The Andrew and Erna Viterbi Faculty of ELECTRICAL & COMPUTER ENGINEERING

TECHNION Israel Institute of Technology

The Prof. Israel Cederbaum High Speed Digital Systems Laboratory HS DSL המעבדה למערכות ספרתיות מהירות ע"ש פרופ' ישראל צדרבאום

Project Number : 7462

SPINNING TOP HARDWARE

Written by Keselman Liya & Jerushalmi Iddo
Winter 23z24
Supervisor: Mizrachi Boaz

Table of Contents

1. Abstract

This report documents the design and implementation of the first hardware prototype for an advanced, remotely controlled spinning top. Our team was responsible for developing the physical and electrical infrastructure, validating its functionality, and preparing it to support the embedded systems required for future project phases. We focused on the hardware configuration, board design, and real-time data streaming via Wi-Fi. The system elevates the traditional Hanukkah dreidel into a modern, controllable device using smartphone integration.

The project involved building a compact mechanical structure, designing a custom PCB, integrating motors, sensors, and LEDs, and programming an Arduino-based control system. Key challenges included fitting numerous components onto a dreidel-sized board, balancing noise-sensitive and noise-generating elements, and maintaining stability during rotation. The prototype achieved partial synchronization between components, demonstrated stable spinning with a balancing mechanism, and successfully communicated sensor data.

Despite some integration limitations and GPIO constraints, the system lays a strong foundation for future development, potentially expanding into fields like education, robotics, or tactical mobile platforms.

2. Motivation

Although the spinning top may appear as a toy at first glance, it holds vast potential for real-world applications. In future phases, the platform could serve in tactical scenarios—maneuvering in confined environments, detecting and avoiding obstacles, or visually communicate through programmable LEDs in real time.
this device represents a foundation for a multi-phase, multi-team development effort that extends far beyond entertainment.

This project isn't just about spinning; it's about proving that even a small, mobile platform can carry out complex functions when engineered thoughtfully. Our motivation was to demonstrate that a low-profile, agile system could be modular, responsive, and ready for future upgrades.

3. System Goals and the Project's Vision

Our project's goals are:

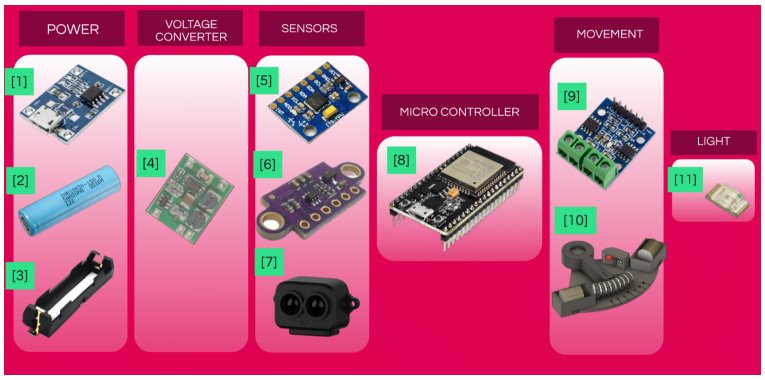
1. Bring up the first functional hardware of the spinning top.
2. Achieve continuous spinning of at least 30 seconds.
3. Ensure that hardware components operate both independently and synchronously.
4. Implement a basic algorithm integrating sensors and hammer actuations

Our team's vision is:

The long-term vision is to develop a mobile, spinning platform that can autonomously navigate a space, react to sensor inputs, and visually communicate data. In future stages, the platform could evolve into a swarm-like device for surveillance, entertainment, or educational use—pushing the boundary of what a small robot can do while in motion.

4. General Overview

The main components are shown in the figure attached:

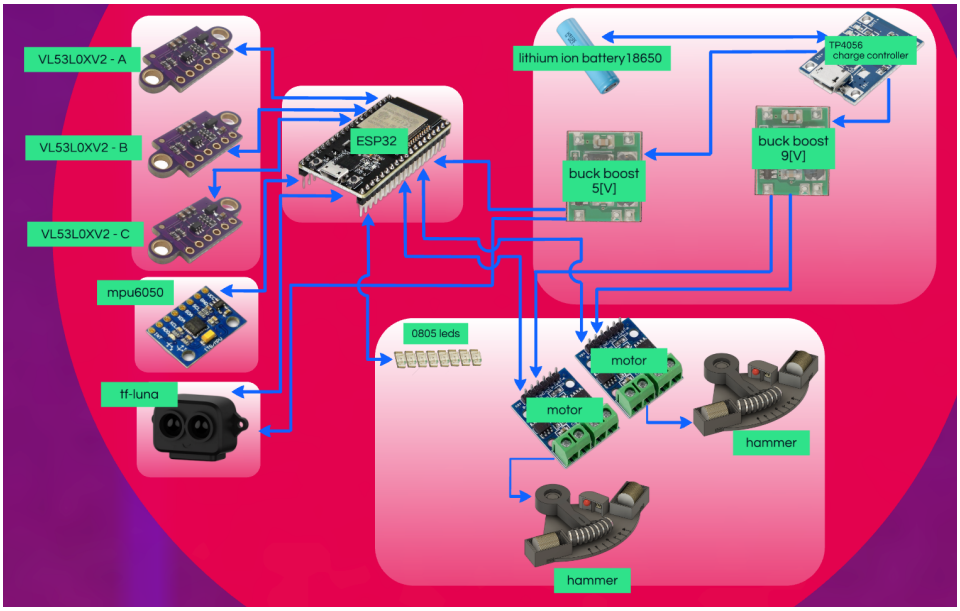


LEGEND

Index	Component	Amount
[1]	TP4056	1
[2]	Battery 18650	1
[3]	Battery House	1
[4]	Buck-Boost	2
[5]	MPU6050	1
[6]	VL53L0XV2	3
[7]	TF-LUNA	1
[8]	ESP32	1
[9]	Motor Driver	2
[10]	Hammer	2
[11]	Led	8

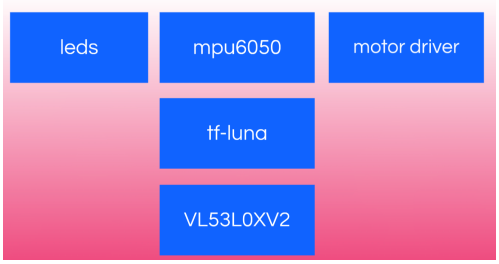
Before we explain the role of each component detailly (on next chapter), we will show a general view of the hardware and software blocks flow.

4.1. Hardware Block Diagram (overview)

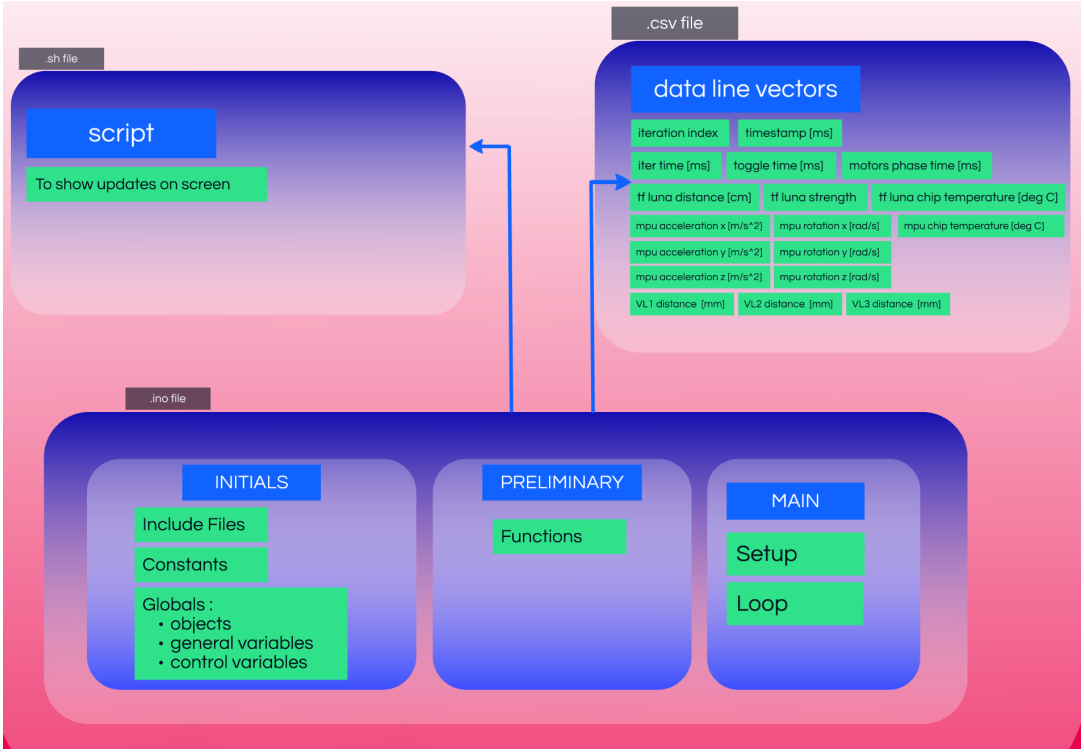


4.2. Software Block Diagram (overview)

Independent Software Block Diagram:



Integration Software Blocks Flow:



5. Components

5.1. ESP32 Dev Module

The ESP32 is the central component of our device, selected for its numerous features, including multiple GPIO pins, built-in Wi-Fi modules, low cost, and the ability to generate PWM (Pulse Width Modulation) signals for motor control. Additionally, it supports I2C and UART communication protocols, which are essential for interfacing with various sensors and peripherals. We programmed the ESP32 to test the basic functionality of each component both individually and as part of the integrated system, ensuring the correctness of our device's routing and overall design.

5.2. Charge Controller TP4056

The TP4056 is a linear charger module designed for single-cell lithium-ion batteries, providing a complete constant-current/constant-voltage charging process with high accuracy and safety features. It supports automatic recharge and includes protection features like over-temperature, over-voltage, and short-circuit protection. Status indicators, usually via LEDs, show charging status and fault conditions.
In the remote-controlled dreidel project, the TP4056 ensures safe and efficient charging of the lithium-ion battery powering the device. It integrates well with the ESP32, providing a stable power supply and safeguarding against power issues. Its compact size fits the limited space on the dreidel-sized board.
Implementation involves connecting the battery to the TP4056, with the module's output powering the ESP32 and other components. A USB power source is used for charging, and the status LEDs monitor the process.

5.3. Buck Boost

A DC-to-DC converter can have an output voltage magnitude that is either greater than or less than the input voltage magnitude. The input voltage range is 3V to 15V, and the adjustable output voltage can be set between 1V and 15V.
On our board, we used two different converters. One provides an output of 5V, which powers the ESP and the TF-Luna sensor. The second converter, with an output named V_{coil}, supplies power to the two motor drivers. Initially, we estimated V_{coil} to be 9V, but after further calculations, we determined the optimal value to be ???V (see the "Testing and Performance" chapter for more details).

5.4. Motor Driver L9110S

The L9110S 2-Channel motor driver module is a compact board designed for driving small robots. It features two independent motor driver chips, each capable of delivering up to 800mA of continuous current. Operating within a voltage range of 2.5V to 12V, it is compatible with both 3.3V and 5V microcontrollers.

On our board, we used two of these motor drivers, each receiving its PWM input from the ESP. These drivers operate hammers by creating a high-voltage drop across the hammers' two inductors.

5.5. Lithium Ion Battery 18650

We used a battery with a nominal voltage of 3.7V, which we positioned on the underside of the board.

5.6. VL53L0XV2 Sensor

The VL53L0X V2 is a time-of-flight (ToF) laser distance sensor that measures distances using an infrared laser. It provides high accuracy and fast response times, making it useful for robotics, drones, and object detection. The sensor communicates via I2C and has a measuring range of up to 4 meters, depending on surface reflectivity and lighting conditions. Its low power consumption and compact size make it ideal for battery-powered and space-constrained applications.
The VL53L0X V2, also known as "the cheap sensors," have a sampling rate approximately one-third that of the TF-Luna. To ensure a "fair" comparison, we use three VL53L0X V2 sensors. These sensors are powered by 3.3V supplied by the ESP. Since they transmit data to the ESP over the same communication line, they share the transmission time, with each sensor operating for one-third of the cycle while the other two remain inactive.

5.7. TF LUNA LiDAR Sensor

The TF-Luna is a compact time-of-flight (ToF) LiDAR sensor designed for accurate distance measurement. It operates using an infrared laser and can measure distances of up to 8 meters with high precision. The sensor supports both UART and I2C communication, making it compatible with various microcontrollers. It has a higher sampling rate compared to other budget ToF sensors like the VL53L0X V2, making it ideal for applications requiring real-time distance tracking, such as robotics, drones, and obstacle detection.
In our project, the TF-Luna sensor operates using the UART protocol. It receives a 5V power supply from the voltage converter.

