Smart Sprinter

Capstone Final Report

Team NPGSO

By: Nick Flora, Patrick Gajewski, Garrett Delaney,
Shah Zaib Hashmi, Owen Singley

Statement of Work

Nicholas Flora (N):

I fully designed the height laser sensor. I first ordered an infrared laser sensor receiver and transmitter system that would cover the necessary width of a track lane with the proper voltage (9 V). I then tested the laser transmitter and receiver to fully understand how it worked as it did not have an accessible data sheet. With the voltages found, Shah Zaib and I then designed a comparator on the PCB to have the PCB output either 0V or 3.3V into the STM32 based on the input of the transmitter, which we then confirmed worked with testing. Owen and I designed it so that 0V (falling edge) flagged the STM32 that the runner was too high and blocked the laser.

Aside from the laser system, I did a lot of miscellaneous tasks and assisted other members of the project, as the laser was one of the first items completed. I came up with the initial design to embed the force track sensors into the track block, which was slightly modified by Owen later through testing. I also designed the container for the PCB, STM32, Battery pack and buzzer. I also assisted Owen and Garrett with force calibration. Finally, in addition to assisting in the comparator design with Shah Zaib, Shah Zaib and I coordinated with WWW electronics to get the components soldered on the board.

Patrick Gajewski (P):

I fully designed the power system used for the circuit. This included the input power of eight 1.5V batteries in series and the buck converters and linear dropout regulators to bring the voltage down to usable DC power rails. I also soldered all the test points onto our PCB and added the extra through-hole components that WWW did not solder. After, I checked to make sure there were no cold solder joints and confirmed that all the nodes were as designed from the PCB Gerber files. I did this by using the short indicator on the multimeter. After, I ran tests to ensure the pressure sensors' hardware was working properly and confirmed that it was. In addition, I helped debug some software coding issues. For example, I helped directly in solving the byte dropping error and fixing the error with the height sensor interrupt.

Garrett Delaney (G):

I was in charge of developing the graphical user interface (GUI) companion application for the laptop, created in Python. To make the application, I utilized the wxPython GUI framework, which is a Python wrapper for the C++ framework wxWidgets. I also used the pySerial library to gain serial port access in Python, since the data is sent from the STM32 microcontroller to the laptop via a universal asynchronous receiver/transmitter (UART). While Owen led the embedded code development for the STM32 in C, I helped and worked closely with him on that as well, since our two programs had to be interoperable. I developed the packet format which we use to transmit data between the microcontroller and the laptop. Owen and I created a transport layer on top of UART to ensure reliable data transfer which includes a cyclic redundancy check (CRC) for data corruption and a timeout for data loss. I handled all of the references throughout the report for the team with Zotero.

Shah Zaib Hashmi (S):

I was responsible for designing the start module circuit for our prototype. This module consisted of two systems, a buzzer and a microphone, which would both be used to calibrate time=0 for a run. The buzzer system consisted of a piezoelectric buzzer and a driving IC. The STM32 would send a 3.3V enable pulse to the IC, causing the buzzer to produce a loud tone for the duration of the pulse. The microphone system consisted of an electret microphone and four op amp circuits that in sequence amplify, peak detect, buffer, and compare the signal from the microphone to output either 3.3V or 0V to the STM32 depending on the detected sound volume. I constructed the KiCad schematics, selected all components involved, conducted simulation and breadboard testing, and tested the final PCB implementation of these systems.

In addition to this, I was also responsible for designing and manufacturing our PCB. I first compiled schematics in KiCad for the start module, laser sensor, pressure sensors, and power system. I selected and modified the components within the individual schematics to best fit the constraints of our power rails. I chose the footprints and did the layout of the board. We opted for a shield design that mounts directly to the header pins at the bottom of the STM32 so I placed and measured corresponding headers. I made some of the more variable components, such as gain or biasing resistors, through holes so that they could be modified post manufacturing. I selected the male and female connectors for the various external sensors so that they could be attached and detached from the PCB. I chose the battery pack for our system and its connection to the PCB. I designed the test points that flank the board and allow for voltage testing of each input, output, and intermediary stage for all modules. I prepared a bill of materials (BOM) for all PCB components as well as the assembly drawings required for manufacturing at 3W.

Owen Singley (O):

My primary responsibility was designing the microcontroller software and integrating the force sensors with this software. Initially I performed research to determine which force sensor and microcontroller best fulfilled the team's design requirements. Once we purchased the force sensors and microcontrollers, I began designing the software system in the C programming language. This involved combining STM32 timing programs with general-purpose input/output (GPIO) interrupts and inter-integrated circuit (I2C) programs. While I took the lead on all the preliminary STM32 programming and developed the first working test of the system, I worked closely with Garrett on much of the software development. Garrett and I specifically spent much of our time together debugging the interoperability between my STM32 program and his Python graphical user interface (GUI). Beyond developing software, I also designed much of the force sensing system. This involved soldering wires to the sensor power and communication pins, embedding the sensors into the foot pads of the sprinting block, and calibrating the sensors with known force values.

Table of Contents

Table of Figures

Table of Tables

Abstract

The Smart Sprinter aims to help a sprinter improve their short-distance sprinting start by analyzing three quantifiable metrics. This will be done using a variety of sensors and a centralized microprocessor. Force sensors will obtain the force produced by the runner during the push off. The force data over time will be used to generate a block exit time statistic and a height sensing laser will report the sprinter's height during the start. This data will be displayed within a Python graphical user interface (GUI). Our primary goal is to provide sprinters with an advanced training tool to help them win more races.

Background

Analytics and data have taken over the sports world in recent years. Athletes are constantly searching for new technology to add to the array of training tools at their disposal. Sports analytics aim "to gather and analyze player and team stats," helping the players to "outsmart their opponents and get the winning result" [1]. Our product seeks to give sprinters in particular an edge over their competitors by providing them with real-time analytics detailing how efficiently the sprinter starts off the block.

In a sprinting race, the start is arguably the most important phase of the race in determining finishing positions. Sprinters must have a quick reaction, a strong push off the block, and they must keep their body shifted forward to maximize the horizontal force produced. Prior studies have shown that these three kinematic parameters, while not the only parameters of significance, have a large impact on how well a sprinter runs the race [2]. Quick reaction to the starting pistol allows the sprinter to begin accelerating as fast as possible. If the sprinter can couple this quick reaction with a strong push off the starting block (Fig. 1), the sprinter will likely jump to an early lead in the race. Finally, staying low and shifting weight forward at the start of the race can help the sprinter maximize the horizontal force produced when pushing off the starting block. As a result, we decided that our product will focus on measuring and presenting the sprinters with three key data points for training assistance: force, block exit time, and whether they keep their body shifted forward (measured through height). Block exit time is defined as when the sprinter leaves the block [3] and is closely related to reaction time, which is when they start reacting to the stimulus of the start pistol.

Fig. 1. Sprinting Start Block

Unlike previous projects that focus on capturing motion data during the middle of the race, our project focuses solely on the start of the race [4]. Additionally, these previous projects used smart sensors as wearable technology whereas our project will remain static and affixed to the ground. Our vision is to create a simple training tool for sprinters that modifies a sprinting starting block to measure the three key kinematic parameters of a sprint start and display these statistics in a clean and satisfying manner to the user.

This project will draw on knowledge gained from our various electrical and computer engineering classes throughout our time in university. The circuit schematic, design, simulation, and PCB layout will use our knowledge from the Fundamentals (FUN) series. Working with the STM32 microcontroller will use our skills developed in Intro to Embedded as well as Computer Systems and Organization (CSO 1&2) since both those classes used the C programming language. Regarding the laptop-side companion application for data collection and representation, we will use skills from Data Structures and Algorithms (DSA 1) and Software Development Essentials (SDE).

Project Description

Performance Objectives

The Smart Sprinter's objective is to gather three key pieces of data from a sprint start and present the data on an application that is easy for the user to use. The three key pieces of data are force, block exit time, and the height of the runner off the start. Block exit time is a piece of data that has changed from the original proposal. The Smart Sprinter was originally proposed to gather reaction time; however block exit time was easier for the group to automatically calculate for the end user. A runner could have a fast reaction time but be very slow at actually leaving the blocks. The block exit time combines these two times; to let the user know how quickly they reacted to the start noise and completely cleared the starting blocks [3]. The force over time was

gathered from the force sensors and presented in a graph that the user can analyze. This was also a shift from the original proposal, as the group found it useful for the user to have access to 3 seconds worth of force readings rather than the single maximum force value initially proposed. Furthermore, the user can still analyze their peak force per foot, and also how long each individual foot took to leave the block.

The purpose of the laser height sensor was to have the runner stay low when running after they have started. The idea is that the track blocks can only measure the initial force off the blocks, but not the trajectory of the force. Ideal starts have the force translated in a horizontal direction, where runners are trained to stay low off the start. This stops the runner from "jumping up" off the blocks, as most amateurs do, and instead explode forward, increasing acceleration and making the start faster. The runner will set the distance away from the block and the height of the sensors based on their form, height, and preference from training previously. Further research in this can be found in reference [5] this result is also shown on the application for the user to document.

Power

Power is integral to all systems; without it nothing would work. To create the power system, we relied on information given to us from ECE 3750: Fundamentals of Electrical and Computer Engineering. The system power is based on 12V input from 8 AA batteries in series with each other. This allows Smart Spinter to avoid using wall power, as track fields do not necessarily have power outlets close to the track itself. The 12V input is then converted to 3 power rails. First, is a 9V power rail. Since a 3 volt drop in voltage is not too high, a linear dropout regulator was used as it is simple and the power efficiency loss is minimal (Fig. 3). This power rail is used to power both the laser and the speaker. Next, a 3.3V rail is required to power many subcircuits including the STM32 microcontroller, the comparator, the FX29 force sensors and the linear dropout regulator for the 2V power rail. This power rail was obtained by using a buck converter to drop 12V to 3.3V (Fig. 2). Despite the extra space and components required, a buck converter is used because an 8.7V drop over a linear dropout regulator (LDO) has horrible power efficiency and a high-power voltage drop off. Power efficiency is more important than space because the goal is to make the batteries last as long as possible. Finally, a 2V rail is created using a linear dropout regulator from the 3.3V LDO (Fig. 4) This power rail is unused as it is an artifact of the microphone which we removed in our final design.

