interfacing options.
Challenges: Strain-gauge sensors provide unstable long-term readings, affecting research accuracy.

2.4 User Journey: From Manufacturing to End-of-Life

2.4.1 1. Manufacturing and Assembly

- Components such as capacitive plates, microcontrollers, and housings are sourced.
- Fabrication and calibration of sensing elements are performed.
- Final assembly and quality control are completed.

2.4.2 2. Distribution and Supply Chain

- Sensors shipped to OEMs and distributors in robotics, automotive, and medical industries.
- Integrated into robotic arms, test benches, and machinery.

2.4.3 3. Deployment and Operation

- Engineers and researchers implement sensors in real-world applications.
- Sensors provide real-time torque feedback.
- Interfaces like EtherCAT and CAN ensure compatibility.

2.4.4 4. Maintenance and Calibration

- Regular calibration ensures accuracy.
- Firmware updates maintain compatibility.
- Inspections identify wear and damage.

2.4.5 5. End-of-Life: Repair, Recycling, or Disposal

- Repair includes calibration, part replacement, or firmware updates.
- Recyclable materials (e.g., aluminum, PCBs) are salvaged.
- Disposal complies with environmental safety regulations.

2.5 Needs List & Key Design Considerations

Based on user observations, the following needs and considerations are identified:

- **High sensitivity** – Detect small torque variations with precision.
- **Non-contact measurement** – Capacitive sensing avoids mechanical wear.
- **Multi-protocol communication** – EtherCAT and CAN support for integration.
- **Compact and lightweight** – Important for robotics and aerospace.
- **High environmental resistance** – Resilient to temperature, dust, and moisture.
- **Real-time data acquisition** – Critical for dynamic applications.
- **Sustainable End-of-Life Strategy** – Emphasis on recyclability and repairability.

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 2):

- **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.

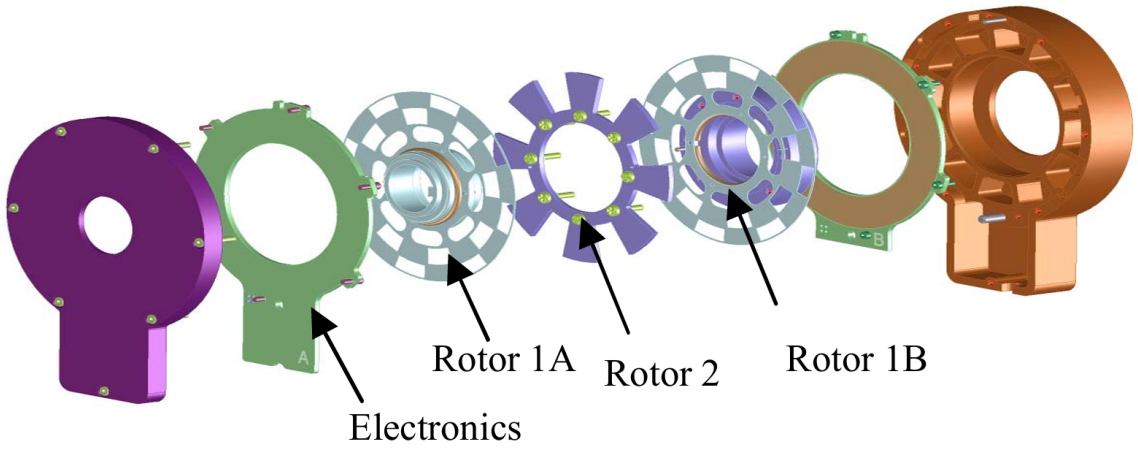


Figure 2: Exploded view of the sensor assembly. The central dielectric disk (Rotor 2) rotates with the shaft, while the outer metal disks (Rotor 1A/B, gray) remain stationary.

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 3).
3. The capacitance variation is measured by electronics connected to the metal disks, proportional to the applied torque.

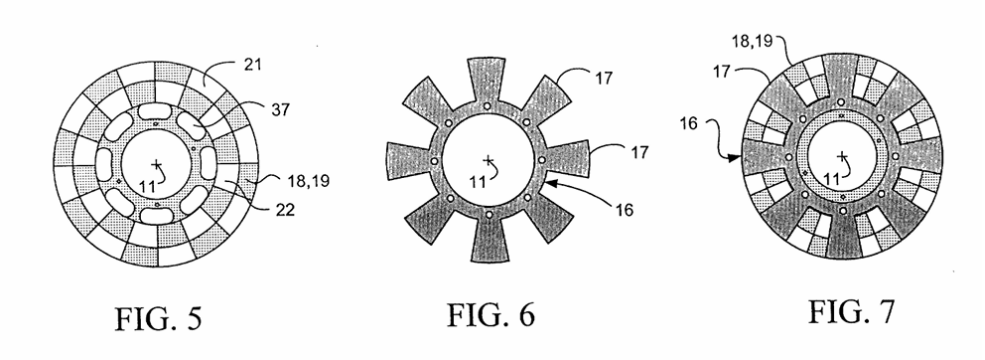


Figure 3: Top-down views of disk geometries: (a) Metal disks (Rotor 1A/1B) with symmetric electrode patterns; (b) Dielectric disk (Rotor 2) with alternating permittivity regions.

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.

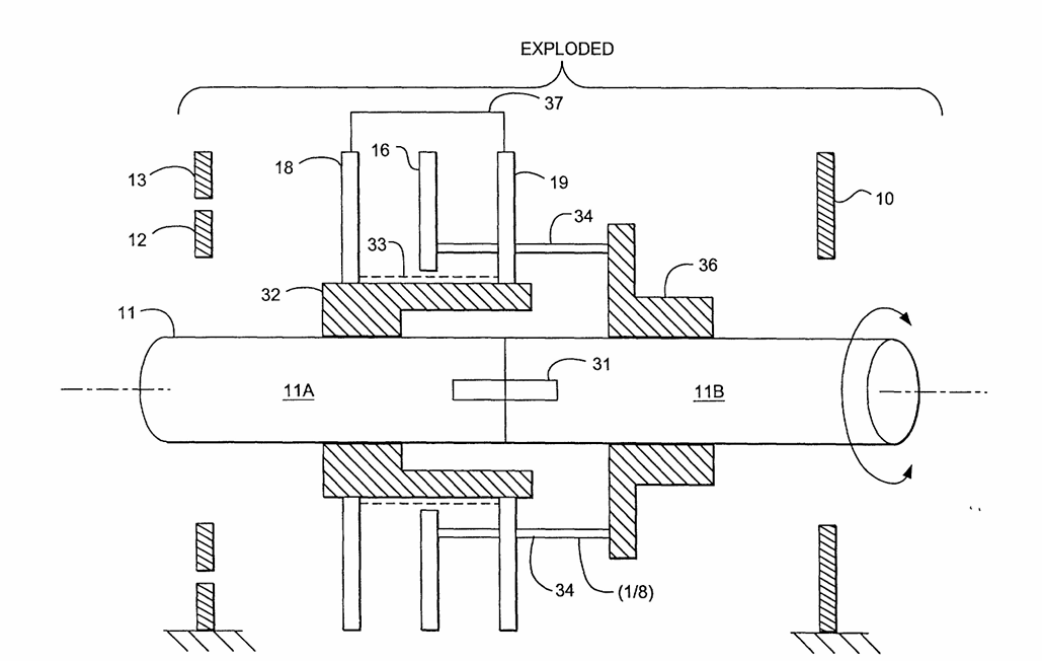


Figure 4: Cross-sectional schematic showing torque transfer (blue arrows) and capacitive coupling (red fields) between disks. The dielectric disk's rotation modulates the electric field between the metal disks.

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

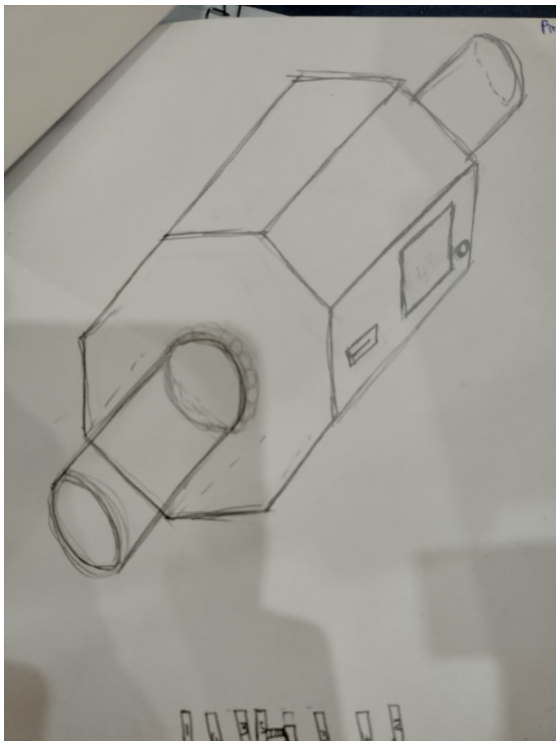
The product development followed a systematic approach with the following chronological stages:

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

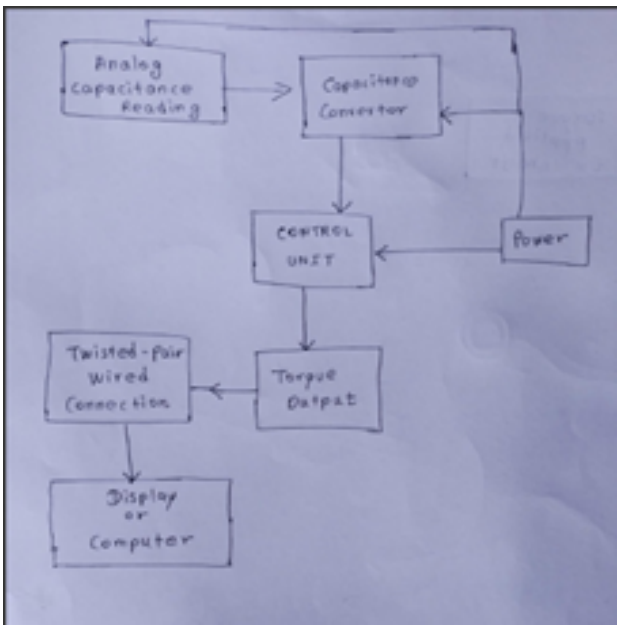
9. **PCB Design - (Week 12)**
    - Created 4-layer board layout in Altium Designer
    - Optimized for signal integrity and EMI
    - Incorporated capacitive sensing functionality
10. **Metal Cutting and Making Enclosures - (Week 13)**
    - Manufactured disks and mechanical components
    - Fabricated 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.

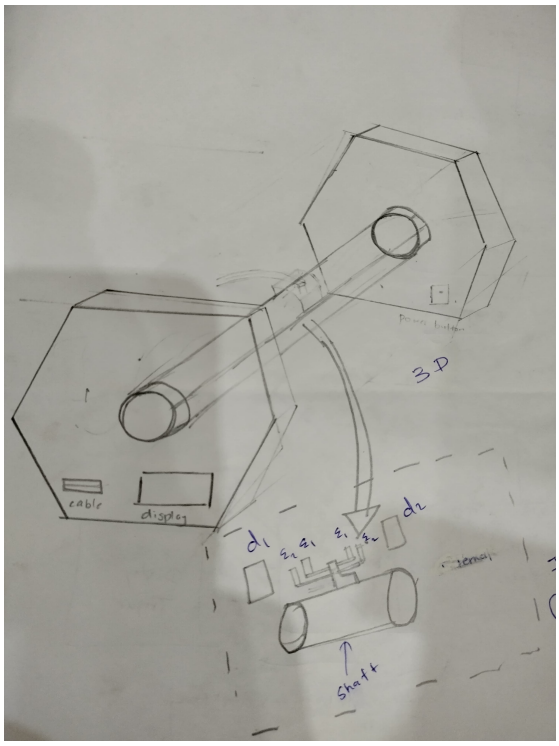
Shaft Design

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

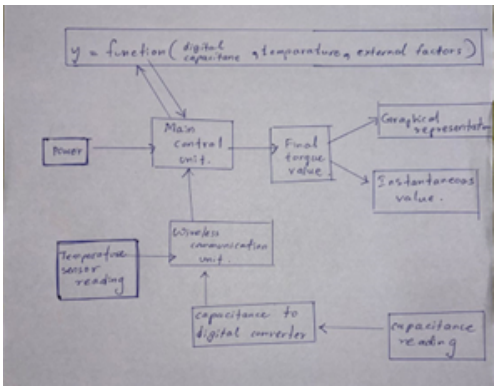
Block Diagram

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 6: 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.

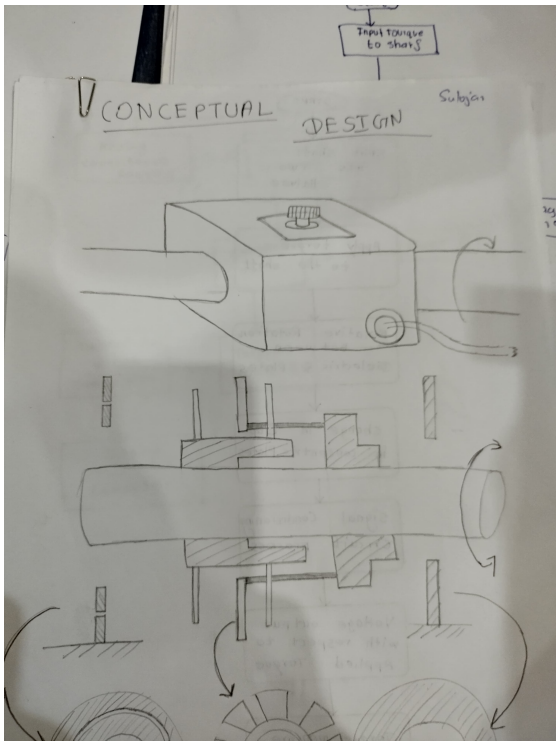
Shaft Design

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.

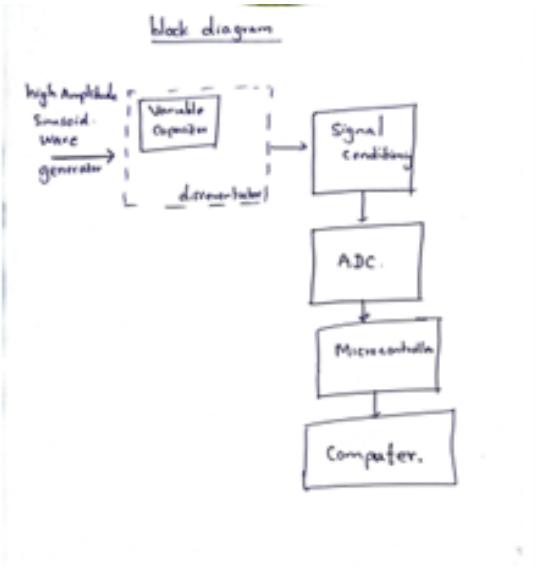
Block Diagram

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 7: 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.

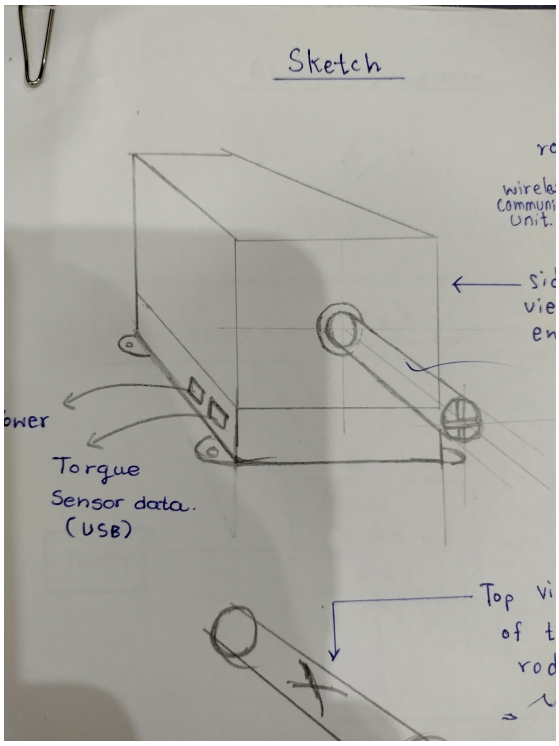
Shaft Design

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.

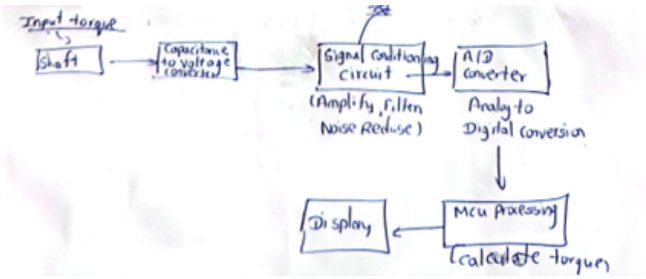
Block Diagram

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 8: 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.

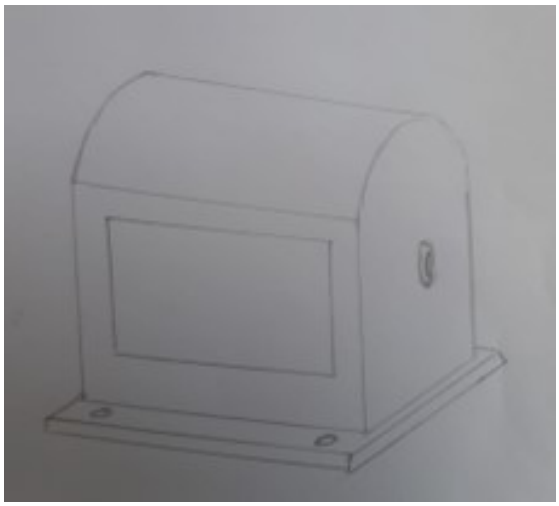
Shaft Design

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.

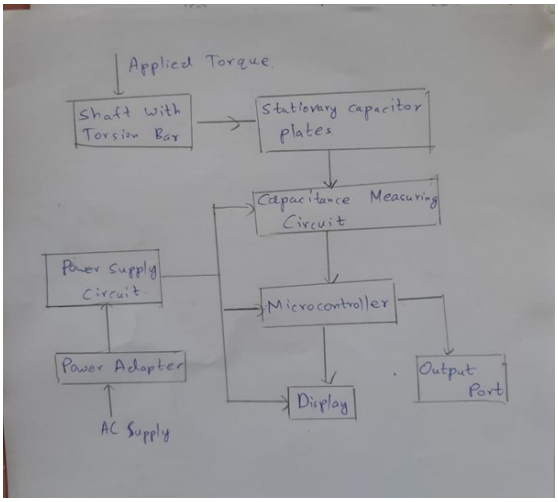
Block Diagram

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



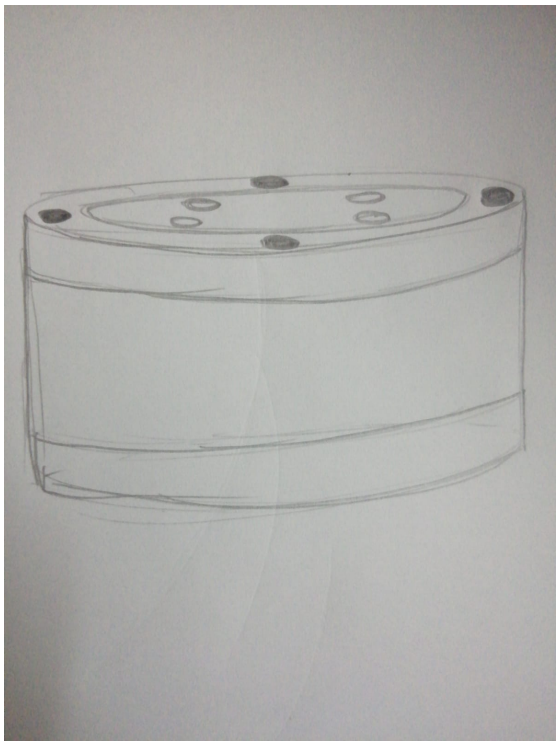
(a) Enclosure and Shaft Design



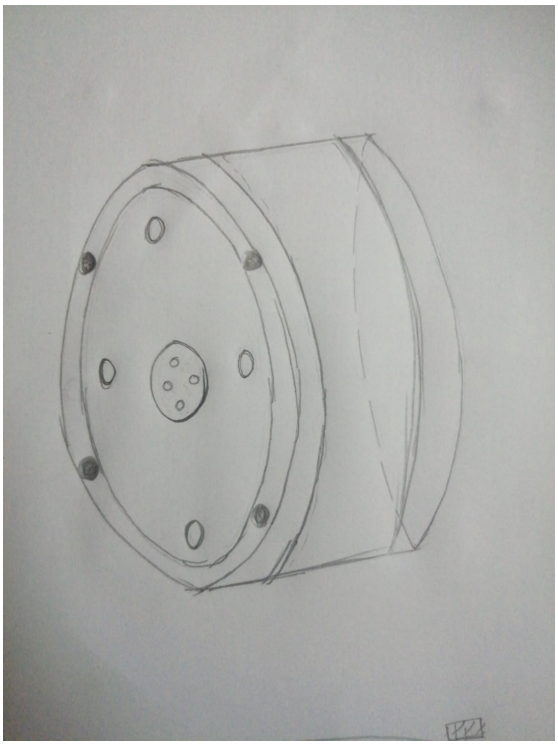
(b) Block Diagram

Figure 9: Final Selected Design

6.6 Selected Design Final



(a) View 01



(b) View 02

Figure 10: 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 2: Evaluation Table for Design Aspects