5.8. MPU6050 Sensor

The MPU6050 is a 6-axis motion tracking sensor that combines a 3-axis gyroscope and a 3-axis accelerometer in a single chip. It communicates via the I2C protocol, making it easy to interface with microcontrollers like ESP32 and Arduino. The sensor is widely used in applications such as robotics, motion tracking, gesture detection, and stabilization systems. It operates on a 5V power supply, depending on the module version, and provides real-time motion data with high accuracy.

5.9. Hammers

The "Hammer Modules" are a unique component of our project. Each module consists of an arm that rotates around a pivot point, with magnets positioned at its end.
Additionally, there are two coil housings, each containing copper windings wrapped around iron cores. When an electric current flows through the coils, a magnetic field is generated, attracting the coils. As a result of the repeated impacts on the edges of the coil housings, the dreidel moves in a controlled manner across the surface.

6. Protocols

6.1. I2C

Introduction to I2C Protocol

I2C (Inter-Integrated Circuit) is a widely used serial communication protocol designed for connecting low-speed peripherals to microcontrollers and processors, including the ESP32 used in our project. This protocol has become a standard for communication between integrated circuits on a circuit board, making it the default method for connecting multiple devices within embedded systems.
Over time, I2C has gained popularity due to its simplicity, efficiency, and cost-effectiveness, making it an ideal choice for applications where devices need to communicate over short distances within a circuit board or a confined physical space.

I2C and ESP32:

The ESP32 microcontroller comes equipped with built-in peripheral interfaces, including hardware support for I2C communication. This allows efficient and reliable data transfer with various sensors, such as the MPU6050 (accelerometer & gyroscope) and VL53L0XV2 (distance sensor).

By default, the ESP32 assigns the following pins for I2C communication:
• GPIO21 → SDA (Serial Data Line)
• GPIO22 → SCL (Serial Clock Line)

The standard I2C frequency used with the Wire library is 100 kHz.

How I2C Works:

I2C operates using a two-wire interface:
• SDA (Serial Data Line) → Transfers data between the master and slaves.
• SCL (Serial Clock Line) → Synchronizes the data transfer, controlled by the master.

Since these lines are open-drain, they require pull-up resistors to maintain a high state when not actively driven low by a device. In our project, the pull-up resistors are integrated within the MPU6050 and VL53L0XV2 components and are connected to the SDA and SCL lines, as shown in the schematic (Chapter 8.1).

Master-Slave Architecture in I2C:

I2C follows one-to-multiple master-slave communication model:
The master initiates communication and controls the clock signal, while the slaves respond to the master's requests. Each slave has a unique I2C address, allowing the master to communicate with multiple devices using a shared bus.

I2C Setup in Our Project:

Our project consists of:
• 1 I2C Master (ESP32).
• 4 I2C Slaves (3 VL53L0XV2 distance sensors and 1 MPU6050 accelerometer & gyroscope sensor).

Since all slaves share the same I2C bus, they initially have the same default address. If multiple sensors have the same address, they will attempt to respond simultaneously, causing data conflicts. To prevent this, we configured the ESP32 to dynamically assign unique addresses to each sensor during their initialization.

Master-Slave Communication Process in Our Project:

1. The ESP32 (master) initializes the I2C bus.
2. It disables all sensors by setting their XSHUT pins to LOW to avoid address conflicts.
3. It activates each sensor one at a time, assigns a new I2C address, and then enables the next sensor.
4. Once all sensors have unique addresses, the ESP32 communicates with each sensor individually using its assigned address.
5. Each sensor responds only when addressed, ensuring smooth and conflict-free communication.

I2C Read/Write Process:

1. The ESP32 sends a command to a sensor at its specific I2C address.
2. The selected sensor responds with its measurement data.
3. The ESP32 reads and processes the data.
4. This process is repeated for each sensor.

The Role of SCL and SDA in I2C Communication

I2C communication relies on two main lines: SCL and SDA.

	SCL (Serial Clock Line)	SDA (Serial Data Line)
Purpose	Synchronizes data transfer.	Transfers data between the master and slaves.
Direction	Controlled by the master.	Bidirectional (data flows in both directions).
Function	The master generates clock pulses, and all connected slaves follow this clock to send/receive data.	The master sends a command to a specific slave, and the slave responds with data.
ESP32 Default Pin	GPIO22 (can be reassigned).	GPIO21 (can be reassigned).

How SCL and SDA Work Together

1. Start Condition:
   The ESP32 (master) initiates communication by generating a start condition, which is signaled by a high-to-low transition on SDA while SCL remains high.
   This informs all devices on the bus that a transmission is about to begin.
2. Addressing the Target Device:
   The master sends an address byte over the SDA line, specifying the I2C address of the intended slave device. This is followed by a read/write (1/0) bit, indicating whether the master intends to read from or write to the slave.
3. Clock Synchronization & Data Transfer Timing:
   The SCL line generates clock pulses, ensuring synchronized data transmission between the master and slave. The slave devices may hold the SCL line low (clock stretching) if they need more time to process data, temporarily pausing the master's transmission.
4. Data Transmission & Acknowledgment:
   The target slave recognizes its address, acknowledges receipt, and data transfer begins over the SDA line. Afterwards, data is transmitted in 8-bit packets (byte packets), where each byte is followed by an acknowledgment (ACK) or no-acknowledgment (NACK) bit:
      • ACK (Acknowledgment Bit): The receiver pulls SDA low to confirm successful reception of the byte.
      • NACK (No-Acknowledgment Bit): The receiver leaves SDA high, indicating the end of data transmission or an issue with data reception.
5. Stop Condition & Communication Termination:
   Once data transmission is complete, the master sends a stop condition, signaled by a low-to-high transition on SDA while SCL remains high.
   This releases the bus, allowing other devices to initiate communication when necessary.

This structured approach - where SCL synchronizing data transfer and SDA carrying the actual data, eventhough multiple sensors share the same I2C bus, ensures reliable and efficient communication between master and its slaves. This approach is reliable since it prevents data conflicts and ensures proper synchronization. And, efficient since it uses only two I2C lines.

6.2. UART

Introduction to UART Protocol

Universal Asynchronous Receiver-Transmitter (UART) is a widely used serial communication protocol that enables asynchronous, full-duplex communication between two devices, enabling continuous exchange between them – data can be sent and received simultaneously between two devices.
The UART interface consists of two main signal lines: TX (transmit) and RX (Receive).
Unlike I2C, which supports multiple devices on a shared bus, UART establishes a direct, point-to-point connection between a transmitter and a receiver.
TX (Transmit Data Line) : responsible for sending data to the receiver.
RX (Receive Data Line) : responsible for receiving data from the transmitter.

UART and ESP32:
The ESP32 microcontroller includes multiple hardware UART interfaces, allowing seamless communication with peripheral devices such as sensors, GPS modules, and displays.
UART simplifies implementation by eliminating the need for device addressing, but it is limited to one-to-one communication.
TX and RX work together to ensure reliable data transfer, following a structural format of start, data transmission, synchronization and stop conditions.
This structured integration ensures reliable, high-speed distance measurement from the sensor, contributing to the efficiency of our ESP32-based system.

How UART Works:
UART operates without a clock signal (asynchronous communication), relying on both devices being configured to the same baud rate (bits per second) to ensure data synchronization. This makes UART a simple and efficient protocol for real-time data exchange, commonly used in sensors, microcontrollers, and serial interfaces.

UART follows One-to-One Master-Slave communication Model, where the master controls the communication process and reads incoming data (data is transferred in packets), and the slave continuously transmits distance measurement data to the master.

Since UART does not support multiple devices on the same bus, each device requires a dedicated serial connection for continuous data transmission - TX/RX connections in master's device and respectively RX/TX connections in slave's device.
The sender always writes data over its TX line, and the receiver always reads data over it RX line.

Data Transmission Process:

1. Start Condition
   The transmitter sends a start bit to indicate the beginning of data transmission and this is detected by the receiver.
2. Data Packet Transmission
   The transmitter sends 8-bit data packets (bytes) at the preconfigured baud rate.
   Each byte consists of *start-bit (indicates the beginning of transmission) , Data-bits (5-9 bits, the actual data being transmitted), optional parity bit (for error detection), stop bit (1 or 2, signals the end of transmission).*
   The receiver captures these bytes and processes them accordingly.
3. Data Synchronization
   Both the transmitter and receiver must operate at the same baud rate (e.g., 115200 bps) to ensure proper timing. If the baud rates are mismatched, data corruption will occur.
4. Stop Condition
   The transmitter sends a stop bit, signaling the end of transmission and then, the receiver detects the stop bit and prepares for the next data packet.
5. Handling Data Flow
   Since the transmitter continuously transmits data, the receiver reads and processes incoming bytes in real-time. In case the receiver wants to transmit too, it has to send commands over the TX line.

UART Setup in Our Project:
In this project, we utilize UART communication to interface with the TF Luna LiDAR sensor (slave). The ESP32 (master) receives real-time measurements from the TF-Luna through the TX/RX serial interface. Baud rate synchronization (115200 bps) is essential for correct data interpretation.
The ESP32 and the TF Luna distance sensor supports both I2C and UART communication modes. However, to configure the TF Luna to operate in UART mode, its *Configuration Input* pin must be connected to 3.3[V], and the ESP32 need to be used with dedicated UART TX/RX serial connections: GPIO17_TX2/ GPIO16_RX2 respectively.
In master's device, the TX is used to send configuration commands to TF-Luna (if needed), while the RX is used to receive continuous distance measurements. This setup ensures seamless and real-time data exchange between the ESP32 and the TF-Luna sensor. At the beginning of the UART connection, the ESP32, initiates UART communication by setting the correct baud rate (both set to 115200 bps). Afterwards, TF-Luna continuously transmits distance measurements over the TX line, while the ESP32 receives data via the RX line and processes the sensor readings. If required, ESP32 can send commands to modify sensor parameters (e.g., frame rate, sensitivity). At the end, ESP32 integrates the TF Luna data into the overall system for further processing.

UART Read/Write Process and Error Detection Mechanisms:

UART communication follows a structured read and write process to ensure data is transmitted and received correctly between the ESP32 (master) and the TF Luna (slave).

The TF Luna continuously transmits (writes/ sends) data containing measurements to the ESP32, while the ESP32 reads/ receives this data and processes it accordingly.
The processing of the data is a decoding of the received data and using it for real-time measurement.

Although the TF Luna LiDAR sensor primarily operates in a continuous transmission mode, there are cases where the ESP32 needs to send commands to modify settings such as frame rate, measurement mode, or output format.

To ensure accurate communication, UART includes error detection mechanisms such as baud rate synchronization (both devices must use the same baud rate (115200 bps) to avoid data corruption), parity bit (optional, for basic error detection), checksum validation (TF Luna includes a checksum byte at the end of each data packet to verify data integrity), buffer management (the ESP32 checks Serial2.available() to ensure the full data packet is received before processing).

Comparison: UART vs. I2C:

Feature	**UART (TF LUNA)**	**I2C (VL53L0XV2, MPU6050)**
Bus Type	Point-to-Point	Multi-Device Bus
Clock Signal	Not Required	Required (SCL)
Number of Wires	2 (TX, RX)	2 (SDA, SCL)
Addressing	No Addressing Needed	Each Slave Needs a Unique Address
Data Transfer	Continuous Streaming	Master Requests Data
Speed	Adjustable Baud Rate (e.g., 115200 bps)	Typically Fixed 100 kHz / 400 kHz

6.3.WiFi

Introduction to WiFi Protocol

Wi-Fi (Wireless Fidelity) is a wireless communication protocol that allows devices to connect to a local network or the internet without physical cables. It operates based on IEEE 802.11 standards and uses radio waves to transmit data between devices and routers. Wi-Fi supports multiple frequency bands (2.4 [GHz], 5 [GHz]) and employs security protocols like WPA2 and WPA3 to protect data transmission. WiFi protocol allows high-speed, long-range, and reliable wireless communication.

WiFi and ESP32:
The ESP32 microcontroller includes built-in WiFi module.
It supports three modes:

- Station Mode (STA) : ESP32 connects to an existing WiFi network (like a router or hotspot).
- Access Point Mode (AP) : ESP32 acts like a WiFi access point, allowing other devices to connect to it.
- SoftAP + STA Mode : ESP32 connects to WiFi while also acting as an access point.

