ECE 445

Fall 2025

Final Report

Project #29: Modular Wafer Track for Semiconductor Fabrication

Team Members:

Jack Schnepel (jackks2)

Hayden Kunas (hkunas2)

Nathan Pitsenberger (nmp5)

TA: Shenyang Liu

Professor: Rakesh Kumar

Abstract

This report describes the process of designing, building, and testing our Modular Wafer Track for low-cost semiconductor fabrication. The goal behind this project was to provide an inexpensive alternative to semiconductor fabrication for hobbyists and small-scale laboratories. The track is to accept a multitude of tool types and provide an easy-to-use interface. The project is centered around an ESP32 and a three-axis motor system, all interfaced with a Raspberry Pi acting as the graphical user interface (GUI). After extensive design, the track met all key performance goals.

Table of Contents

1. Introduction

1.1. Problem

In today's world, where semiconductors drive nearly every aspect of technological innovation, little room is left for small-scale fabrication and experimentation. Commercial wafer processing equipment ranges from tens of thousands to hundreds of millions of dollars, putting it far out of reach for hobbyists, educational laboratories, and early-stage researchers. Existing systems are not only cost-prohibitive but also lack the flexibility and modularity needed for experimentation on a smaller scale. As a result, innovation outside of large industrial fabs is limited, leaving students, independent researchers, and small labs without access to tools that enable exploration of semiconductor device fabrication.

1.2. Solution

Our team's solution to this problem is to design, build, and demonstrate a modular, cost-effective wafer track system that lowers the barrier to entry for small-scale semiconductor processing. Along with the cost reduction, our system offers a level of customizability that many industry-standard machines do not offer. Our team developed a machine that transports wafers between processing modules, allowing the user to execute repeatable, user-defined fabrication recipes and communicate a set of standardized instructions to each sub-module.

By creating this machine, our *Modular Wafer Track* solves many key issues. First is the user interface running on our integrated Raspberry Pi, which acts as the front-end bridge between the user and the modules. This UI is the hub for all recipe creation and module management. This allows the user to create their own recipes, save and load them to the Raspberry Pi. In turn, any recipe can be run multiple times and be repeatable, without the worry of variability in its semiconductor development process.

Secondly, our machine uses a standardized set of instructions to communicate with the modules and the ESP32 Microcontroller. This standardized communication is what allows the user to build their own personalized modules for the *Modular Wafer Track* and work seamlessly with the UI. Pairing this with the open-source capabilities, anyone can design and build any combination of modules that fit their processing needs

Finally, the *Modular Wafer Track* is designed and built off easily sourced materials, motors, and custom 3D printed models, which factor into its cost-effective nature. All of this contributes to a design and machine that brings the power back to hobbyists and students who are looking to get their foot in the semiconductor landscape.

1.3. Visual Aid

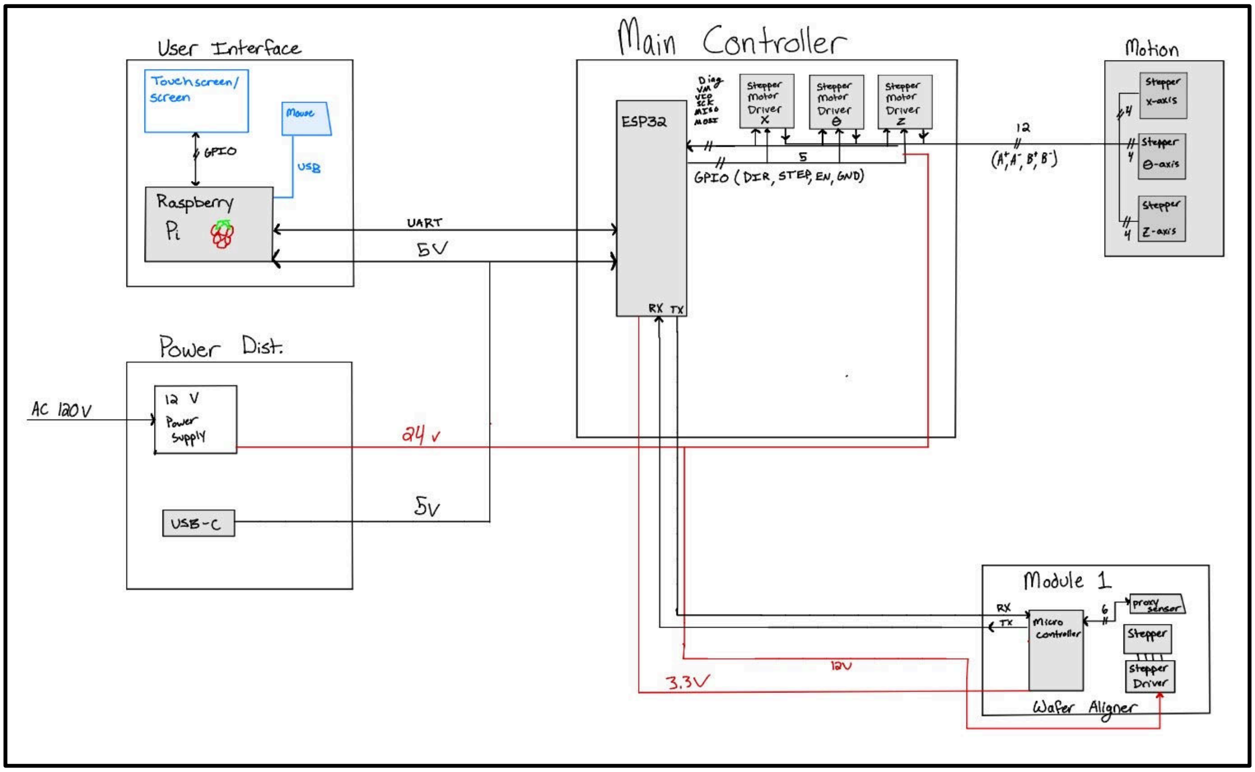


Figure 1.1: Modular Wafer Track Block Diagram

1.4. High-Level Requirements

To achieve a successful final project, our group established a set of achievable and standard high-level requirements. The first requirement for our team was to set a list of actions that users of the *Modular Wafer Track* can achieve. This was:

- Create a recipe
- Run and save the recipe to the system,
- Insert the wafer on the *Modular Wafer Track* arm,
- Move the wafer to our wafer aligner sub-module
- Return the wafer to the user once the recipe is finished.