8 Capacitance and CDC Requirements Evaluations

8.1 Capacitance Calculation

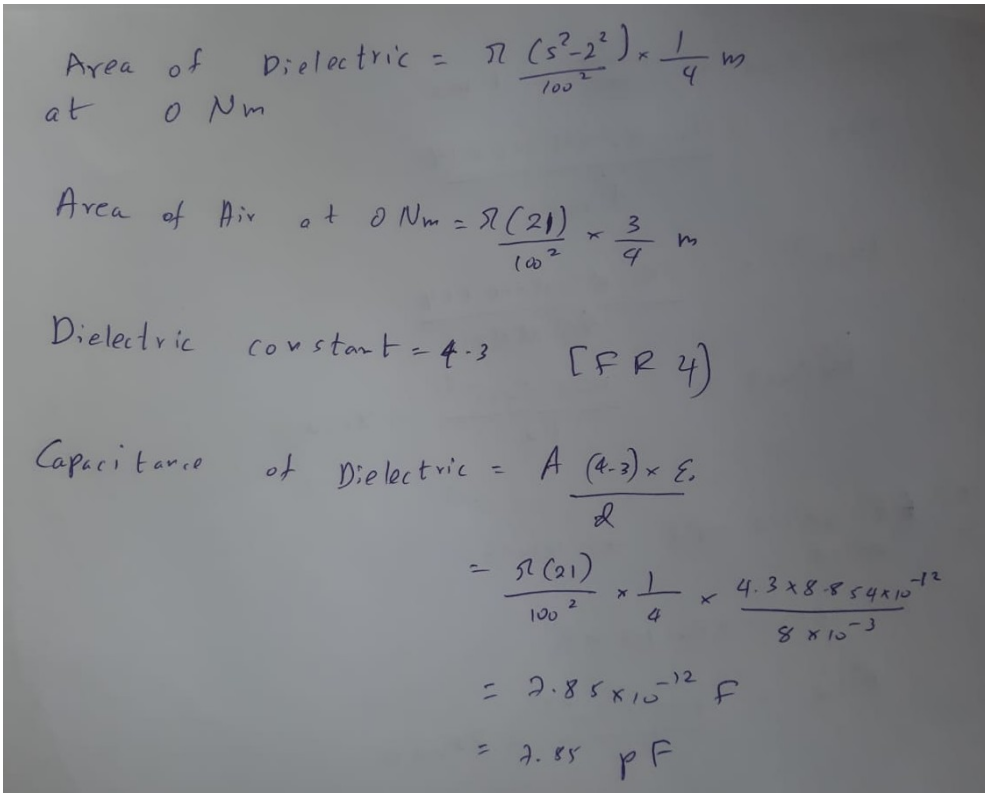


Figure 11: Area and Capacitance Calculation

8.2 Change of Capacitance

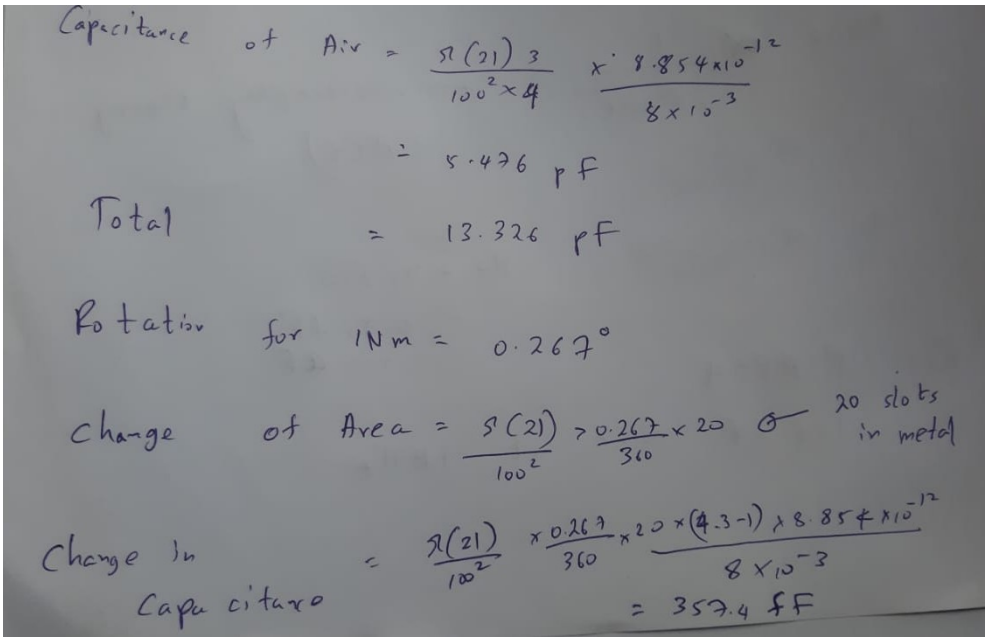


Figure 12: Capacitance calculation

8.3 CDC Calculations

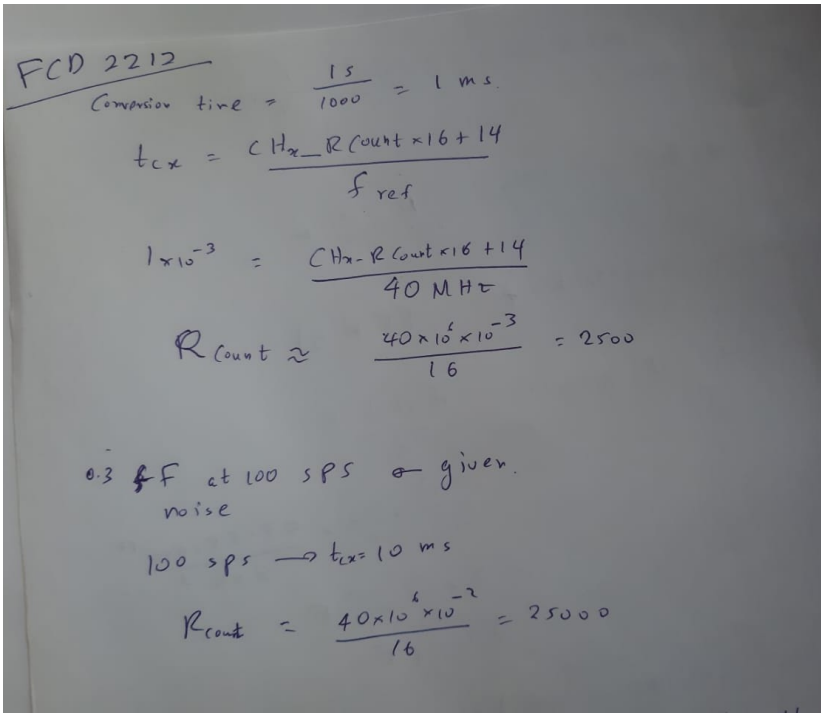


Figure 13: CDC calculation(1)

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 14: 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 15: PACP04 CDC

10 MCU Calculation

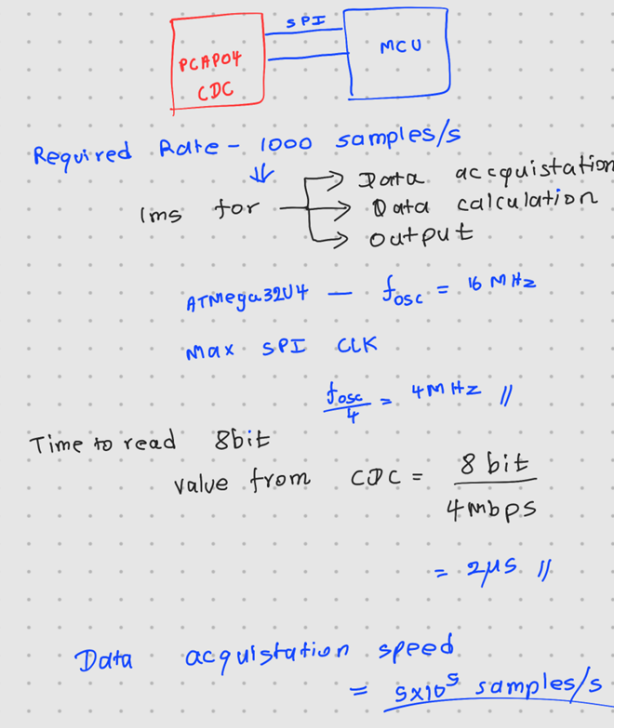


Figure 16: MCU calculation(1)

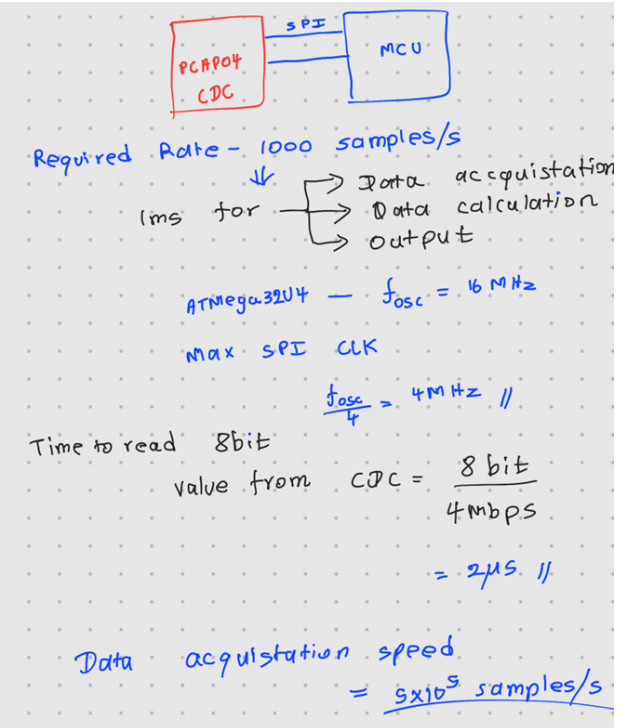


Figure 17: MCU calculation(2)

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

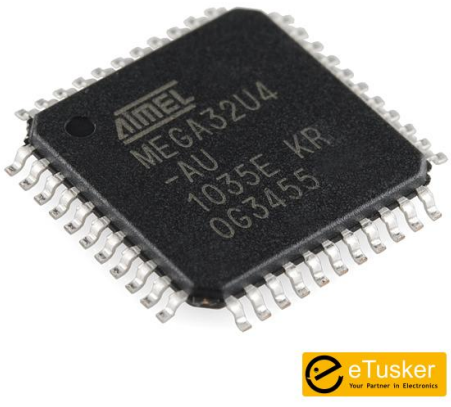


Figure 18: ATMega32U4

12 Dielectric Material Evaluation

Desired Specifications

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

	PTEF (Teflon)	Polyimide (Kapton)	Alumina Ceramic (Al2O3)	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 19: 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 20: 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