Client-Server Architecture Model
The server is responsible for listening request and sending responses, while the client initiates the request and waits for the server's response.
In our project, the server is the ESP32 and the client is another devices connected to the same router.
The server listens for incoming connections on a specific port (port 80 for HTTP), and the clients sends an HTTP GET request to the server.
The server processes the response and processes the data.

WiFi Setup in Our Project

In our project, we utilize WiFi to wirelessly transmit sensors' data collected from I2C and UART devices over the local network from ESP32 to local computers.
It means that the ESP32 operates in Station Mode (STA), where ESP32 and our computer both are connected to the same router.

WiFi supports multiple network communication protocols.
It means that WiFi does not replace I2C and UART and is used in addition to those protocols to transmit the data wirelessly after collecting it from sensors.
To transmit data wirelessly over the network we involve a local HTTP server that is created on ESP32 itself – i.e., the ESP32 acts like a local HTTP web server.

The ESP32 performs two main actions which does not involve an external server:

1. Writing sensors' data to SPIFFS storage (Internal Flash Memory on ESP32).
2. Reading the stored file and serving it through ESP32's web server over HTTP.

Local computers can access this server and get a CSV file that includes data of measurements (vector lines). The local computers act like local clients of the ESP32's HTTP web server and send HTTP requests to ESP32's local IP address (`http://ESP32_IP_ADDRESS/sensors_data.csv`).
The router forwards this request to ESP32 inside the local network. ESP32 processes the request and responds with the CSV file and then the router forwards ESP32's response back to the local computer.

How Data Collected from Sensors is Transferred via WiFi to A Local Device?

1. ESP32 Connects to the local network:
   ESP32 runs an Arduino appropriate commands, which connect it to the local WiFi network (the local router).
   It uses a credentials (SSID and PASSWORD) which are set in the code.
   The local router assigns the ESP32 a local IP address dynamically using DHCP (meaning that the IP address might change (unless we set a static IP).
   This IP address belongs to the local network and is accessible only by devices connected to the same router.
   ESP32 can be accessible from the Internet (Wide Area Network) only if a forwarding port is set up (we did not implement that).

2. ESP32 Starts the Web Server (AsyncWebServer) to allow clients to access a CSV file stored on ESP32's flash memory, which contains data about measurements :
   Several Arduino commands create an HTTP server on ESP32, using port 80.
   This web server runs directly on ESP32, making it a local web server.
   This means the ESP32 starts listening for incoming HTTP requests.
   During the code running, the ESP32 collects sensor data from I2C (VL53L0X, MPU6050) and UART (TF Luna). It also writes sensor data to a CSV file store and serve it from SPIFFS storage when requested.
   The CSV file never moves to another external server- it stays on ESP32's SPIFFS storage and is only served to clients that request it.

3. A client (computer/mobile) requests the CSV file, and ESP32 serves it over WiFi.
   Once ESP32 is connected to WiFi, it gets an IP address.
   ESP32 creates, stores and reads a CSV file from its SPIFFS (internal flash storage) and ESP32's HTTP web server waits for requests on `http://<ESP32_IP>/sensor_data.csv` from the local devices in the network. When ESP32 gets an HTTP request, it sends it over HTTP to the local computer.

   We created 2 different ways to access the ESP32's HTTP WEB Server:
   a. The first method is via running a Bash script- which automates real-time monitoring of CSV file stored on ESP32's SPIFFS.
   It does this via *curl* command which requests the file from the HTTP server:
   curl -s -o "sensors_data.csv" http://$ESP_IP_ADDRESS/sensors_data.csv
   The ESP32 reads the file from the SPIFFS and sends it over HTTP to the local computer.
   The script fetches the CSV file on the HTTP server and monitors changes.
   If the CSV file is updated, the script detects new data and logs it.

   b. The second method is viewing data in local browser – refreshing the browser manually updates the displayed date.
   By opening a local browser (Chrome, Firefox, Edge, etc.) in a device that is connected to the same local network, i.e., local computer, and access ESP32's HTTP web-server via entering the one of the following lines on browser's address-line:
   - To watch the last updated file:
     `http://ESP32_IP_address`
   - To download the last updated file:
     http:// `ESP32_IP_address` */sensors_data.csv*

   When we refresh the page, the browser sends a new HTTP GET request from the local computer to the ESP32's HTTP web server. The ESP32 responds with the latest version of `sensors_data.csv`.
   If new line was written to SPIFFS (internal storage of ESP32), the CSV file contains updated values. Refreshing manually fetches the newest data.

Utilization of Reading from SPIFFS Via WiFi

WiFi is only used when another device (computer, browser, or script) requests the data from ESP32's web server.

ESP32 collects sensors' data from I2C (VL53L0XV2, MPU6050) and UART (TF Luna), formats it into CSV format and writes it to `sensors_data.csv` inside SPIFFS (internal flash storage). This means the data is not sent to an external/remote server and remains stored locally inside the ESP32. The SPIFFS acts like a small local database and no WiFi involved.

The ESP32 acts as an HTTP web server, and the local device (or browser, script) acts as a client.
ESP32's HTTP web server listens for requests on port 80.
The client uses WiFi only to read (receive) data from SPIFFS, not to write to SPIFFS.
ESP32's SPIFFS does not use WiFi to write data to ESP32's HTTP local server. Instead, SPIFFS writes and stores data internally in ESP32's flash memory (local storage).

When the client (local browser or Bash script in our case) uses WiFi to read data from SPIFFS, it means that it sends an HTTP GET request to the web server when it wants to access the file.
The Bash script does this by fetching the file periodically from ESP32's web server and compareing it for updates via `curl` command.
The script runs on the local device (the local computer) and logs updates whenever new data is added to SPIFFS.

The Browser does it via accessing `http://ESP32_IP_address` on the address line.
When ESP32's HTTP web server receives the HTTP GET request, the ESP32 reads the current file on its SPIFFS and then serves it over the WiFi by sending the file to the local device in a response.
ESP32 acts as a web server that delivers the file.

7. Mechanical Overview

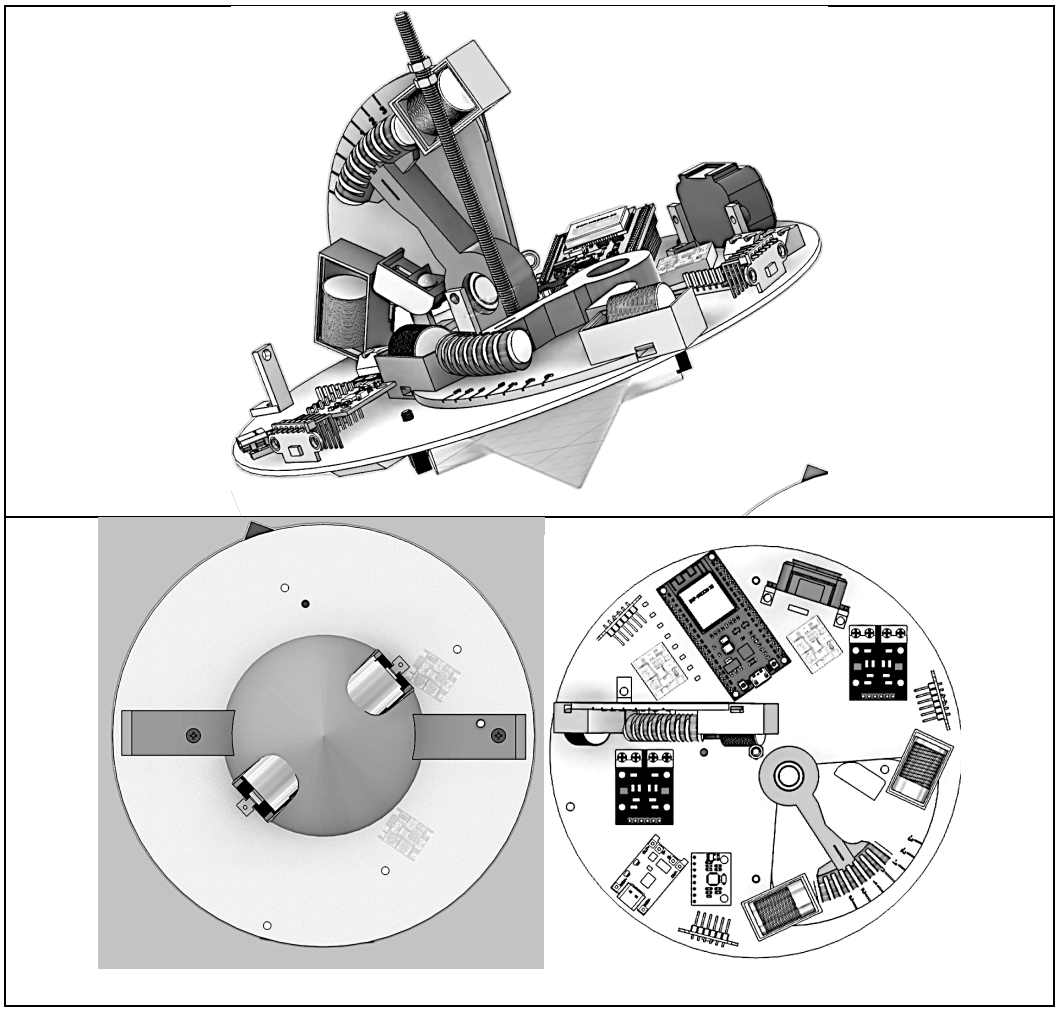
7.1. Placing and Sizing

We chose a round PCB with a diameter of 162 mm, which represented a compromise between wanting a small and compact spinning top and having enough space for all the components. First, we placed the larger and heavier components. We aimed to evenly distribute the weight with radial symmetry, with mass concentrated near the edges to provide greater momentum for spinning. We decided to place the battery under the PCB in a horizontal position. Although we considered placing it vertically on the axis of rotation of the top, we opted not to, to better distribute the weight towards the edges. Additionally, we tried to position components sensitive to noise (such as the sensors) as far as possible from noisy components (such as the coils).

7.2. 3D model

The 3D model is designed by Fusion360 modeling software.

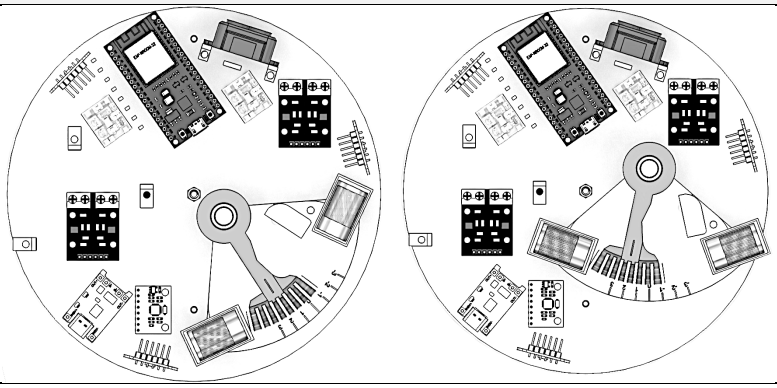
We created the model of the PCB, handles and cones. However, other models were taken from other sources (hammers were taken from previous spinning-top's group, screws were taken from Fusion360 local's library and other electrical components were taken from the Internet).



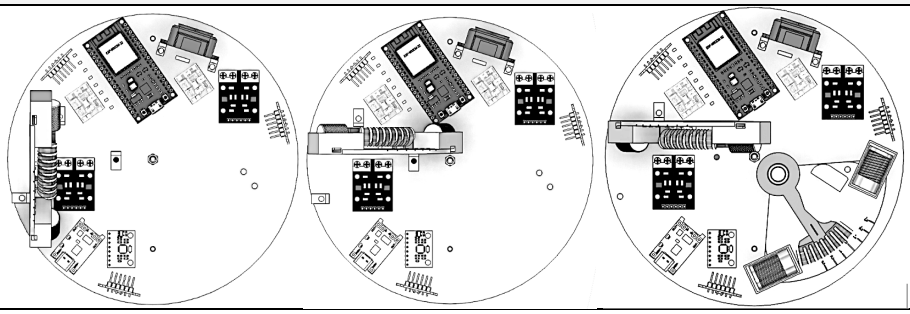
7.3. Optional Positions

Our device supports 2 positions for the horizontal hammer and 3 positions for the vertical hammer, for a total of 6 possible positions. The idea behind this flexible design is to examine different options to determine the optimal pair of positions for spinning-top's rotation.
The positions of the horizontal hammer represent the options of the horizontal hammer to be pounded the angular direction, and in radial direction.
The positions of the vertical hammer represent the options of the vertical hammer to be pounded to the center of the circle, to the out of the circle and to the edge of the circle.