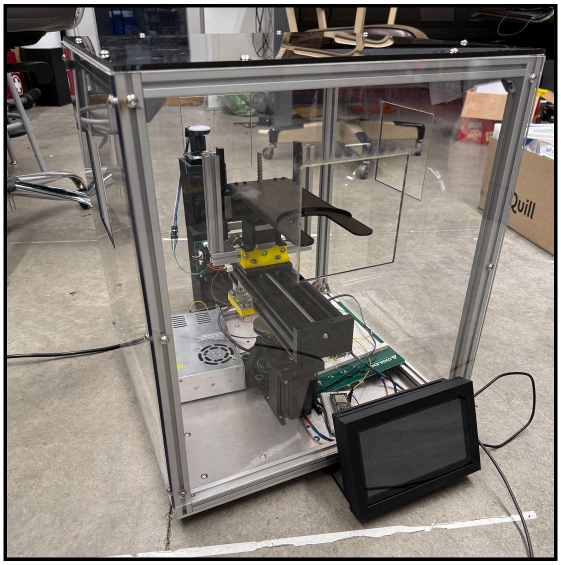


Figure 1.2 Modular Wafer Track In-Person Picture

The second requirement was to have subsystem functionality, which included the following:

- Power distribution,
- Wafer track,
- Graphical user interface (GUI),
- Wafer aligner

The last requirement was reusability. This means a user can:

- Create their own subsystems for future use
- Run the same processes in succession with similar precision and tolerances,
- Use our system across a wide range of settings (including lab spaces and clean rooms).

2. Design

2.1. Design Procedure

2.1.1. Motor System

The motor system is paramount to the successful completion of the project, so much of our initial design work focused on the motor system. By the end of our project, the system consisted of three axes: two linear and one rotational. The rotational axis rotates the wafer arm to the desired slot; the horizontal axis loads the wafer into the slot; and the vertical axis mounts or dismounts the wafer from the slot.

The drivers used for the system are BigTreeTech 2130v3 stepper motor drivers. The design choice for using these drivers was for several reasons. Firstly, they were readily available. We already had a few of these drivers at our disposal, so it was a no-brainer. Secondly, these drivers were able to be easily mounted to a printed circuit board (PCB), which made it much more convenient from a physical design standpoint. Third, and most importantly, these drivers offered sensorless homing. What this means is that there was no need for limit switches for homing the motors to their default position at startup. The sensorless homing works by reading the current draw, and once the current reaches a specific threshold, it assumes that it is home.

An alternative approach to the motor system at the beginning of the design process was to use two horizontal linear motors and one rotational motor. One horizontal motor would transport the other two motors down the track to different slots. The other horizontal motor and the rotational motor would act similarly to the final design. The problem with this design is that the additional horizontal axis is redundant. Additionally, there was no mount/dismount procedure for the wafer. We were going to make it such that the modules were responsible for taking the wafer from the arm; however, we decided this would be more expensive in the long run for the user, since they would need to implement their own dismount system for each module.

Another design choice for the motor system we had to make was for the transfer arm itself, not just the motor assembly. This was more of a materials choice rather than the physical design. We had wanted to use a material that would be rigid enough such that the arm would not deflect when the wafer is loaded onto the arm. Our first thought was to use a metal arm. However, using a metal arm would reduce prototyping efficiency and would increase costs due to metal machining; additionally, there was a concern that the metal would be incompatible with some semiconductor processes, possibly chemically reacting with the wafer or ESD on the wafer, damaging the components. The next best thought was 3D printing the wafer arm, since it would improve prototype efficiency, and it was cost-effective.

After doing some research on different filaments, we found that CF-PETG (Carbon Fiber Polyethylene Terephthalate Glycol) would be the most effective for this project, especially when considering rigidity. The calculations for beam deflection are shown in Chapter 3.1. We also designed a slot for the IR proximity sensor in the wafer arm to determine whether a wafer was present.

2.1.2. Main Controller

The main controller coordinates the three motor axes, orchestrates automatic load and unload procedures, and supervises external modules that are attached to the track (wafer aligner, spin coater, hotplate, etc.) over a serial link. The design for the main controller considers determinism for motion and simplifies user-system interaction for process modules.

The microcontroller unit (MCU) used for the main controller was an ESP32. The ESP32 offered a multitude of benefits that other MCUs could all offer. Firstly, the ESP32 provided sufficient performance for many of the main peripherals, such as the SPI interface with the stepper motor drivers, the pulse width modulation general purpose input output (GPIO) for the three stepper motor drivers, and UART communication with each external module and the GUI. Additionally, the ESP32 was able to handle many of the mature Arduino libraries, such as AccelStepper, TMCStepper, and ArduinoJson (which is covered more shortly). These libraries ended up being vital to a sufficient amount of our program and could not be sacrificed. Finally, and probably most importantly, is cost. Compared to many other MCUs (Arduino, Raspberry Pi, etc.), the ESP32 offered a significant cost advantage. Oftentimes, you can find ESP32s for roughly five dollars. This made it a clear choice to choose the ESP32 for prototyping and maintaining the low cost of our project compared to industrial equipment.

In addition to the MCU choice, we also had to decide on a stepper motor driver. While the stepper motor driver was already discussed in Chapter 2.1.1, we will discuss the inherent benefits the driver offers rather than the physical and cost evaluations here. Firstly, the driver offers SPI register access that allows us to have repeatable configuration of the drivers and deeper diagnostics; the drivers' settings can also be changed via SPI, whereas many budget-friendly drivers require you to manually change settings via dip switches. Secondly, the driver offers stallGuard, which is the sensorless homing option that was mentioned before. Finally, the drivers offer two unique settings called spreadCycle and stealthChop. The spreadCycle setting offers higher torque during fast moves, whereas stealthChop provides quiet movement by sending a fixed-frequency voltage-controlled chop to create a sinusoidal current wave into the stepper motor coils[1]. We utilized spreadCycle for the rotational axis since it has a high moment of inertia, requiring more torque for controlled movements, and stealthChop for the linear axes for silent movement.

For the step generation and movements, we used the AccelStepper library in conjunction with helper functions to determine motor movement. We decided to use the AccelStepper because it is a proven, working library with trapezoidal motion (built-in acceleration functions) and minimized development time. Additionally, the AccelStepper library provided blocking segments for recipe steps and non-blocking segments in the main loop to keep motion alive during I/O waits; since our process is mostly sequential, we utilized the blocking functions significantly.