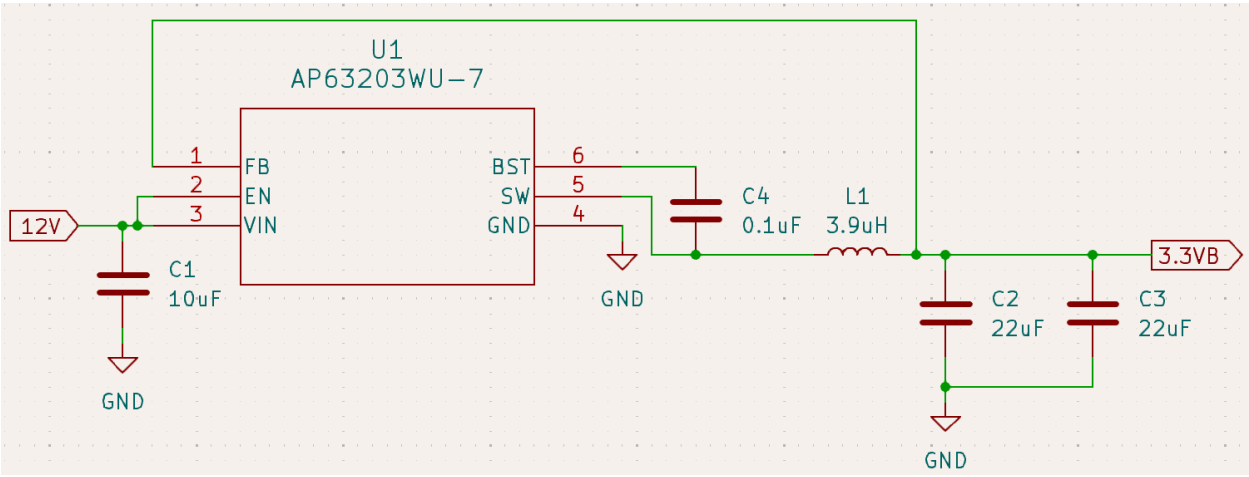


Fig. 2. 12-to-3.3V Buck Converter Schematic

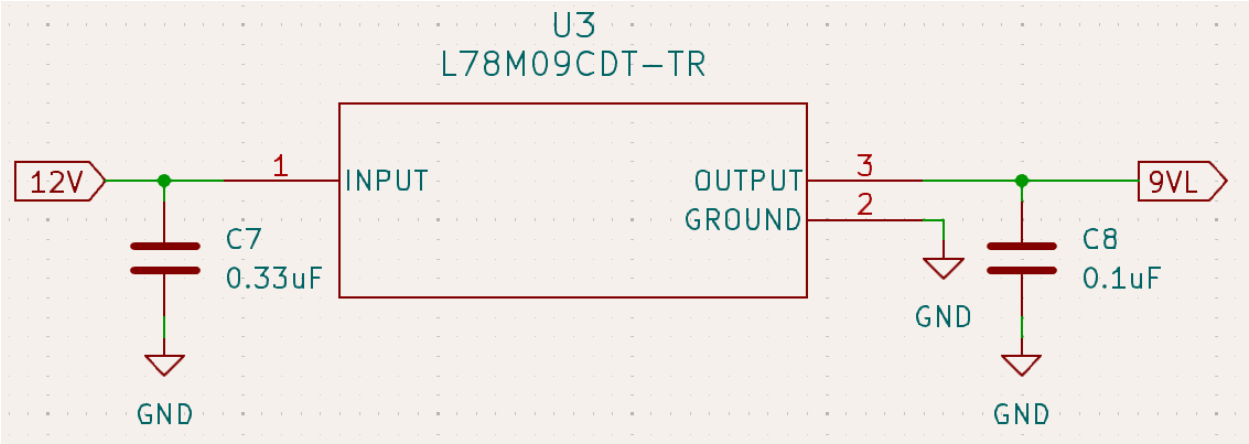


Fig. 3. 12-to-9V Low-Dropout Regulator (LDO) Schematic

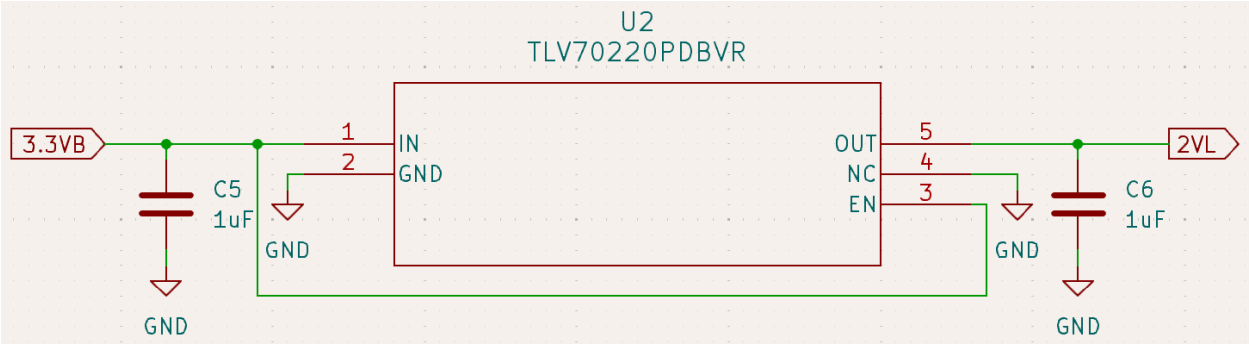


Fig. 4. 3.3-to-2V Low-Dropout Regulator (LDO) Schematic

Start Module

The start module is an essential part of the Smart Sprinter system as it allows for measurement of the initial time value necessary for calculating block exit time, one of the three data points our device gathers for runners. The initial time value can be thought of as T=0 for the user; it is the point at which the user has been alerted that a run has begun. The delta between this time and the time at which the peak force is detected is used to calculate the block exit time data point. The module consists of two independent circuits, a buzzer circuit and microphone circuit. The purpose of having these two different circuits within the same module is to provide the user with the option to operate the system either internally, through the companion program, or externally, through something like a whistle.

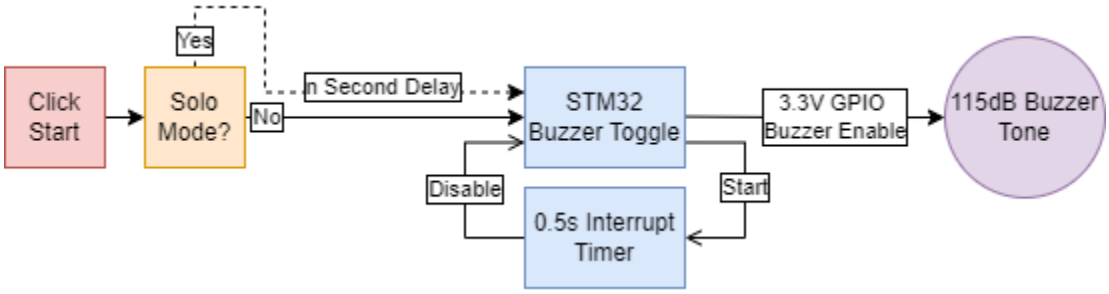


Fig. 5. Buzzer Operation Block Diagram

The buzzer circuit consists of a piezoelectric buzzer that outputs a loud audio tone when powered on. This is designed for an internal start initiated through the python interface program. After the start button is pressed within the program, the laptop either sends a signal immediately, or if operating in solo mode after a delay, to the microcontroller which initiates a brief pulse on the buzzer's enable line as shown in Fig. 5. This produces a sound pulse intended to mimic a coaches whistle or a track starting pistol, alerting the runner that they need to start running. The specific model of buzzer chosen is capable of a maximum output volume of 115dB [6]. This volume at first glance appears to be unnecessarily loud, but it was chosen for two distinct reasons. First, the volume of an actual track starting pistol once it reaches the ear of a runner is around 130dB - 145dB, several orders of magnitude louder than the buzzer[7]. We wanted our design to be as representative of actual track conditions as possible so this choice of buzzer makes sense. Second, it is quite simple to decrease the volume by physically obstructing the buzzer, but there is no way to make it louder. We did not want to risk the buzzer being too quiet once it was in real world conditions with ambient noise, lower than expected voltages, and the physical construction of the hardware mount.

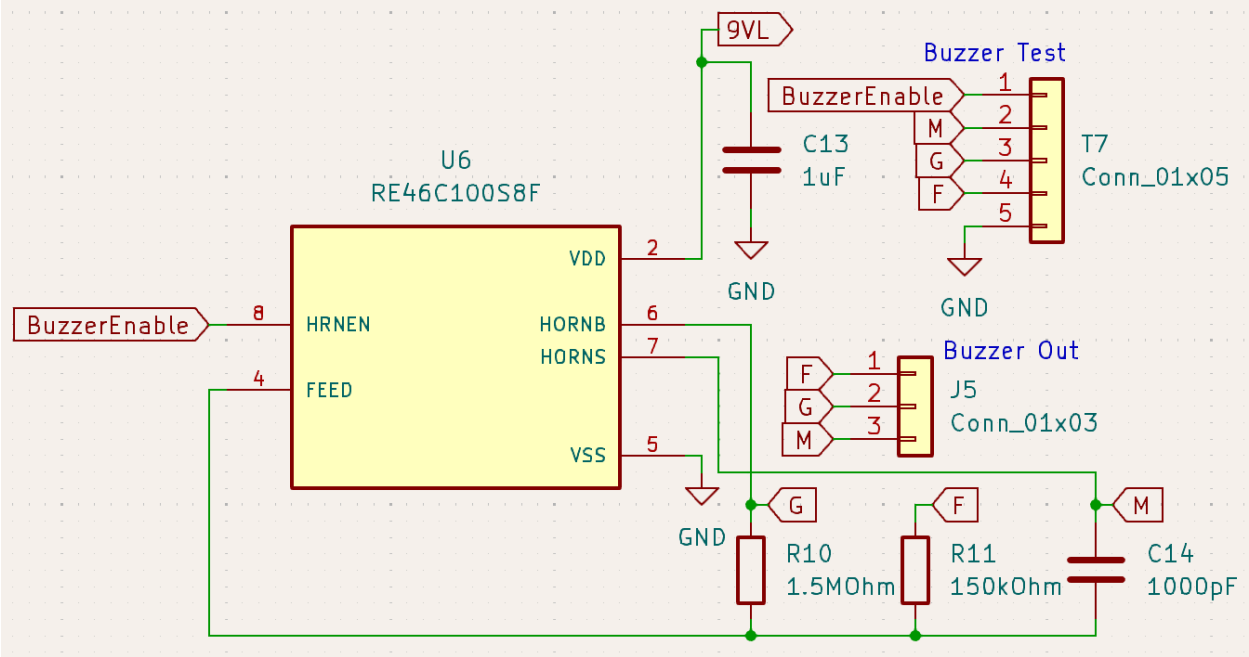


Fig. 6. Buzzer Circuit Schematic

The buzzer circuit schematic is shown in Fig. 6. The buzzer itself is not within the schematic but its three pin input is represented by the F G M labels (datasheet did not state what these stand for). The buzzer datasheet [6] recommended the usage of the RE46C100S8F horn driving IC to safely power and operate the buzzer. We opted to include it and go with the typical application circuit as there was no special functionality that we needed. The component values of R10, R11, and C14 were chosen based on the typical application circuit. The buzzer operates on a 9V rail with C13 as a bypass capacitor to produce the outputs described in the datasheet. The signal labeled BuzzerEnable comes from the microcontroller as a 3.3V GPIO pulse that determines the duration of the audio pulse. The J5 connector serves as the physical connection point for the buzzer while T7 serves as a test point.

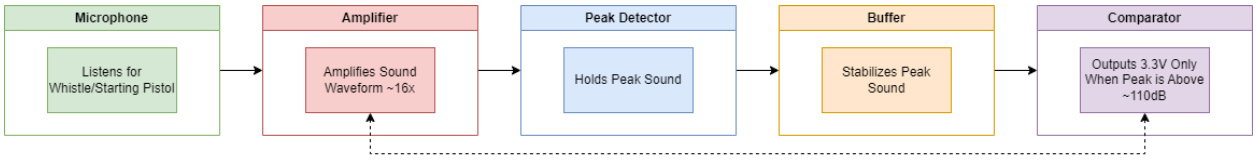


Fig. 7. Microphone Circuit Design Block Diagram

The microphone circuit consists of an electret microphone and several stages of op amp filtering that continuously detects loud audio impulses within the environment as shown in Fig. 7. This is designed for an external start initiated through a second person using a whistle, starting pistol, or similarly loud impulse. The microphone detects a raw sound signal which gets amplified, peak detected, buffered, and compared to a reference value in order to provide the microcontroller with a stable reading once a sound above a certain threshold is detected. The

addition of this circuit allows the Smart Sprinter to be operated by an athlete and coach in a realistic competition like environment, making it a more effective practice tool.