2 Positions for Horizontal Hammer



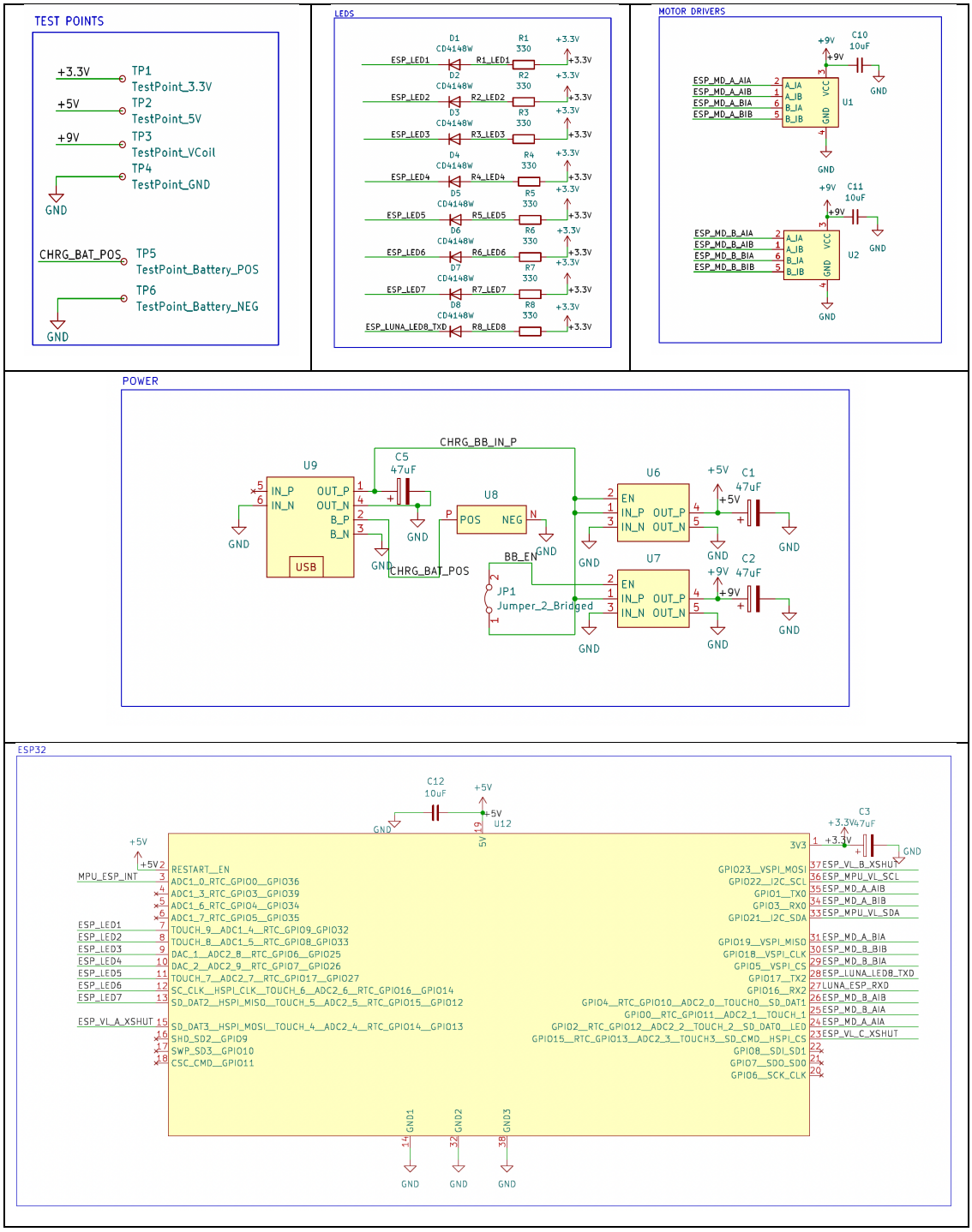
3 Positions for Vertical Hammer

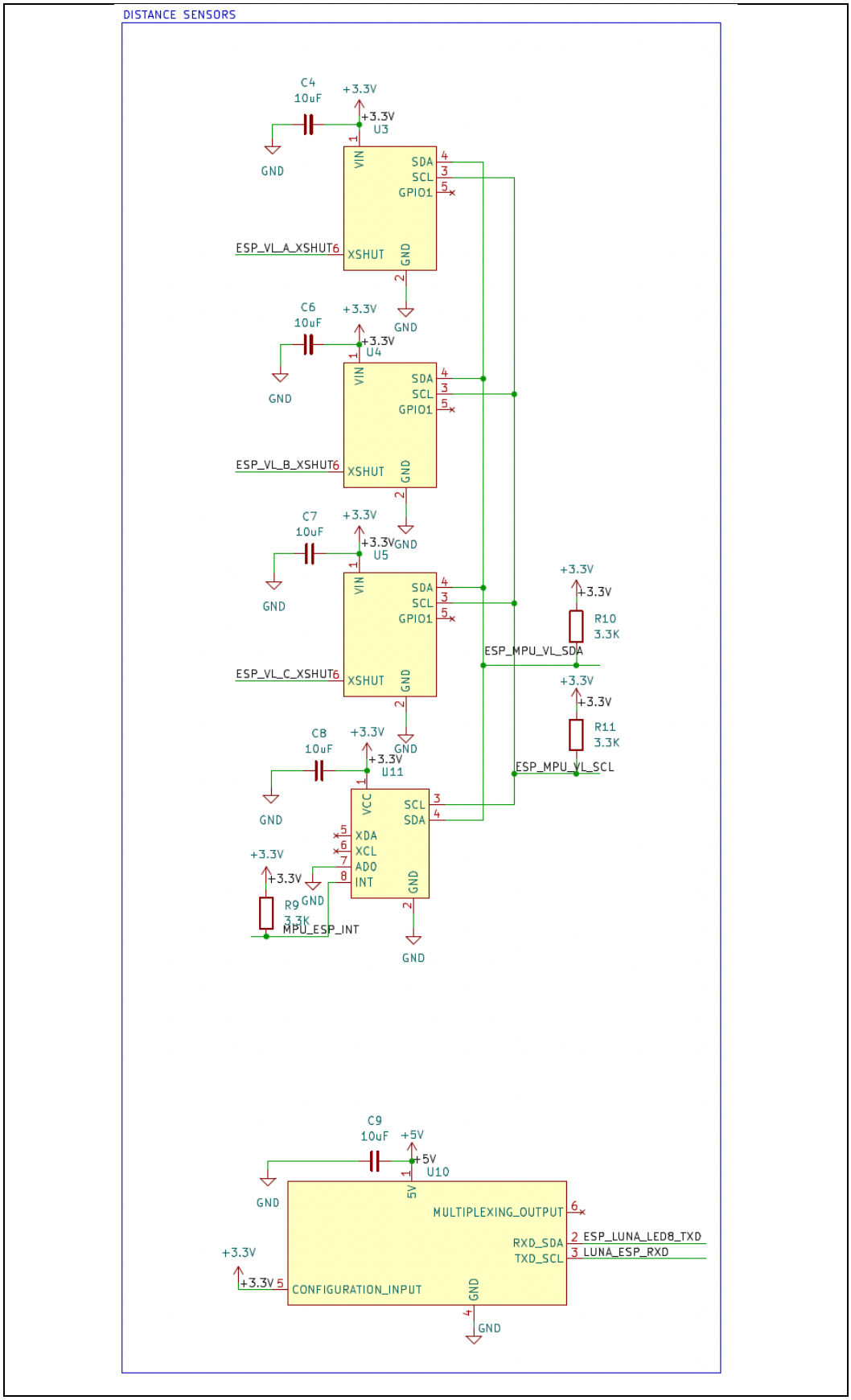


8. Electrical Overview

8.1. Schematic

The full schematic file is attached to [GitHub repository].




DISTANCE SENSORS
C4
10uF
+3.3V
U3
VIN
SDA
SCL
GPIO1
XSHUT
GND
ESP_VL_A_XSHUT
C6
10uF
U4
ESP_VL_B_XSHUT
C7
10uF
U5
ESP_VL_C_XSHUT
C8
10uF
U11
VCC
XDA
XCL
ADO
INT
R9
3.3K
MPU_ESP_INT
R10
3.3K
ESP_MPU_VL_SDA
R11
3.3K
ESP_MPU_VL_SCL
C9
10uF
+5V
U10
MULTIPLEXING_OUTPUT
RXD_SDA
TXD_SCL
ESP_LUNA_LED8_TXD
LUNA_ESP_RXD
CONFIGURATION_INPUT

Power Block:
The battery serves as the power source for the entire device. It is connected to a charge controller, which regulates the battery voltage, prevents harmful voltage spikes, and enables battery charging. From there, power is distributed to buck-boost DC/DC converters, which generate the required 5V and V_coil (9.13[V]) voltages for the other components in the device.

Distance Sensor Block:
We have five sensors of three different types: three VL sensors, an MPU sensor, and a TF-Luna sensor. The VL sensors are powered by the 3.3[V] output of the ESP, while the TF-Luna sensor is powered via a voltage converter.
The three VL sensors transmit data over the same line to the ESP, sharing it equally in each cycle so that each sensor transmits for one-third of the time. This was necessary due to the limited number of available GPIO inputs on the ESP.
Additionally, we added several pull-up resistors because some inputs and outputs cannot receive a pure logical "1." The VL sensors and the MPU sensor communicate using the I2C protocol, while the TF-Luna sensor operates via UART.

ESP Block:
The ESP is the heart of our board, serving as both a processor and a controller. It receives data from the sensors, processes it according to the embedded code, and, as a result, controls the motor drivers, which supply current to the inductors.

Motor Driver Block:
The motor drivers receive signals from the ESP and, based on them, supply current to the coils in the "forward" or "reverse" direction or stop the current flow altogether. They receive their power supply from the voltage converter, which outputs -V_coil (9.13[V]).

LED Block:
All LEDs are connected on one side to a GPIO pin on the ESP, except for LED number 8, which shares the same line with the TF-Luna sensor. This setup prevents LED 8 and the TF-Luna from operating simultaneously.
The LEDs are wired with inverted logic because it is easier to sink current into the ESP than to source it. Additionally, each LED is connected in series with a resistor to reduce the voltage drop across the LEDs and prevent them from burning out.

Test Points Block:

The test points block shown in the schematic is an essential part of the circuit's development, testing, and maintenance processes. Test points are predefined connections on the printed circuit board (PCB) that provide convenient access for electrical measurements during system development, quality control in manufacturing, and troubleshooting in future maintenance.
This block defines several test points related to the power supply and battery charging system. TP1, TP2, and TP3 represent key voltage nodes within the system. TP1 provides access to a 3.3[V] rail, TP2 to a 5[V] rail, and TP3 is connected to a node labelled V_Coil, which allows monitoring of the voltage at this specific junction. TP4 serves as a ground (GND) test point, acting as a reference for voltage measurements.
Additionally, the block includes two test points associated with the battery system: TP5 and TP6. TP5 is connected to CHRG_BAT_POS, representing the positive terminal of the charging system or battery, while TP6 corresponds to the negative terminal or ground. These test points enable monitoring of the battery’s charging state and related voltage levels, ensuring proper power delivery within the system.

8.2. Layout

The PCB layout process was carried out using KiCad, following a systematic and structured workflow. The layout design was based on an accurate mechanical representation that was first modeled in Fusion360 and later refined through electrical design in KiCad.

Mechanical Foundation with Fusion360

The process began in Fusion360, where a 2D model of the PCB wad derived from a full 3D model. This model included: component frames (outlines for mechanical placement), drill locations and dimensions, PCB outline geometry.
This mechanical drawing was exported as a .DXF file, which is used to import precise mechanical constraints into KiCad for layout referencing. The DXF ensures mechanical accuracy and alignment with enclosures or mechanical interfaces.

Schematic Capture in KiCad

After the mechanical model, we constructed the electrical design via custom symbol library and custom footprint library. Each component in the circuit was assigned a unique symbol and corresponding footprints were created and linked to each symbol.
Once the schematic was completed, a .NET file (netlist) was generated. This netlist defines all electrical connections between the component pins and serves as the input for layout.

Layout Creation in KiCad

With both the mechanical and electrical bases ready, the layout phase included:

- Importing the .DXF file to define the board outline, mounting holes, and component placement frames.
- Importing the .NET file, which generated ratsnest (airwires) showing all required electrical connections.
- Component Placement: Components were places precisely according to the DXF frames.