Finally, the main controller was responsible for inter-module communication, which is, arguably, the most critical component of the main controller, or the entire project. The following description will be for what we planned to use, but later, we will comment on what

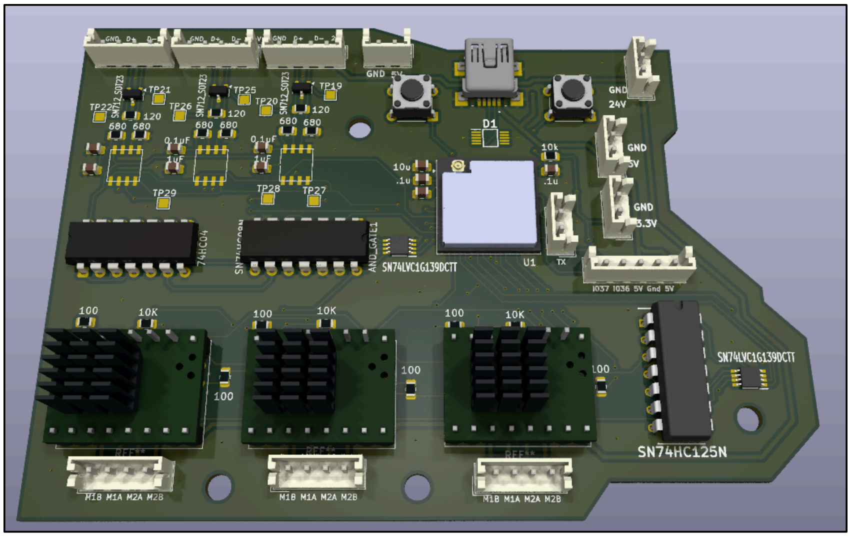


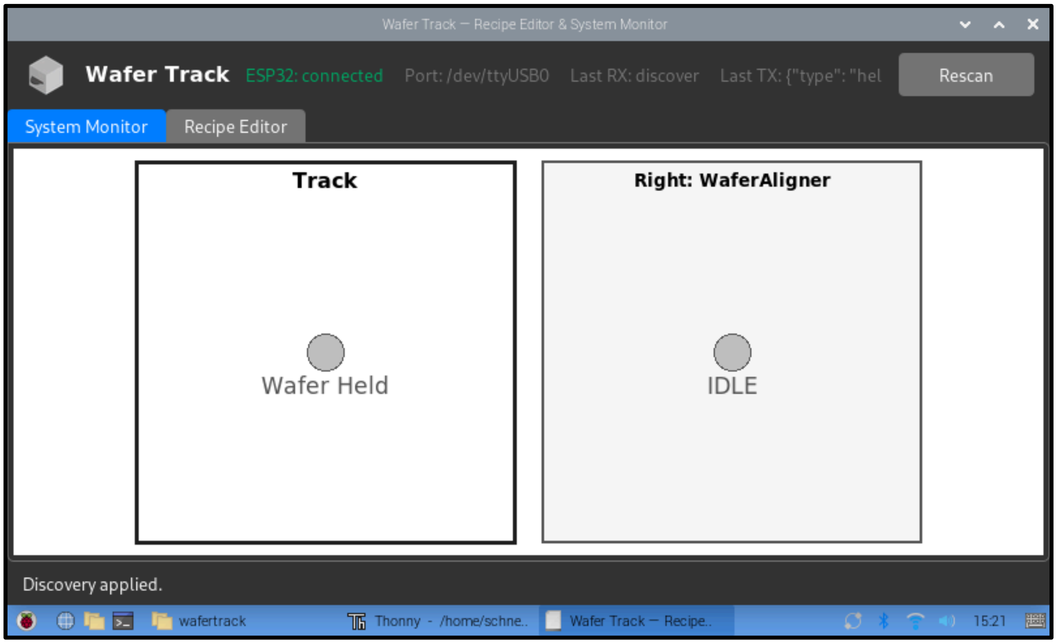
Figure 2.1: Final Logic PCB Render including ESP32, Stepper Motor Drivers, and RS-485 Communication

we resorted to for the final demonstration of our project. The framing, for both the Raspberry Pi-Main and Main-Module communication, is newline-delimited JSON; messages are UTF-8 JSON objects, terminated by a newline, which keeps parsing non-trivial[2]. The transport differs by link: USB serial at 1152000 bps between the Raspberry Pi and the main controller, and (by design) an RS-485 bus between the main and the modules. Other alternatives we could have used were GPIO handshakes or a binary framed protocol. However, GPIO handshakes, as the name suggests, are GPIO limited, which limits scaling; binary framed protocol is faster than JSON, but it is less transparent (difficult to know what data is being sent). As a patch to not having the RS-485 communication fully functional, we used GPIO handshakes, but, without loss of generality, we will stick to the RS-485 scheme for the description of the communication protocol.

2.1.3. Graphical User Interface (GUI)

The graphical user interface played an important part as the median between the user and the subsystems. The GUI consisted of three main parts: the Raspberry Pi, the touchscreen, and a USB mouse. Conceptually, the GUI had three main responsibilities: (i) discovery and status, where it asked the main controller to probe for connected modules and then rendered slot occupancy; (ii) operation control, where an operator could trigger parameterized commands and run predefined recipes; and (iii) observability and safety, where the interface streamed JSON logs and presented progress. The GUI itself was intentionally “dumb”: it did not schedule motors or directly control modules; instead, it issued well-formed JSON requests from the main controller and updated based on those responses. Figure 2.2 shows one of the UI pages for system monitoring.

This client approach was chosen for three main reasons. First, it keeps real-time responsibilities off the GUI. The Pi’s UI loop never blocks the track motion; that is the main controller’s responsibility. Secondly, it reduces coupling. Because the GUI only depends on messages received from the main controller, we could evolve the main controller's code



ESP32: connected Port: /dev/ttyUSB0 Last RX: discover Last TX: {"type": "hel

Discovery applied.

Figure 2.2: Screen Capture of the System Monitor Screen on the GUI showing a wafer aligner and hot plate attached to the system.

without rewriting the GUI. Third, it lowers implementation risk; a front-end that renders and publishes JSON is much easier to verify and debug.

Some alternative designs we considered were a full web app and a terminal-based UI. Regarding the full web app, this design would present a richer UI panel, multi-user access, and cross-platform compatibility. However, this design would provide more challenges than it is worth, such as building a web server and needing to secure the client. The next alternative was a terminal-based application. The terminal-based option would be significantly easier to build and provide easier debugging; however, this design choice is not very user-friendly and slows down overall use.

2.1.4. Power Subsystem

The power requirements for the Modular Wafer Track include 24 V and 3 A (max) for the stepper motors, 5 V and 3 A (max) for the Raspberry Pi 4 Model B, and 3.3 V and 1 A (max) for all of the logic circuits. The original design for this included a 120 VAC to 24 VDC adapter for the main source of power for the circuit. The 24VDC would then be stepped down to 5 VDC using a LM2678 and then stepped down further using a TPS62162QDSGRQ1. The LM2678 is a high-efficiency switching buck converter capable of handling up to 5A[3], which is 1 A lower figure than the maximum draw of the Raspberry Pi and the estimated maximum draw of the logic. Both of the loads must be considered since the 3.3 VDC converter worked off the 5 V buck converter. This would later be changed in the final iteration for a few reasons.