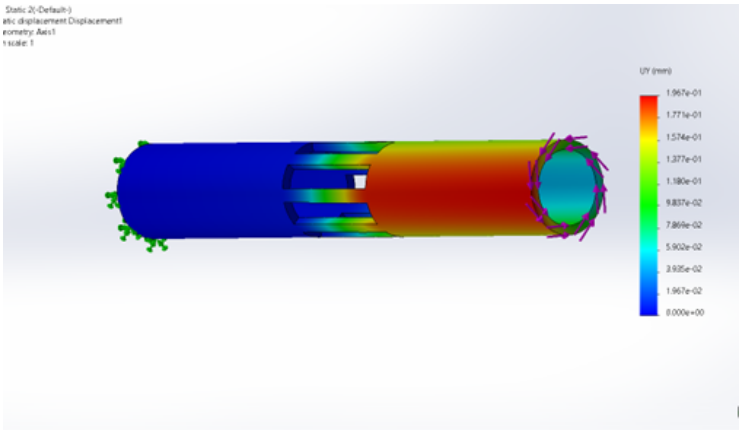


Figure 21: Short shaft – similar to the reference design

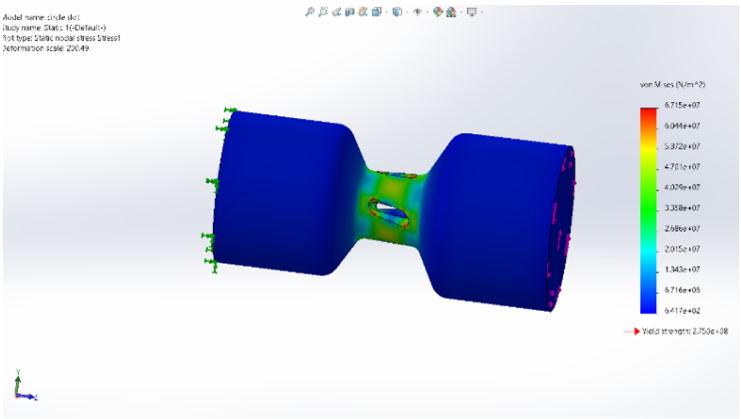


Figure 22: Shaft with pill-shaped cutouts

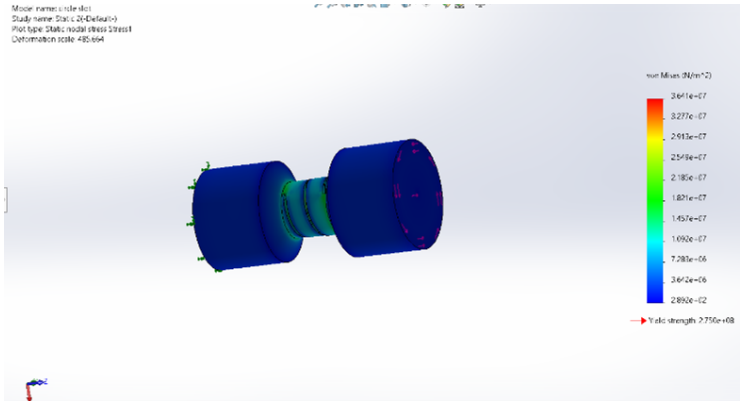


Figure 23: Shaft with helical groove

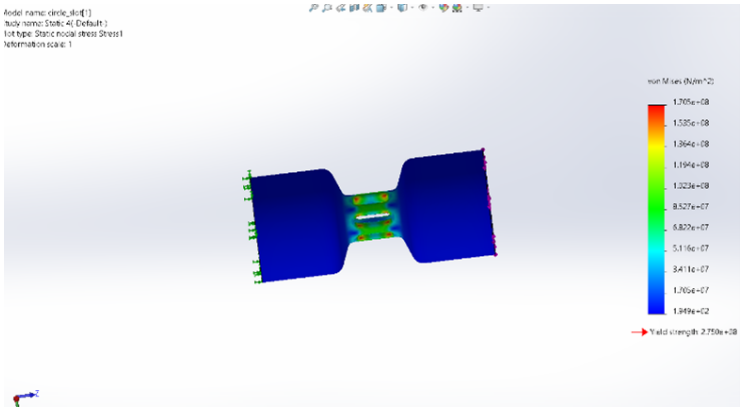


Figure 24: Pill-shaped cutouts + fillets

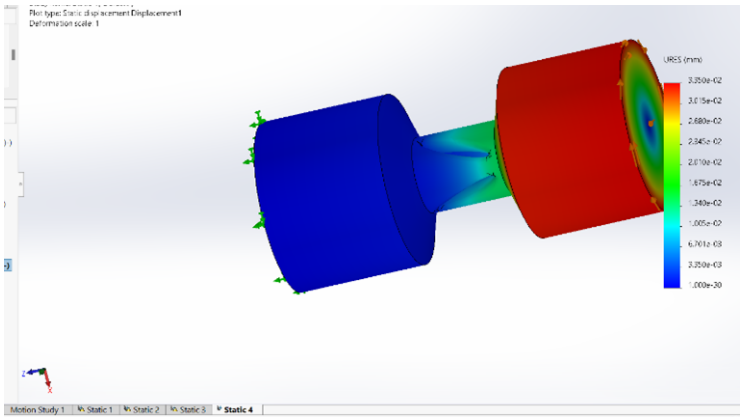


Figure 25: Angled pill-shaped cutouts + fillets

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

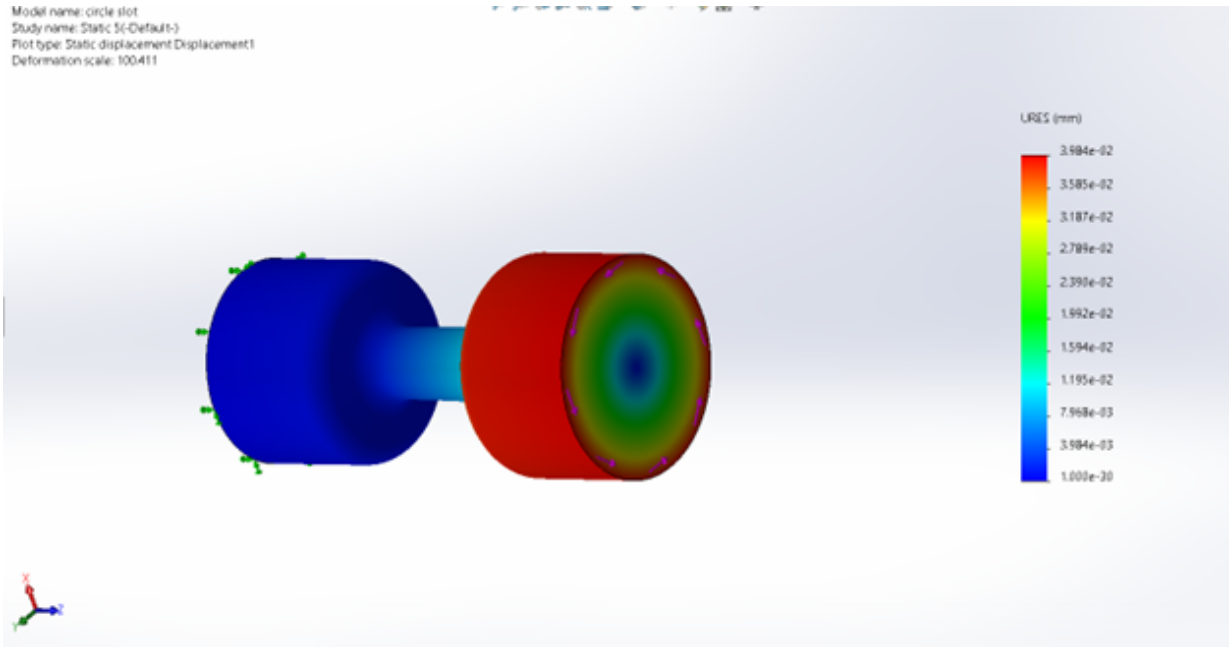


Figure 26: Final Shaft

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 Solidworks 3D Design

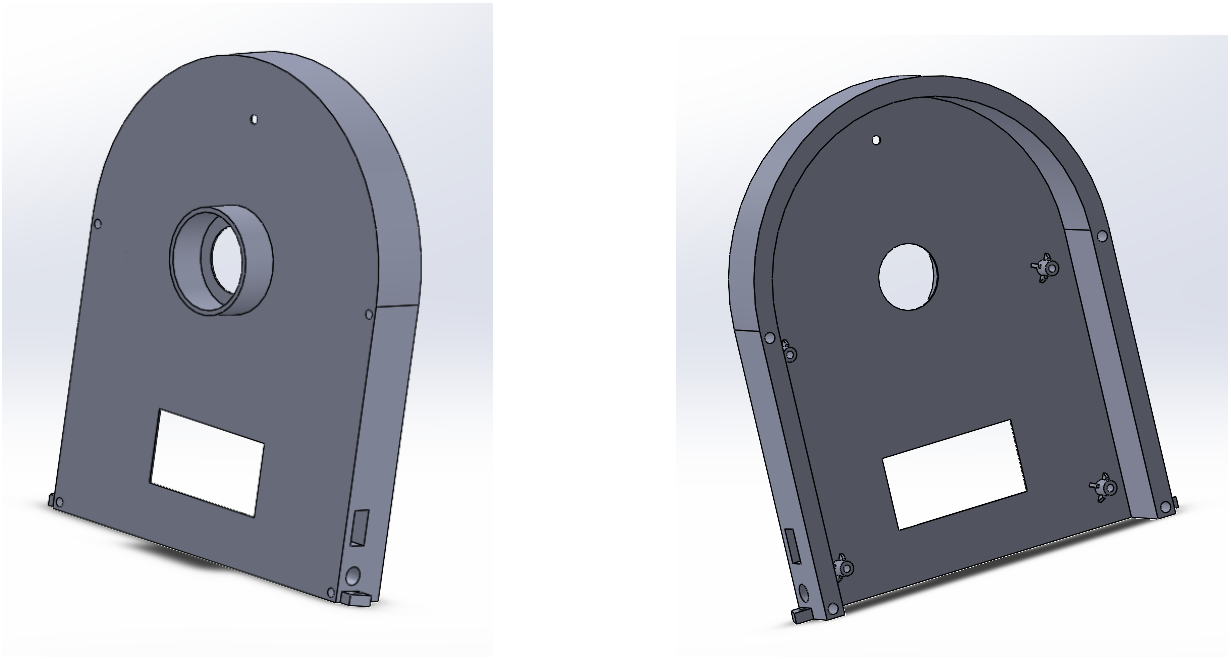
14.1 Initial Design - Version 01

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

14.1.1 Outer Enclosure

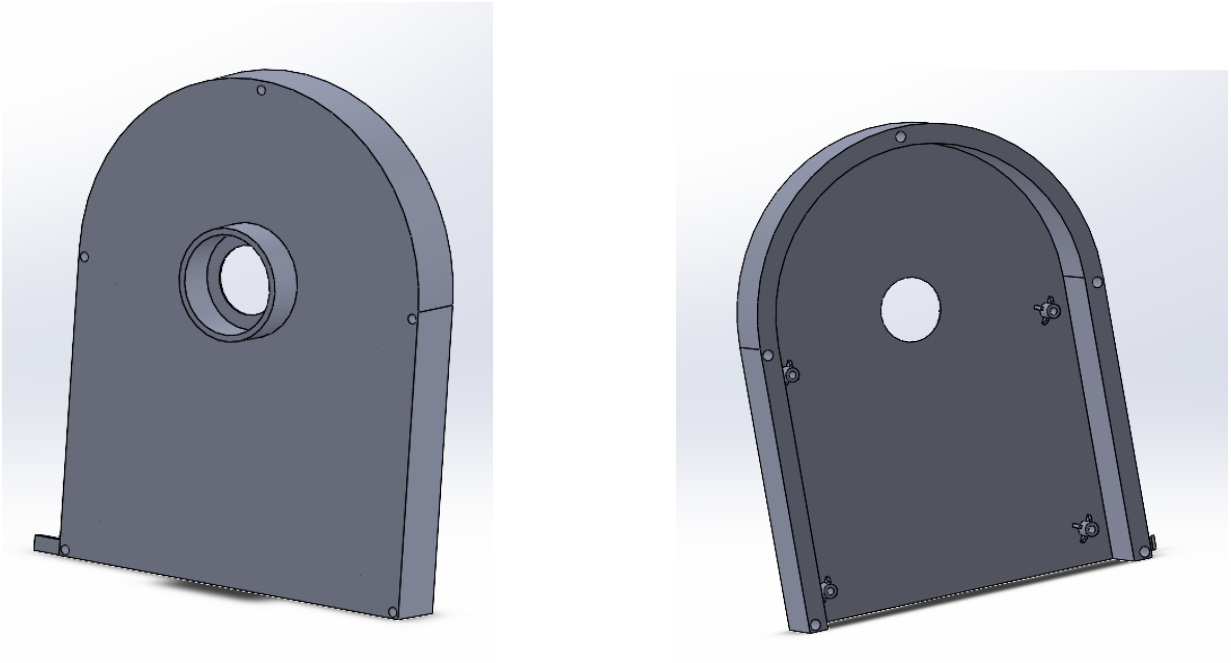
Frontside view



(a) Outer view (b) Inner view

Figure 27: Front Half

Backside view



(a) Outer view

(b) Inner view

Figure 28: Backside Half

Bottom Part

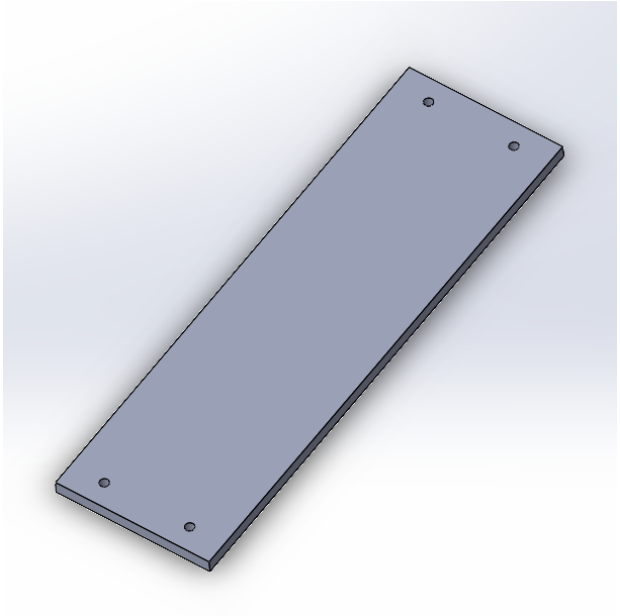


Figure 29: Bottom Part

14.1.2 Inner Parts

Disks

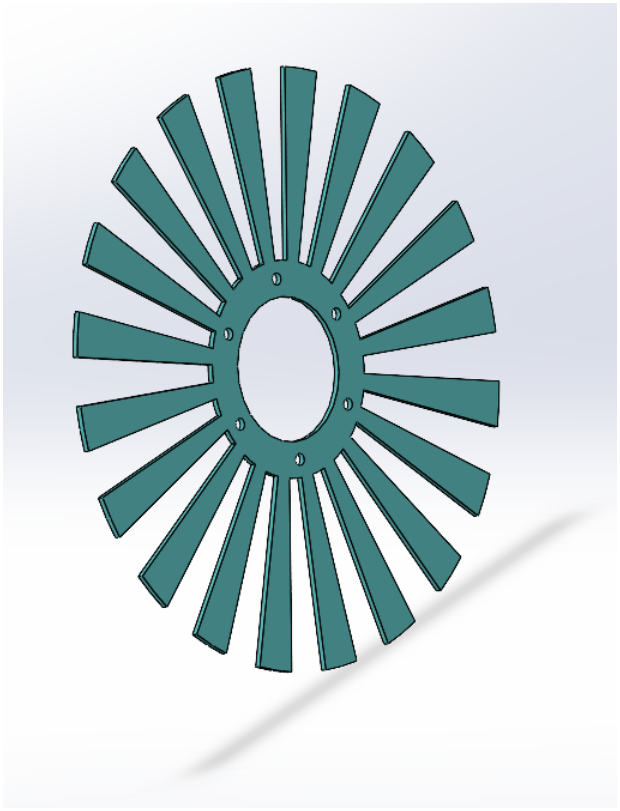


Figure 30: Dielectric Disk

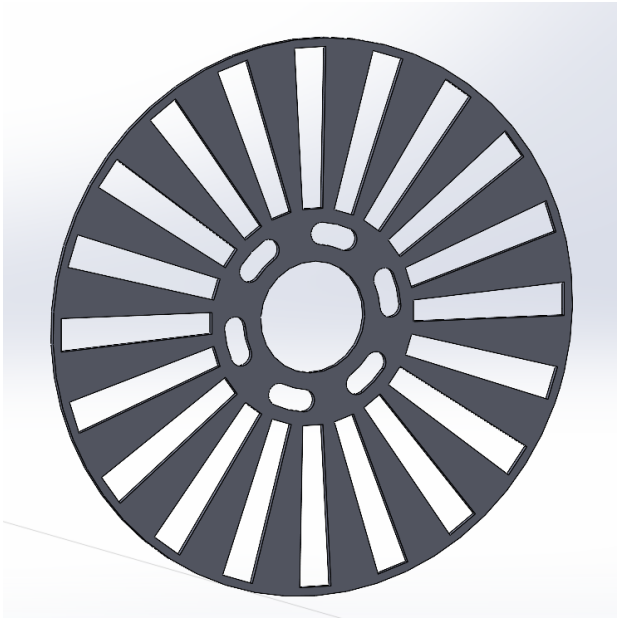
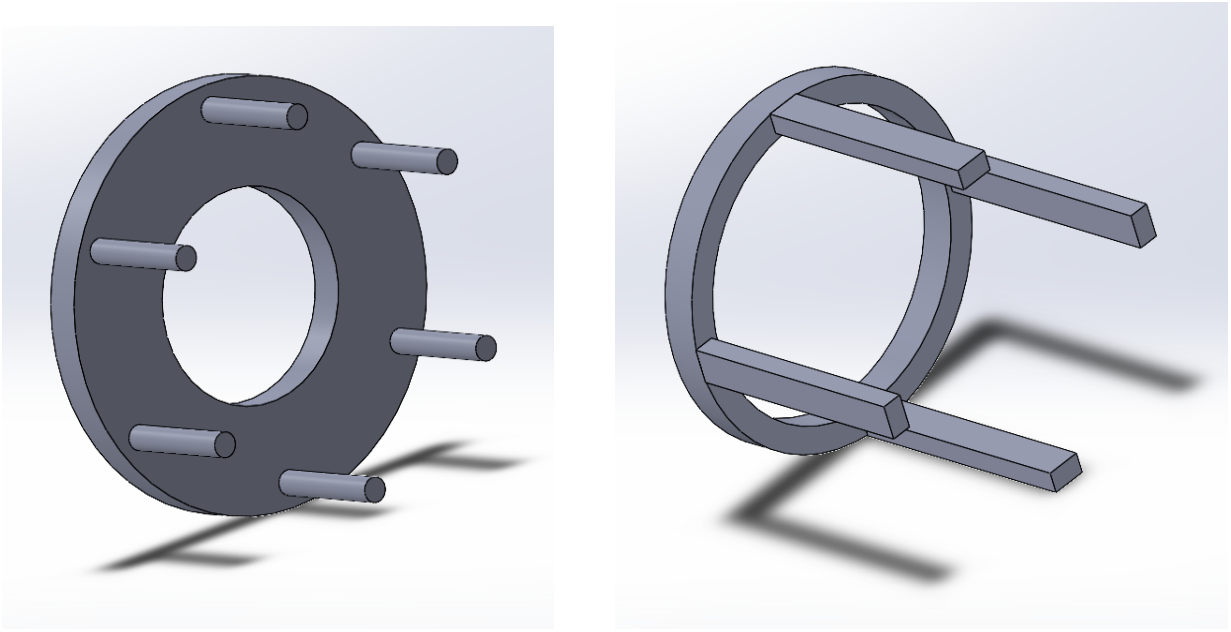


Figure 31: Metal Disk

Connecting Parts



(a) Dielectric Holder (b) Metal Plate Holder

Figure 32: Connecting Holders

Shaft

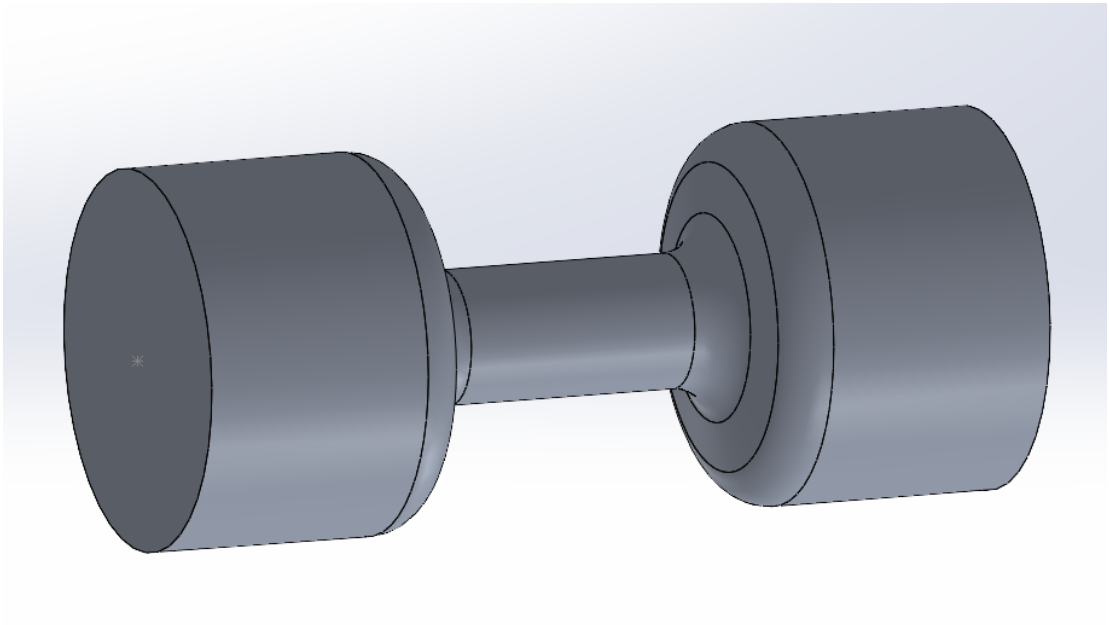


Figure 33: Shaft

14.1.3 Final Assembly

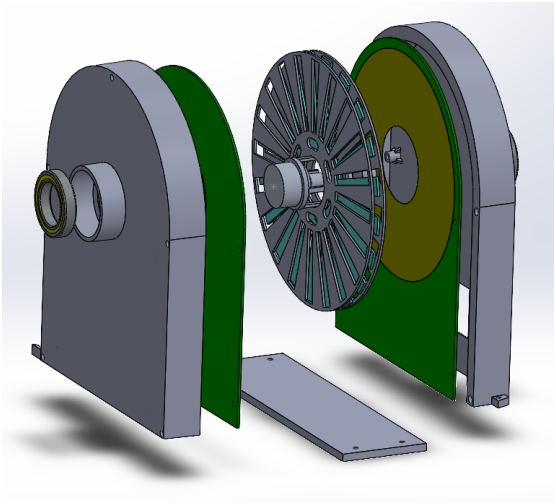


Figure 34: Backside View

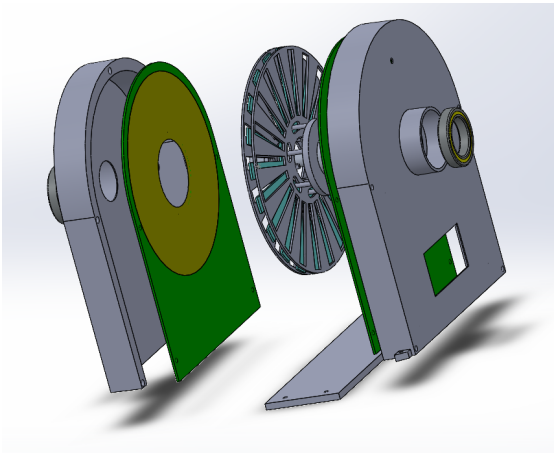


Figure 35: Frontside View

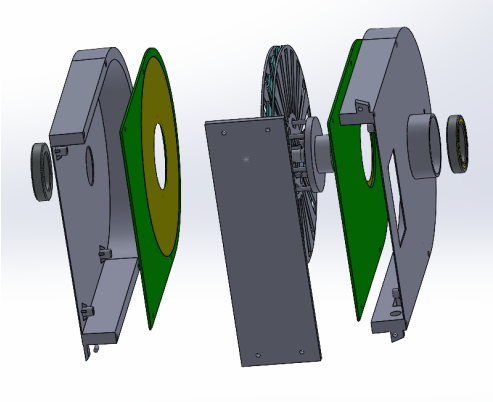


Figure 36: Bottom view

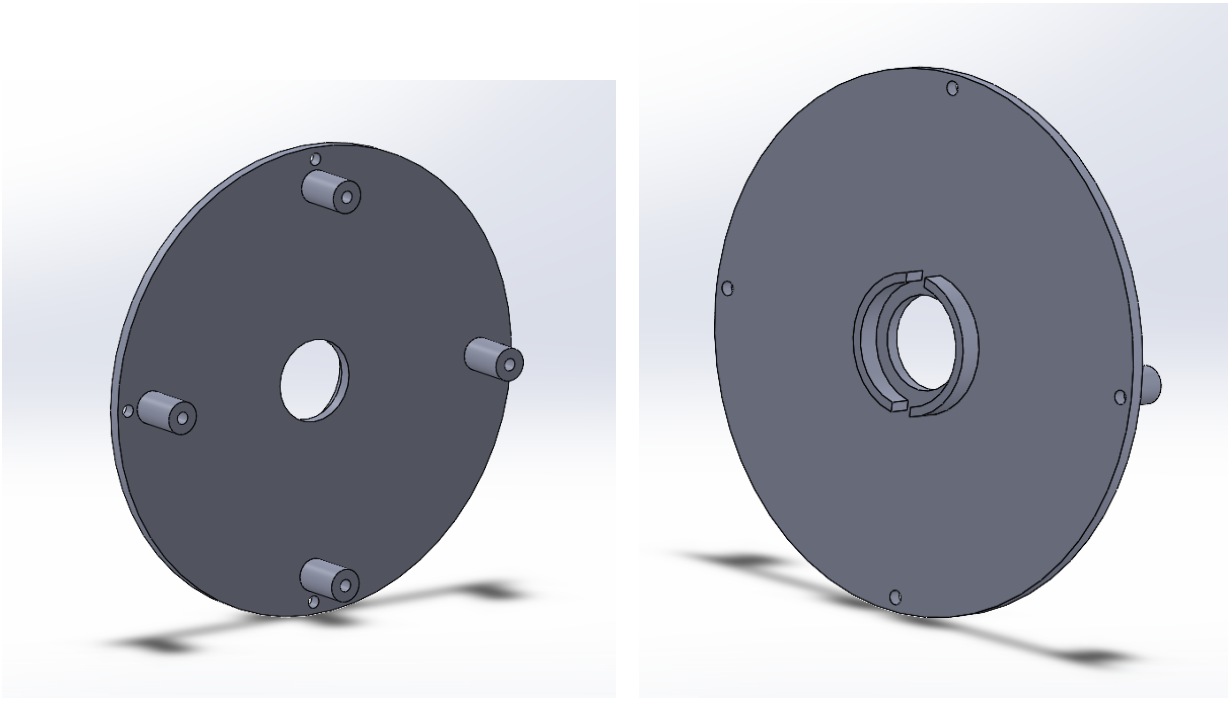
14.2 Final Design - Version 02

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

14.2.1 Outer Enclosure

Front and back half



(a) Frontside (b) Backside

Figure 37: Enclosure caps

Cylindrical Body

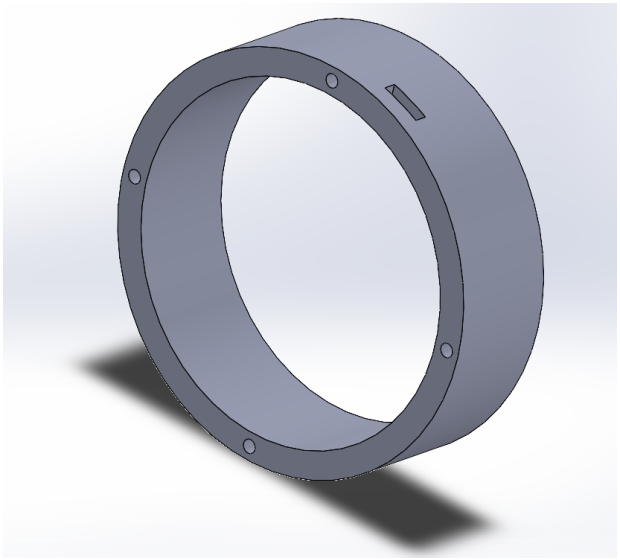


Figure 38: Cylinder Enclosure

14.2.2 Shaft

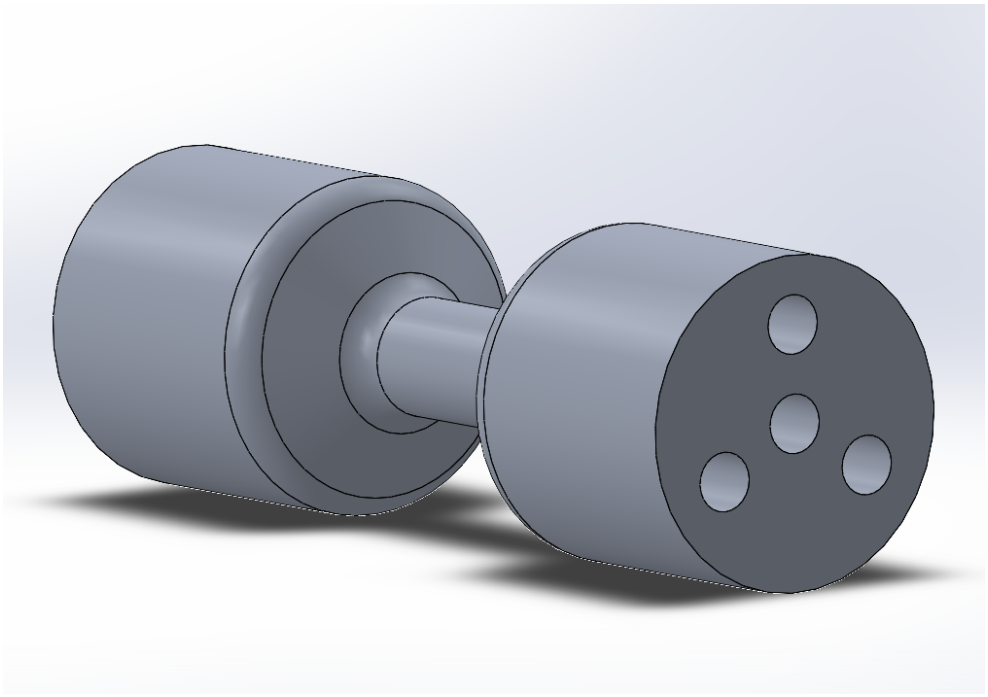
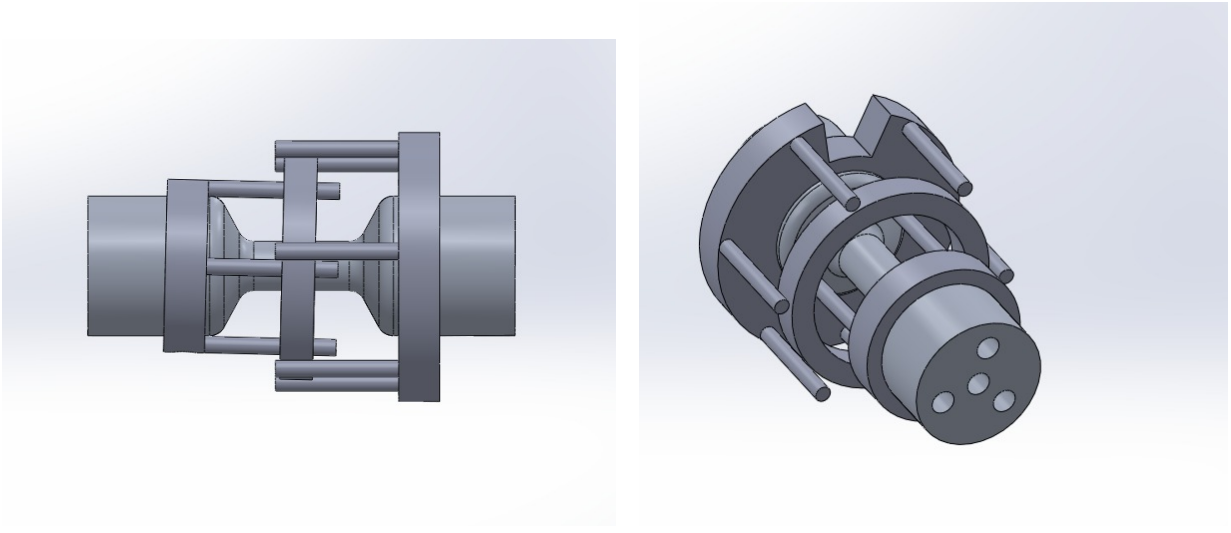