This PCB was design using two layers: Top Layer (F.CU) and Bottom Layer (B.Cu).
This approach provides a balance between complexity and cost.
One layer handles signals and power, while the other serves as ground plane and additional routing.

<u>Routing Considerations and Interferences Reduction</u>

When electrical signal travels through a PCB trace, a signal degradation might occur. It means a loss of quality, strength or clarity of the electrical signal. It can result in slower signals, incorrect logic levels (0 or 1), crosstalk or noise, data errors. The further the signal travels, the more distorted or weaker it becomes. The causes of signal degradation are long trace lengths (more resistance, more signal loss), high frequencies (more sensitive to interference), impedance mismatch (can cause reflections), poor routing (like sharp 90-degree angles), too many vias.

Tracks were routed to avoid crossing lines and minimize interference:
Design rules were applied to control minimum trace width, spacing, via sizes, etc.
The design rules consist of 3 categories:

1. Net Classes : separated power, signal and ground nets with specific rules (clearance, trace width).
2. Constraints: defined physical constraints such as annular width, copper-to-hole clearance, etc.
3. Violation Severity: set to Error or Warning to detect and prevent design mistakes.

ESP32 Antenna Considerations:
The ESP32 includes an onboard antenna, which must be treated carefully.
It means that no copper or traces should be routed under the antenna zone, and a clear area around the antenna should be maintained for optimal signal transmission.

Copper Bending Angle Considerations:
When a fast signal hits a sharp corner, part of the signal can reflect back, creating noise, data errors, or even EMI (electromagnetic interference).
Hence, we used smooth 45-Degree Trace Angles to prevent these problems.

Vias Usage Considerations:
In addition, we used vias to allow smart routing for escaping tight spots or avoid crossing traces, to enable layer switching that is essential in multilayer boards, and to help with layout clarity and reduce interference between traces.
We minimized via's usage and length, since overusing vias can cause signal degradation, especially if not handled properly.

- A via introduces a small change in impedance, so high-speed signals can partially reflect at that point.
- every via add a bit of parasitic inductance and capacitance, which can distort fast signals or introduce delay.
- unused sections of a via (long via stubs, like in multilayer boards) can resonate and act like antennas – which is very bad in RF.

Hence, the best practices to reduce signal degradation are to minimize the number of vias for fast signals, keep vias' lengths short, avoid unnecessary transitions between layers.

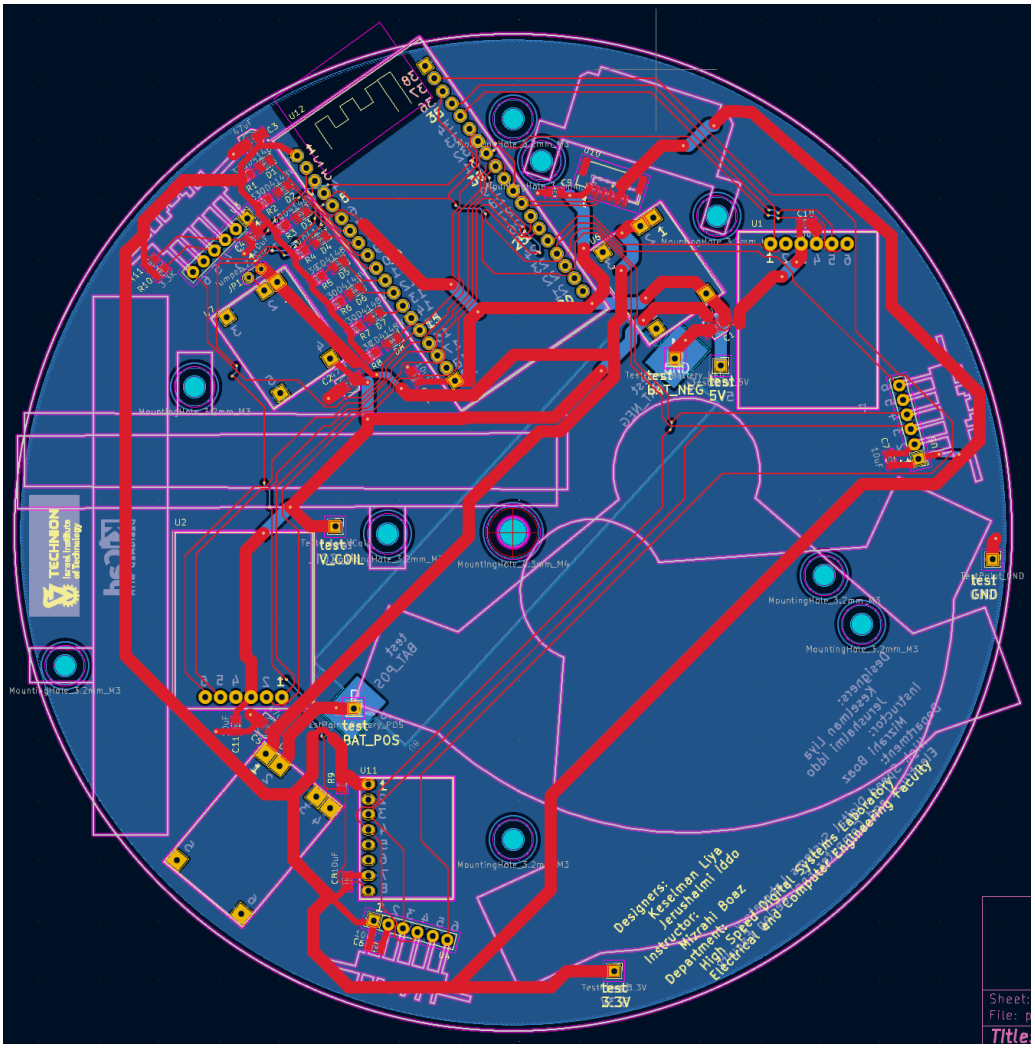
Area Around Drills Considerations:
When a hole is drilled in a PCB (for mounting, or for through-hole component), we must leave a safe area around it. This safe area called clearance.
Clearance around drills means: the distance between the edge of the drilled hole and the nearest copper features (like traces, pads, or ground pours).
Clearance around drills is important to prevent short circuits, since if copper (a trace or pad) is too close to the edge of the hole, and the drill shifts slightly during manufacturing, it can cut into the copper. That could create a short between signals or power and ground. Drill machines aren't perfectly accurate. They have a small allowed error (called drill tolerance). We had to set clearance to make sure that even if the hole is slightly off, it doesn't damage nearby copper. Also, having copper too close to the hole weakens the PCB mechanically.
On KiCad settings we used common clearance value: Copper to hole clearance = 0.25 [mm]

KiCad Settings Configuration

Layer Stackup and Materials	As defined in the Physical Stackup: Material: FR4 core (standard PCB dielectric). Total thickness: 1.6 [mm]. Solder Mask and paste layers defined with standard tolerances.
Board Design Finishing Settings	No castellated pads or plated edges were used. Copper finish and edge connector options were left as none for simplicity.
Solder Mask & Paste Settings	Solder mask expansion: 0.38 [mm]. Tent vias enabled to protect vias from oxidation. Clearance and web width values were based on manufacturer recommendations.
Predefined Sizes and Classes	Trace widths and via dimensions were predefined to match fabrication limits and power needs. Differential pairs left unused in this design.

(*) Screenshots of KiCad's settings for design rules are attached to GitHub repository.

Scheme of the Layout:



Production and Gerber File Generation (JLCPCB)

After completing the layout, the design was prepared for manufacturing by exporting the necessary files in **Gerber format**. These files are an industry-standard file format used to describe the various layers of a PCB design, and available for generation via KiCad by menu options.
The PCB manufacturer we worked with is JLCPCB, and they use the Gerber files to fabricate and assemble the board accurately.

Each file represents a different aspect of the PCB:

- Top and Bottom Copper layers **(.GTL / .GBL)**
- Top and Bottom Solder Mask **(.GTS / .GBS)**
- Top and Bottom Silkscreen **(.GTO / .GBO)**
- Drill files **(.TXT or .DRL)**
- Board outline **(.GML or .Edge.Cuts)**

This process bridges the gap between **digital PCB design** and **physical PCB fabrication**, allowing our design to be manufactured reliably by a professional fabrication service.

9. Testing and Performance

9.1. Independent Testing

Electrical Testing:

After receiving the PCB from manufacturing, we used a resistance meter to check for shorts or open circuits between all connection points, ensuring they matched our design.

Once some of the electrical components and their connections were soldered onto the board, we carefully connected each component one by one. We verified that each component received the correct supply voltage, that each input was properly connected to its designated line, and that no component placement interfered with others.

Finally, we rechecked for any unintended shorts or open circuits between various points. Once all these checks were completed, we proceeded to flash the ESP with code and observed whether it influenced the system as expected.

Component Software Testing with Arduino

Before integrating all components onto the PCB, each module and sensor was independently tested using an Arduino board to verify its basic functionality. This included checking whether sensors could properly communicate (via I2C, UART) and whether they responded with valid data on the Serial Monitor.

Digital output pins were toggled HIGH/LOW to ensure LEDs turned on/off as expected, and results were shown on the Serial Monitor of Arduino IDE.

I2C sensors (3 VL53L0XV2 and 1 MPU6050) received addresses dynamically to avoid data transmission conflicts (explained detailly on chapter 6.2 I2C). We examined the ability of each sensor to extract required several parameters (for 3 VL's sensors: distance from an object [mm], and for MPU6050 acceleration and gyroscope on X,Y,Z axis and chip temperature).
We presented the measurements on the Serial Monitor of Arduino IDE.

The UART sensor (1 TF-Luna LiDAR Sensor) communicates with the ESP32 when both set to 115200 [bps] baud rate. We examined the ability of the sensor to extract several parameters from the environment:
distance from an object [in cm units], signal strength, chip temperature [in Celsius degrees].
We presented the measurements on the Serial Monitor of Arduino IDE.
(*) Note: the distance from TF-Luna sensor is unreliable when strength < 100 or strength = 65535 (overexposure).

Motor Drivers were checked by rotating the motor clockwise and counterclockwise.
By alternating the state of digital output pins of the same channel we could rotate motor forward and backward.
When one edge of a channel is toggled HIGH and the second edge is toggled LOW, and then we change it to the opposite states (the first edge is toggled LOW and the second edge is toggled HIGH), we make the current to flow in an opposite direction, so it moves the hammer on the board.
We tested the motor movement by sending digital signals to confirm that motors responded correctly in terms of speed and direction.

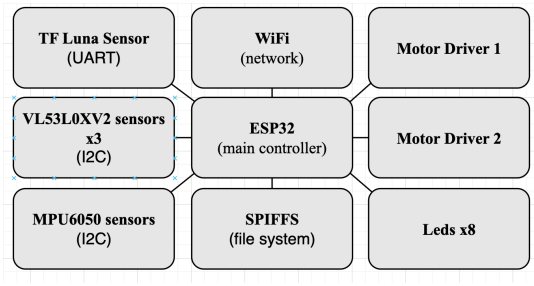
This early-stage testing helped confirm that each component functioned reliably in isolation, minimizing debugging time after full PCB assembly.

(*) full code to each component is attached.

9.2. Integration Test

BIST (Build-In Self-Test)

BIST Architecture – Spinning Top Hardware



The BIST code was written as part of the final production and testing phase to ensure hardware integrity and sensors' accuracy. It gives a comprehensive diagnostic and data collection routine designed for the *Spinning-Top Hardware* project and enables real-time validation of sensor connections, motor drivers, and wireless functionality on an ESP32-based embedded system.

This software validates that all sensors respond correctly, data sampling works at the expected rate, motors operate with accurate phase timing, WiFi and filesystem features are functional for logging purposes.

In addition, the BIST allows the ESP32 to communicate with various sensors and peripheral devices using industry-standard protocols (I^2C, UART, and WiFi). These protocols allow seamless data acquisition, motor control, and real-time diagnostics.

We designed the structure of the BIST to allows modularity and scalability (components can be tested independently). We considered in the design a robust data formatting and timestamping, for synchronization and deeper system analysis. In addition, we considered in the design a debug-mode and selective component testing, to make it ideal for rapid iteration and hardware diagnostics.

Throughout the main loop of the BIST code, sensor readings, timestamps, and operational debug values are collected into a structured string buffer.
Each line follows a CSV (Comma-Separated Values) format, including:

- Iteration index and timestamp.
- Motor timing data.
- Sensor readings from TF Luna, MPU6050, and all three VL53L0X units.

Once a certain number of lines are collected (as defined by BUFFER_SIZE), the buffer is flushed to a file named /sensors_data.csv stored in the ESP32's SPIFFS.