The final subsystem ended up being a separate PCB, with plugs that may be plugged into the main logic board. This allows for more leeway and a way to make the logic work in the case of the power board not working. As mentioned before, the final design included a 24 V to 5 V buck and a 24 V to 3.3 V buck, which is different from the 5 V to 3.3 V buck in the previous design. This means the converters are not dependent on each other to work. Also in the final design are different buck chips, which include a simpler functional design. The LMQ61460-Q1[4] was used for the 24 VDC to 5 VDC buck, and the TPS54302[5] was used for the 24VDC to 3.3VDC converter. These were chosen for their simplicity of implementation and ability to handle the maximum currents required in each circuit with enough headroom. The image of the final subsystem design is seen in Figure 2.3.

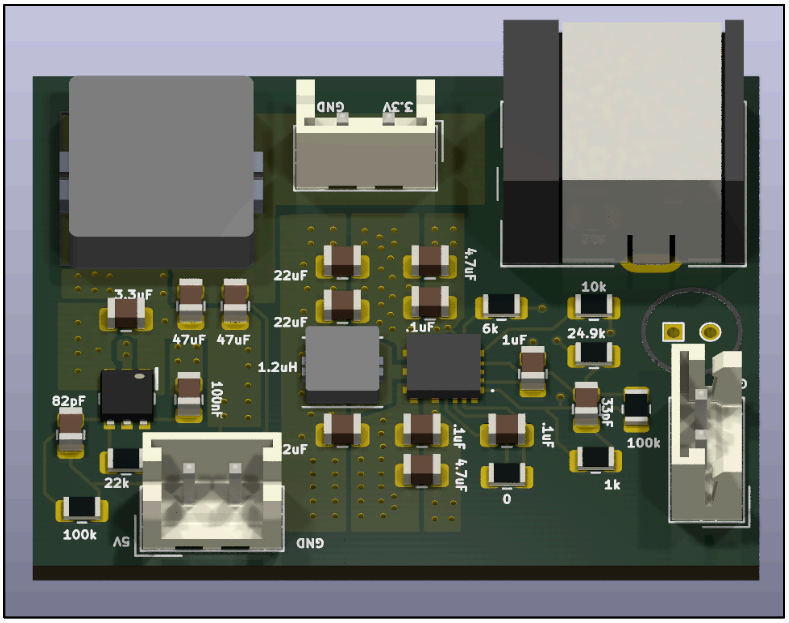


Figure 2.3: Final Power Subsystem Render

2.2. Design Details

2.2.1. Motor System

What has not been talked about yet regarding the motor system is the enclosure details. Once the physical dimensions were all determined for the motor assembly, we were able to design the enclosure for the system. The main obstacle to consider was optimizing the size of the enclosure: make it small enough to maintain its compactness, but large enough for the motors to move freely. So, the optimization was how small we could make the enclosure without the motors hitting the walls? The calculation for the dimensions was done by simply using the Pythagorean theorem on the horizontal axis. While we went with a cubic design for our enclosure, we also considered a cylindrical enclosure, which would minimize the volume of the system. However, this design choice would lead to more difficult mounting techniques for the modules to attach to the track. The framing for the system was made of 20-Series Aluminum extrusions and secured to one another via right-angle aluminum brackets.

Another aspect that many would disregard as being important is the base of the enclosure. While it’s not complex, it’s important enough to note. Because the entire system is being mounted to the plate, the plate must be rigid and sturdy enough to support the weight of the motor assembly and framing without deflecting, which would cause inaccuracies in wafer traversal. The plate consists of mounting holes for the framing, motor assembly, and PCB mounting. The plate was machined from Xometry and made from 5mm thick 5052 Aluminum. To support the plate, we constructed a 360 x 360 mm subframe made of 20-Series aluminum extrusions, mounted to the plate via T-nuts and M5 bolts. The dimension of the enclosure (excluding the subframe) was 440 mm x 440 mm x 500mm.

As mentioned before, the wafer arm had to undergo some beam deflection calculations to determine whether the CF-PETG would be substantial enough, since it can be modeled as a cantilever (since its fulcrum is the mounting point).[6]

Table 2.1: Values Used for Beam Deflection Calculation

Input	Value
Length L	206.5mm
Width W	99.8mm
Height $\bar{h}$	5mm
Load at tip F = $(m_{wafer} + m_{arm})g$	≈ 2.45N
Modulus of elasticity E	≈ 2200MPa
Poisson’s Ratio v	0.38

$$Max\ Beam\ Deflection\ =\ \delta_{bend} = \frac{FL^3}{3EI} = \frac{FL^3}{3E(1.04E-9\ m^4)} = 1.15mm$$

On the PCB, the motor system required its own section with particular detail. Because the ESP32 has limited GPIOs, we had to minimize the GPIO cost. If all the drivers were hooked up directly, we would need 16 GPIO pins (four for SPI and four for CS, STEP, DIR, and DIAG) for the three stepper motor drivers. However, we designed the motion subsystem to multiplex the high-rate signals, serialize the low-rate controls, and aggregate the status lines.

First, we reduced the STEP and DIR pin count from three to one by introducing a 2:4 demultiplexer (actually a 3:8 74HC138[7], but only used two control bits) as a driver selector. The STEP and DIR signals were fanned out to three SN74HC125[8] buffers for each respective pin, which would pass the STEP and DIR signals to the selected stepper motor driver. The result of this was reducing the GPIO pin count by two.

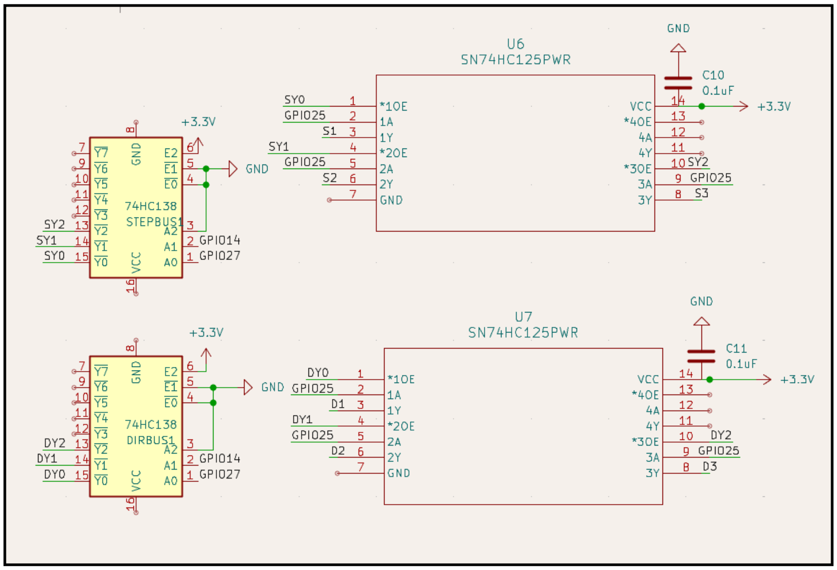


Figure 2.4: Logic for stepper motor drivers including the buffer and demultiplexer

Additionally, the DIAG pin (used for sensorless homing) count could be reduced from three to one GPIO pin. To do this, we only need to fan the three DIAG pins into a single line with a 100 Ω resistor at each fan. The consequence of this design choice for the DIAG, STEP, and DIR pins is that axis movement must be sequential; however, this is not a problem for us, since each motion is designed to be sequential anyway.