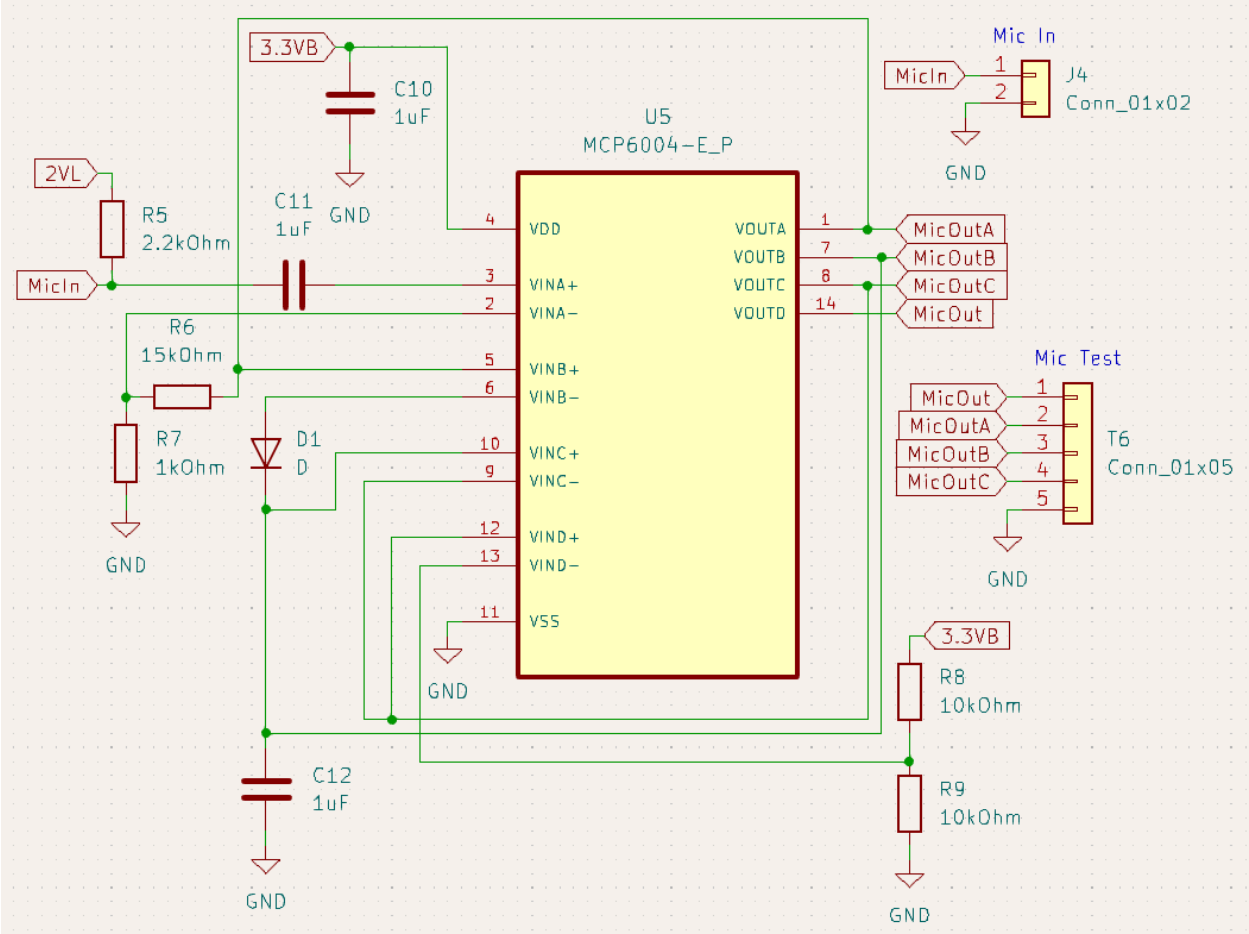


Fig. 8. Microphone Circuit Schematic

The microphone circuit schematic is shown in Fig. 8. The microphone is not within the schematic but its output pin is represented by the MicIn label. The microphone is powered by a 2V rail and has the DC bias from this rail removed from its output through C11 as described in the microphone datasheet typical application circuit [8].This AC waveform feeds into a non-inverting amplifier with a gain of ~16 configured on op amp A. This gain value is based on a calculation of the mic sensitivity, -36dB, and the target volume threshold, ~110dB. With this gain, an input volume of ~110dB would produce a ~0.1V output from the microphone which would then be amplified to ~1.6V.

$$V_{Mic} = 10^{(\frac{-36}{20})} * 10^{(\frac{110}{20})} * 20 * 1 \approx 0.1V$$

The output of the amplifier feeds into a diode capacitor peak detector configured on op amp B. This holds the peak value from the amplifier for further processing. This peak value feeds into a unity gain buffer on op amp C. This helps stabilize the waveform ensuring the peak does not drop too fast. The buffered waveform feeds into a final comparator configured on op

amp D. This comparator sets the output to 3.3V when the signal is above a 1.65V threshold and 0V when the signal is below the threshold.

The buzzer circuit fully works as intended, producing the proper sound pulse when triggered by the STM32. The microphone circuit however, does not work as intended and was thus not fully integrated to function with the STM32. An explanation as to why this is can be found in the Final Results section.

PCB

The PCB within the Smart Sprinter serves as the central interface between all sensors and the microcontroller. We chose to design the PCB with KiCad as it offered a good balance of functionality and simplicity. We first began by compiling the individual system schematics in the KiCad schematic editor. Schematics for the power, buzzer, microphone, and laser as shown in Fig. 2, Fig. 3, Fig. 4, Fig. 6, Fig. 8, above and Fig. 17 below respectively were put together in one file.

We created an additional schematic for the two I2C pressure sensor connections and added pullup resistors for the SDA and SCL lines as shown in Fig. 9. Due to the sensitivity of the I2C data and the potential ripple of the 3.3V buck converter, we opted to place two bypass capacitors of varying sizes in parallel near the power connections for increased stability.

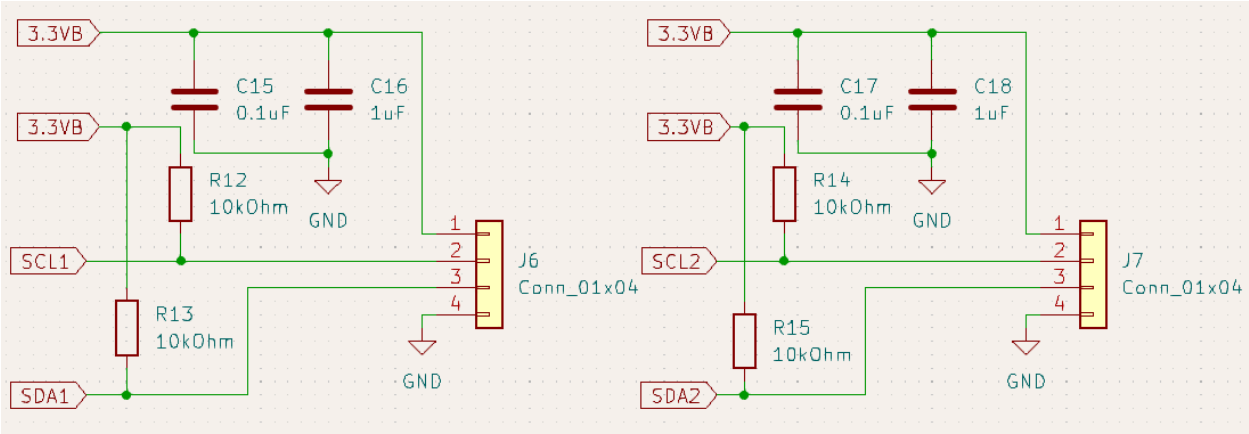


Fig. 9. Pressure Sensor I2C Pull Up Resistors and Connector Schematic

We created schematics for the power rail tests pins and battery input connector as shown below in Fig. 10.

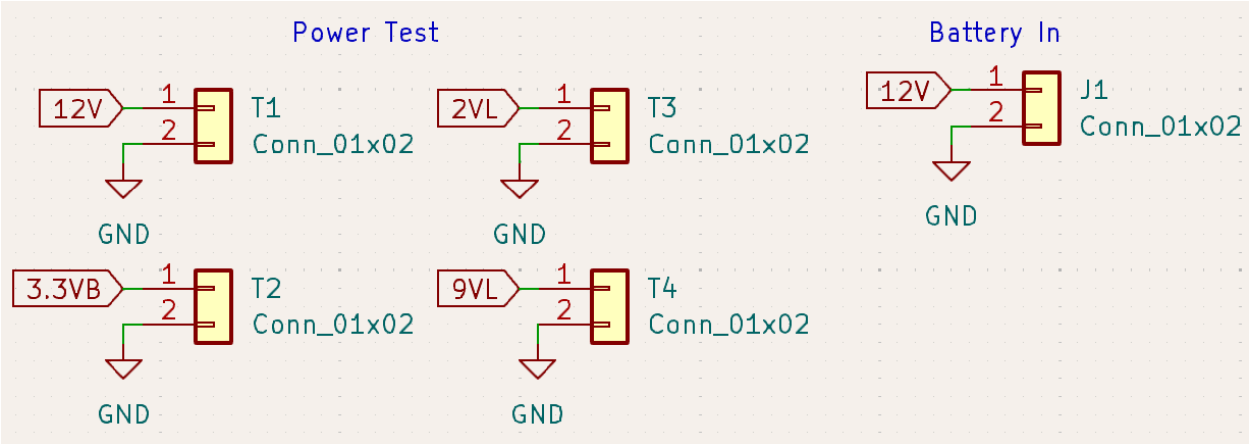


Fig. 10. Power Test Pins and Battery Connector Schematic

We chose to design the PCB as a shield that mounts to the header pins on the bottom of the STM32. This complicates the design somewhat as we now have to consider the physical spacing between components, but offers the benefits of minimizing the number of connections to the STM32 and an overall cleaner final design. Fig. 11 below shows the schematics for the headers and the STM32 pins that have connections.

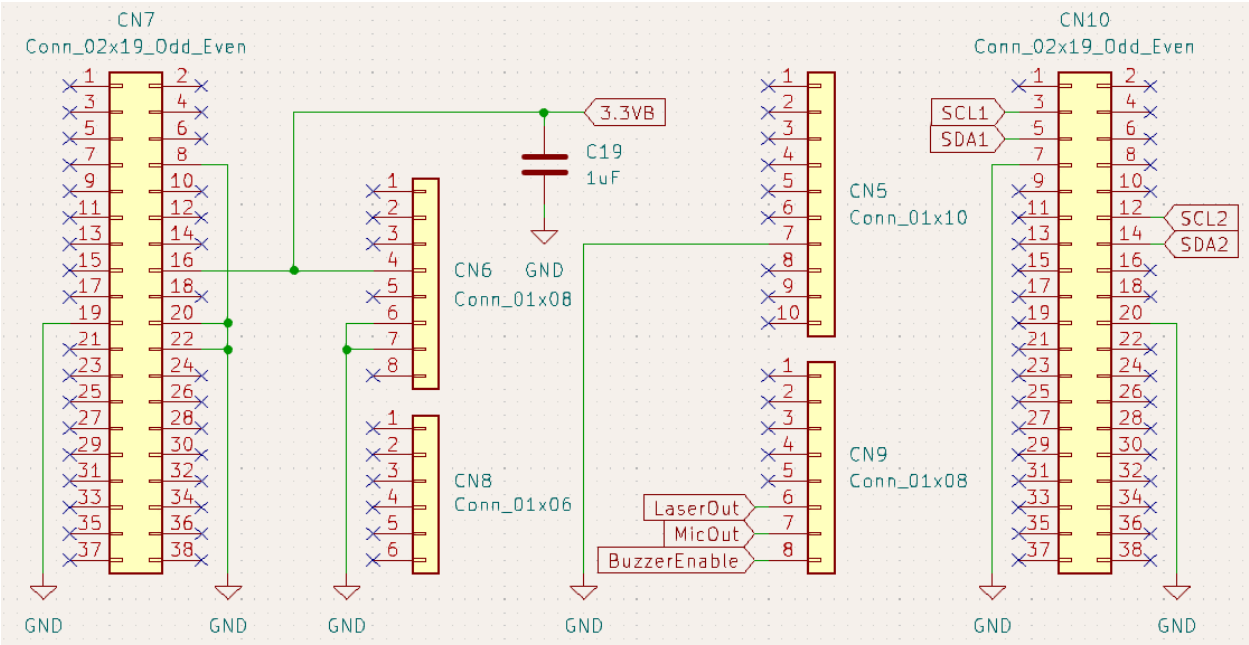


Fig. 11. PCB Shield Header Connector Schematic

After completing the schematic, the next step to designing the PCB was selecting footprints for all of the components. These footprints are the physical representation of the circuit components and are what get arranged on the PCB.