Figure 39: Final Shaft

14.2.3 Shaft and holders with modified Dimensions



(a) Side View (b) Angle view

Figure 40: Internal assembly of Shaft and Holders

14.2.4 Full assembly

Full Product Overview

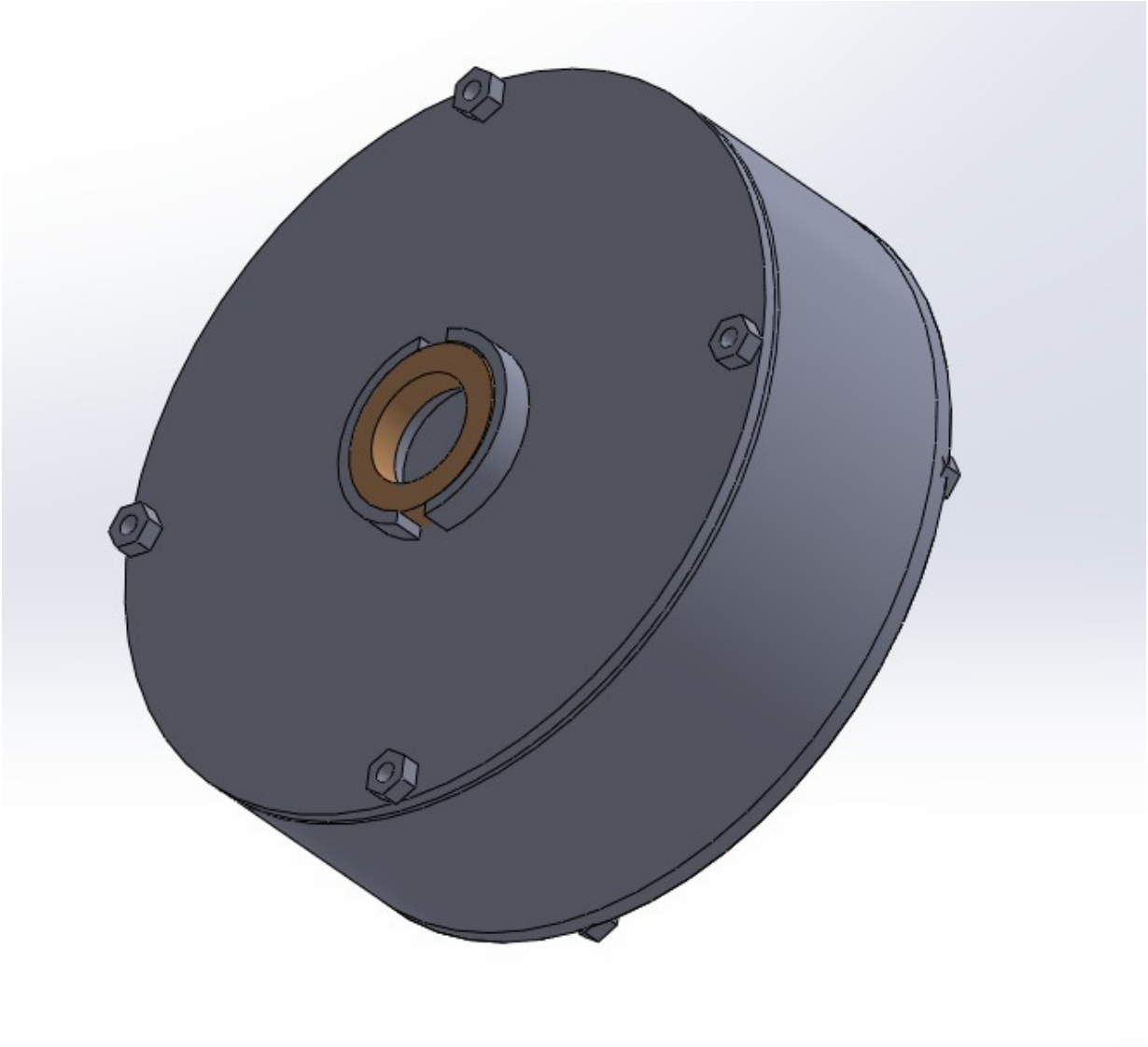


Figure 41: Full Closed view

Exploded view with internal positions

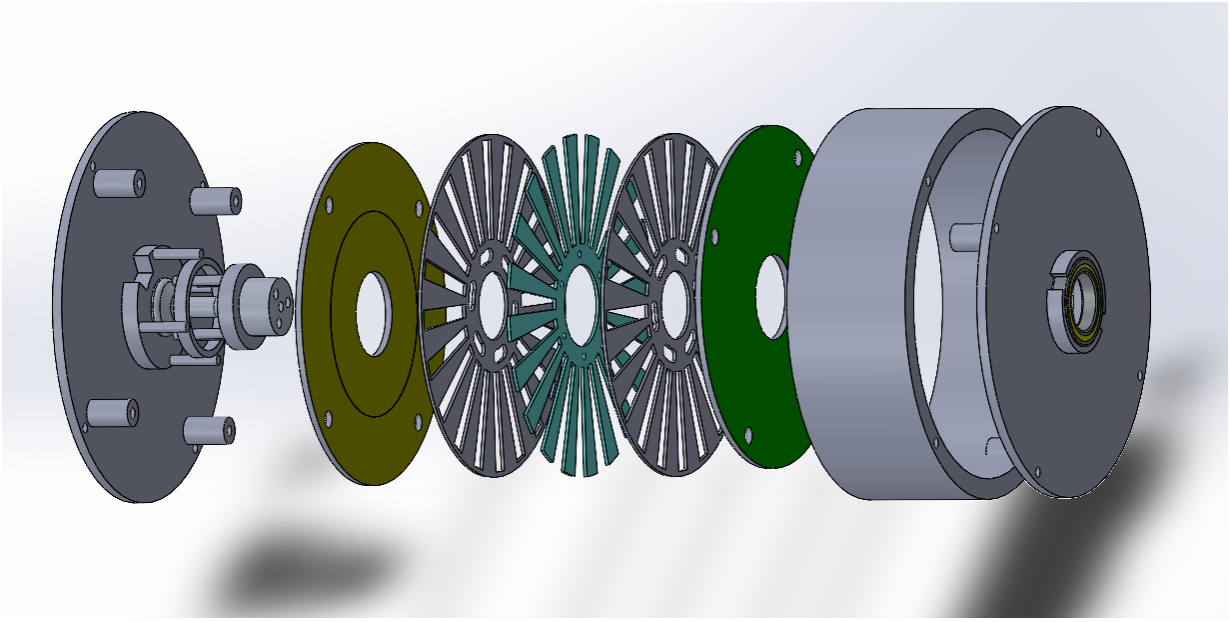


Figure 42: Exploded view 01

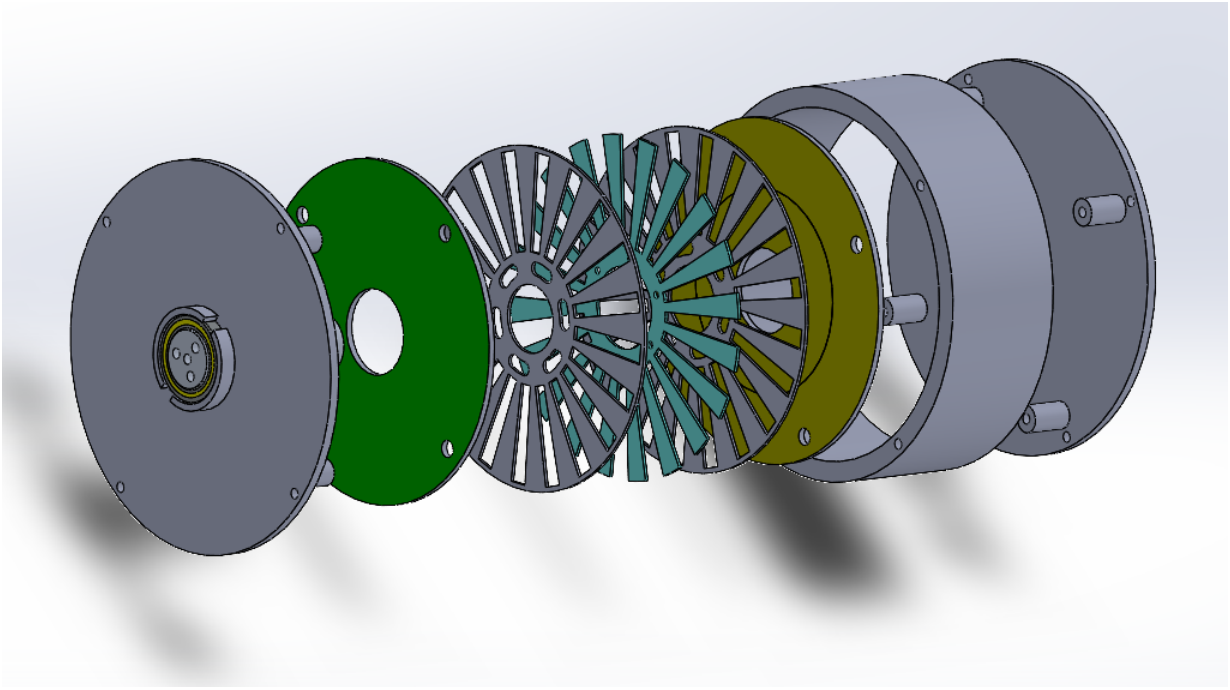
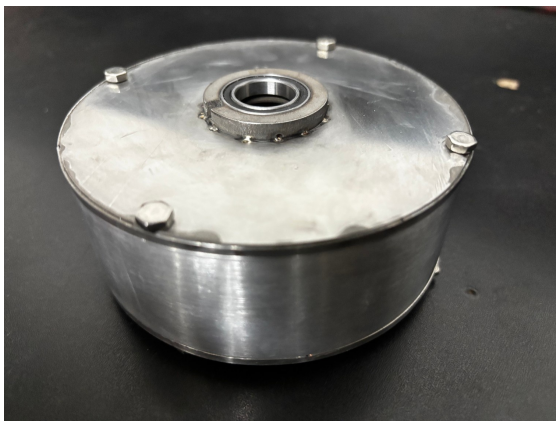
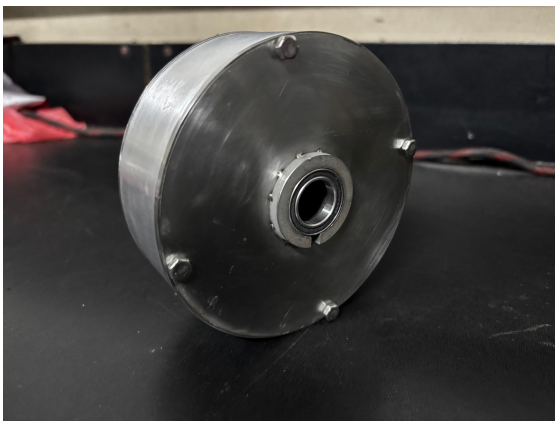


Figure 43: Exploded view 02

15 Actual Product (Physical)

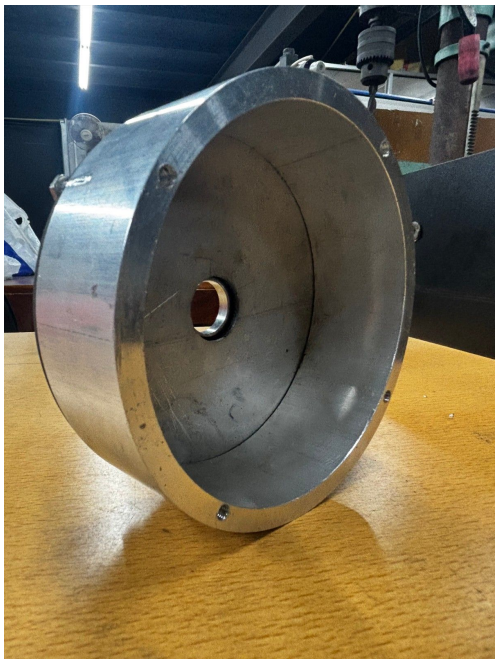


(a) Side View

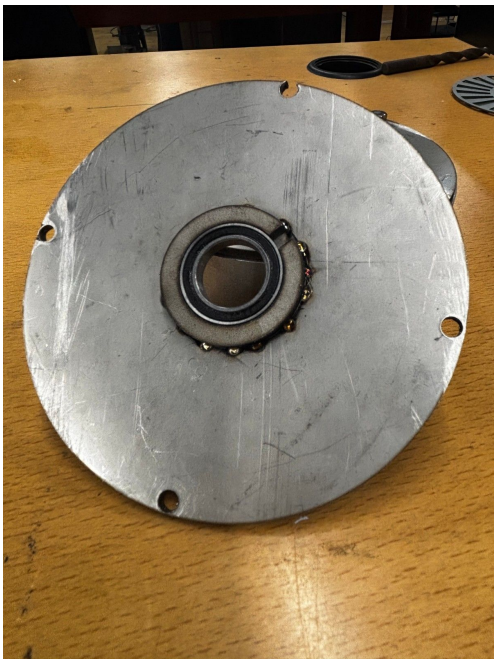


(b) Angle view

Figure 44: Actual Product Outer look



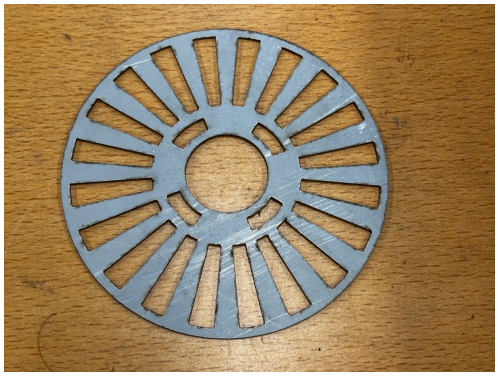
(a) Side View



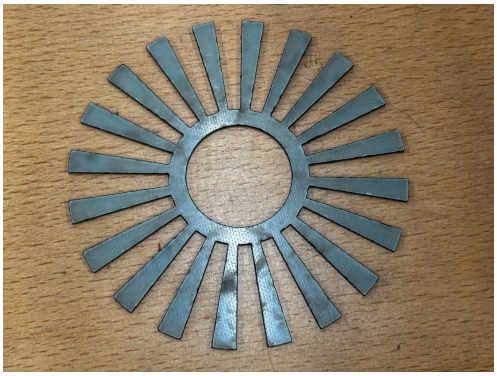
(b) Front cap with Bearings

Figure 45: Enclosure Outer look

Figure 46: Dielectric and Metal holders



(a) Side View



(b) Front cap with Bearings

Figure 47: Inner Disks

(a) Side View

(b) Front cap with Bearings

Figure 48: Shaft

16 Schematic Circuit Design

16.1 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)