This design choice allowed us to minimize the GPIO count by reducing it from 16 to 12 GPIO pins. This provided more headway for additional peripherals (such as the proximity sensor) and RS-485 communication.

2.2.2. Main Controller

Regarding the design details corresponding to the main controller, the first thing to mention is the packet communication protocol we developed. At startup, the main controller performs discovery by sending a “who” probe on each module line. Any listening module replies with its presence: an identity “iam”, a “module present” marker, and a “module hello” that declares the operation names it supports. The main controller consolidates these announcements into an upstream discovery to the Raspberry Pi.

Command/response semantics are uniform and two-phase (requires both parties to participate). The Pi invokes a module operation by sending a “run” message specifying the module, operation, and a parameters object. Upon receipt, the module immediately returns an acknowledgement, confirming it accepted the command; when execution finishes, the module emits a “done” statement with an “ok” flag and an optional reason on failure. The main controller enforces ordering by waiting briefly for an acknowledgement, followed by a “done” before continuing to the next dependent step. To maintain visibility, modules also publish periodic “module status” heartbeats. If a module is busy, it fails fast with “ok = false” and “reason: busy”, allowing the main controller to try again or abort cleanly.

The planned RS-485 layer focuses on electrical robustness and scaling. This provides ESD protection and DE/RE control lines on the controller, so only one module can communicate at a time. Logically, the modules are addressed in the message by the “module” field; physically, the controller enables exactly one port receiver when it expects traffic to prevent collisions. This allows us to add many modules while keeping the higher-level JSON protocol unchanged.

For the final demonstration, we had to rely on only one UART line for the modules because we ran out of pins for module communication. Crucially, however, this does not change the fundamental design of the project: discovery, command flow, status heartbeats, and timeout remained the exact same. See Appendix A for a summary of some of the query/response vocabulary used between the Raspberry Pi, the main controller, and the modules.

Another important aspect to talk about for the main controller is the RS-485 communication circuit, which, similar to the stepper motor driver circuit, was designed to minimize the GPIO count. We adopted a hub-and-spoke design, where each module has its own half-duplex RS-485 transceiver (SN65HVD72[9]). Each transceiver produces an A/B differential pair to transport JSON packets between the modules and the main controller.

To minimize GPIO, we share a single ESP32 UART across all ports and control which transceiver is live via its driver enable and its receiver enable pins. Thus, the two UART pins are common to all ports. Each transceiver also has a Driver Enable (DE) and an active low Receive Enable ($\overline{\text{RE}}$) pin. Similar to the stepper motor drivers, we utilized a 2:4 demultiplexer to select which driver we wish to enable. So each output of the demultiplexer was tied to the respective $\overline{\text{RE}}$ pin on the transceiver. To obtain the DE signal for each, we simply had to invert the $\overline{\text{RE}}$ pin for each, since if we are not transmitting, then we are receiving.

With this logic configuration, this brings the total GPIO pins used for RS-485 communication to three modules from 12 GPIO pins to 4 GPIO pins. The final RS-485 schematic is shown in Figure 2.5.

2.2.3. Power Subsystem

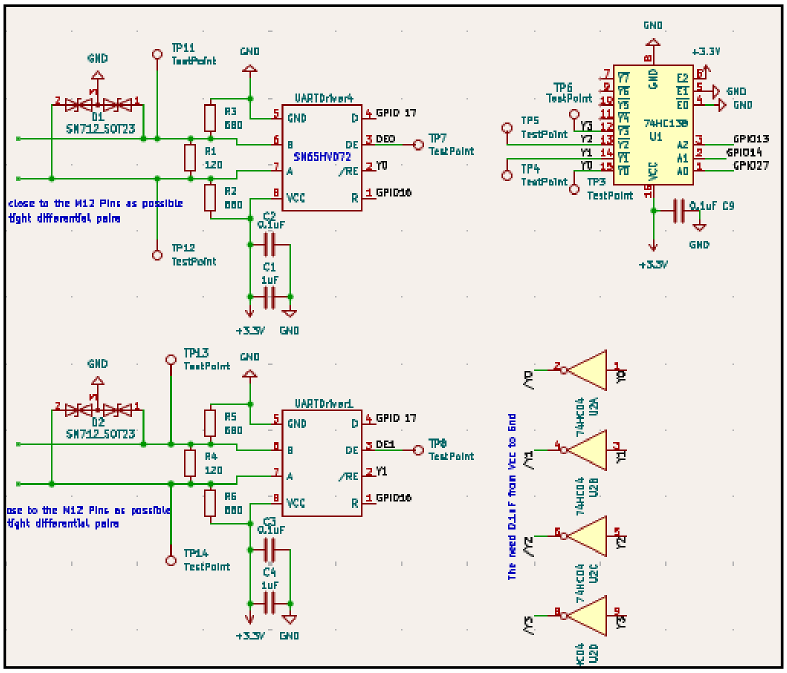


Figure 2.5: Partial RS-485 Communications Circuit Diagram

In the original buck designs, the circuit was made from the recommended designs included in the datasheets for each respective switching chip, but the 5 V buck ended up not working, which meant that the 3.3 V buck could not work. The design was checked and verified along with component values and solder joints. At the end of the datasheet was a recommended layout section, which detailed the best way to lay out the components relative to each other for the best performance. This section of the datasheet stuck out because it made connections with zones instead of traces, which was not done in the original design. This would be an important thing to consider in the design of the other buck converter circuits. Not a ton of time was spent debugging the design of the power circuit, as the final round was due very soon. The main error that was found was a short between the switch output pin, feedback pin, and VCC pin, which would cause the chip to go into shutdown mode.