We started by selecting the female header connectors as these had to match the exact specifications of the STM32. We found components on DigiKey with the same 2.54mm pitch of

the STM32 pins and added them to our BOM. Equivalent footprints were found in the standard KiCad library and imported into the PCB editor. This process was repeated for the male test points. For the non power ICs, op amps, comparator, and horn driver, we opted for chip holders instead of a direct connection for easy troubleshooting and replacement if a system broke or was overloaded.

Next, we selected the connectors for the various sensors. This process was a little more involved than the rest of the footprints as we had to find matching pairs of male and female connectors on DigiKey. We chose JST brand connectors for the battery pack, microphone, laser, and buzzer as they were readily in stock, relatively minimal, and provided an easy to understand scheme for the male and female counterparts. We chose Molex brand connectors for the pressure sensors since at this point in the project we had already ordered a set of Molex style I2C cables to extend the short built in wires for testing.

We then selected footprints for the buck converter and two LDOs by importing EDA files available on SnapEDA into KiCad [9]. The footprints for the passive components were split into through hole and surface mount based on their role within the circuit. Components that impacted bias voltages or gain values were set as through hole to allow for adjustments post manufacturing. All other passive components were selected as surface mount to save on space. For all of the passive components, footprints were imported from the standard KiCad library and equivalents were found on digikey.

Now that the footprints were selected, we exported the schematic files into the PCB editor of KiCad. We started by configuring the board setup to match the constraints of our PCB manufacturer JLPCB. We opted for a four layer stackup as it did not have a significant impact on cost and allows for easier routing of our three different power rails. We selected 0.1524mm, 0.3048mm, and 0.2286mm track widths for the digital, power, and signal traces respectively. The board does not draw nearly enough current to require adjustment to the default track width but these adjustments helped us visualize the various connections.

To lay out the board, we began by placing the shield header connectors based on the measurements provided in the STM32 Nucleo G071RB mechanical drawing [10]. These measurements were critical as any deviation from the technical specifications would result in the board being unable to properly connect to the microcontroller. Once we had these in place, we locked them down and began to group the other components by their functionality and subcircuits. We then placed the sensor and battery connectors strategically around the board based on what would minimize the path length in the final track block. The pressure sensor connectors were placed on the bottom as this is closest to the footpads. The laser connectors were placed near the bottom facing outward as they would go out and then forward, connecting to the laser stands. The microphone, buzzer, and battery pack connectors were placed near the top of the board furthest away from the runner.

Next, we placed the power components prioritizing a placement near the 12V battery input for the 12V-9V LDO and 12V-3.3V Buck converter. The 3.3V-2V LDO for the microphone was placed near its connector as that was the only place it is used in the circuit. We then placed

the ICs in the open central areas as close to their connectors as possible. It took several iterations of passive component placement around the ICs to get a version that minimized overlapping paths in the ratsnest. We placed the test point pins on the outside edge of the header connectors closest to them so that they could be accessed even while the microcontroller was mounted to the PCB. We finalized the layout by defining an edge cut around the board, creating a final size of 80mm by 80mm. Fig. 12 below shows the board with all components placed in respective locations.

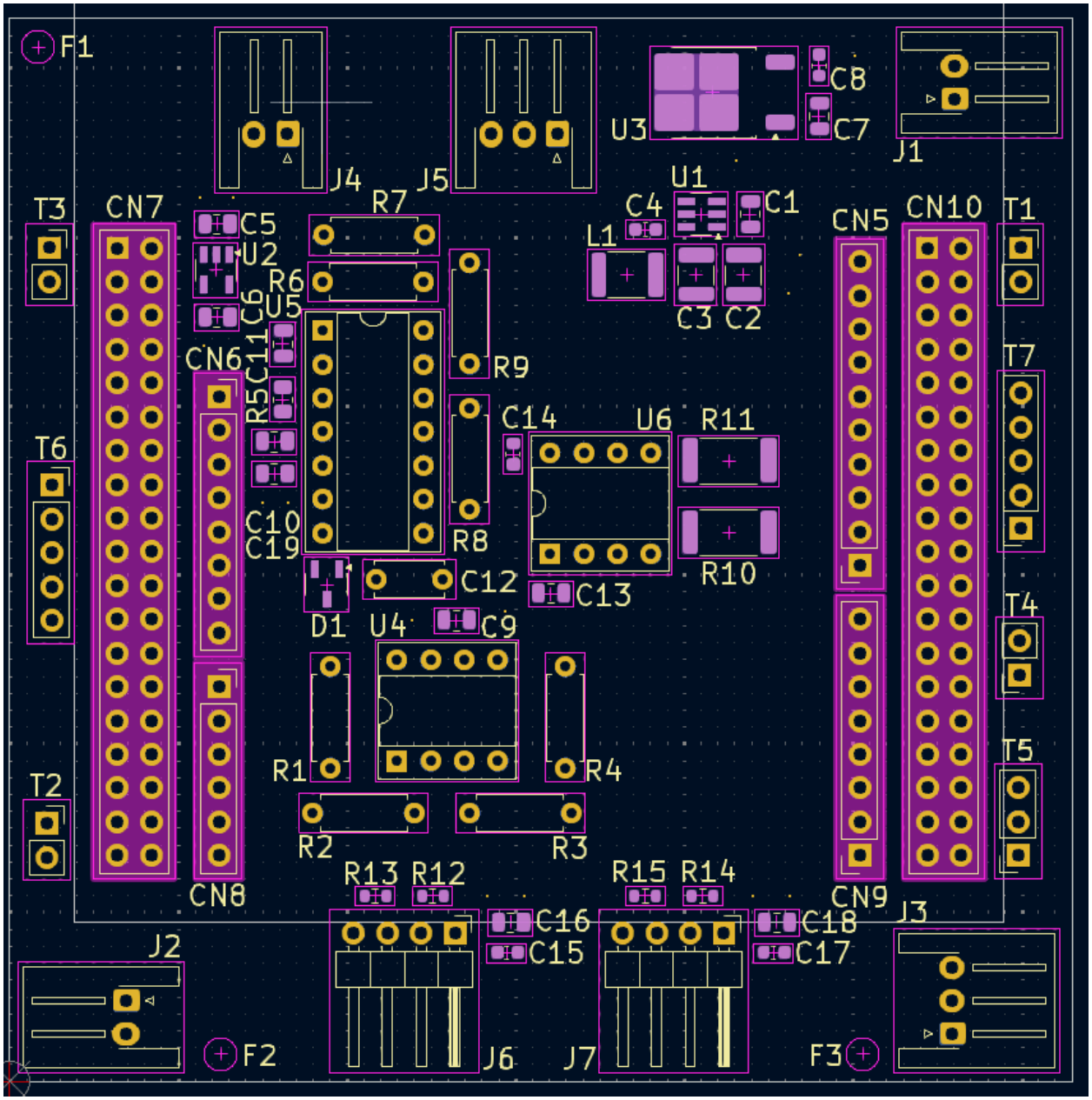


Fig. 12. PCB Component Layout

After completing the component placement, we began to route the connections. We started by creating copper fills for a ground and 3.3V power planes on layers 2 and 3 respectively. This allowed us to leverage the four layer stackup effectively and eliminated the most frequent connections. The surface mount components were unable to connect to these planes as they exist exclusively on layer 1 of the board. We first routed the surface mount connections within the individual power systems and then used vias to connect them to the ground or power plane as necessary.

Now with the power systems setup within themselves, we routed the 2V and 9V power to their respective components. The 2V rail was routed on the surface due to the short path length to the microphone. The 9V rail was cut into the ground plane and routed along the edge of the board as it had to travel all the way across the board and we wanted to minimize any interference with other surface traces.

Next, we routed the I2C connections to their respective passive components and points on the STM32 headers. Then we routed the laser comparator connections and buzzer IC connections again starting with internal passive component connections and then STM32 header connections. The microphone circuit was routed last as it had the most complexity with its connections. Wherever a surface trace couldn't be used layer 4, the back of the board, was used. Layer 4 was also used to route a majority of the test point connections on the edges of the board. Again vias were used whenever a surface mount component could not reach a connection it needed.

With the routing complete, the final step in the PCB design was inspecting the routing for any errors and adjusting the silkscreen layer to be readable. This process was tedious but relatively quick compared to the routing. The fully routed four layer board can be seen in Fig. 13 below. Additional figures for each individual layer can be found in the Appendix under Fig. 61, Fig. 62, Fig. 63, and Fig. 64.