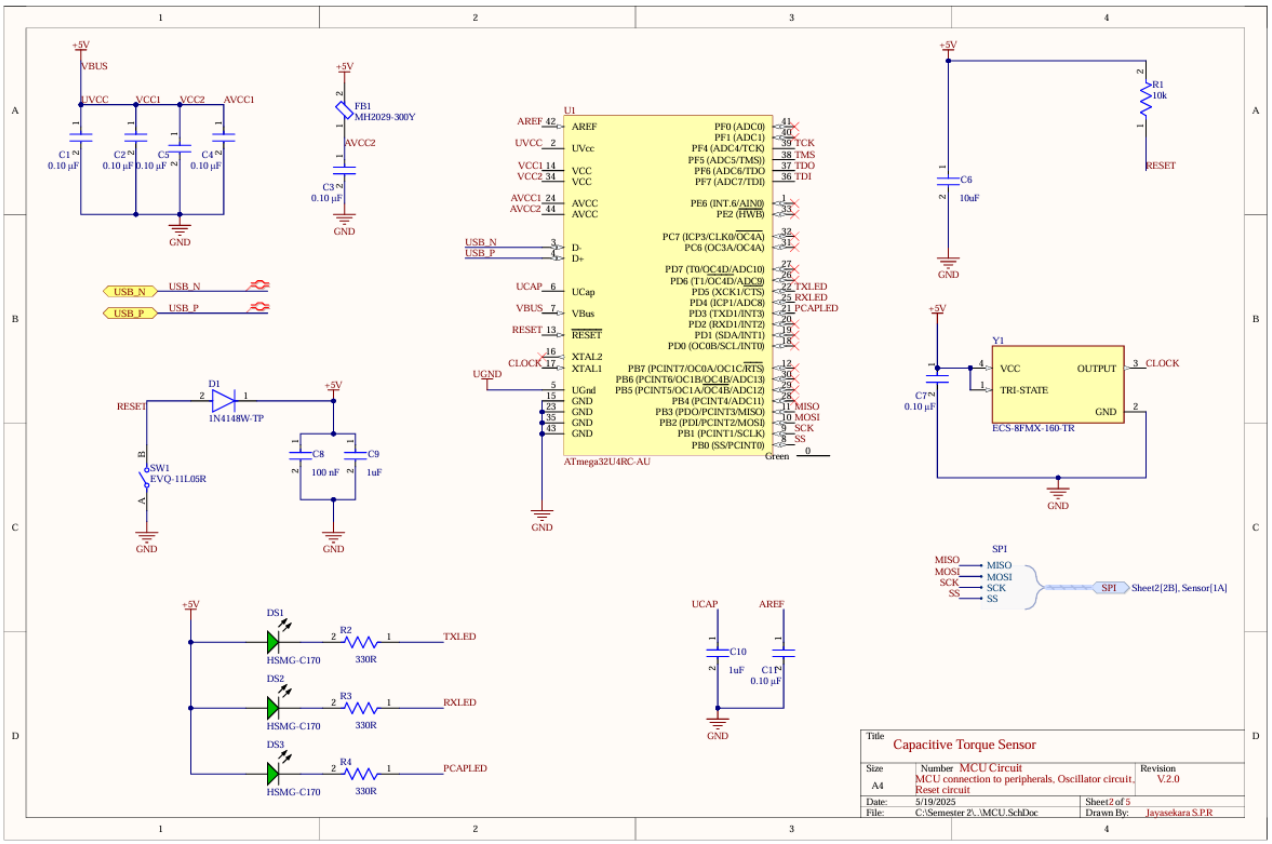


Figure 49: MCU Schematic

16.2 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

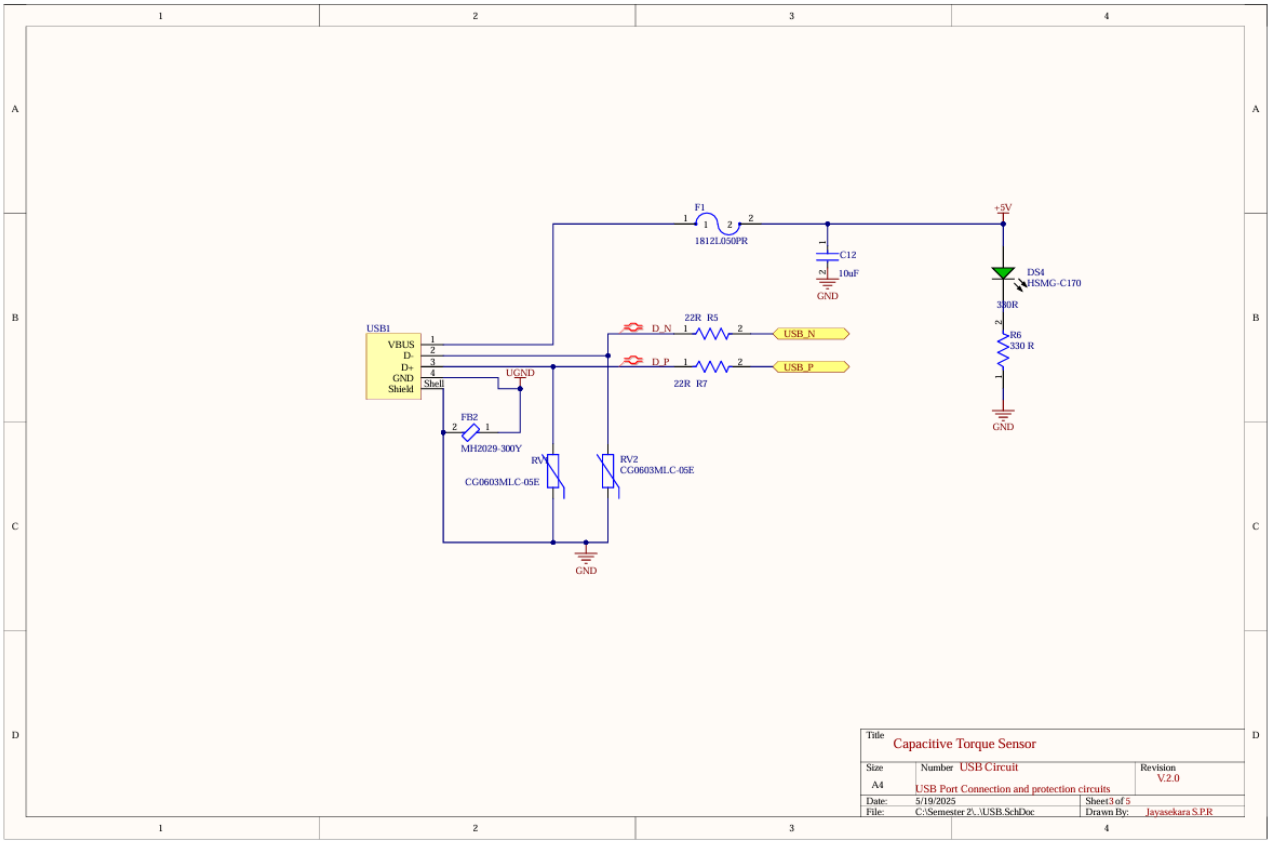


Figure 50: USB Schematic

16.3 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

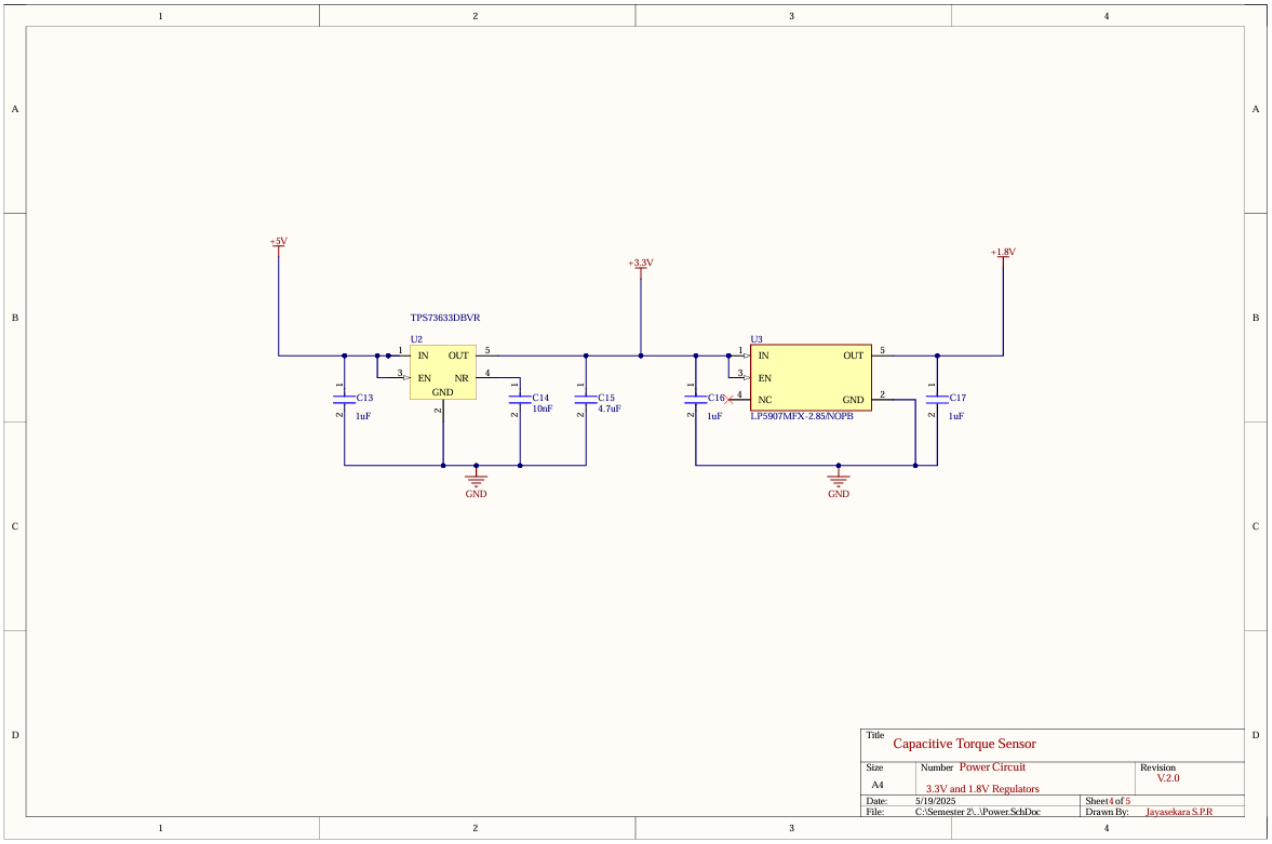


Figure 51: Power Supply Schematic

16.4 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)

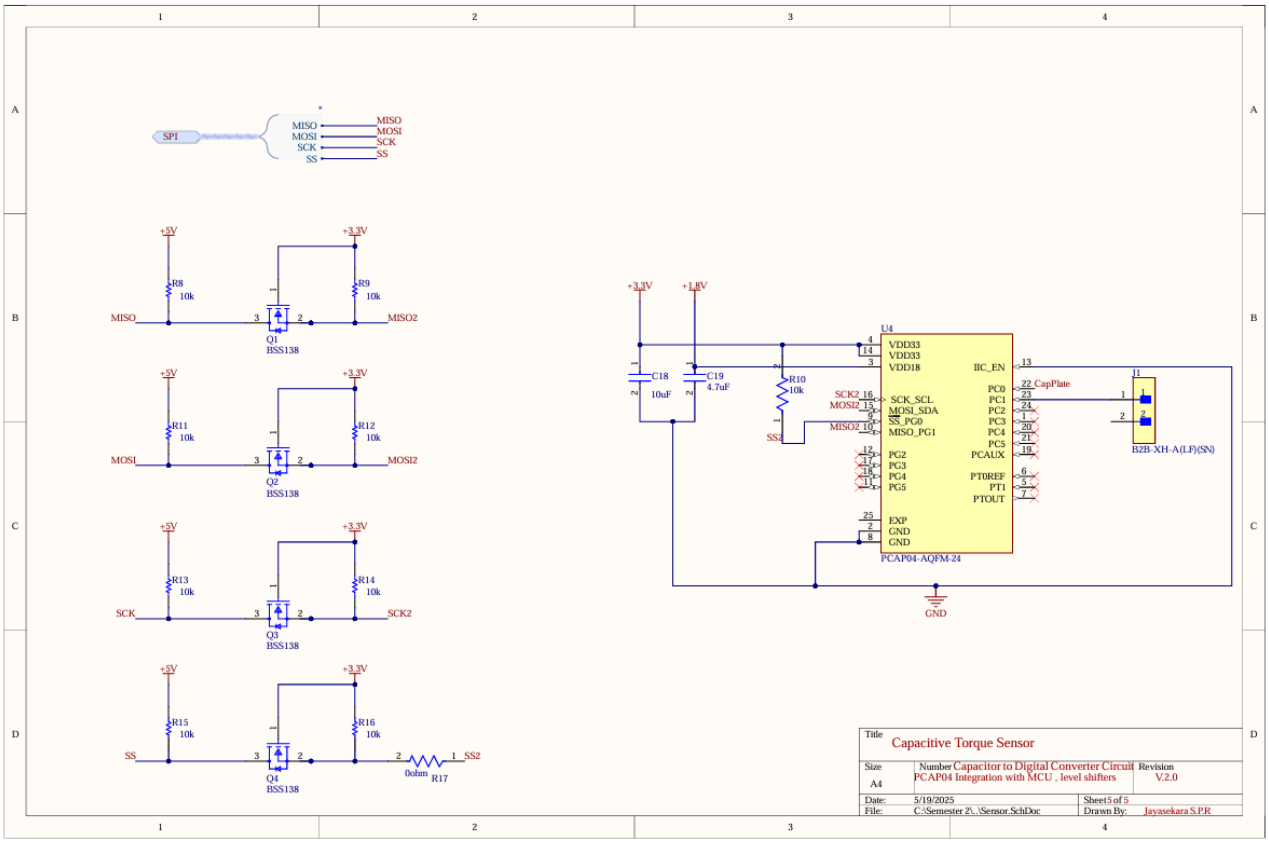


Figure 52: CDC Sensor Schematic

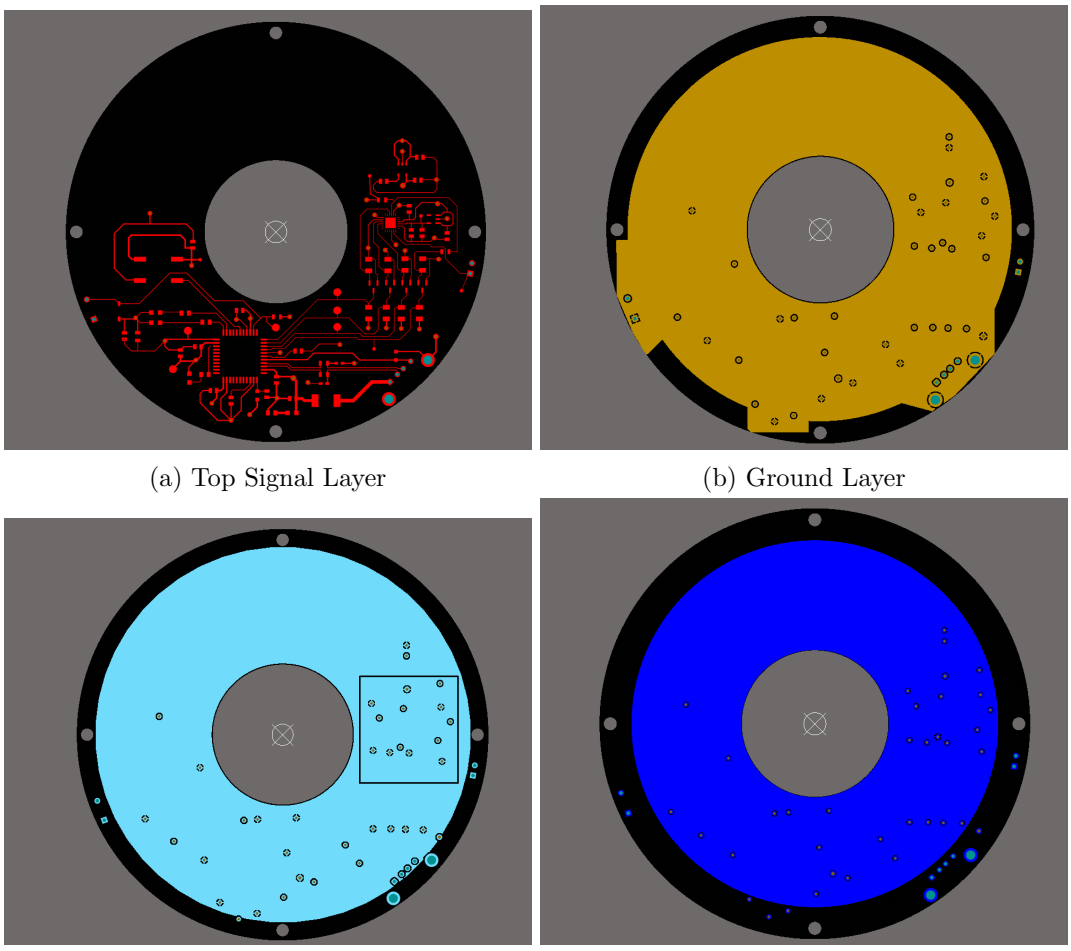
17 PCB Design

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

17.1 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 53: PCB Layers view

17.2 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

17.3 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

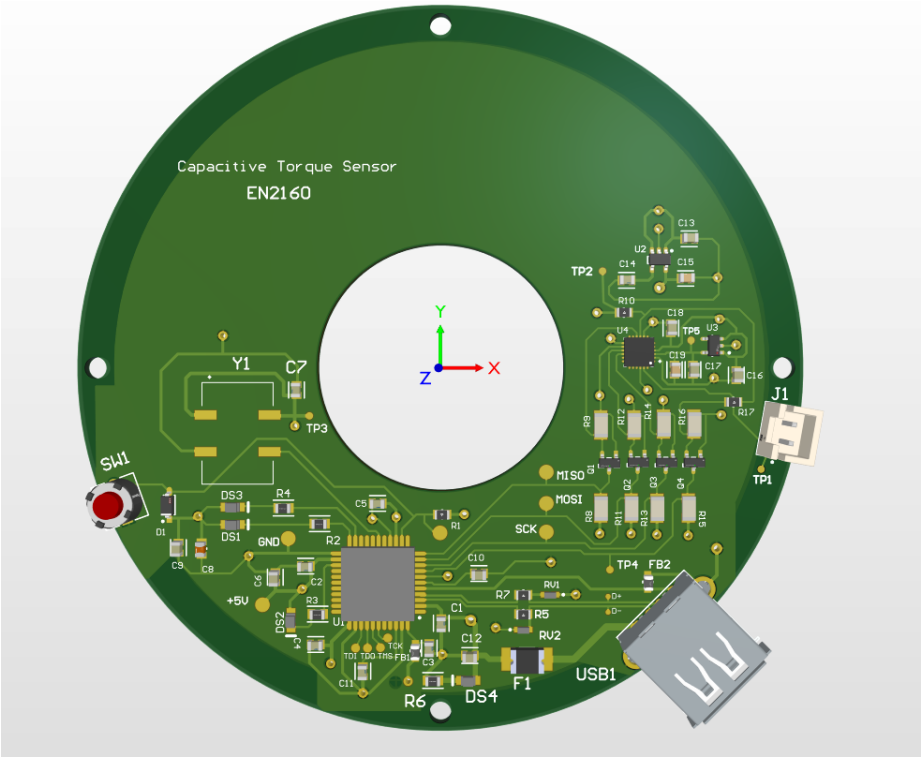


Figure 54: Front View

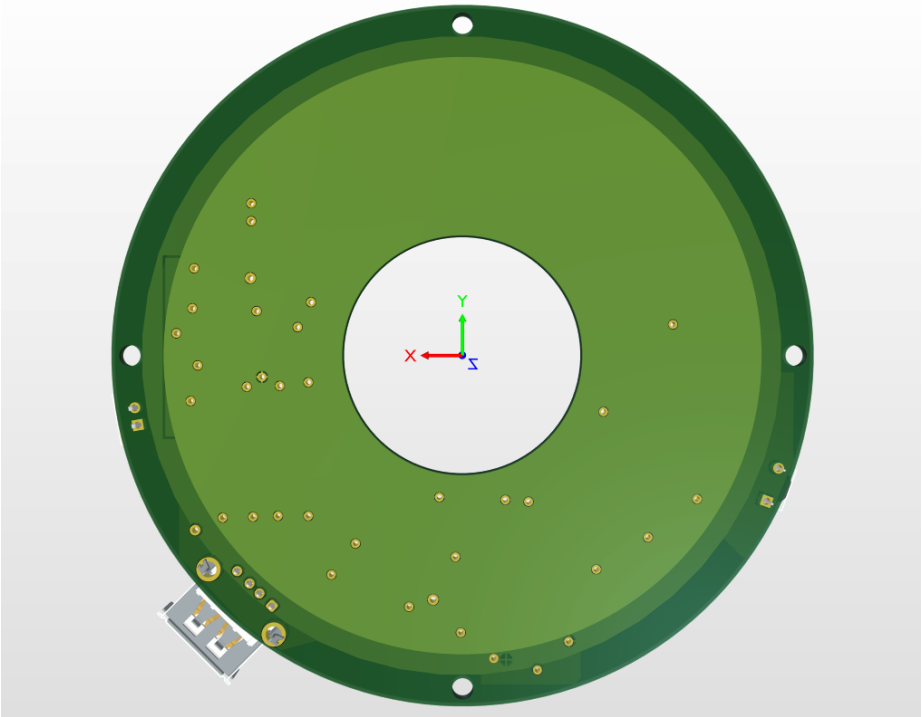


Figure 55: Backside View

18 Testing Procedure

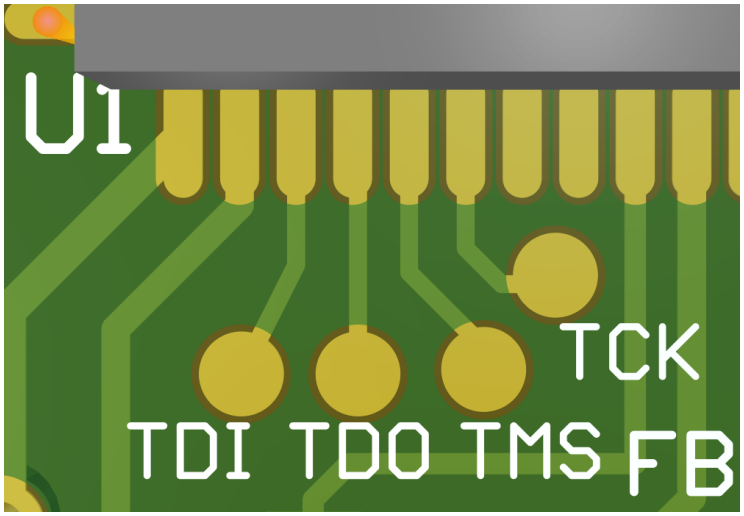


Figure 56: Test Points Implementation on PCB

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:

18.1 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

18.2 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.

19 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 57: Complete Bill of Materials for the Capacitive Torque Sensor

The complete Bill of Materials shown in Figure 57 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.