The final power subsystem PCB used two new switchers, which were implemented using the recommended component values as well as the optimal layout. The 3.3 V buck actual layout and the PCB, and the layout on the datasheet are included in the image below. My design differs slightly as there are more vias included to further reduce the impedance and slightly improve the thermal management. The issue with our circuit was that it was outputting 6.93 V instead of the intended 3.3 V. Equation 3 on the datasheet describes the value of Vout relative to Vref and the voltage divider resistors R3 and R2. The equation states that Vout=Vref(R2/R3+1) where R3 is 22 kΩ, R2 is 100 kΩ, and Vref (FB pin) is .596 V. This leads to Vout being 3.3V, but when the actual Vref pin is probed, the value is 1.255 V, which projects the output to be the same as the measured value of 6.93 V when plugged into the equation. The IC uses the feedback (FB) voltage as a way to adjust the duty cycle of the output to adjust the voltage so the FB pin should always be .596 V, as that is the value when the duty cycle has stabilized. The FB pin stabilizing at 1.255 V leads us to believe that maybe the chip is faulty, as shorts and other errors were thoroughly checked for. This also leads us to believe that, considering that Vref is 1.255 V, you could make the 22 kΩ resistor a 38 kΩ resistor to make up for FB being a higher value, but we unfortunately did not have the components on hand to immediately test this hypothesis.

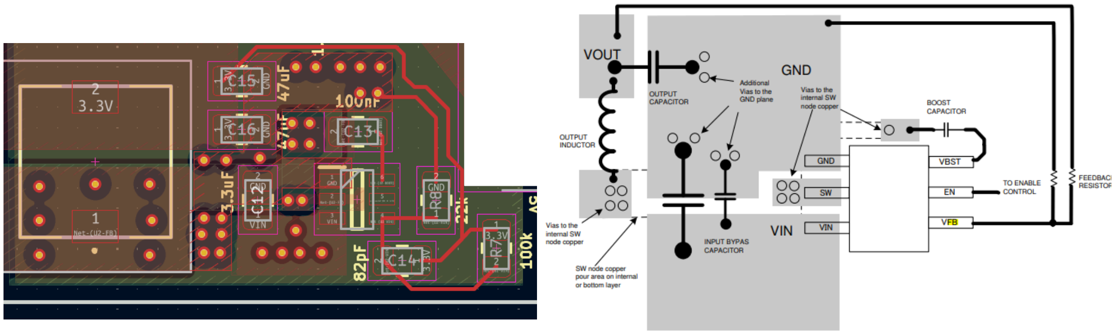


Figure 2.6: Layout Comparison of 3.3V Buck

The final 25 V to 5 V buck proved harder to debug. When the 5 V output pin of the system was probed with a multimeter, it returned a 0 V reading regardless of the board receiving power from the 24 V main. This led to the logical conclusion that the LMQ61460 was not getting power at all. We first visually inspected the components to verify there were no solder bridges, which may be the source of a short circuit, and none were found. We believe that a major shortcoming in the circuit came not in the design of the circuit but in the selection of components. The capacitors C1, C2, C3, C4, and C5 (referenced in Figure 2.7) all had a rating of 25V. This is much too close to the actual input voltage, and any spike that may occur from the source can and will break these capacitors and could directly affect the chip itself. In fact, when the resistance between the ground and the output of the chip is measured, the resistance is very low, which means the chip is no longer functional. A future iteration of the Buck would include capacitors with the correct voltage ratings to hopefully fix this issue.

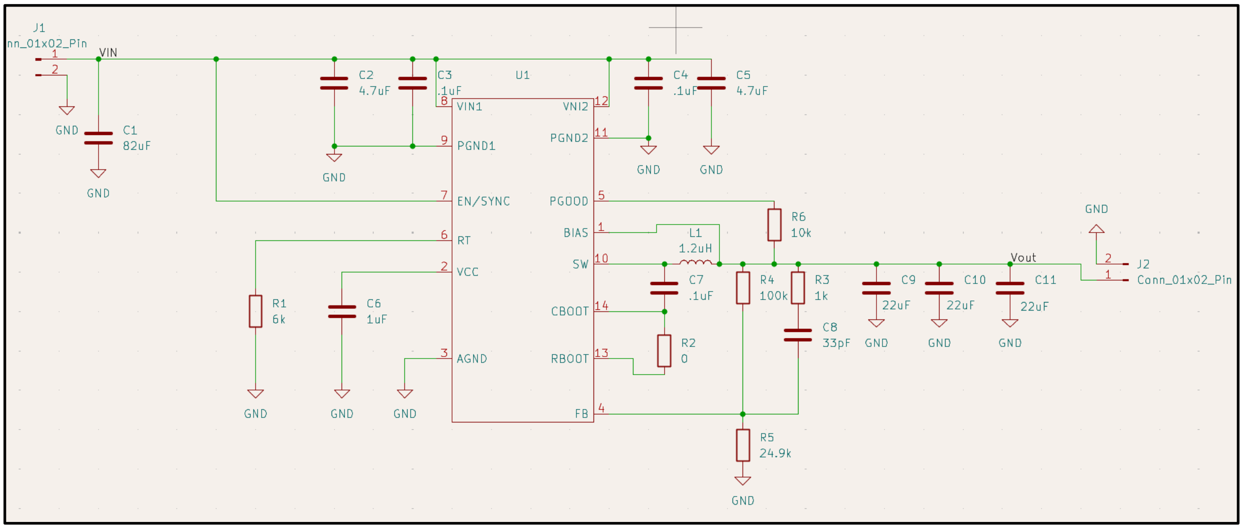


Figure 2.7: 24VDC to 5VDC Buck Converter

A future iteration of the design would also include larger chips with legs that come out to easily inspect the connection integrity at the solder joints. The use of small ICs saved a lot of space, but caused more issues than we believe they fixed. The 5 V buck specifically had the contact pads beneath the chip, which made it very difficult to see if a connection was made and if there was one at all. The overall power PCB ended up being 30 mm x 40 mm, which is much smaller than it had to be, considering that there was a lot of space for electronics at the base of the box that contained the transport arm.

3. Verification

3.1. Main Controller

The ESP32 main controller firmware should not exceed 16MB of storage to ensure headroom for future features and library growth. We built the firmware in an Arduino IDE, suitable for Arduino/ESP32 programming, then recorded the linker output reported from the Arduino IDE console.

The result was that the final compiled file size was only 0.36 MB. This is 15.64 MB under our target. This is considered a very acceptable result and leaves us with a significant amount of headroom for future development.

Additionally, we required the Raspberry PI and the main controller to have a handshake latency of ≤ 100 ms. To test this, we created a Python function within the existing UI code that would print out the latency. It does so by first recording the current time, the instant before an instruction is sent, sending the instruction from the Raspberry Pi to the main controller, waiting for the ESP32 to respond, and recording the time when the response arrives; it then takes the difference in time markers and reports the latency.

The results of this test are that the average latency was 51.74 ms with a standard deviation of 30.98 and a sample size of 420. Again, this result well exceeds our expectations. There are times when the latency is in the hundreds of milliseconds; we attribute this to the instruction arriving at the main controller when it’s in the middle of another operation, such as listening to a module. The data for this test is provided in Appendix B.