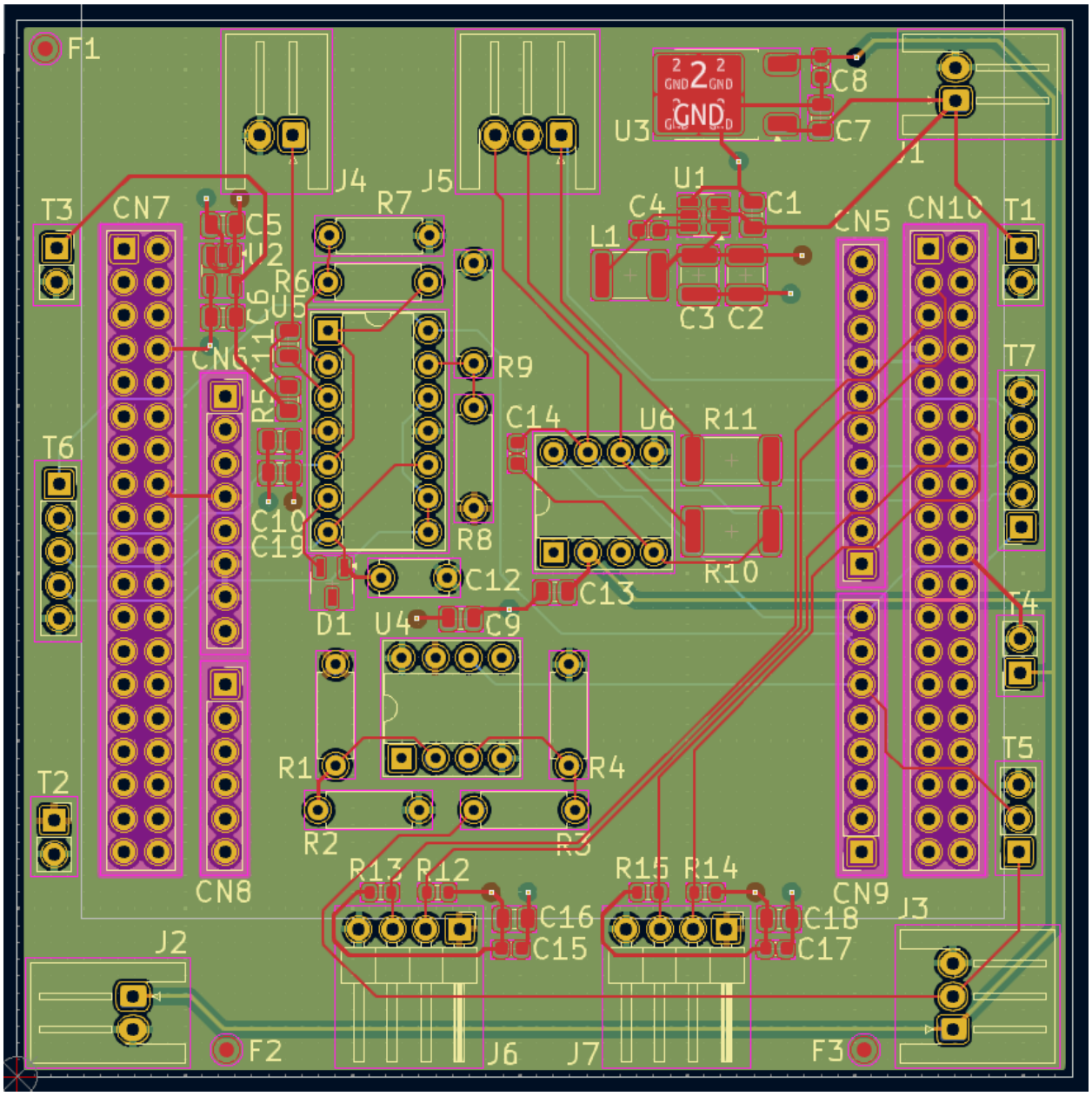


Fig. 13. Full PCB Layout and Routing

With the PCB design complete, we now had to finalize it for manufacturing with JLPCB. We exported the Gerber files from KiCad and formatted them in the way they were requested. We uploaded the files to the JLPCB website and selected the specifications with which we wanted the board manufactured. Nothing on the board was particularly technical with its design so we opted for the most basic specifications they offered for four layer boards. The unpopulated PCBs that were manufactured can be seen in Fig. 14 below.

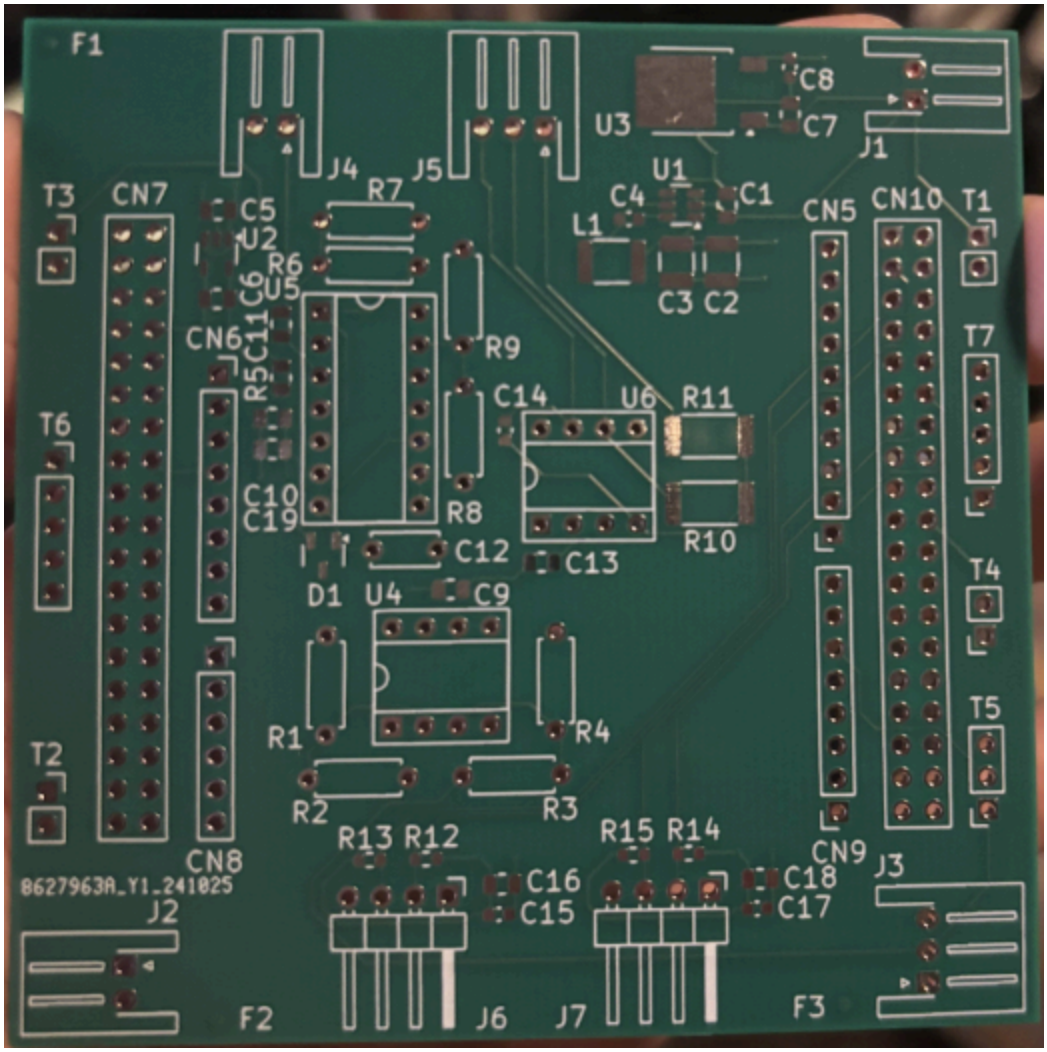


Fig. 14. Unpopulated PCB

Now that we had the PCB manufactured, we prepared it for assembly with 3W Electronics. We created an assembly schematic that showcased the placement of each component in a readable, printable format, a BOM that explained the values each component corresponds to, and some paperwork to identify our board and team. These documents, our unpopulated PCB, and components were sent to 3W. They were returned to us with all components soldered on. Fig. 15 below shows the board in a semi populated state with the headers and chip holders in place.

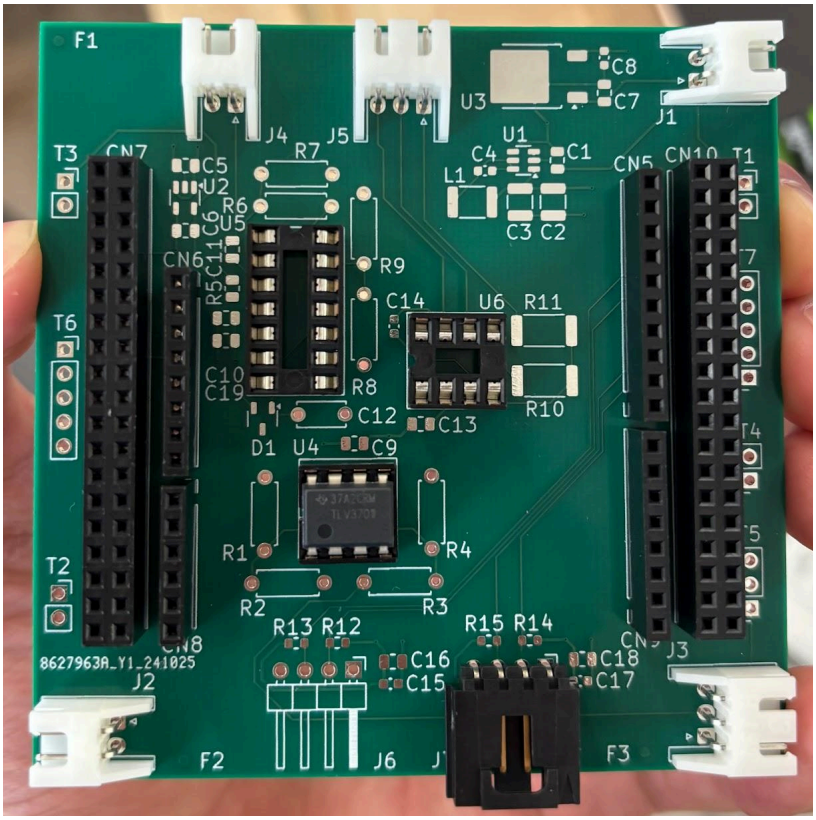


Fig. 15. Partially Populated PCB

With a fully populated PCB, we now had to ensure that everything was connected as intended. The test points made this process straightforward as we could simply run down them and see if the correct voltages read out. All points passed connectivity testing and every functioning subsystem came out to the values we expected.

Laser Height Sensor

The laser height sensor is the third data point that the track system gathers, giving the runner more details on optimizing their start. The idea behind the laser height sensor is to make sure that after a certain distance from the track blocks, the runner is running below a set height. If the runner is too upright after starting off the track blocks, the laser is blocked by the runner's upper body and flags the start as “BAD”. If the runner remains low off the blocks and goes under the laser, the start is flagged as “GOOD”. This means the runner’s body is shifted forward enough and their force will be properly translated horizontally.

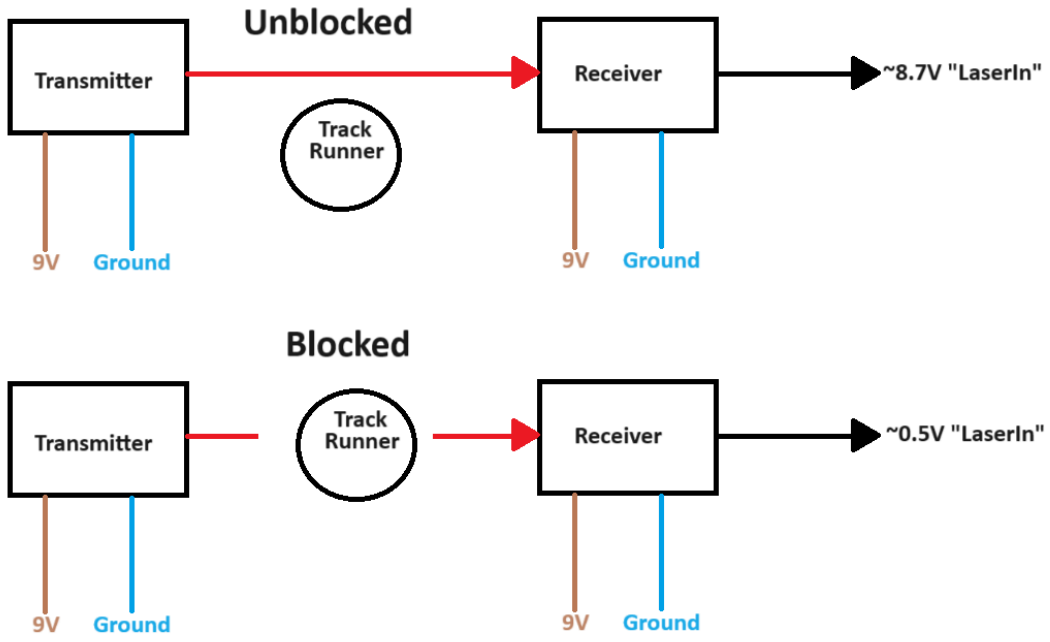


Fig. 16. Wire setup block diagram, blocked and unblocked

The laser setup works by powering an infrared transmitter laser and receiver laser [11] with 9V. Nine Volts was the value settled on as this gave the laser setup more than enough power to cover the width of a track lane (roughly 2 meters), while not risking the power dropping below 6V (which would turn the lasers off). Additionally, this power rail was already designed for the starting buzzer. The receiver laser had a third wire that would output around 8.7V when the transmitter hit the receiver, and around 0.5V when blocked. This setup and the two scenarios are seen in Fig. 16 above. A notable design change from the initial proposal was to have this voltage feed into a voltage divider that feeds into a comparator on a PCB, instead of just a simple voltage divider. This was due to inconsistencies in the voltage output of the receiver wire when testing, and the input of the STM32 needing to be a true 0V when blocked and not 0.5V (or anything lower, but higher than 0V).