20 RTL Code for PCB (ATmega32U4)

20.1 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

20.2 Code - Iteration 01

20.2.1 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;
}
```

```
37
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); //  CFG9 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) {
90      float ratio = (float)raw / (1UL << 27); // 5.27 fixed-point
91      return ratio * Cref;
92  }
93
94  float calculateTorque(float x) {
95      return A3 * pow(x, 3) + A2 * pow(x, 2) + A1 * x + A0;
96  }
97
98  int main(void)
99  {
100     SPI_init();
101     m_usb_init();
102     _delay_ms(10);
103     PORTB |= (1 << PB0); // CS HIGH idle
104
105     PCAP04_init();
106     _delay_ms(100);
107
108     while (!m_usb_isconnected()) { }
109
110     while (1) {
111         uint32_t raw = PCAP04_read_result();
112         float cap_pf = PCAP04_to_pF(raw, 10.0f); // Assuming Cref = 10.0pF
113         float torque = calculateTorque(cap_pf);
114         m_usb_tx_long(torque);
115         m_usb_tx_push();       // Ensure transmission
116         _delay_ms(1);// 1 kHz loop        // Prevent buffer overflow
117     }
118 }
```

20.3 Code - Iteration 02

20.3.1 main.c

Listing 2: USB header file

```
1   #define F_CPU 16000000UL
2   #include <avr/io.h>
3   #include <util/delay.h>
4   #include <stdio.h>
5   #include "m_usb.h"
6
7   // SPI Pin Definitions for ATmega32U4
8   #define SPI_DDR  DDRB
9   #define SPI_PORT PORTB
10  #define MOSI     PB2
11  #define MISO     PB3
12  #define SCK      PB1
13  #define SS       PB0
14
15  // PCAP04 SPI Commands
16  #define OPCODE_WRITE_CFG    0xA3
17  #define OPCODE_READ_CFG     0x23
```

```
#define OPCODE_READ_RES     0x40
#define OPCODE_POR          0x88
#define OPCODE_INIT         0x8A
#define OPCODE_CDC_START    0x8C
#define OPCODE_TEST_READ    0x7E
#define WR_MEM              0xA0
#define RD_MEM              0x20

uint8_t standard_fw[548] = {
    // Standard firmware data
    // ... (truncated for brevity)
};

void SPI_Init(void) {
    SPI_DDR |= (1 << MOSI) | (1 << SCK) | (1 << SS);
    SPI_DDR &= ~(1 << MISO);
    SPI_PORT |= (1 << SS);
    SPCR = (1 << SPE) | (1 << MSTR) | (1 << CPHA) | (1 << SPR0);
}

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

void PCAP04_WriteRegister(uint8_t address, uint8_t data) {
    SPI_PORT &= ~(1 << SS);
    _delay_ms(1);
    SPI_Transfer(OPCODE_WRITE_CFG);
    SPI_Transfer(address | 0xC0);
    SPI_Transfer(data);
    _delay_ms(1);
    SPI_PORT |= (1 << SS);
    _delay_ms(1);
}

uint8_t PCAP04_ReadRegister(uint8_t address) {
    uint8_t val;
    SPI_PORT &= ~(1 << SS);
    _delay_ms(1);
    SPI_Transfer(OPCODE_READ_CFG);
    SPI_Transfer(address | 0xC0);
    val = SPI_Transfer(0x00);
    _delay_ms(1);
    SPI_PORT |= (1 << SS);
    _delay_ms(1);
    return val;
}

void Write_Byte_Auto_Incr(uint8_t opcode, uint8_t address, const uint8_t*
   byte_array, int length) {
    SPI_PORT &= ~(1 << SS);
    _delay_ms(1);
```

```
71      SPI_Transfer(opcode);
72      SPI_Transfer(address);
73      for (int i = 0; i < length; i++) {
74          SPI_Transfer(byte_array[i]);
75          _delay_ms(5);
76      }
77      _delay_ms(10);
78      SPI_PORT |= (1 << SS);
79      _delay_ms(1);
80  }
81
82  void Read_Byte_Auto_Incr(uint8_t opcode, uint8_t address, int length) {
83      uint8_t val=0;
84      SPI_PORT &= ~(1 << SS);
85      _delay_ms(1);
86      SPI_Transfer(opcode);
87      SPI_Transfer(address);
88      for (int i = 0; i < length; i++) {
89          val=SPI_Transfer(0x00);
90          m_usb_tx_uint(val);
91          m_usb_tx_push();
92          _delay_ms(10);
93      }
94      SPI_PORT |= (1 << SS);
95      _delay_ms(1);
96  }
97
98  void PCAP04_CDC_START(void) {
99      SPI_PORT &= ~(1 << SS);
100     _delay_ms(1);
101     SPI_Transfer(OPCODE_CDC_START);
102     _delay_ms(1);
103     SPI_PORT |= (1 << SS);
104     _delay_ms(1);
105 }
106
107 void PCAP04_Init(void) {
108     SPI_PORT &= ~(1 << SS);
109     _delay_ms(1);
110     SPI_Transfer(OPCODE_INIT);
111     _delay_ms(1);
112     SPI_PORT |= (1 << SS);
113     _delay_ms(1);
114     PCAP04_WriteRegister(0x00, 0x01);
115     PCAP04_WriteRegister(0x04, 0b10010001);
116     PCAP04_WriteRegister(0x06, 0x3F);
117     PCAP04_WriteRegister(0x07, 0x08);
118     PCAP04_WriteRegister(0x08, 0x00);
119     PCAP04_WriteRegister(0x09, 0x19);
120     PCAP04_WriteRegister(0x0A, 0x00);
121     PCAP04_WriteRegister(0x0B, 0x00);
122     uint8_t reg17 = PCAP04_ReadRegister(0x11);
123     reg17 &= 0x03;
124     reg17 |= (7 << 2);
```

```
    PCAP04_WriteRegister(0x11, reg17);
}

uint32_t PCAP04_ReadResult(uint8_t address) {
    uint32_t val = 0;
    SPI_PORT &= ~(1 << SS);
    _delay_ms(1);
    SPI_Transfer(OPCODE_READ_RES | address);
    uint8_t byte1 = SPI_Transfer(0x00);
    uint8_t byte2 = SPI_Transfer(0x00);
    uint8_t byte3 = SPI_Transfer(0x00);
    uint8_t byte4 = SPI_Transfer(0x00);
    val = (byte1 << 24) | (byte2 << 16) | (byte3 << 8) | byte4;
    _delay_ms(1);
    SPI_PORT |= (1 << SS);
    _delay_ms(1);
    return val;
}

float binaryFractionToDecimal(uint8_t frac_bits, uint8_t num_bits) {
    float result = 0.0;
    for (uint8_t i = 0; i < num_bits; i++) {
        if (frac_bits & (1 << (num_bits - 1 - i))) {
            result += 1.0f / (1 << (i + 1));
        }
    }
    return result;
}

int main(void) {
    uint32_t result;
    m_usb_init();
    _delay_ms(1);
    SPI_Init();
    _delay_ms(1);
    PCAP04_Init();
    _delay_ms(1);

    Write_Byte_Auto_Incr(WR_MEM, 0x00, standard_fw, 548);
    _delay_ms(10);
    Read_Byte_Auto_Incr(RD_MEM, 0X00, 10);
    _delay_ms(10);
    PCAP04_WriteRegister(0x2F, 0x01);
    _delay_ms(10);
    PCAP04_CDC_START();
    _delay_ms(20);

    while (1) {
        result = PCAP04_ReadResult(0x00);
        _delay_ms(10);

        for (int i = 0; i < 200; i++) {
            char buffer[16];
            float r = (float)rand() / RAND_MAX;
```

```
            float val = r/5;
            dtostrf(val, 6, 3, buffer);
            for (int i = 0; buffer[i] != '\0'; i++) {
                m_usb_tx_char(buffer[i]);
            }
            m_usb_tx_char('\n');
            m_usb_tx_push();
            _delay_ms(100);
        }

        for (int i = 0; i < 200; i++) {
            char buffer[16];
            float r = (float)rand() / RAND_MAX;
            float val = (0.2 + r);
            if (val<= 1) {
                dtostrf(val, 6, 3, buffer);
                for (int i = 0; buffer[i] != '\0'; i++) {
                    m_usb_tx_char(buffer[i]);
                }
                m_usb_tx_char('\n');
                m_usb_tx_push();
                _delay_ms(100);
            }
        }
    }
}
```

20.3.2 usb.h & usb.c

Listing 3: USB Communication codes

```
#ifndef m_usb__
#define m_usb__

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

void m_usb_init(void);
char m_usb_isconnected(void);
unsigned char m_usb_rx_available(void);
char m_usb_rx_char(void);
void m_usb_rx_flush(void);
char m_usb_tx_char(unsigned char c);
void m_usb_tx_hexchar(unsigned char i);
void m_usb_tx_hex(unsigned int i);
void m_usb_tx_int(int i);
void m_usb_tx_uint(unsigned int i);
void m_usb_tx_long(long i);
void m_usb_tx_ulong(unsigned long i);
#define m_usb_tx_string(s) print_P(PSTR(s))

// Back compatibility macros
```

```
#define usb_init()          m_usb_init()
#define usb_configured()    m_usb_isconnected()
#define usb_rx_available()  m_usb_rx_available()
#define usb_rx_flush()      m_usb_rx_flush()
#define usb_rx_char()       m_usb_rx_char()
#define usb_tx_char(val)    m_usb_tx_char(val)
#define usb_tx_hex(val)     m_usb_tx_hex(val)
#define usb_tx_decimal(val) m_usb_tx_uint(val)
#define usb_tx_string(val)  m_usb_tx_string(val)
#define usb_tx_push()       m_usb_tx_push()

#endif

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

#define USB_SERIAL_PRIVATE_INCLUDE
#include "m_usb.h"
// ---- OVERLOADS FOR M1 BACK COMPATIBILITY ----
#define usb_init()          m_usb_init()
#define usb_configured()    m_usb_isconnected()

#define usb_rx_available()  m_usb_rx_available()
#define usb_rx_flush()      m_usb_rx_flush()
#define usb_rx_char()       m_usb_rx_char()

#define usb_tx_char(val)    m_usb_tx_char(val)
#define usb_tx_hex(val)     m_usb_tx_hex(val)
#define usb_tx_decimal(val) m_usb_tx_uint(val)
#define usb_tx_string(val)  m_usb_tx_string(val)
#define usb_tx_push()       m_usb_tx_push()

#define m_usb_rx_ascii()    m_usb_rx_char()
#define m_usb_tx_ascii(val) m_usb_tx_char(val)


// EVERYTHING ELSE
// ************************************************************


// setup

int8_t usb_serial_putchar(uint8_t c);   // transmit a character
int8_t usb_serial_putchar_nowait(uint8_t c);  // transmit a character, do not wait
int8_t usb_serial_write(const uint8_t *buffer, uint16_t size); // transmit a buffer
void print_P(const char *s);
void phex(unsigned char c);
void phex16(unsigned int i);
void m_usb_tx_hex8(unsigned char i);
void m_usb_tx_push(void);


// serial parameters
```

```
73 uint32_t usb_serial_get_baud(void); // get the baud rate
74 uint8_t usb_serial_get_stopbits(void);  // get the number of stop bits
75 uint8_t usb_serial_get_paritytype(void);// get the parity type
76 uint8_t usb_serial_get_numbits(void);   // get the number of data bits
77 uint8_t usb_serial_get_control(void);   // get the RTS and DTR signal
      state
78 int8_t usb_serial_set_control(uint8_t signals); // set DSR, DCD, RI, etc
79
80 // constants corresponding to the various serial parameters
81 #define USB_SERIAL_DTR          0x01
82 #define USB_SERIAL_RTS          0x02
83 #define USB_SERIAL_1_STOP       0
84 #define USB_SERIAL_1_5_STOP     1
85 #define USB_SERIAL_2_STOP       2
86 #define USB_SERIAL_PARITY_NONE      0
87 #define USB_SERIAL_PARITY_ODD       1
88 #define USB_SERIAL_PARITY_EVEN      2
89 #define USB_SERIAL_PARITY_MARK      3
90 #define USB_SERIAL_PARITY_SPACE     4
91 #define USB_SERIAL_DCD          0x01
92 #define USB_SERIAL_DSR          0x02
93 #define USB_SERIAL_BREAK        0x04
94 #define USB_SERIAL_RI           0x08
95 #define USB_SERIAL_FRAME_ERR        0x10
96 #define USB_SERIAL_PARITY_ERR       0x20
97 #define USB_SERIAL_OVERRUN_ERR      0x40
98
99 // This file does not include the HID debug functions, so these empty
100 // macros replace them with nothing, so users can compile code that
101 // has calls to these functions.
102 #define usb_debug_putchar(c)
103 #define usb_debug_flush_output()
104
105 #define EP_TYPE_CONTROL         0x00
106 #define EP_TYPE_BULK_IN         0x81
107 #define EP_TYPE_BULK_OUT        0x80
108 #define EP_TYPE_INTERRUPT_IN        0xC1
109 #define EP_TYPE_INTERRUPT_OUT       0xC0
110 #define EP_TYPE_ISOCHRONOUS_IN      0x41
111 #define EP_TYPE_ISOCHRONOUS_OUT     0x40
112 #define EP_SINGLE_BUFFER        0x02
113 #define EP_DOUBLE_BUFFER        0x06
114 #define EP_SIZE(s)  ((s) == 64 ? 0x30 : \
115 ((s) == 32 ? 0x20 : \
116 ((s) == 16 ? 0x10 : \
117 0x00)))
118
119 #define MAX_ENDPOINT        4
120
121 #define LSB(n) (n & 255)
122 #define MSB(n) ((n >> 8) & 255)
123
124 #define HW_CONFIG() (UHWCON = 0x01)
125
```

```
126 #ifdef M1
127 #define PLL_CONFIG() (PLLCSR = 0x02) // fixed to 8MHz clock
128 #else
129 #define PLL_CONFIG() (PLLCSR = 0x12) // 0001 0010 For a 16MHz clock
130 #endif
131
132 #define USB_CONFIG() (USBCON = ((1<<USBE)|(1<<OTGPADE)))
133 #define USB_FREEZE() (USBCON = ((1<<USBE)|(1<<FRZCLK)))
134
135 // standard control endpoint request types
136 #define GET_STATUS          0
137 #define CLEAR_FEATURE           1
138 #define SET_FEATURE         3
139 #define SET_ADDRESS         5
140 #define GET_DESCRIPTOR          6
141 #define GET_CONFIGURATION       8
142 #define SET_CONFIGURATION       9
143 #define GET_INTERFACE           10
144 #define SET_INTERFACE           11
145 // HID (human interface device)
146 #define HID_GET_REPORT          1
147 #define HID_GET_PROTOCOL        3
148 #define HID_SET_REPORT          9
149 #define HID_SET_IDLE            10
150 #define HID_SET_PROTOCOL        11
151 // CDC (communication class device)
152 #define CDC_SET_LINE_CODING     0x20
153 #define CDC_GET_LINE_CODING     0x21
154 #define CDC_SET_CONTROL_LINE_STATE  0x22
155
156
157 /*
    ************************************************************************

158  *
159  *  Configurable Options
160  *
161  ************************************************************************
     */
162
163 #define STR_MANUFACTURER    L"J. Fiene"
164 #define STR_PRODUCT     L"M2"
165 #define STR_SERIAL_NUMBER   L"410"
166 #define VENDOR_ID       0x16C0 // must match INF file in Windows
167 #define PRODUCT_ID      0x047A // must match INF file in Windows
168 #define TRANSMIT_FLUSH_TIMEOUT  5   /* in milliseconds */
169 #define TRANSMIT_TIMEOUT    25   /* in milliseconds */
170 #define SUPPORT_ENDPOINT_HALT // can save 116 bytes by removing, makes
    fully USB compliant
171
172 /*
    ************************************************************************

173  *
```

```
174  *  Endpoint Buffer Configuration
175  *
176  ************************************************************************
        */
177
178 #define ENDPOINT0_SIZE      16
179 #define CDC_ACM_ENDPOINT    2
180 #define CDC_RX_ENDPOINT     3
181 #define CDC_TX_ENDPOINT     4
182 #define CDC_ACM_SIZE        16
183 #define CDC_ACM_BUFFER      EP_SINGLE_BUFFER
184 #define CDC_RX_SIZE     64
185 #define CDC_RX_BUFFER       EP_DOUBLE_BUFFER
186 #define CDC_TX_SIZE     64
187 #define CDC_TX_BUFFER       EP_DOUBLE_BUFFER
188
189 static const uint8_t PROGMEM endpoint_config_table[] = {
190     0,
191     1, EP_TYPE_INTERRUPT_IN,  EP_SIZE(CDC_ACM_SIZE) | CDC_ACM_BUFFER,
192     1, EP_TYPE_BULK_OUT,      EP_SIZE(CDC_RX_SIZE) | CDC_RX_BUFFER,
193     1, EP_TYPE_BULK_IN,       EP_SIZE(CDC_TX_SIZE) | CDC_TX_BUFFER
194 };
195
196
197 /*
       ************************************************************************

198  *
199  *  Descriptor Data
200  *
201  ************************************************************************
        */
202
203 static const uint8_t PROGMEM device_descriptor[] = {
204     18,                 // bLength
205     1,                  // bDescriptorType
206     0x00, 0x02,             // bcdUSB
207     2,                  // bDeviceClass
208     0,                  // bDeviceSubClass
209     0,                  // bDeviceProtocol
210     ENDPOINT0_SIZE,             // bMaxPacketSize0
211     LSB(VENDOR_ID), MSB(VENDOR_ID),     // idVendor
212     LSB(PRODUCT_ID), MSB(PRODUCT_ID),   // idProduct
213     0x00, 0x01,             // bcdDevice
214     1,                  // iManufacturer
215     2,                  // iProduct
216     3,                  // iSerialNumber
217     1                   // bNumConfigurations
218 };
219
220 #define CONFIG1_DESC_SIZE (9+9+5+5+4+5+7+9+7+7)
221 static const uint8_t PROGMEM config1_descriptor[CONFIG1_DESC_SIZE] = {
222     // configuration descriptor, USB spec 9.6.3, page 264-266, Table 9-10
223     9,                  // bLength;
```

```
224     2,                  // bDescriptorType;
225     LSB(CONFIG1_DESC_SIZE),         // wTotalLength
226     MSB(CONFIG1_DESC_SIZE),
227     2,                  // bNumInterfaces
228     1,                  // bConfigurationValue
229     0,                  // iConfiguration
230     0xC0,                   // bmAttributes
231     50,                 // bMaxPower
232     // interface descriptor, USB spec 9.6.5, page 267-269, Table 9-12
233     9,                  // bLength
234     4,                  // bDescriptorType
235     0,                  // bInterfaceNumber
236     0,                  // bAlternateSetting
237     1,                  // bNumEndpoints
238     0x02,                   // bInterfaceClass
239     0x02,                   // bInterfaceSubClass
240     0x01,                   // bInterfaceProtocol
241     0,                  // iInterface
242     // CDC Header Functional Descriptor, CDC Spec 5.2.3.1, Table 26
243     5,                  // bFunctionLength
244     0x24,                   // bDescriptorType
245     0x00,                   // bDescriptorSubtype
246     0x10, 0x01,             // bcdCDC
247     // Call Management Functional Descriptor, CDC Spec 5.2.3.2, Table 27
248     5,                  // bFunctionLength
249     0x24,                   // bDescriptorType
250     0x01,                   // bDescriptorSubtype
251     0x01,                   // bmCapabilities
252     1,                  // bDataInterface
253     // Abstract Control Management Functional Descriptor, CDC Spec
        5.2.3.3, Table 28
254     4,                  // bFunctionLength
255     0x24,                   // bDescriptorType
256     0x02,                   // bDescriptorSubtype
257     0x06,                   // bmCapabilities
258     // Union Functional Descriptor, CDC Spec 5.2.3.8, Table 33
259     5,                  // bFunctionLength
260     0x24,                   // bDescriptorType
261     0x06,                   // bDescriptorSubtype
262     0,                  // bMasterInterface
263     1,                  // bSlaveInterface0
264     // endpoint descriptor, USB spec 9.6.6, page 269-271, Table 9-13
265     7,                  // bLength
266     5,                  // bDescriptorType
267     CDC_ACM_ENDPOINT | 0x80,        // bEndpointAddress
268     0x03,                   // bmAttributes (0x03=intr)
269     CDC_ACM_SIZE, 0,            // wMaxPacketSize
270     64,                 // bInterval
271     // interface descriptor, USB spec 9.6.5, page 267-269, Table 9-12
272     9,                  // bLength
273     4,                  // bDescriptorType
274     1,                  // bInterfaceNumber
275     0,                  // bAlternateSetting
276     2,                  // bNumEndpoints
```

```
277     0x0A,                   // bInterfaceClass
278     0x00,                   // bInterfaceSubClass
279     0x00,                   // bInterfaceProtocol
280     0,                  // iInterface
281     // endpoint descriptor, USB spec 9.6.6, page 269-271, Table 9-13
282     7,                  // bLength
283     5,                  // bDescriptorType
284     CDC_RX_ENDPOINT,            // bEndpointAddress
285     0x02,                   // bmAttributes (0x02=bulk)
286     CDC_RX_SIZE, 0,             // wMaxPacketSize
287     0,                  // bInterval
288     // endpoint descriptor, USB spec 9.6.6, page 269-271, Table 9-13
289     7,                  // bLength
290     5,                  // bDescriptorType
291     CDC_TX_ENDPOINT | 0x80,         // bEndpointAddress
292     0x02,                   // bmAttributes (0x02=bulk)
293     CDC_TX_SIZE, 0,             // wMaxPacketSize
294     0                   // bInterval
295 };
296
297 // If you're desperate for a little extra code memory, these strings
298 // can be completely removed if iManufacturer, iProduct, iSerialNumber
299 // in the device desciptor are changed to zeros.
300 struct usb_string_descriptor_struct {
301     uint8_t bLength;
302     uint8_t bDescriptorType;
303     int16_t wString[];
304 };
305 static const struct usb_string_descriptor_struct PROGMEM string0 = {
306     4,
307     3,
308     {0x0409}
309 };
310 static const struct usb_string_descriptor_struct PROGMEM string1 = {
311     sizeof(STR_MANUFACTURER),
312     3,
313     STR_MANUFACTURER
314 };
315 static const struct usb_string_descriptor_struct PROGMEM string2 = {
316     sizeof(STR_PRODUCT),
317     3,
318     STR_PRODUCT
319 };
320 static const struct usb_string_descriptor_struct PROGMEM string3 = {
321     sizeof(STR_SERIAL_NUMBER),
322     3,
323     STR_SERIAL_NUMBER
324 };
325
326 // This table defines which descriptor data is sent for each specific
327 // request from the host (in wValue and wIndex).
328 static const struct descriptor_list_struct {
329     uint16_t    wValue;
330     uint16_t    wIndex;
```

```
331     const uint8_t   *addr;
332     uint8_t     length;
333 } PROGMEM descriptor_list[] = {
334     {0x0100, 0x0000, device_descriptor, sizeof(device_descriptor)},
335     {0x0200, 0x0000, config1_descriptor, sizeof(config1_descriptor)},
336     {0x0300, 0x0000, (const uint8_t *)&string0, 4},
337     {0x0301, 0x0409, (const uint8_t *)&string1, sizeof(STR_MANUFACTURER)},
338     {0x0302, 0x0409, (const uint8_t *)&string2, sizeof(STR_PRODUCT)},
339     {0x0303, 0x0409, (const uint8_t *)&string3, sizeof(STR_SERIAL_NUMBER)}
340 };
341 #define NUM_DESC_LIST (sizeof(descriptor_list)/sizeof(struct
       descriptor_list_struct))
342
343
344 /*
       **************************************************************************