3.2. Graphical User Interface (GUI)

Our requirement for the GUI was that the touchscreen GUI should (1) create, save, and recall multi-step recipes as JSON files; (2) enumerate attached modules at startup and display their capabilities; (3) allow one-tap execution of recipes; (4) boot to a usable state in ≤ 20 s.

The final result was that the GUI reliably created and executed recipes via touch input, and recipes were saved/loaded with no data loss. Module discovery populated and operations at startup as expected; if a module did not appear, a rediscovery was applied, and the module would then populate. End-to-end readiness, which included Raspberry Pi OS boot, UI launch, and motor homing, averaged 31 s, exceeding the 20 s target; this is attributed primarily to OS boot and the homing sequence, not UI latency.

The 31 s latency can be reduced by trimming OS services, such as disabling the desktop. We elected to keep a full Raspberry Pi OS environment for flexibility; given this, the 31-second ready-to-run time was deemed acceptable for the demo.

3.3. Power Subsystem

Considering the aforementioned issues with the power PBC, we were unable to successfully implement it into our final design, which means the verifications could not be completed. Instead, we opted for a prebuilt AC to DC power module, which can provide a sufficient 12 V to the stepper motor drivers. The unit we chose was the S-600-12 because we had one on hand, and it could output a staggering 50A[10] to handle our needs with ease. The Raspberry Pi (GUI) got power from a computer that it was plugged into via USB. The USB ports on the Pi were then used to power the ESP32 DevKit3.0s that were used in place of the logic PCB, as that experienced implementation issues as well. The ESP dev boards had on-board power conversion, which would automatically and reliably step the 5 V down to 3.3 V used to power logic circuits such as those on the programmable BigTreeTech stepper motor drivers. The 5 V going into the dev boards was also used as a power source for the distance sensor used on the wafer aligner module.

4. Costs

4.1. Enclosure Costs (Appendix C)

4.2. Mechanical and Electrical Costs (Appendix C)

4.3. PCB - Power (Appendix C)

4.4. PCB - Logic (Appendix C)

4.5. Labor Costs and Product Production Viability

The total labor cost is calculated to be:

$$Ideal\ Salary(ideal\ rate)\ *\ Actual\ Hours\ spent\ *\ 2.5$$

$$\frac{\$45}{hr} * \ 60\frac{hrs}{engineer}\ *\ 2.5\ =\ \$6,750\ per\ engineer$$

Based on the cost tables, if we were to buy parts in bulk, assume a bulk discount of 10%, and a 10 % profit margin, the sticker price for our tool would be:

$$(1\ +\ Profit\ Margin)(Labor\ Cost\ +\ Manufacturing\ Cost(1\ -\ Discount))\ =\ Sale\ Price$$

$$1.1\ *\ (3\ *\ \$6750\ +\ \$755.75\ *\ (0.9))\ =\ \$23,023.19$$

However, we believe that once we finalize on an effective design, labor hours will drop significantly, reducing the overall sale cost.

5. Conclusions

The Modular Wafer Track successfully demonstrated a low-cost, automated solution for semiconductor fabrication suitable for hobbyists and small-scale R&D laboratories. By integrating a custom stepper motor system with a Raspberry Pi interface, the design met all specified performance goals for accessibility and repeatability. Despite necessary modifications to the power subsystem, the final prototype validates the modular architecture and its operational viability.

Our team successfully verified the functionality of the mechanical and logic subsystems. The stepper motor assembly achieved precise wafer loading and alignment, while the GUI reliably executed multi-step recipes without data loss. The system exceeded efficiency requirements, and the communication latency was less than we expected. These results convince us that the logic and the mechanical design are robust and capable of handling sequential fabrication tasks.

The primary uncertainty remaining lies in the custom power PCB, which failed due to component voltage ratings and layout issues. To resolve this for the final demo, we implemented a standard external power supply and utilized the ESP32 power regulators. A future iteration of the design must mitigate this by closely paying attention to component ratings as well as their recommended layouts. Additionally, while a direct UART was used for the demo to pin constraints, the design supports RS-485 for future scalability.

In adherence to the IEEE Code of Ethics, we prioritized the safety of the public and the user by enclosing the moving mechanisms to prevent injury. We also upheld the obligation to be honest in our claims by transparently reporting the failure of the power PCB and our shift to commercial modules to ensure safe operation.

References

[1] Analog Devices, "stealthChop Performance (qualitative)," Application Note AN-015, Oct. 2014. [Online]. Available: https://www.analog.com/en/resources/app-notes/an-015.html.

[2] B. Blanchon, ArduinoJson (v6). [Online]. Available: https://arduinojson.org/. Accessed: Dec. 9, 2025.

[3] Texas Instruments, "LM2678 SIMPLE SWITCHER Power Converter, High-Efficiency, 5A, Step-Down Voltage Regulator," datasheet, Rev. L, Jun. 2025. [Online]. Available: https://www.ti.com/lit/ds/symlink/lm2678.pdf.

[4] Texas Instruments, "LMQ61460-Q1 Automotive 3-V to 36-V, 6-A, Low EMI Synchronous Step-Down Converter," datasheet, Rev. C, Jan. 2021. [Online]. Available: https://www.ti.com/lit/ds/symlink/lmq61460-q1.pdf.

[5] Texas Instruments, "TPS54302 4.5-V to 28-V Input, 3-A Output, EMI-Friendly Synchronous Step-Down Converter," datasheet, Rev. B, Apr. 2021. [Online]. Available: https://www.ti.com/lit/ds/symlink/tps54302.pdf.

[6] Engineering ToolBox, “Cantilever Beam Calculations: Formulas, Loads & Deflections,” 2013. [Online]. Available: https://www.engineeringtoolbox.com/cantilever-beams-d_1848.html . [Accessed: Nov. 5, 2025].

[7] Texas Instruments, SNx4HC138 3-Line to 8-Line Decoders/Demultiplexers, SCLS107G datasheet, rev. G, Oct. 2021. [Online]. Available: https://www.ti.com/lit/ds/symlink/sn74hc138.pdf. Accessed: Dec. 9, 2025.

[8] Texas Instruments, SNx4HC125 Quadruple Buffers with 3-State Outputs, SCLS104F datasheet, rev. F, Apr. 2021. [Online]. Available: https://www.ti.com/lit/ds/symlink/sn74hc125.pdf. Accessed: Dec. 9, 2025.

[9] Texas Instruments, SN65HVD7x 3.3-V Supply RS-485 With IEC ESD Protection, SLLSE11H datasheet, rev. H, Mar. 2019. [Online]. Available: https://www.ti.com/lit/ds/symlink/sn65hvd72.pdf. Accessed: Dec. 9, 2025.