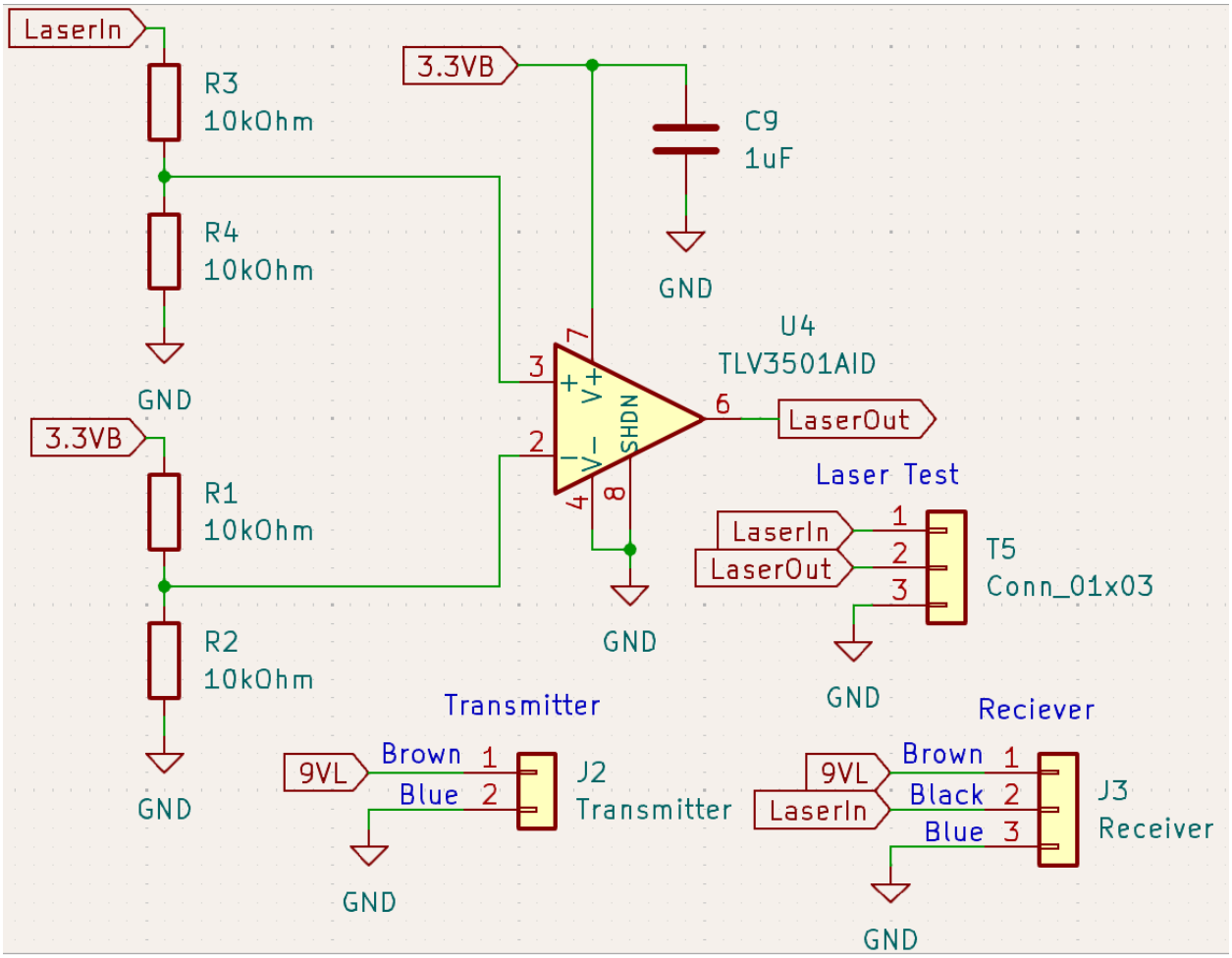


Fig. 17. Comparator Schematic

The comparator schematic is shown in Fig. 17 above. The receiver wire is the “LaserIn” label, which first feeds into a 1:1 ratio voltage divider that feeds into pin 3 of U4. This is due to the comparator chip (U4) only being able to take a max voltage of 8.5V. Through testing it was found the receiver laser was around 8.7V, which would exceed this chip’s maximum input rating. This input voltage was compared to 1.65V, the input on pin 2. This design will output the positive rail (3.3V) from pin 6 (“LaserOut”) when LaserIn is 8.7V and pin 3 is therefore 4.35V, as it is higher than the pin 2 voltage of 1.65V. When the laser is blocked, LaserIn will be 0.5V, making pin 3 equal to 0.25V, which is below the pin 2 voltage of 1.65V, meaning the comparator outputs ground (0V) from pin 6 (“LaserOut”). This was confirmed through testing and is shown in Fig. 42 and Fig. 43 in the test plan section.

Force Sensor Calibration

The FX29 Compact Compression Load Cells are the force sensors used in the Smart Sprinter, one embedded in each footpad of the track block. The manufacturer TE Connectivity offers analog and digital options. We decided to use the digital option which utilizes the I2C

protocol, a standard the STM32 Nucleo can interface with. TE Connectivity sells various load ranges, we went with the 100-pound-force (lbf) as this was the greatest load range that was readily available and not on backorder. The data sheet for the FX29 gives a linear transfer function to convert between the I2C readings and pounds-force [12]. Theoretically, the minimum I2C reading of 1,000 corresponds to 0 lbf and the max I2C reading of 15,000 translates to 100 lbf.

Fig. 18. Force Sensor Calibration Weights

For various reasons such as manufacturing tolerances and the way we embedded the force sensors into the foot pads, the manufacturer's transfer function supplied in the datasheet was consistently off. For example, when no force was applied, we often saw I2C readings in the 900s, below the theoretical minimum of 1000. Thus, we calculated our transfer function for each foot through experimentation. We used kettlebells and a dumbbell to collect I2C readings corresponding to 0, 10, 20, and 25 lbf, as seen in Fig. 18. We then used linear least-squares regression to find best-fit lines for the left and right feet. Fig. 19 shows the 4 data points for each foot, the least squares regression line (LSRL), as well as the manufacturer's supplied transfer function.

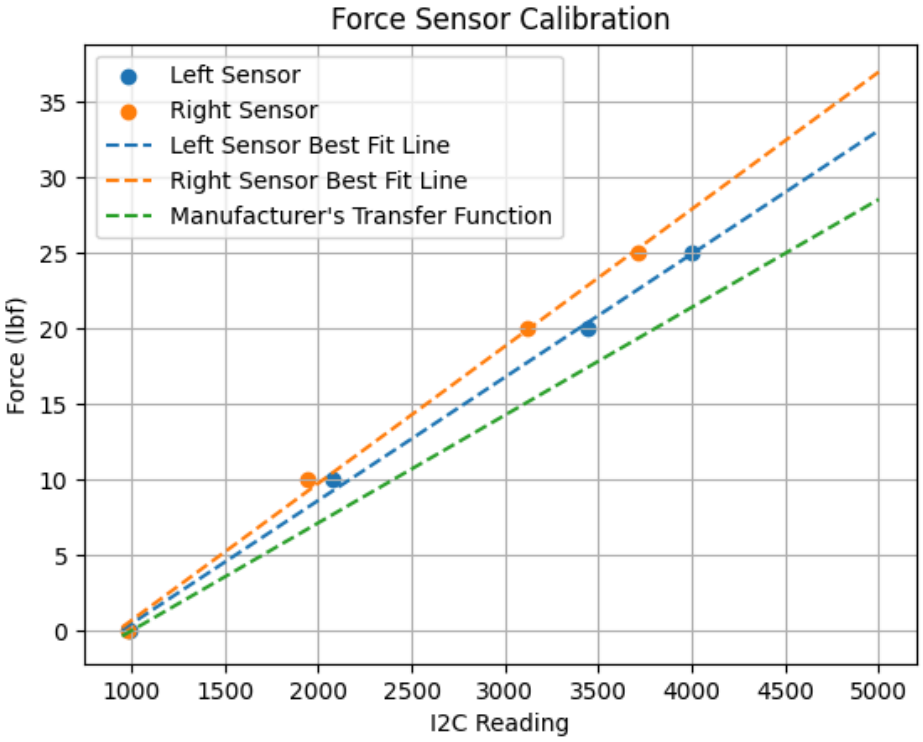


Fig. 19. Force sensor calibration and best-fit lines for each foot

We extrapolated the best-fit lines and used this as our transfer functions to convert the left and right I2C readings to force. This is performed in the Python program since the microcontroller just sends the raw I2C readings. In the final product, we converted from pounds-force to Newtons because metric units are used in track and field. Fig. 20 depicts a zoomed-out view of the extrapolation over the entire I2C reading range. It is justified to extrapolate linearly like this because the manufacturer's supplied transfer function is linear. Additionally, the coefficient of determination (R^2) values for left and right force sensors were both greater than 0.99. Thus more than 99% of the variation in the force value is predictable by the I2C reading.

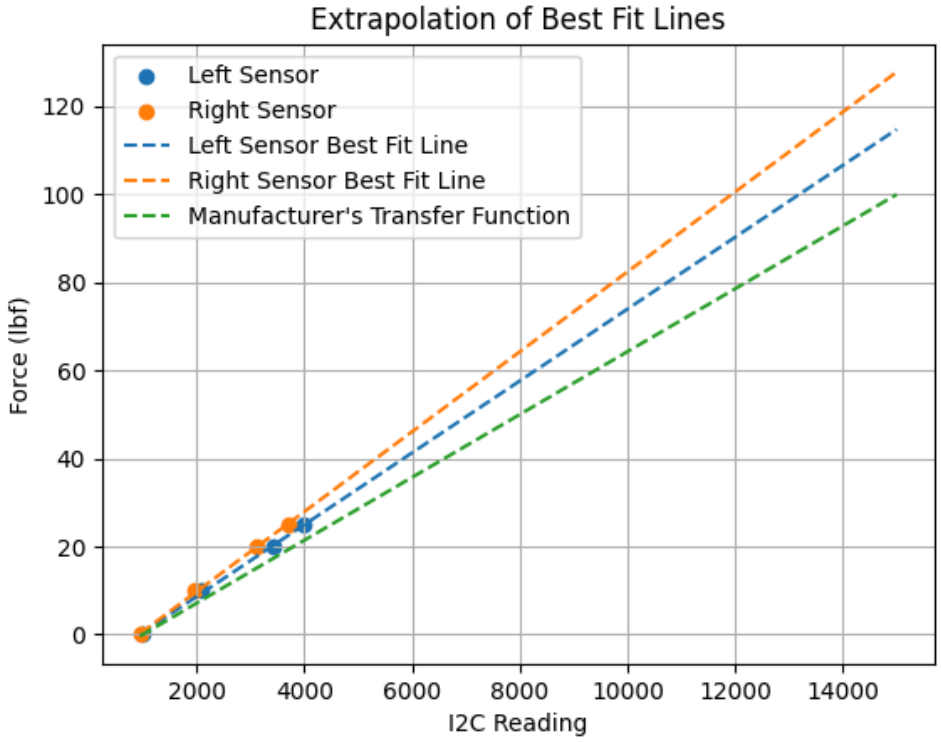


Fig. 20. Extrapolation of the force sensor best fit lines

STM32 Microcontroller

The STM32 microcontroller is the hub of this system. It performs data collection, system timing, lightweight data processing, and sends all the data to an external laptop equipped with a Python GUI for display. We drew on knowledge from ECE 3430: Intro to Embedded Computing Systems, where we learned how to program the STM32 for various applications, to design this subsystem. Power is delivered to the STM32 via the 3V3 input on pin 16 of CN7 and the ground on pin 20 of CN7 (Fig. 21). To run the STM32 in this power configuration, the ST-LINK shunt must be removed completely before powering on the board (Fig. 22).