345  *
346  *  Variables - these are the only non-stack RAM usage
347  *
348  ***************************************************************************
       */
349
350 // zero when we are not configured, non-zero when enumerated
351 static volatile uint8_t usb_configuration=0;
352
353 // the time remaining before we transmit any partially full
354 // packet, or send a zero length packet.
355 static volatile uint8_t transmit_flush_timer=0;
356 static uint8_t transmit_previous_timeout=0;
357
358 // serial port settings (baud rate, control signals, etc) set
359 // by the PC.  These are ignored, but kept in RAM.
360 static uint8_t cdc_line_coding[7]={0x00, 0xE1, 0x00, 0x00, 0x00, 0x00, 0
       x08};
361 static uint8_t cdc_line_rtsdtr=0;
362
363
364 /*
       **************************************************************************

365  *
366  *  Public Functions - these are the API intended for the user
367  *
368  ***************************************************************************
       */
369
370 // initialize USB serial
371 void m_usb_init(void)
372 {
373     HW_CONFIG();
374         USB_FREEZE();               // enable USB
375         PLL_CONFIG();               // config PLL, 16 MHz xtal
376         while (!(PLLCSR & (1<<PLOCK))) ;    // wait for PLL lock
```

```
        USB_CONFIG();                  // start USB clock
        UDCON = 0;                // enable attach resistor
    usb_configuration = 0;
    cdc_line_rtsdtr = 0;
        UDIEN = (1<<EORSTE)|(1<<SOFE);
    sei();
}

// return 0 if the USB is not configured, or the configuration
// number selected by the HOST
char m_usb_isconnected(void)
{
    return (char)usb_configuration;
}

// get the next character, or -1 if nothing received
char m_usb_rx_char(void)
{
    uint8_t c, intr_state;

    // interrupts are disabled so these functions can be
    // used from the main program or interrupt context,
    // even both in the same program!
    intr_state = SREG;
    cli();
    if (!usb_configuration) {
        SREG = intr_state;
        return -1;
    }
    UENUM = CDC_RX_ENDPOINT;
    if (!(UEINTX & (1<<RWAL))) {
        // no data in buffer
        SREG = intr_state;
        return -1;
    }
    // take one byte out of the buffer
    c = UEDATX;
    // if buffer completely used, release it
    if (!(UEINTX & (1<<RWAL))) UEINTX = 0x6B;
    SREG = intr_state;
    return (char)c;
}

// number of bytes available in the receive buffer
unsigned char m_usb_rx_available(void)
{
    uint8_t n=0, intr_state;

    intr_state = SREG;
    cli();
    if (usb_configuration) {
        UENUM = CDC_RX_ENDPOINT;
        n = UEBCLX;
    }
```

```
431     SREG = intr_state;
432     return (unsigned char)n;
433 }
434 
435 // discard any buffered input
436 void m_usb_rx_flush(void)
437 {
438     uint8_t intr_state;
439 
440     if (usb_configuration) {
441         intr_state = SREG;
442         cli();
443         UENUM = CDC_RX_ENDPOINT;
444         while ((UEINTX & (1<<RWAL))) {
445             UEINTX = 0x6B;
446         }
447         SREG = intr_state;
448     }
449 }
450 
451 // transmit a character.  0 returned on success, -1 on error
452 char m_usb_tx_char(unsigned char c)
453 {
454     uint8_t timeout, intr_state;
455 
456     // if we're not online (enumerated and configured), error
457     if (!usb_configuration) return -1;
458     // interrupts are disabled so these functions can be
459     // used from the main program or interrupt context,
460     // even both in the same program!
461     intr_state = SREG;
462     cli();
463     UENUM = CDC_TX_ENDPOINT;
464     // if we gave up due to timeout before, don't wait again
465     if (transmit_previous_timeout) {
466         if (!(UEINTX & (1<<RWAL))) {
467             SREG = intr_state;
468             return -1;
469         }
470         transmit_previous_timeout = 0;
471     }
472     // wait for the FIFO to be ready to accept data
473     timeout = UDFNUML + TRANSMIT_TIMEOUT;
474     while (1) {
475         // are we ready to transmit?
476         if (UEINTX & (1<<RWAL)) break;
477         SREG = intr_state;
478         // have we waited too long?  This happens if the user
479         // is not running an application that is listening
480         if (UDFNUML == timeout) {
481             transmit_previous_timeout = 1;
482             return -1;
483         }
484         // has the USB gone offline?
```

```
485         if (!usb_configuration) return -1;
486         // get ready to try checking again
487         intr_state = SREG;
488         cli();
489         UENUM = CDC_TX_ENDPOINT;
490     }
491     // actually write the byte into the FIFO
492     UEDATX = (uint8_t)c;
493     // if this completed a packet, transmit it now!
494     if (!(UEINTX & (1<<RWAL))) UEINTX = 0x3A;
495     transmit_flush_timer = TRANSMIT_FLUSH_TIMEOUT;
496     SREG = intr_state;
497     return 0;
498 }
499
500
501 // transmit a character, but do not wait if the buffer is full,
502 //   0 returned on success, -1 on buffer full or error
503 int8_t usb_serial_putchar_nowait(uint8_t c)
504 {
505     uint8_t intr_state;
506
507     if (!usb_configuration) return -1;
508     intr_state = SREG;
509     cli();
510     UENUM = CDC_TX_ENDPOINT;
511     if (!(UEINTX & (1<<RWAL))) {
512         // buffer is full
513         SREG = intr_state;
514         return -1;
515     }
516     // actually write the byte into the FIFO
517     UEDATX = c;
518         // if this completed a packet, transmit it now!
519     if (!(UEINTX & (1<<RWAL))) UEINTX = 0x3A;
520     transmit_flush_timer = TRANSMIT_FLUSH_TIMEOUT;
521     SREG = intr_state;
522     return 0;
523 }
524
525 // transmit a buffer.
526 //  0 returned on success, -1 on error
527 // This function is optimized for speed!  Each call takes approx 6.1 us
    overhead
528 // plus 0.25 us per byte.  12 Mbit/sec USB has 8.67 us per-packet overhead
     and
529 // takes 0.67 us per byte.  If called with 64 byte packet-size blocks,
    this function
530 // can transmit at full USB speed using 43% CPU time.  The maximum
    theoretical speed
531 // is 19 packets per USB frame, or 1216 kbytes/sec.  However, bulk
    endpoints have the
532 // lowest priority, so any other USB devices will likely reduce the speed.
     Speed
```

```
533 // can also be limited by how quickly the PC-based software reads data, as
        the host
534 // controller in the PC will not allocate bandwitdh without a pending read
        request.
535 // (thanks to Victor Suarez for testing and feedback and initial code)
536
537 int8_t usb_serial_write(const uint8_t *buffer, uint16_t size)
538 {
539     uint8_t timeout, intr_state, write_size;
540
541     // if we're not online (enumerated and configured), error
542     if (!usb_configuration) return -1;
543     // interrupts are disabled so these functions can be
544     // used from the main program or interrupt context,
545     // even both in the same program!
546     intr_state = SREG;
547     cli();
548     UENUM = CDC_TX_ENDPOINT;
549     // if we gave up due to timeout before, don't wait again
550     if (transmit_previous_timeout) {
551         if (!(UEINTX & (1<<RWAL))) {
552             SREG = intr_state;
553             return -1;
554         }
555         transmit_previous_timeout = 0;
556     }
557     // each iteration of this loop transmits a packet
558     while (size) {
559         // wait for the FIFO to be ready to accept data
560         timeout = UDFNUML + TRANSMIT_TIMEOUT;
561         while (1) {
562             // are we ready to transmit?
563             if (UEINTX & (1<<RWAL)) break;
564             SREG = intr_state;
565             // have we waited too long?  This happens if the user
566             // is not running an application that is listening
567             if (UDFNUML == timeout) {
568                 transmit_previous_timeout = 1;
569                 return -1;
570             }
571             // has the USB gone offline?
572             if (!usb_configuration) return -1;
573             // get ready to try checking again
574             intr_state = SREG;
575             cli();
576             UENUM = CDC_TX_ENDPOINT;
577         }
578
579         // compute how many bytes will fit into the next packet
580         write_size = CDC_TX_SIZE - UEBCLX;
581         if (write_size > size) write_size = size;
582         size -= write_size;
583
584         // write the packet
```

```
585         switch (write_size) {
586             #if (CDC_TX_SIZE == 64)
587             case 64: UEDATX = *buffer++;
588             case 63: UEDATX = *buffer++;
589             case 62: UEDATX = *buffer++;
590             case 61: UEDATX = *buffer++;
591             case 60: UEDATX = *buffer++;
592             case 59: UEDATX = *buffer++;
593             case 58: UEDATX = *buffer++;
594             case 57: UEDATX = *buffer++;
595             case 56: UEDATX = *buffer++;
596             case 55: UEDATX = *buffer++;
597             case 54: UEDATX = *buffer++;
598             case 53: UEDATX = *buffer++;
599             case 52: UEDATX = *buffer++;
600             case 51: UEDATX = *buffer++;
601             case 50: UEDATX = *buffer++;
602             case 49: UEDATX = *buffer++;
603             case 48: UEDATX = *buffer++;
604             case 47: UEDATX = *buffer++;
605             case 46: UEDATX = *buffer++;
606             case 45: UEDATX = *buffer++;
607             case 44: UEDATX = *buffer++;
608             case 43: UEDATX = *buffer++;
609             case 42: UEDATX = *buffer++;
610             case 41: UEDATX = *buffer++;
611             case 40: UEDATX = *buffer++;
612             case 39: UEDATX = *buffer++;
613             case 38: UEDATX = *buffer++;
614             case 37: UEDATX = *buffer++;
615             case 36: UEDATX = *buffer++;
616             case 35: UEDATX = *buffer++;
617             case 34: UEDATX = *buffer++;
618             case 33: UEDATX = *buffer++;
619             #endif
620             #if (CDC_TX_SIZE >= 32)
621             case 32: UEDATX = *buffer++;
622             case 31: UEDATX = *buffer++;
623             case 30: UEDATX = *buffer++;
624             case 29: UEDATX = *buffer++;
625             case 28: UEDATX = *buffer++;
626             case 27: UEDATX = *buffer++;
627             case 26: UEDATX = *buffer++;
628             case 25: UEDATX = *buffer++;
629             case 24: UEDATX = *buffer++;
630             case 23: UEDATX = *buffer++;
631             case 22: UEDATX = *buffer++;
632             case 21: UEDATX = *buffer++;
633             case 20: UEDATX = *buffer++;
634             case 19: UEDATX = *buffer++;
635             case 18: UEDATX = *buffer++;
636             case 17: UEDATX = *buffer++;
637             #endif
638             #if (CDC_TX_SIZE >= 16)
```

```
639             case 16: UEDATX = *buffer++;
640             case 15: UEDATX = *buffer++;
641             case 14: UEDATX = *buffer++;
642             case 13: UEDATX = *buffer++;
643             case 12: UEDATX = *buffer++;
644             case 11: UEDATX = *buffer++;
645             case 10: UEDATX = *buffer++;
646             case  9: UEDATX = *buffer++;
647             #endif
648             case  8: UEDATX = *buffer++;
649             case  7: UEDATX = *buffer++;
650             case  6: UEDATX = *buffer++;
651             case  5: UEDATX = *buffer++;
652             case  4: UEDATX = *buffer++;
653             case  3: UEDATX = *buffer++;
654             case  2: UEDATX = *buffer++;
655             default:
656             case  1: UEDATX = *buffer++;
657             case  0: break;
658         }
659         // if this completed a packet, transmit it now!
660         if (!(UEINTX & (1<<RWAL))) UEINTX = 0x3A;
661         transmit_flush_timer = TRANSMIT_FLUSH_TIMEOUT;
662     }
663     SREG = intr_state;
664     return 0;
665 }
666
667
668 // immediately transmit any buffered output.
669 // This doesn't actually transmit the data - that is impossible!
670 // USB devices only transmit when the host allows, so the best
671 // we can do is release the FIFO buffer for when the host wants it
672 void m_usb_tx_push(void)
673 {
674     uint8_t intr_state;
675
676     intr_state = SREG;
677     cli();
678     if (transmit_flush_timer) {
679         UENUM = CDC_TX_ENDPOINT;
680         UEINTX = 0x3A;
681         transmit_flush_timer = 0;
682     }
683     SREG = intr_state;
684 }
685
686 // functions to read the various async serial settings.  These
687 // aren't actually used by USB at all (communication is always
688 // at full USB speed), but they are set by the host so we can
689 // set them properly if we're converting the USB to a real serial
690 // communication
691
692 //uint32_t usb_serial_get_baud(void)
```

```
693 //{
694 //  return *(uint32_t *)cdc_line_coding;
695 //}
696 uint8_t usb_serial_get_stopbits(void)
697 {
698     return cdc_line_coding[4];
699 }
700 uint8_t usb_serial_get_paritytype(void)
701 {
702     return cdc_line_coding[5];
703 }
704 uint8_t usb_serial_get_numbits(void)
705 {
706     return cdc_line_coding[6];
707 }
708 uint8_t usb_serial_get_control(void)
709 {
710     return cdc_line_rtsdtr;
711 }
712 // write the control signals, DCD, DSR, RI, etc
713 // There is no CTS signal.  If software on the host has transmitted
714 // data to you but you haven't been calling the getchar function,
715 // it remains buffered (either here or on the host) and can not be
716 // lost because you weren't listening at the right time, like it
717 // would in real serial communication.
718 // TODO: this function is untested.  Does it work?  Please email
719 // paul@pjrc.com if you have tried it....
720 int8_t usb_serial_set_control(uint8_t signals)
721 {
722     uint8_t intr_state;
723
724     intr_state = SREG;
725     cli();
726     if (!usb_configuration) {
727         // we're not enumerated/configured
728         SREG = intr_state;
729         return -1;
730     }
731
732     UENUM = CDC_ACM_ENDPOINT;
733     if (!(UEINTX & (1<<RWAL))) {
734         // unable to write
735         // TODO; should this try to abort the previously
736         // buffered message??
737         SREG = intr_state;
738         return -1;
739     }
740     UEDATX = 0xA1;
741     UEDATX = 0x20;
742     UEDATX = 0;
743     UEDATX = 0;
744     UEDATX = 0; // TODO: should this be 1 or 0 ???
745     UEDATX = 0;
746     UEDATX = 2;
```

```
747     UEDATX = 0;
748     UEDATX = signals;
749     UEDATX = 0;
750     UEINTX = 0x3A;
751     SREG = intr_state;
752     return 0;
753 }
754
755
756
757 /*
    ************************************************************************

758  *
759  *  Private Functions - not intended for general user consumption....
760  *
761  ************************************************************************
        */
762
763
764 // USB Device Interrupt - handle all device-level events
765 // the transmit buffer flushing is triggered by the start of frame
766 //
767 ISR(USB_GEN_vect)
768 {
769     uint8_t intbits, t;
770
771         intbits = UDINT;
772         UDINT = 0;
773         if (intbits & (1<<EORSTI)) {
774         UENUM = 0;
775         UECONX = 1;
776         UECFG0X = EP_TYPE_CONTROL;
777         UECFG1X = EP_SIZE(ENDPOINT0_SIZE) | EP_SINGLE_BUFFER;
778         UEIENX = (1<<RXSTPE);
779         usb_configuration = 0;
780         cdc_line_rtsdtr = 0;
781         }
782     if (intbits & (1<<SOFI)) {
783         if (usb_configuration) {
784             t = transmit_flush_timer;
785             if (t) {
786                 transmit_flush_timer = --t;
787                 if (!t) {
788                     UENUM = CDC_TX_ENDPOINT;
789                     UEINTX = 0x3A;
790                 }
791             }
792         }
793     }
794 }
795
796
797 // Misc functions to wait for ready and send/receive packets
```

```
798 static inline void usb_wait_in_ready(void)
799 {
800     while (!(UEINTX & (1<<TXINI))) ;
801 }
802 static inline void usb_send_in(void)
803 {
804     UEINTX = ~(1<<TXINI);
805 }
806 static inline void usb_wait_receive_out(void)
807 {
808     while (!(UEINTX & (1<<RXOUTI))) ;
809 }
810 static inline void usb_ack_out(void)
811 {
812     UEINTX = ~(1<<RXOUTI);
813 }
814
815
816
817 // USB Endpoint Interrupt - endpoint 0 is handled here.  The
818 // other endpoints are manipulated by the user-callable
819 // functions, and the start-of-frame interrupt.
820 //
821 ISR(USB_COM_vect)
822 {
823         uint8_t intbits;
824     const uint8_t *list;
825         const uint8_t *cfg;
826     uint8_t i, n, len, en;
827     uint8_t *p;
828     uint8_t bmRequestType;
829     uint8_t bRequest;
830     uint16_t wValue;
831     uint16_t wIndex;
832     uint16_t wLength;
833     uint16_t desc_val;
834     const uint8_t *desc_addr;
835     uint8_t desc_length;
836
837         UENUM = 0;
838         intbits = UEINTX;
839         if (intbits & (1<<RXSTPI)) {
840                 bmRequestType = UEDATX;
841                 bRequest = UEDATX;
842                 wValue = UEDATX;
843                 wValue |= (UEDATX << 8);
844                 wIndex = UEDATX;
845                 wIndex |= (UEDATX << 8);
846                 wLength = UEDATX;
847                 wLength |= (UEDATX << 8);
848                 UEINTX = ~((1<<RXSTPI) | (1<<RXOUTI) | (1<<TXINI));
849                 if (bRequest == GET_DESCRIPTOR) {
850             list = (const uint8_t *)descriptor_list;
851             for (i=0; ; i++) {
```

```
852                 if (i >= NUM_DESC_LIST) {
853                     UECONX = (1<<STALLRQ)|(1<<EPEN);  //stall
854                     return;
855                 }
856                 desc_val = pgm_read_word(list);
857                 if (desc_val != wValue) {
858                     list += sizeof(struct descriptor_list_struct);
859                     continue;
860                 }
861                 list += 2;
862                 desc_val = pgm_read_word(list);
863                 if (desc_val != wIndex) {
864                     list += sizeof(struct descriptor_list_struct)-2;
865                     continue;
866                 }
867                 list += 2;
868                 desc_addr = (const uint8_t *)pgm_read_word(list);
869                 list += 2;
870                 desc_length = pgm_read_byte(list);
871                 break;
872             }
873             len = (wLength < 256) ? wLength : 255;
874             if (len > desc_length) len = desc_length;
875             do {
876                 // wait for host ready for IN packet
877                 do {
878                     i = UEINTX;
879                 } while (!(i & ((1<<TXINI)|(1<<RXOUTI))));
880                 if (i & (1<<RXOUTI)) return;    // abort
881                 // send IN packet
882                 n = len < ENDPOINT0_SIZE ? len : ENDPOINT0_SIZE;
883                 for (i = n; i; i--) {
884                     UEDATX = pgm_read_byte(desc_addr++);
885                 }
886                 len -= n;
887                 usb_send_in();
888             } while (len || n == ENDPOINT0_SIZE);
889             return;
890                 }
891         if (bRequest == SET_ADDRESS) {
892             usb_send_in();
893             usb_wait_in_ready();
894             UDADDR = wValue | (1<<ADDEN);
895             return;
896         }
897         if (bRequest == SET_CONFIGURATION && bmRequestType == 0) {
898             usb_configuration = wValue;
899             cdc_line_rtsdtr = 0;
900             transmit_flush_timer = 0;
901             usb_send_in();
902             cfg = endpoint_config_table;
903             for (i=1; i<5; i++) {
904                 UENUM = i;
905                 en = pgm_read_byte(cfg++);
```