There are two primary methods to retrieve and inspect this CSV file:

1. Over Terminal (running Bash Script):
   A companion .sh script automates downloading the file over serial. By running it on terminal it monitors data output in real time, and also saves the stream to a local CSV file on your development machine.
2. Over a Browser (HTTP Server):
   After powering up the device and connecting to WiFi, the ESP32 prints its IP address to the Serial Monitor. Navigating to this address via any browser (e.g., http://192.168.x.x) opens a page where the latest CSV can be downloaded.

BIST's structure:

1. System Initialization:
   The system initializes a variety of component including *WiFi* and *File System (SPIFFS)* for network-based interaction and local data storage, *LEDs* for visual status indicators, *sensors* (tfLuna (ToF), VL53L0XV2 (x3 distance sensors), and MPU6050 (accelerometer & gyroscope)) and *motor drivers* (two channels, four control pins each).
2. Hardware Definitions and Configurations:
   Constants and GPIO pin mappings are defined for each hardware module.
   Macros and helper functions (e.g., FILLARRAY) assist in repetitive array initializations.
   Configuration flags allow developers to enable/disable hardware components individually to speed up testing.
3. Data Collection and Buffering
   Sensor data is sampled periodically and stored in a cyclic buffer (dataBuffer[]), which allows for efficient storage in chunks.
   Sample fields include:
   - Timestamps, iteration timing, and motor phase timing
   - Sensor readings from all components (ToF distances, acceleration, rotation, temperature).
   - A CSV-formatted string is built to be stored in the file system for further analysis.
4. Self-Test Logic and Execution
   The main loop likely handles:
   - Sensor setup verification (with per-sensor flags like is_done_setup_mpu).
   - Real-time data polling.
   - Writing results to a file (/sensors_data.csv).
   - Reporting via Serial/Network (debugging purposes).
   - Conditional execution based on actual hardware connected (controlled by flags like is_tfLuna_connected, is_md2_connected, etc.).

Limitations Observed During Software Testing

While most components functioned as expected in isolation, some integration issues were observed. For example, LED8 failed to operate when the TF Luna sensor was active, and Motor Driver A could not function reliably due to GPIO conflicts. These limitations are discussed in detail in the "Results and Conclusions" chapter, along with proposed solutions.

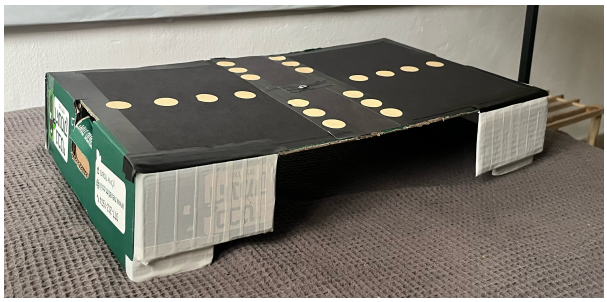
Spinning-Top Balance Assessment and Stabilization

During the integration testing phase, we encountered a significant imbalance in the spinning top's weight distribution. This imbalance negatively impacted its stability and rotation efficiency. To address this issue, we initially attempted to balance the spinning top by attaching small amounts of modeling clay to its edges. However, this method proved insufficient in achieving the desired stability.
To find a more effective solution, we sought a material that was soft, slightly adhesive, and sufficiently dense, allowing for easy adjustment and redistribution as needed. After evaluating various options, we selected plasticine as the preferred material for fine-tuning the spinning top's weight distribution.

Balancing Procedure

To determine the optimal placement of the plasticine, we designed a simple yet effective balancing device. This device consisted of a meshed metal basket and two low-friction fishing lines. The spinning top was placed on its side within this setup and given a small push, allowing it to rotate like a pendulum. When the rotation ceased, the heaviest side naturally settled at the lowest point. Based on this observation, we adjusted the weight by either adding plasticine to the upper side or removing it from the lower side. This process was repeated iteratively until the spinning top exhibited more uniform weight distribution. The criterion for successful stabilization was that, upon stopping, the spinning top would settle randomly in different positions rather than favoring a particular side.
Despite our best efforts within the given time frame, the spinning top still exhibited a tendency to topple after being spun.

To further improve its stability, we devised an additional support mechanism. This secondary stabilization device was constructed from a piece of cardboard with a central hole reinforced by a washer for added durability. The threaded rod serving as the spinning top's handle passed through this hole, providing external support without impeding its ability to rotate freely.
This solution successfully prevented the spinning top from falling over prematurely while still allowing it to exhibit rotational momentum.

First Stabilization Device we built	Secondary Stabilization Device
	

Recommendations for Future Iterations

Based on our findings, we recommend that future versions of the spinning top incorporate a more precise weight distribution strategy from the outset. Integrating weight-balancing considerations into the design phase would significantly improve the spinning top's performance and stability, reducing the need for extensive post-fabrication adjustments. By optimizing mass distribution and refining the support mechanisms, future iterations can achieve greater rotational efficiency and prolonged stability.

9.3. Simulations

Signal-to-Noise Ratio Test

To evaluate the quality of the DC voltage output, we measured the Signal-to-Noise ratio (SNR) under the following conditions:

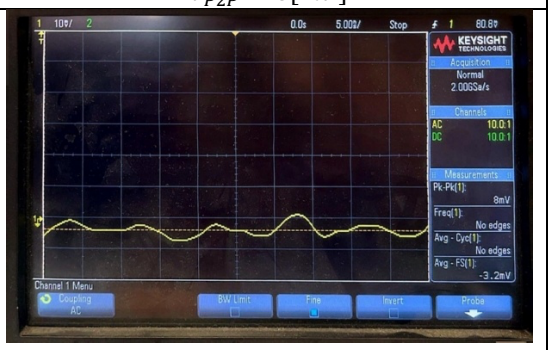
1. Charge controller connected to a charging source:
    1.1 DC-DC converter is disconnected from board, in cases with and without a resistive load.
2. Charge controller is disconnected from the charging source:
    2.1 DC-DC converter is connected to board, in cases with and without a resistive load.
    2.2 DC-DC converter is disconnected from board, in cases with and without a resistive load.

MARKS:

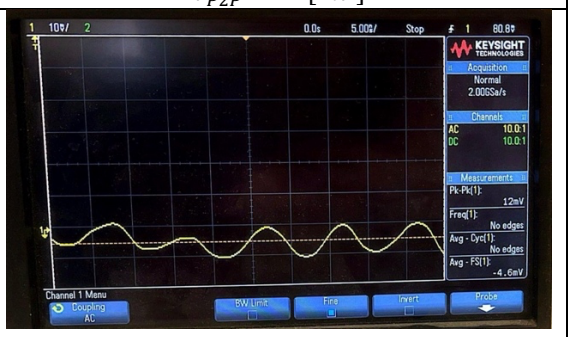
1. In all tests, battery is connected to board.
2. When DC-DC converter is connected, its output is tuned to 9[V] , denotes V_{coil}.
3. When resistive load is connected, loads the output of charge controller.
4. SNR calculation corresponds the formula $SNR_{dB} = 10\log_{10}\left(\frac{P_{signal}}{P_{noise}}\right) = 20\log_{10}\left(\frac{V_{signal}}{V_{noise}}\right)$.

Charge controller connected to the charging source Without DC-DC Converter:

- No Load:

The measured DC voltage was: $V_{DC} \approx 4.056[V]$	The measured average voltage noise was: $V_{noise,avg} \approx -3.2[mV]$ The measured Peak-to-Peak voltage was: $V_{P2P} \approx 8[mV]$
	
calculation of the SNR derives: $SNR_{dB} = 20\log_{10}\left(\frac{4.056[V]}{8[mV]-3.2[mV]}\right) \approx 58.537\ [dB]$ It indicates that the noise is 0.1183% of the DC signal ($\frac{1}{10^{\frac{58.537}{20}}} \times 100\% = 0.1183\%$).	

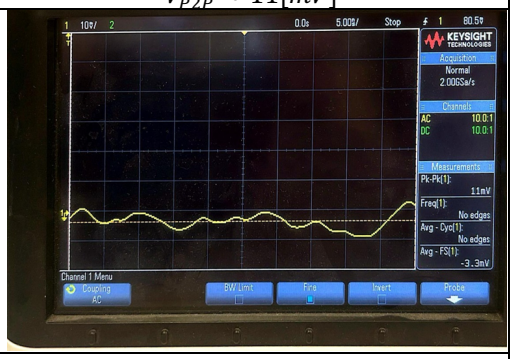
- With $29[\Omega]$ Load:

The measured DC voltage was: $V_{DC} \approx 4.038[V]$	The measured average voltage noise was: $V_{noise,avg} \approx -4.6[mV]$ The measured Peak-to-Peak voltage was: $V_{P2P} \approx 12[mV]$
	
calculation of the SNR derives: $SNR_{dB} = 20\log_{10}\left(\frac{4.038[V]}{12[mV]-4.6[mV]}\right) \approx 54.738\ [dB]$ This corresponds to 0.1832% noise relative to the DC signal ($\frac{1}{10^{\frac{54.738}{20}}} \times 100\% = 0.1832\%$).	

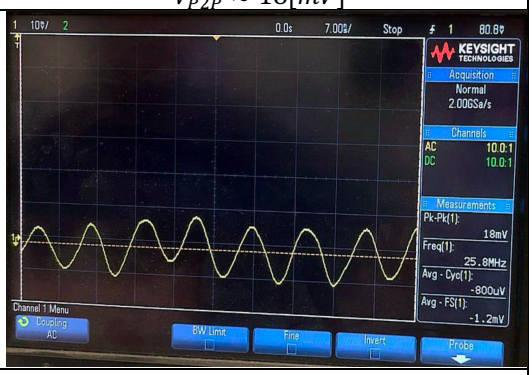
Charge controller is disconnected from the charging source:

- Without DC-DC converter:

- No Load:

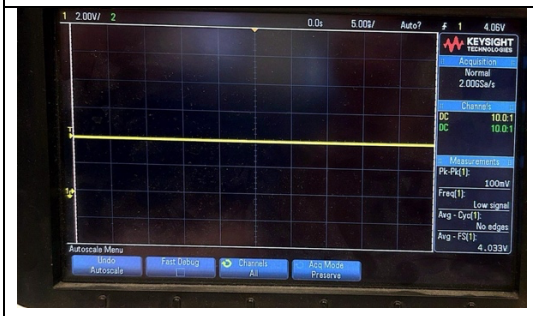
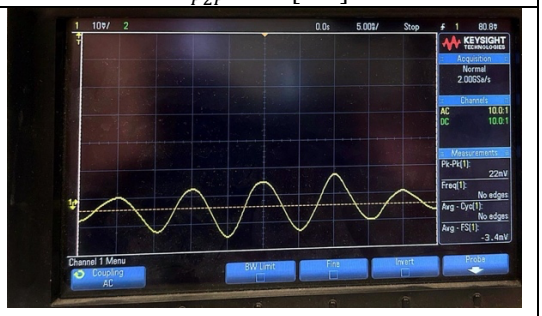
The measured DC voltage was: $V_{DC} \approx 4.024[V]$	The measured average voltage noise was: $V_{noise,avg} \approx -3.3[mV]$, The measured Peak-to-Peak voltage was: $V_{P2P} \approx 11[mV]$
	
calculation of the SNR derives: $SNR_{dB} = 20\log_{10}\left(\frac{4.024[V]}{11[mV]-3.3[mV]}\right) \approx 54.3633\ [dB]$ This corresponds to 0.1913% noise relative to the DC signal ($\frac{1}{10^{\frac{54.3633}{20}}} \times 100\% = 0.1913\%$).	