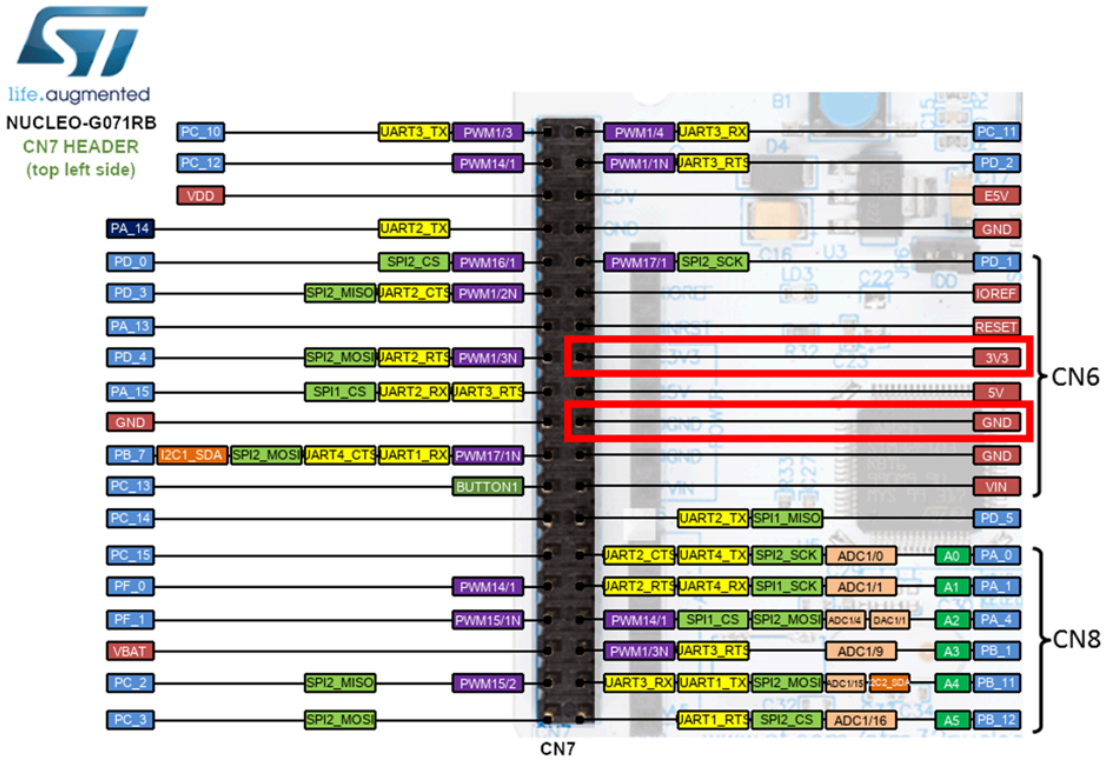


Fig. 21. STM32 Power Pins

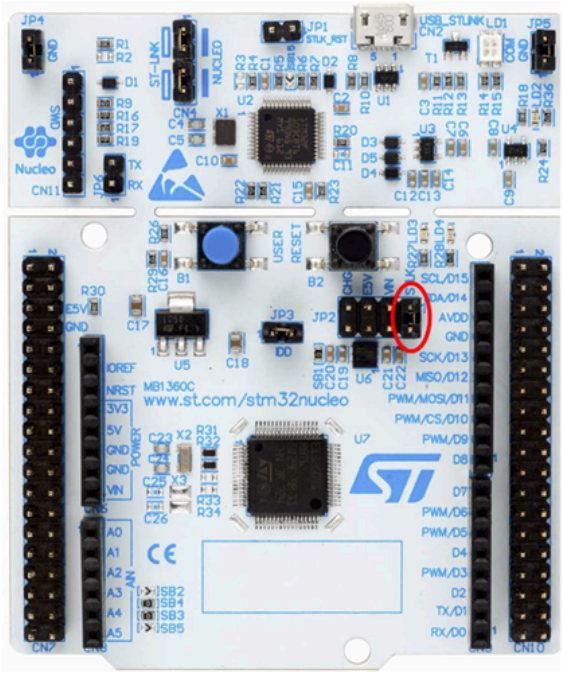


Fig. 22. ST-LINK Shunt on STM32 Top View

The STM32 is equipped with two I2C interfaces that were used for communication with the two FX29 force sensors [13]. The height sensing laser and start buzzer were attached to GPIO pins on the microcontroller. Two hardware interrupt timers were also configured. One acted as a system timeout to indicate that the STM32 should begin sending data to the laptop. The other was configured to delay the toggling of the start buzzer. Finally, the microcontroller was configured for normal UART communication, as well as UART interrupts. UART communications via micro-USB were necessary for interaction between the microcontroller and the Python GUI. These peripherals were configured to align with the design of the system sequence.

Prior to configuring and programming the microcontroller, we designed a system sequence that would model how our microcontroller responded to different events in the track start. This sequence begins with the microcontroller in an idle state where data is read from the force sensors every millisecond, but not written to memory. Once a start event is triggered, meaning the start button is pressed on the Python GUI, the microcontroller toggles the GPIO-attached buzzer, begins writing the force values it reads to memory, and the timeout hardware timer is started. From the start time until the timeout, the microcontroller waits to receive a signal from the height sensing laser. Once the timeout occurs, the microcontroller packages the force data and height data with a cyclic redundancy check (CRC) checksum. Then this data is sent in 4-byte chunks to the external device via UART on a micro-USB cable. At this point, the microcontroller re-enters the idle state, listening for either a start event signal or a request from the external device to resend the data.

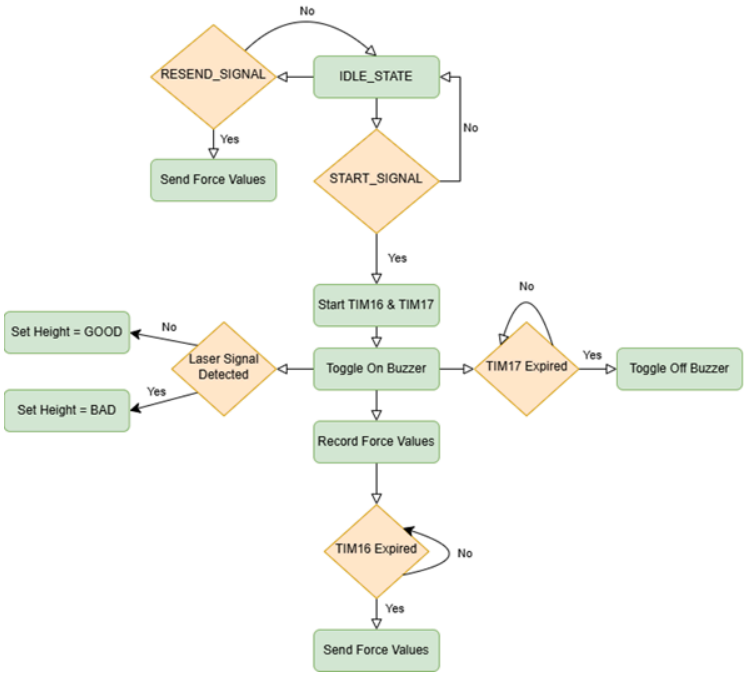


Fig. 23. Block Diagram of the STM32 System Sequence

In all, the STM32 records 6000 force readings during the first 3 seconds of the sprint. That is 3000 force readings per foot, and 1 reading per foot per millisecond. Each of the force readings are 2 Bytes in length, resulting in a total data size of 12kBytes. Given the large amount of data being transferred over UART, we felt it necessary to implement a system where the STM32 and external device coordinate to use a transport layer protocol to strengthen communication fault tolerance.

Communication Protocols (I2C & UART)

With the high-level overview explained, now we will detail the finer aspects of the STM32 subsystem. The microcontroller used two communication protocols to interact with different devices. The first protocol was the inter-integrated circuit protocol (I2C), which was used for communication from the FX29 force sensors and the STM32. I2C is a two-wire communication protocol that uses a serial data bus and serial clock bus to support communication between multiple target devices [13]. It is important to note that a pair of pull-up resistors (typically 10kOhms) must be placed on the positive supply and data and clock lines for successful communication to occur. In most implementations of I2C, multiple target devices are connected to the same data and clock buses and the controllers differentiate between the targets by using the I2C address assigned to the target hardware (Fig. 24).

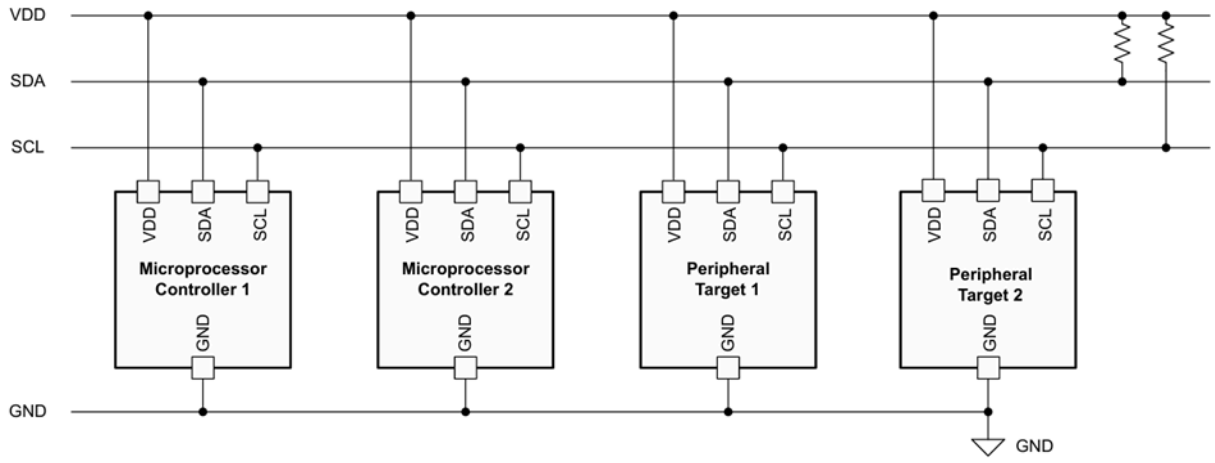


Fig. 24. Typical I2C Implementation [13]

However, in our implementation we decided to use two separate I2C buses because the I2C addresses on the FX29 force sensors were the same: 0x28 [2]. While there was a process available to change the I2C address on the force sensors, we did not think this was a wise use of our limited time given the difficulty of the task and the availability of two I2C interfaces. As a result, the FX29 force sensors were each connected to a dedicated I2C interface on the STM32. We used the STM32Cube IDE to configure pins PB8 and PB9 as I2C1_SCL and I2C1_SDA

respectively, and pins PA11 and PA12 as I2C2_SCL and I2C2_SDA respectively (Fig. 25). None of the default settings were altered after configuring the pins for I2C communication.