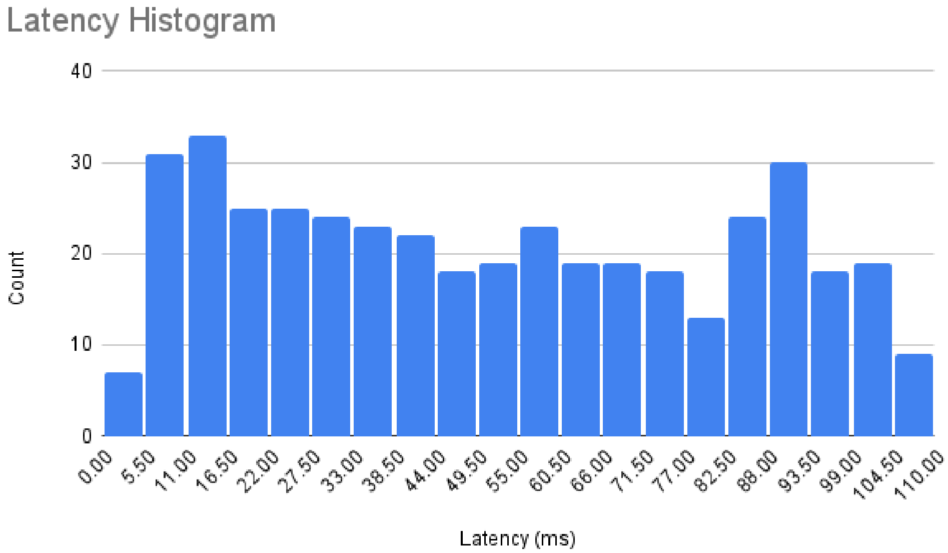
[10] WeHo, "S-600W Series Normal Single Switching Power Supply," WeHo Power. [Online]. Available: https://www.wehopower.com/product/s-600w-series-normal-single-switching-power-supply/. [Accessed: Dec. 9, 2025].

Appendix A: Example JSON Packets

Communication Vocabulary Examples

Direction and Purpose	Message (JSON)
Pi→Main (ask to discover)	{"type":"hello","want":"discover"}
Main→Module (probe on startup):	{"type":"who"}
Module→Main (basic identity):	{"type":"iam","module":"WaferAligner"}
Module→Main (capabilities and presence):	{"type":"module_hello","module":"WaferAligner","ops":["WaferAligner.AlignTo","WaferAligner.AlignFlat","WaferAligner.Zero"]}
Main→Pi (aggregate discover):	{"type":"discover","modules":{"Motor":true,"WaferAligner":true,"Spincoater":true},"slots":{"Top":"","Bottom":"Spincoater","Left":"","Right":"WaferAligner"}}
Main→Module (start an operation with parameters):	{"type":"run","module":"WaferAligner","op":"AlignFlat","params":{"angle_deg":99,"rpm":20,"timeout_ms":3000}}
Module→Main (acknowledge receipt):	{"type":"ack","module":"WaferAligner","op":"AlignFlat"}

Appendix B: Verification Data



Appendix C: Cost Breakdown

Enclosure Cost Table			
Component	Quantity	Price	Ext Price
Custom-Cut Aluminum Baseplate	1	$73.83	$73.83
2020 Aluminum Extrusions	7490.8	$0.033/mm	$247.62
M5 T-Nut 200pc Set	1	$10.89	$10.89
UCF201 Pillow Block Flange Bearing	1	$8.50	$8.50
36” x 24”x 1/8" Plexiglass Sheets (2 Pack)	1	$39.99	$39.99
FLASHFORGE Carbon Fiber PETG Filament	1	$29.99	$29.99
		Total	$410.82

Mechanical and Electrical Cost Table			
Component	Quantity	Price	Ext Price
Raspberry Pi 4	1	$64.78	$64.78
Raspberry Pi 4 Touch Screen	1	$42.50	$42.50
250mm High-Torque C-Beam Linear Actuator	1	$122.50	$122.50
BigTreeTech 2130v3.0 Stepper Motor Driver	3	3	$14.99
Aluminum Timing Pulley 1:4	1	$8.50	$8.50
APDS-9930 ALS IR RGB and Proximity Sensor	2	$2.00	$4.00
Hardened Chrome 150mm Shaft	1	$7.51	$7.51
UCF201 Pillow Block Flange Bearing	1	$10.00	$10.00
JST Connector Kit	1	$7.98	$7.98
12mm Flange Coupling Connector	1	$2.00	$4.00

DIP Socket Kit	1	$9.99	$9.99
		Total	$292.75

Power Subsystem PCB Cost Table			
Component	Quantity	Price	Ext Price
6.04kΩ Resistor	1	$0.25	$0.25
22kΩ Resistor	1	$0.33	$0.33
10kΩ Resistor	1	$0.49	$0.49
24.9kΩ Resistor	1	$0.41	$0.41
100kΩ Resistor	2	$0.40	$0.80
1kΩ Resistor	1	$0.10	$0.10
0Ω Jumper	1	$0.22	$0.22
3.3µF Capacitor	1	$0.51	$0.51
22µF Capacitor	3	$0.68	$2.04
47µF Capacitor	2	$0.61	$1.22
82pF Capacitor	1	$0.35	$0.35
33pF Capacitor	1	$0.22	$0.22
0.1µF Capacitor	4	$0.27	$1.08
4.7µF Capacitor	2	$0.34	$0.68
1µF Capacitor	1	$0.17	$0.17
82µF Capacitor	1	$0.53	$0.53
10µH Inductor	1	$1.16	$1.16
1.2µH Inductor	1	$3.13	$3.13
LMQ61460-Q1	1	$5.41	$5.41
TPS54302	1	$1.85	$1.85
PJ-063AH	1	$1.33	$1.33

Power Subsystem PCB Cost Table

Component	Quantity	Price	Ext Price
6.04kΩ Resistor	1	$0.25	$0.25
22kΩ Resistor	1	$0.33	$0.33
10kΩ Resistor	1	$0.49	$0.49
		Total	$22.28

Logic PCB Cost Table

Component	Quantity	Price	Ext Price
100Ω Resistor	6	$0.10	$0.60
10KΩ Resistor	4	$0.10	$0.40
120Ω Resistor	3	$0.10	$0.30
680Ω Resistor	6	$0.10	$0.60
1μF Capacitor	3	$0.17	$0.51
0.1μF Capacitor	6	$0.30	$1.80
SM712-02HTG	3	$0.64	$1.92
SN65HVD72D	3	$3.38	$10.14
ESP32-S3-MINI-1U-N8	1	$6.15	$6.15
SP4045-04ATG	1	$0.50	$0.50
SN74LVC1G139DCTT	2	$1.19	$2.38
SN74HC125N	2	$1.26	$2.52
1-1734035-1	1	$1.60	$1.60
B3F-1000	2	$0.24	$0.48
		Total	$29.90