- With Load of $29[\Omega]$ resistor:

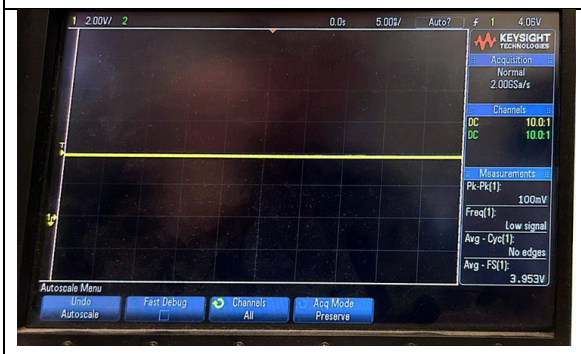
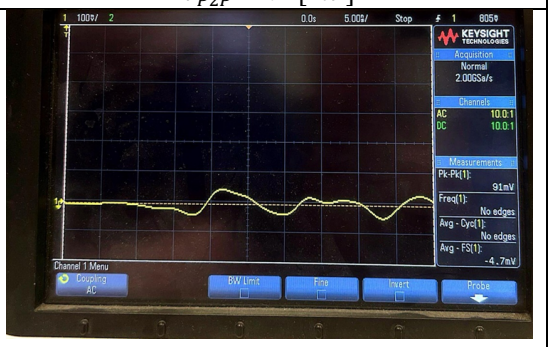
The measured DC voltage was: $V_{DC} \approx 3.994[V]$	The measured average voltage noise was: $V_{noise,avg} \approx -1.2[mV]$, The measured Peak-to-Peak voltage was: $V_{P2P} \approx 18[mV]$
	
calculation of the SNR derives: $SNR_{dB} = 20\log_{10}\left(\frac{3.994[V]}{18[mV]-1.2[mV]}\right) \approx 47.5219\ [dB]$ This corresponds to 0.4206% noise relative to the DC signal ($\frac{1}{10^{\frac{47.521}{20}}} \times 100\% = 0.4206\%$).	

- With DC-DC converter:

- No Load:

The measured DC voltage was: $V_{DC} \approx 4.033[V]$	The measured average voltage noise was: $V_{noise,avg} \approx -3.4[mV]$, The measured Peak-to-Peak voltage was: $V_{P2P} \approx 22[mV]$
	
calculation of the SNR derives: $SNR_{dB} = 20\log_{10}\left(\frac{4.033[V]}{22[mV]-3.4[mV]}\right) \approx 46.722\ [dB]$ This corresponds to 0.4612 % noise relative to the DC signal ($\frac{1}{10^{\frac{46.722}{20}}} \times 100\% = 0.4612\%$).	

- With Load of $30[\Omega]$ resistor:

The measured DC voltage was: $V_{DC} \approx 3.953[V]$	The measured average voltage noise was: $V_{noise,avg} \approx -4.7[mV]$, The measured Peak-to-Peak voltage was: $V_{P2P} \approx 91[mV]$
	
calculation of the SNR derives: $SNR_{dB} = 20\log_{10}\left(\frac{3.953[V]}{91[mV]-4.7[mV]}\right) \approx 33.2183[dB]$ This corresponds to 2.1831 % noise relative to the DC signal ($\frac{1}{10^{\frac{33.2183}{20}}} \times 100\% = 2.1831\%$).	

Conclusions:

The SNR measurements conducted under various operating conditions reveal several important trends regarding the behavior of the charge controller's DC output. These insights are summarized below:

1. DC Voltage Stability:
   Across all test scenarios, the DC voltage remained relatively stable, ranging from approximately 3.95 V to 4.06 V, regardless of whether the charge controller was connected or disconnected from the charging source and regardless a DC-DC converter was connected or not and regardless a resistive load was applied.

This indicates good voltage regulation by the charge controller under all tested conditions.

2. Impact of Resistive Load:
   Introducing a 29–30 Ω resistive load consistently resulted in a lower SNR (i.e., more noise), an increase in peak-to-peak voltage noise and a higher percentage of noise relative to the DC signal.
   For example:
   • Without load and no DC-DC converter: SNR ≈ 58.5 dB (0.1183% noise),
   • With 29 Ω load: SNR dropped to ≈ 54.7 dB (0.1832% noise),
   • With both DC-DC converter and 30 Ω load: SNR dropped significantly to ≈ 33.2 dB (2.18% noise).

This confirms that adding a load introduces more fluctuations in the output signal, likely due to increased current demand and associated switching behavior.

3. Effect of the DC-DC Converter:
   The presence of the DC-DC converter introduced significant additional noise:
   • Even without a load, SNR dropped to 46.7 dB (0.46% noise),
   • Under load, noise increased drastically, reducing SNR to 33.2 dB (2.18%).

This suggests that the DC-DC converter is the dominant source of noise in the system, especially under load conditions.

4. Charging Source Disconnection:
   Disconnecting the charge controller from the charging source showed only slight changes in SNR, with minor variations in voltage noise:
   SNR stayed in the 47–54 dB range depending on load and noise percentages remained below 0.5% when no converter was involved.

This implies that the presence of the charging source does not have a major impact on noise performance by itself, compared to the effects of load or the DC-DC converter.

Calculating the parasite resistance of the inductor

Coil Parameters	
Coil Diameter	$d_{coil} = 2 \times r_{coil} = 0.25 \cdot 10^{-3}[m]$
Cross-Sectional Area	$A_{coil} = \pi \cdot r_{coil}^2 = \pi \cdot \left(\frac{0.25 \cdot 10^{-3}[m]}{2}\right)^2 = 4.91 \cdot 10^{-8}[m^2]$
# loops in one layer	80
# layers in one inductor	20
# total loops in one inductor	$80 \times 20 = 1600$
Iron Core Parameters	
Radius	$3[mm]$

This means that the average radius of the inductor is:

$$r_{avg} = r_{core} + \frac{num\ of\ loops}{2} \cdot r_{coil} = 3[mm] + \frac{20}{2} \cdot 0.25[mm] = 5.5[mm]$$

And the total length of the wire is:

$$l_{tot} = \#\ total\ loops\ in\ one\ inductor \times average\ loop\ parimeter =$$
$$= 1{,}600 \times 2\pi \cdot 5.5 \cdot 10^{-3}[m] = 55.29\ [m]$$

So the resistance of one inductor can be calculates as following:

$$R_{inductor} = \rho_{copper}[\Omega \cdot m] \cdot \frac{l_{tot}[m]}{A_{coil}[m^2]} = 1.72 \cdot 10^{-8}[\Omega \cdot m] \cdot \frac{55.29[m]}{4.91 \cdot 10^{-8}[m^2]} = 19.37$$

In practice, we used 4 iron rods attached to each other with a clay, which disrupted our model based on the average radius. The measured resistance of one inductor was 10.1[Ω].

Calculation of the maximum current in the inductor

To generate the strongest possible magnetic attraction in the hammer mechanism, the coils must carry the highest possible current. As engineers, the key parameter within our control that directly influences the coil current is the selection of **V_coil**.
In the laboratory, we measured the effect of **V_coil** on the coil current.
Results are shown in the table below:

$I_{DC}[mA]$	$V_{on\ board}[V]$
175.6	4.09
212.5	5.12
269.1	8.84
284.3	9.11
284.2	9.15
60	9.19
54	9.30
63.3	9.37
54	9.53
55	9.83

The drop in current at higher voltages is due to the **battery's power limitation (2[W])**.
At higher voltages, the battery is unable to supply sufficient current to maintain **Ohm's Law**.
Based on these findings, we set **V_coil** to **9.13[V]**.

10. Results and Conclusions

Throughout the development process, our project has yielded tangible results, bringing us close to fully achieving our objectives. We successfully developed and assembled the first functional hardware prototype of the advanced spinning top.
We created a 3D mechanical model, developed a functioning hardware prototype, and implemented core software elements to operate and test the system.

The mechanical system operates smoothly, with the spinning motion stabilized by a custom-built balancing device and the strategic addition of plasticine to fine-tune weight distribution.
Individual components such as the motors, LEDs, sensors, and hammers operated independently.
The hammers are striking at their designated frequencies, and the onboard sensors are effectively receiving and transmitting data.
On the software side, we've managed to get all components up and running individually, with partial synchronization already in place.

However, we encountered a few integration challenges, primarily related to ESP32 port limitations, component interface, and performance constraints:

1. Limited Safe GPIO Ports on ESP32
   The ESP32 has several GPIO pins that are not safe or stable for general use due to their roles during boot or flashing (according to unsafe GPIOs list). These GPIOs are marked as unsafe or restricted, further increasing the risk of instability or programming errors.
   In our design, motor driver A (U1 in the electrical schematic) uses GPIO1 (TX0) and GPIO3 (RX0), which are also Serial0 pins that are used by default for the Serial Monitor of Arduino IDE.
   Motor Driver A (U1 in electrical schematic) doesn't work when Serial0 is active.
   Even if we never call Serial.begin(), the onboard USB-to-Serial chip on the ESP32 DevKit still physically connects to GPIO 1 (TX0) and GPIO 3 (RX0).
   So even with Serial0 "disabled" in code, the hardware might still interfere.
   If the motor driver tries to drive signals that conflict with what the USB chip is trying to send or receive — we get signal corruption, voltage clashes, or inconsistent behavior.
   In addition, GPIO1 and GPIO3 are used during flashing, and must not be pulled high/low in a way that interferes (according to details on unsafe GPIOs list). If the motor driver is connected during boot, it may mess up the upload process or cause boot failure.
   Also, motor drivers (especially L9110S that we use) can generate electrical noise (voltage spikes or timing sensitivity) , and it could interfere the pulling high/low conditions.

2. Not Enough Available GPIOs
   The planned design required more GPIOs than were reliably available on the ESP32.
   This resulted in:
   - The Using pins that are critical for boot behavior (as explained in section 1 for motor drivers).
   - Shared signal line- for example, LED8 cannot function simultaneously with TF Luna sensor.

   TF Luna distance sensor used GPIO 16 (RX2) and GPIO17 (TX2), which are also Serial2 pins that are used for UART2 on ESP32 and communicates at 115200 baud.

   The net `ESP_LUNA_LED8_TXD` is connected to GPIO17 (TX2) on the ESP32 and RXD pin of the TF Luna sensor so ESP32 transmits UART data to TF Luna. But, this net is also connected to the cathode of LED8 diode, meaning that ESP32 is also driving LED8.
   So, this line tries to do two things at once.

   However, UART and LED cannot share the same pin. When ESP32 transmits UART data to TF Luna (configuration commends) it sends rapid pulses (high/low transitions) at 115200 baud rate. These transitions are not meant to drive a LED – and will not result in visible light.
   In fact, the voltage levels and frequency prevent LED8 from being consistently turned on (it may flicker invisibly or remain off).
   There is also a conflict between constant LED bias and UART transmission, since the pull-up and diode-resistor circuit to 3.3[V] assumes the ESP32 pin is a digital LOW to turn on the LED.

But, TX2 (GPIO17) is actively toggling as UART output, and thus **cannot be held LOW** constantly. Because it's switching quickly for transmitting UART data to TF Luna, it cannot reliably sink current through LED8— so LED8 cannot turn on.
LED needs the pin to stay LOW to turn on, and UART TX never stays LOW, so the LED does not get enough time or current to turn on visibly.
In addition, since LED + diode + resistor add electrical load on the UART TX line, this might distort the signal going to the TF Luna sensor and risk UART communication errors.

3. Serial Debugging Limitations
   Activating the motor driver blocked access to the serial port, making it difficult to debug or monitor system behavior in real time.

   From (1)+(2)+(3) problems we conclude that we cannot use:
   - Serial0 for debugging because motor driver A is on GPIO1 and GPIO3.
   - Serial2 because TF Luna uses it.
   - Serial1 (default pins GPIO9 and GPIO10) because they are tied to the ESP32's flash.

4. File Writing Delays
   Writing sensor data directly to file during each iteration caused significant delays, which negatively affected motor movement smoothness and real-time performance.

5. Mechanical and Design Constraints
   •The hammer's aperture was too wide to produce the intended knocking frequency.
   • The spinning top suffered from balance issues that affected stability, though partially mitigated using plasticine.

Suggested Solutions

To address the integration challenges encountered in the current phase of development, we propose a combination of short-term workarounds and long-term design improvements to ensure stable operation, clearer signal routing, and more reliable performance.

1. GPIO Management and Port Conflicts:
   To resolve the issue of unsafe or overloaded GPIO usage, especially with GPIO1 and GPIO3 being used for the motor driver while also tied to the USB-serial interface (Serial0), we recommend migrating motor driver A's control pins to alternative, safer GPIOs. Pins such as GPIOs 4, 5, 18, 19, 21, 22, or 23 are generally considered safe and unreserved by boot or debug processes. These GPIOs offer clean digital output capabilities and avoid electrical contention with the onboard USB-to-serial chip. In addition, using a GPIO expander is highly recommended for future iterations to handle larger pin demand while freeing up critical pins on the ESP32 for time-sensitive or communication-based tasks.

2. Resolving the TF Luna and LED8 Conflict:
   The UART conflict between LED8 and the TF Luna sensor can be resolved by separating their signal lines entirely. Specifically, LED8 should be assigned to a dedicated GPIO not used for UART transmission, thereby allowing it to function correctly as a standard digital output. This avoids signal distortion on the UART TX line and ensures TF Luna communication remains error-free.