```
                UECONX = en;
                if (en) {
                    UECFG0X = pgm_read_byte(cfg++);
                    UECFG1X = pgm_read_byte(cfg++);
                }
            }
                UERST = 0x1E;
                UERST = 0;
            return;
        }
        if (bRequest == GET_CONFIGURATION && bmRequestType == 0x80) {
            usb_wait_in_ready();
            UEDATX = usb_configuration;
            usb_send_in();
            return;
        }
        if (bRequest == CDC_GET_LINE_CODING && bmRequestType == 0xA1) {
            usb_wait_in_ready();
            p = cdc_line_coding;
            for (i=0; i<7; i++) {
                UEDATX = *p++;
            }
            usb_send_in();
            return;
        }
        if (bRequest == CDC_SET_LINE_CODING && bmRequestType == 0x21) {
            usb_wait_receive_out();
            p = cdc_line_coding;
            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)
{
    char c;

    while (1) {
        c = pgm_read_byte(s++);
        if (!c) break;
        if (c == '\n') usb_tx_char('\r');
        usb_tx_char(c);
    }
}

void phex1(unsigned char c)
{
    usb_tx_char(c + ((c < 10) ? '0' : 'A' - 10));
}

void phex(unsigned char c)
{
    phex1(c >> 4);
    phex1(c & 15);
}

void m_usb_tx_hex(unsigned int i)
{
    phex(i >> 8);
```

```
1013     phex(i);
1014 }
1015
1016 void m_usb_tx_hexchar(unsigned char i)
1017 {
1018     phex(i);
1019 }
1020
1021 void m_usb_tx_int(int i)
1022 {
1023     char string[7] = {0,0,0,0,0,0,0};
1024     itoa(i,string,10);
1025     for(i=0;i<7;i++){
1026         if(string[i]){
1027             m_usb_tx_char(string[i]);
1028         }
1029     }
1030 }
1031
1032 void m_usb_tx_uint(unsigned int i)
1033 {
1034     char string[6] = {0,0,0,0,0,0};
1035     utoa(i,string,10);
1036     for(i=0;i<5;i++){
1037         if(string[i]){
1038             m_usb_tx_char(string[i]);
1039         }
1040     }
1041 }
1042
1043 void m_usb_tx_long(long i)
1044 {
1045     char string[11] = {0,0,0,0,0,0,0,0,0,0,0};
1046     ltoa(i,string,10);
1047     for(i=0;i<11;i++){
1048         if(string[i]){
1049             m_usb_tx_char(string[i]);
1050         }
1051     }
1052 }
1053
1054 void m_usb_tx_ulong(unsigned long i)
1055 {
1056     char string[11] = {0,0,0,0,0,0,0,0,0,0,0};
1057     ultoa(i,string,10);
1058     for(i=0;i<10;i++){
1059         if(string[i]){
1060             m_usb_tx_char(string[i]);
1061         }
1062     }
1063 }
```

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.

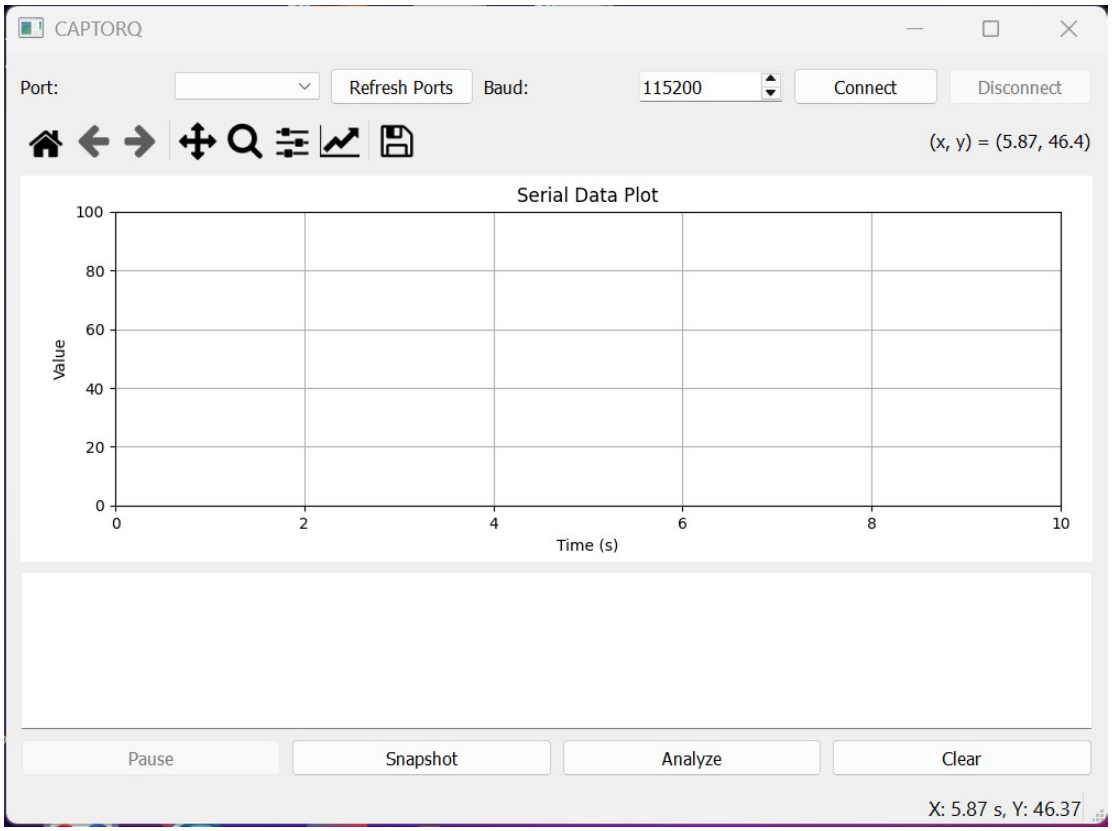


Figure 58: CAPTORQ Software Interface for Visualizing Torque Sensor Output

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 3. All costs are calculated in Sri Lankan Rupees (LKR).

Table 3: 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

23 Amendments Made Since First Evaluation

* Amending the Figure 13. (CDC Calculation)

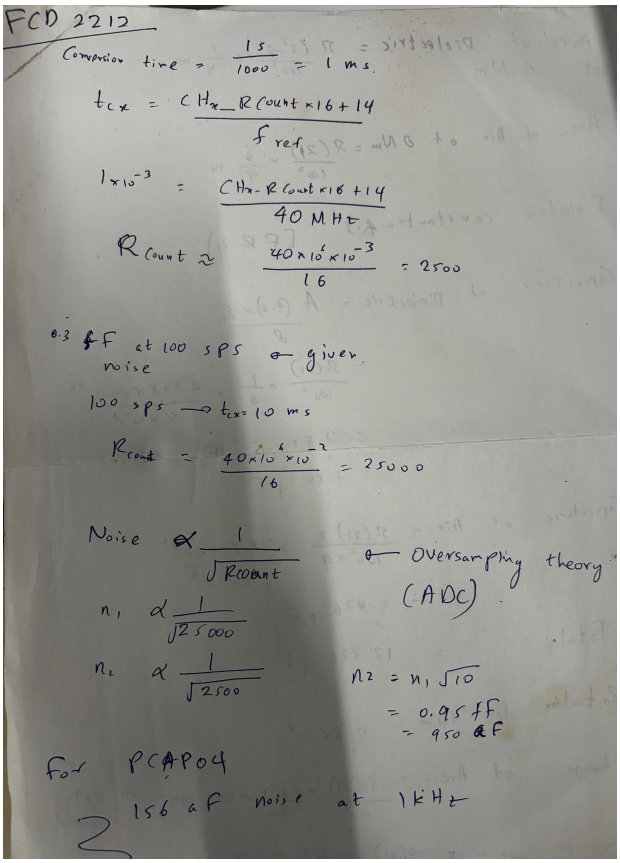
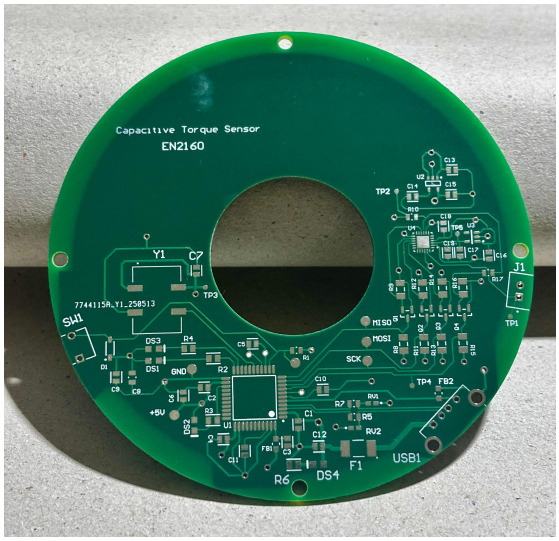


Figure 59: CDC FUll Calculation

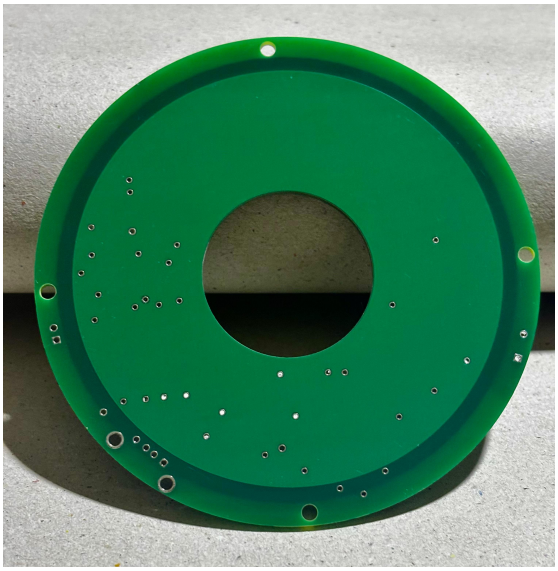
23.1 PCB Soldering and Testing

The printed circuit board arrived on the day of the first evaluation. There were 6 copies from both of the PCBs. They were soldered by 2 teammates on the workshop in the department. Solder-paste which had a mixture of solder and flux was used for soldering. The SMD components were placed by tweezers with the aid of the magnifying camera. The a miniature hot plate was used to heat up the solder to proper temperature. The most challenging part was soldering PCAP04. Its QFN package made it difficult to have enough solder as well as prevent seeing if it was properly soldered. After soldering the SMD components, through hole components such as the JST headers and USB header were soldered using a soldering iron.

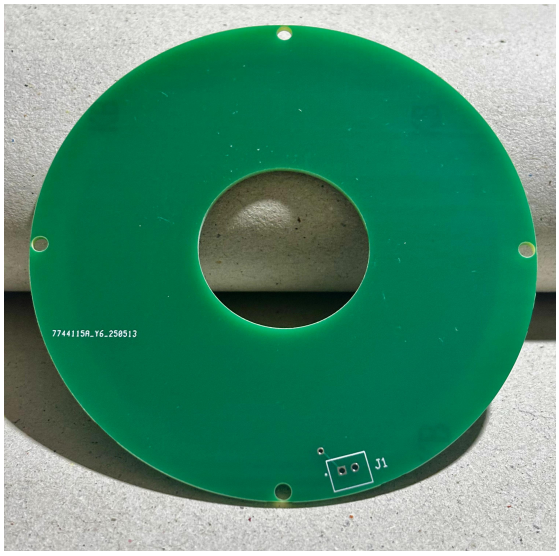
The PCB was tested in the analog laboratory. Firstly, continuity was checked using the continuity method in the digital multimeter. There seemed to be no shorted components. Since JTAG pins were added later, they could not be utilized in this device. The voltage levels of respective test points were measured to ensure the proper function of 3.3 V and 1.8 V regulators. Finally, SPI lines were observed with an oscilloscope to ensure that communication between MCU and PCAP04 was possible.



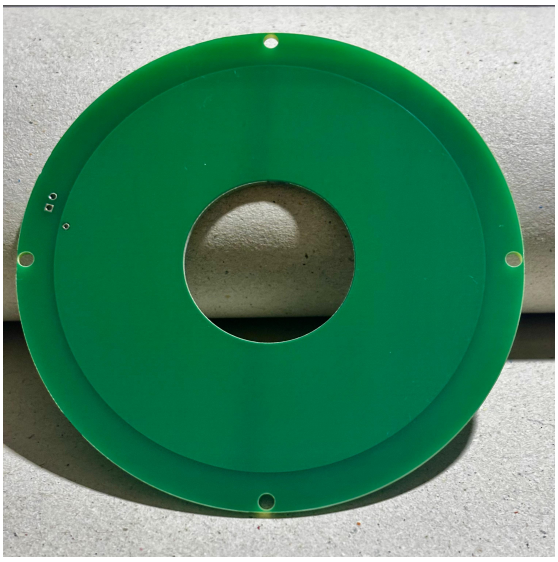
(a) Front Side of Primary PCB



(b) Back Side of Primary PCB



(c) Front Side of Secondary PCB



(d) Back Side of Secondary PCB

Figure 60: Front and Back Appearance of Both PCBs Before Soldering

.

Figure 61: Soldered Primary PCB

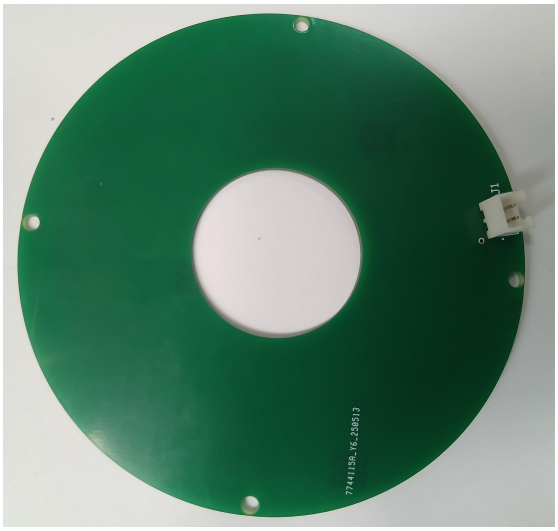


Figure 62: Soldered Secondary PCB

23.2 Final Mechanical Parts + Enclosure

The inner parts made for the first evaluation (metal disks, dielectric disk, and their holders) had slight errors in there dimensions. As a result, the metal disks and dielectric disks were touching, causing friction and therefore, inaccurate torque measurement. These parts were machined again with refined measurements to ensure proper operation of the device. These were then assembled using adhesives. The small size of disk holding poles prevent welding due to risk of melting. Also the closely packed nature makes it difficult.

Figure 63: Final Product

Not much changes were done to the Aluminum enclosure apart from adding a hole for USB port to come out of. This was done using a CNC drilling machine. Then the PCB spacers were attached to the 2 flat walls of the enclosure. These are used to hold the PCB with screws. Finally a coat of white paint was applied using the paint spray machine in the UAV lab.

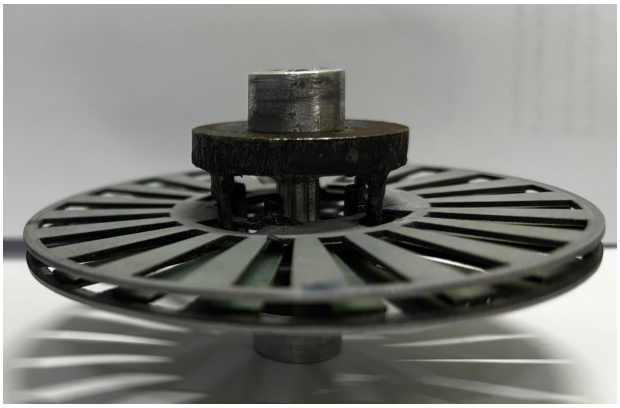


Figure 64: The Internal Mechanical Parts Attached to the Shaft

Figure 65: Inside View

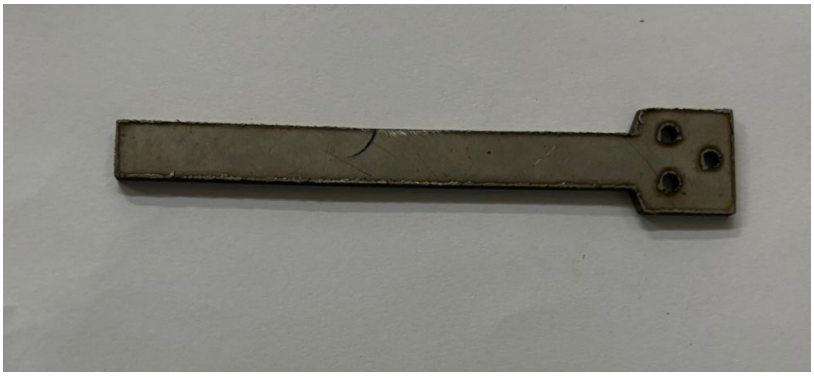


Figure 66: Handles to apply external Torque

23.3 Problems Encountered

At Initial Stage

- Lack of reference material. While strain guage based torque sensors were common, capacitive torque sensors were not. Therefore, it took lot of effort to find a sensor to use as a reference design.
- Lack of information. Even after finding a capacitive torque sensor, its internal working were a mystery. The documentation provided by manufacturers did not describe internal workings. We had to research extensively till we found a patent describing design of a capacitive torque sensor.

At Design Stage

- Component Selection. Since the deformation is miniscule, we needed an extremely accurate way to measure torque. In addition to this, there was sampling rate requirement of 1000Hz. Finding a capacitance to digital converter to satisfy these needs was difficult.
- Finite Element Analysis. None of the group members had much mechanica/material engineering knowledge. So running simulations to choose shaft material and shape with best deformation was challenging. In addition to deformation requirement, there was also the requirement that device does not deform upto five times nominal torque. These constraints made designing shaft difficult.
- PCB Routing. Since the reference torque sensors were ssmall, we also had to reduce size as much as possible. This meant a small PCB. Also, since we use a copper pour on the ottom layer as the capacitor plate, routing was restricted to just 1 layer.

At Final Stage

- Soldering the PCB was difficult because PCAP04 was extremely small an was a QFN package. After soldering, there was no wy to know if it was propely connected.
- Since PCAP04 did not have much documentation available, the coding part was difficult. Also the available documentation contradicted each other at times.
- Due to the small size of PCAP04, it was difficult to debug it using multimeter and oscilloscope probe. Also since it was QFN package no pins were visible to test.
- The internal metal disks were a tight fit to the enclosure. Because of a slight tilt to the shaft, the metal disks tended to touch enclosure wall while rotating. Also JST wire makes the matter even worse because it must fit between metal disk and wall.

23.4 Project Timeline

21 February - Started reading a research paper from IEEE about a design of a novel capacitive torque sensor.
25 February - Watched videos about using torque sensors and identified user needs.
3 March - Read a patent about capacitor torque sensor design and identified a viable internal mechanism.
6 March - Evaluating conceptual designs.
15 March - Evaluated the capacitance measruing methods and decided on using a CDC.
28 March - 2 April - Did finite element analysis on shaft and decided on final shaft shape.
2 April - Evaluated dielectric materials and decided on FR4.
8 April - Evaluated MCUs and CDCs.
18 April - Decided on final MCU selection - ATmega32U4
20 April - Purchased aluminum for shaft and enclosure
23 April -Decided on final CDC selection - PCAP04
25 April - Finished coding the USB communication part on register level. Tested on development board
29 April - Designed the circuit and drew the schematics.
3 May - Ordered components
10 May - Did PCB design and routing
13 May - Ordered the PCB with 3 other teams.
18 May - Did machining and laser cutting for the enclosure and internal mechanical parts.
22 May - Had first evaluation
1 July - Soldered the PCB in department workshop.
2 July - Started coding the PCAP04 interfacing with the MCU.
7 July - Did laser cutting and machined the hole in enclosure for USB.
10 July - Did final assembly and painted the enclosure.

23.5 Complete Project Budget

Electronic Components - Mouser

Component	Mouser Product Code	Quantity	Price (Rs.)
ESD Protection Diodes	652-CG0603MLC-05E	10	454.95
Resettable Fuses	576-1812L050PR	10	973.6
1.8V Voltage Regulator	LP5907MFX-1.8/NOPB	5	743
3.3V Voltage Regulator	TPS73633DBVR	4	2608.4
16MHz Clock Oscillator	ECS-8FMX-160-TR	3	3776.085
Wire Headers	82BXHAM(LF)(SN)	7	360.9
1uF Capacitor	KAM21BR71H104JT	20	770.38
0.1uF Capacitor	KGM21AR71C105JU	10	500.5
10uF Capacitor	C0805C106J8RACAUTO7210	10	1686.34
0.01uF Capacitor	0805ZC103JAT2A	10	506.5
4.7uF Capacitor	C0805X475J8RAC7800	10	1668.15
22 Resistor	ERJ-T06J220V	10	315.4
330 Resistor	560112120014	10	424.6
10k Resistor	RK73B2ATTE103J	10	81.9
Ferrite Bead	MMZ1005F181ETD25	10	461.2
PCAP04 Capacitance to Digital Converter	PCAP04-AQFM-24-V2	4	9293.11
ATmega32U4 Microcontroller	ATMEGA32U4-AU	4	6417.82
Total			**31042.835**

Shipping = Rs. 14200.85
Taxes = Rs. 9000
Total = Rs. 54243.685

Components - Local

Component	Quantity	Price (Rs.)
USB Cable	1	250
USB Ports	5	75
Development Board	1	1760
JST Cable + Header	5	550
PCB Holders + Screws	10	530
Solder Paste	1	1000
Total		**3165**

Components - Ali Express

Component	Quantity	Price (Rs.)
6803 Bearings	10	340.87
1mm Thick FR4 30cm x 30cm	1	3150.20
Total		3491.07

Shipping + Taxes = Rs. 3950

Total = Rs. 7441.07

PCB Expenses

PCB Price + Shipping = Rs. 12350

Tax = Rs. 2612

Total = Rs. 14962

Enclosure and Inner Parts

Description	**Price (Rs.)**
Aluminum Cylinder + Rod	4800
Laser Cutting	7710
Metal Sheets (5mm + 3mm)	2300
Metal Sheet (1mm)	650
Metal Sheet (6mm)	300
Metal Work (Lathe Machining)	3500
Welding	1200
Alan Bolt	450
Sandpaper + Glue	1240
Paint	715
Total	**22865**

Total Expenses (Rs.) = 22865 + 7441.07 + 14962 + 3165 + 54243.685

= 102,676.755