3. Improving Debugging Capabilities:
   To enhance real-time debugging, especially when motor drivers interfere with Serial0, it is advisable to shift development logging and serial output to Serial1, which can be routed through GPIO's expander pins. This separation would allow motor control and debugging to occur simultaneously.

   Serial1 can be remapped. The ESP32 allows to remap Serial1 to any available GPIOs, as long as they support digital I/O (which most do). The default GPIOs 9/10 for UART1 are just defaults — we can override them. So even though GPIO9 and GPIO10 are unusable, we can still use Serial1 on custom pins. For example:

```
HardwareSerial Serial1(1);  // Use UART1
Serial1.begin(115200, SERIAL_8N1, 21, 22);  // RX, TX remapped
```

   In this example, we just need to plug a **USB-to-UART adapter** into GPIO22 (TX) and GND, and then we can receive debug output in a second serial monitor.

   We suggest to use Serial1 with remapped pins, on GPIO expander or if we have two unused GPIO's (in this example GPIO21 and GPIO22), and assign them like the commands above. Afterwards we have to connect a USB-to-Serial converter:
   • TX of ESP32 (GPIO22) → RX of adapter
   • GND → GND
   And then, open a second serial monitor on the PC for debugging.

   This approach will leave Serial0 available for uploading code, and motor drivers won't interfere because they're on GPIO1/3.

4. Buffered Data Logging:
   To mitigate performance degradation caused by frequent file writes, we recommend implementing a data buffer system in software. Data should be collected and temporarily stored in RAM during runtime, then written to file in bulk at defined intervals or when certain thresholds are met. This significantly reduces file I/O overhead and prevents it from interfering with motor timing and responsiveness. For even more robust performance, consider using double-buffering to allow data capture and writing to occur in parallel.

5. Mechanical Adjustments and Balancing:
   On the mechanical side, reducing the aperture of the hammer or redesigning its striking surface can help achieve more precise and higher-frequency impacts. Materials with higher stiffness or improved resonance response may also support better frequency control. For balancing, we recommend incorporating a more systematic mass distribution strategy using calibrated weights or redesigning the chassis to ensure natural symmetry.

Despite these limitations, the current state of the project represents a solid foundation for future iterations.

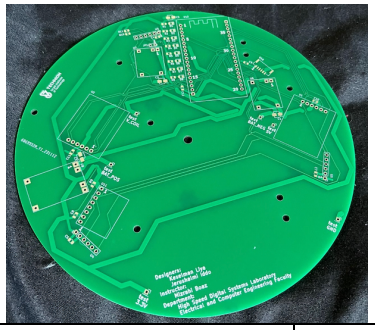
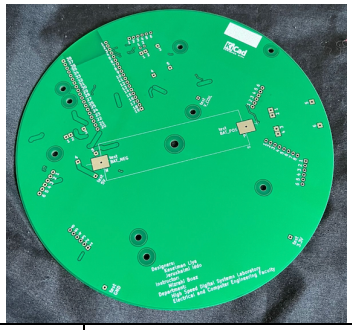
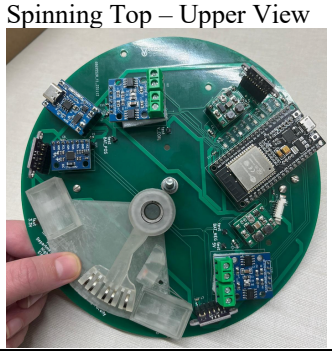
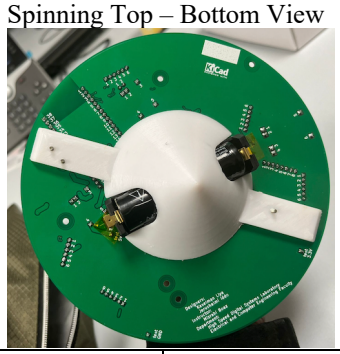
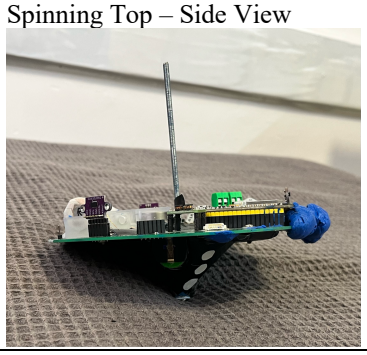
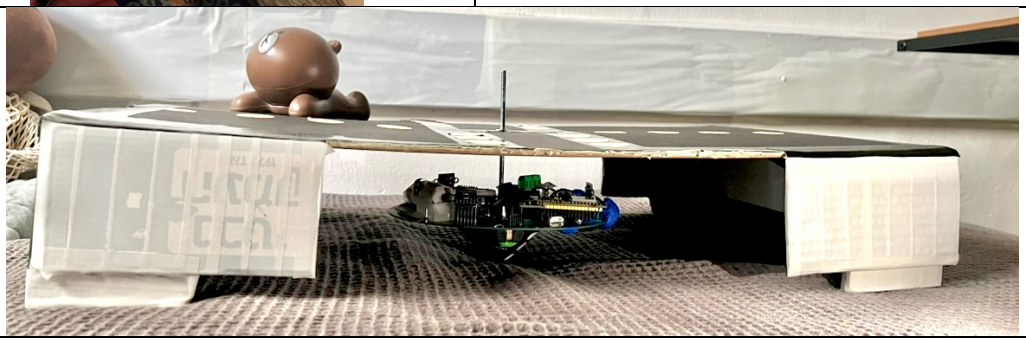
In addition to the technical achievements, we gained hands-on experience in system integration, PCB design and embedded programming. Also, we enhanced our understanding of real-time sensor management, and importance of system modularity in hardware design. Furthermore, during the process we gained valuable experience in troubleshooting issues.
These insights will be invaluable for future iterations of the project.

While this prototype might resemble a toy, its potential applications extend far beyond entertainment. With future improvements, the spinning top could be used in various fields — including tactical, military, or educational contexts — where mobility, obstacle detection, data visualization, and responsive control are critical. This project serves as a robust foundation for further development, and we believe the next teams will be able to build on our work to push the system toward its full potential.

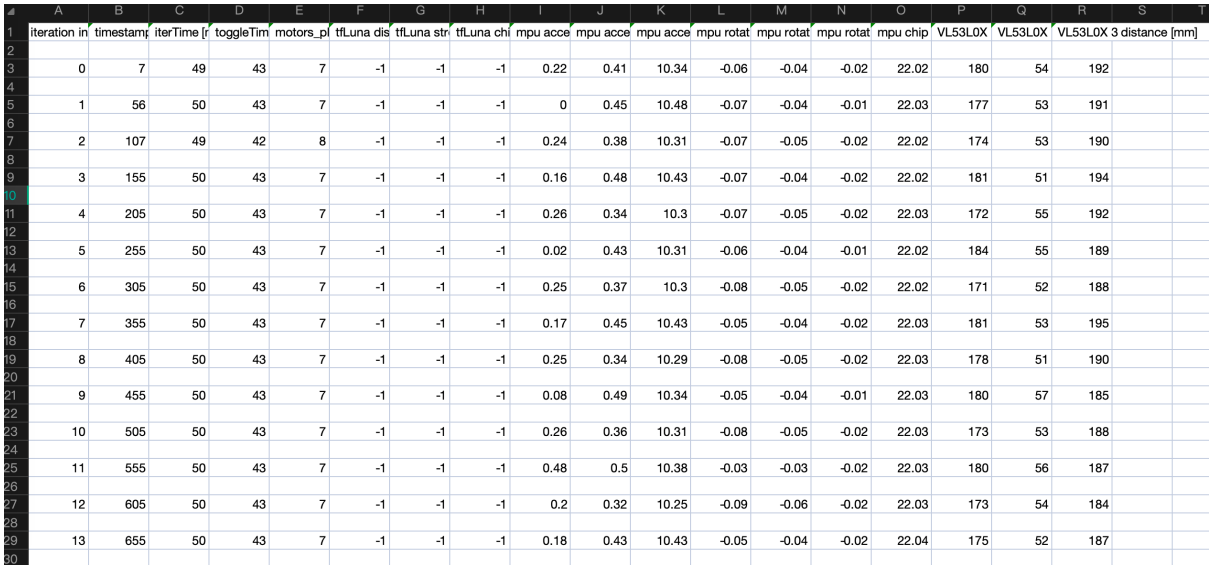
(*) an Error Sheet is attached to GitHub repository.

Integrated System

The following images present the fully integrated system, as well as an example of the live data output displayed on the computer. These visuals highlight the transition from design to implementation, demonstrating how the physical hardware closely follows the initial modeling and planning stages. The system includes all major components—sensors, motors, PCB, and structural elements—assembled and operational. The output screenshot illustrates successful communication between the hardware and software layers, confirming functionality such as sensor data transmission and live status feedback.

<table>
<tr><td>PCB</td><td>PCB – front side
  </td><td colspan="2">PCB – Back side
  </td></tr>
<tr><td>Spinning Top</td><td>Spinning Top – Upper View
  </td><td>Spinning Top – Bottom View
  </td><td>Spinning Top – Side View
  </td></tr>
<tr><td>Playing With the Design</td><td>Designing Cone
  </td><td colspan="2">Designing of Secondary Stabilization Device
  </td></tr>
<tr><td>Final Prototype</td><td colspan="3">  </td></tr>
</table>

Output example:



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
1	iteration in	timestamp	iterTime [r	toggleTim	motors_pl	tfLuna dis	tfLuna str	tfLuna chi	mpu acce	mpu acce	mpu acce	mpu rotat	mpu rotat	mpu rotat	mpu chip	VL53L0X	VL53L0X	VL53L0X 3 distance [mm]		
2																				
3	0	7	49	43	7	-1	-1	-1	0.22	0.41	10.34	-0.06	-0.04	-0.02	22.02	180	54	192		
4																				
5	1	56	50	43	7	-1	-1	-1	0	0.45	10.48	-0.07	-0.04	-0.01	22.03	177	53	191		
6																				
7	2	107	49	42	8	-1	-1	-1	0.24	0.38	10.31	-0.07	-0.05	-0.02	22.02	174	53	190		
8																				
9	3	155	50	43	7	-1	-1	-1	0.16	0.48	10.43	-0.07	-0.04	-0.02	22.02	181	51	194		
10																				
11	4	205	50	43	7	-1	-1	-1	0.26	0.34	10.3	-0.07	-0.05	-0.02	22.03	172	55	192		
12																				
13	5	255	50	43	7	-1	-1	-1	0.02	0.43	10.31	-0.06	-0.04	-0.01	22.02	184	55	189		
14																				
15	6	305	50	43	7	-1	-1	-1	0.25	0.37	10.3	-0.08	-0.05	-0.02	22.02	171	52	188		
16																				
17	7	355	50	43	7	-1	-1	-1	0.17	0.45	10.43	-0.05	-0.04	-0.02	22.03	181	53	195		
18																				
19	8	405	50	43	7	-1	-1	-1	0.25	0.34	10.29	-0.08	-0.05	-0.02	22.03	178	51	190		
20																				
21	9	455	50	43	7	-1	-1	-1	0.08	0.49	10.34	-0.05	-0.04	-0.01	22.03	180	57	185		
22																				
23	10	505	50	43	7	-1	-1	-1	0.26	0.36	10.31	-0.08	-0.05	-0.02	22.03	173	53	188		
24																				
25	11	555	50	43	7	-1	-1	-1	0.48	0.5	10.38	-0.03	-0.03	-0.02	22.03	180	56	187		
26																				
27	12	605	50	43	7	-1	-1	-1	0.2	0.32	10.25	-0.09	-0.06	-0.02	22.03	173	54	184		
28																				
29	13	655	50	43	7	-1	-1	-1	0.18	0.43	10.43	-0.05	-0.04	-0.02	22.04	175	52	187		
30																				

Full output example is attached to GitHub repository.

2. Thanks and Recognition

Mizrachi Boaz

and HSDSL lab’s team: Orbach Mony, Rivkin Ina, Bulkin Lena

12. Bibliography

12.1. Sources to 3D files

GrabCad Community

12.2. Datasheets

attached to GitHub repository

12.3. ESP32 Safe Ports

Which ESP32 GPIOs are Safe to Use? Lastminuteseengineers

13. Appendix

GitHub repository can be accessed by the following link:
Spinning_Top_Hardware_Part_1

Content of repository includes:

- Document Files
- Presentation Files
- Schematic Files
- Program Files
- Production Files
- ErrorSheet
- 3D Print Files
- 3D Modeling Files (Fusion360).
- KiCad Files
- DataSheets
- Components List
- BOM
- Web Files
- Media Files

Project Presentation can be accessed by the following link:
Project Presentation_part_1