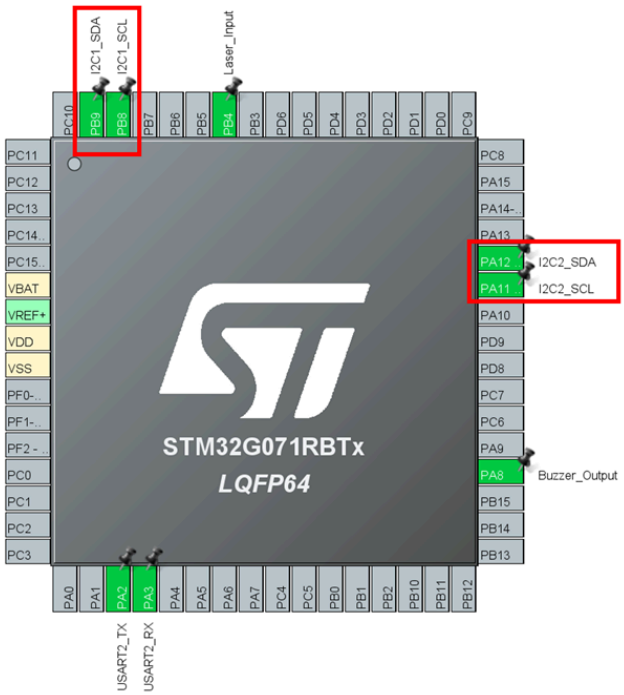


Fig. 25. Configuration of I2C Interfaces in STM32 .ioc File

In programming the microcontroller, only two lines of code made use of the I2C interface. These were two lines calling the function HAL_I2C_Master_Receive. This function "receives in master mode an amount of data in blocking mode," and takes the I2C_HandleTypeDef pointer, target address, output buffer, amount of data to be received, and the timeout length as arguments [14]. Since we used two I2C interfaces, this function was called twice, but the I2C_HandleTypeDef pointer was altered to change the interface.

The second communication protocol we used was Universal Asynchronous Receiver/Transmitter (UART). Like I2C, UART is another two-wire communication protocol. However, unlike I2C, UART allows for communication to and from both connected devices through the use of transmit (Tx) and receive (Rx) lines (Fig. 26).

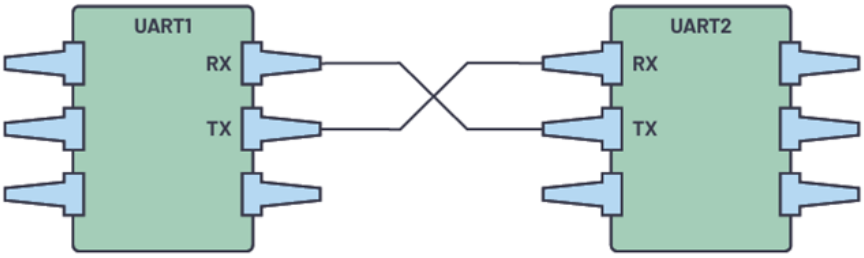


Fig. 26. UART Communication Devices [15]

This aspect of UART communication was important to us because we required a communication protocol that allowed us to send data from the microcontroller to an external laptop as well as have the microcontroller receive data from that same external laptop. Additionally, we used a micro-USB cable to connect the two devices. We configured the microcontroller to use pins PA2 and PA3 as the UART_Tx and UART_Rx lines respectively Fig. 27. We kept all UART default settings except enabling the UART interrupt in the NVIC settings tab.

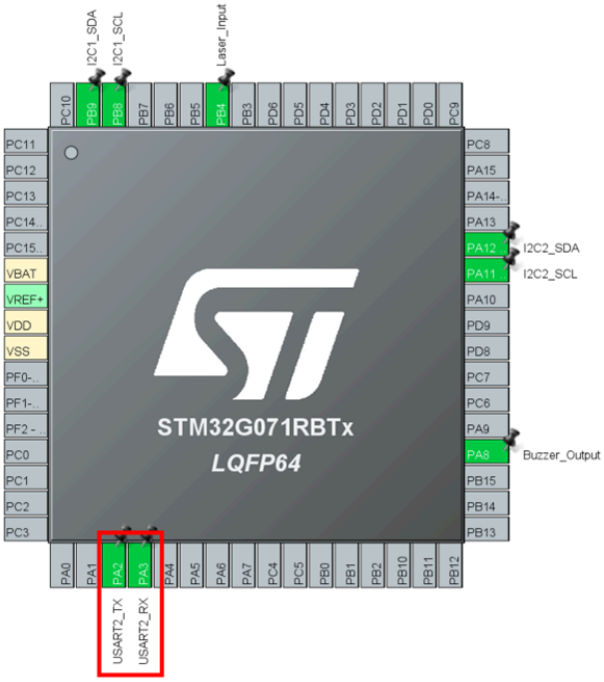


Fig. 27. Configuration of UART Interface in STM32 .ioc File

In the STM32 program, there were two different cases where UART communication occurred. The first case was when the microcontroller needed to send the force data to the laptop. In this case, the function send_force_values() (Fig. 67 in Appendix) is called with the force values array and length of this array passed as arguments. Within this function, a CRC checksum

is calculated for the array of force values using the function HAL_CRC_Calculate [14]. This CRC checksum is 4 bytes long and is sent by UART to the laptop using the HAL_UART_Transmit function [14]. Next, the forces array is sent to the laptop using the same HAL function. This array is 12kB in size, but the array is sent in 4 byte chunks to limit any mishaps in transmission of such a large amount of data. A more detailed explanation of why the CRC checksum was calculated is available in the "Transport Layer" section of this report.

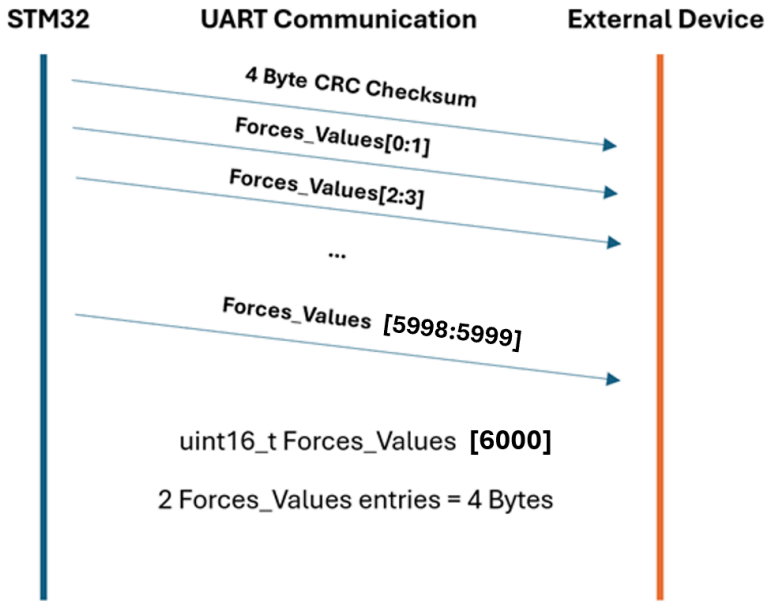


Fig. 28. Example of UART Communication Between STM32 and External Device

The second case where we needed UART communication was when the laptop sent either a start signal or resend signal to the microcontroller. This case is why we enabled the UART interrupt. The interrupt functions such that whenever the microcontroller receives UART communication from the laptop, it enters the UART interrupt callback function (Fig. 70 in Appendix). It is important to note that two calls of the function HAL_UART_Receive_IT are necessary for this interrupt to work correctly. This function tells the microcontroller to listen for UART communication, trigger the interrupt when it occurs, and write the received data to a specified buffer [14]. One call must exist within the main function while the other exists at the bottom of the callback function. Within this callback function, the received data buffer is examined. If the buffer contains the ASCII character 'S', this denotes a start signal and the start sequence is initiated. The start sequence involves toggling the buzzer, starting the buzzer delay timer (TIM17), and incrementing the button_presses variable to prevent an accidental "double start." Additionally, the timeout timer (TIM16) is started and the forces_index variable is reset to zero so that new force measurements are written to memory. The other possible signal, the

resend signal, is denoted by the ASCII character ‘R.’ When the microcontroller receives this signal from the laptop, it triggers a new call of the send_force_values function (Fig. 67 to resend the force data to the laptop. This only occurs when some of the data initially sent to the laptop is either lost or corrupted (more details in the “Transport Layer” section).

Force Sensor Data

After the FX29 force sensor data is read from the I2C interfaces, the values must be converted from binary into a decimal reading. The FX29 force sensor sends two big-endian bytes of data to the STM32, but only bits 0 to 13 are the important bridge data bits Fig. 29. [12].

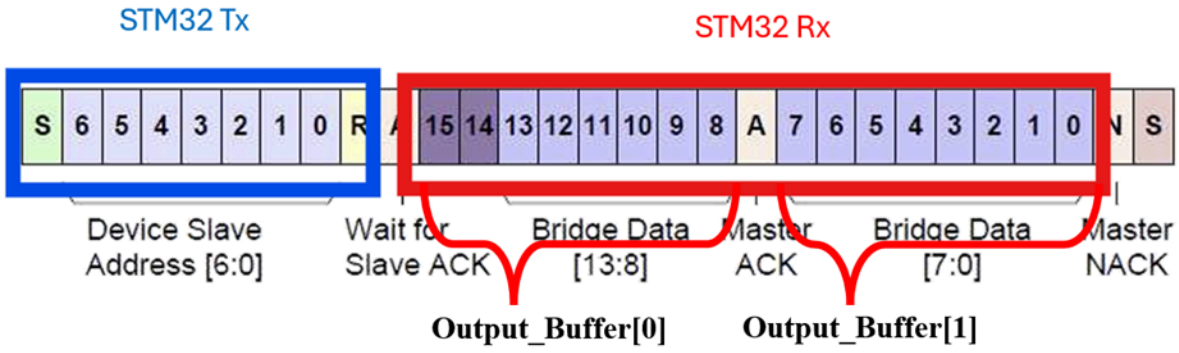


Fig. 29. Diagram of I2C Communication Between FX29 and STM32

These two bytes are written to a uint8_t output buffer and then rearranged by shifting the first buffer entry to the left 8 bits and performing a bitwise “or” on this shifted value and the second buffer entry. This value is then masked with the hex value 0x3FFF to access only bits 0-13. Now we have the decimal value of the force sensor reading. This process must be performed separately for each sensor, but the values are written to memory in the same forces array. This was done to simplify the checksum calculations and send_force_values() function (Fig. 67). If a force value comes from the left foot sensor, it will be written to the first 3000 entries of a uint16_t array, while force values from the right foot will be written to the last 3000 entries. This results in a total force array of 6000 16-bit entries, equivalent to 12,000 Bytes. The specific STM32 model we use, the Nucleo-G071RB, has 36kBytes of SRAM available, so we had no concerns of the force values array exceeding the available memory [16].

The first iteration of the project did not send this large force value array, but instead only sent the maximum force value read from the sensors. We had concerns about the STM32’s ability to store and transmit a large amount of data, but we decided to test it anyway. We discovered that the STM32 outperformed our expectations and transmitted the data effectively. As a result, we moved on from the maximum force idea and moved towards the idea of sending the force values

for the first 3 seconds of the race. This also allowed us to display a graph of the sprinter's force over time during the start in the Python GUI.

Timing Systems

Hardware timers perform the task of counting to a certain value and rolling over once they reach their maximum value. However, it is possible to configure a timer such that an interrupt is triggered when the timer hits its maximum value. In our design we needed two hardware timers: one to act as a full system timeout and another to handle the toggling of the start buzzer. The STM32 is equipped with two general purpose, 16-bit, 16 MHz timers in TIM16 and TIM17 [17]. These timers can only count up to a value of 65536 before rolling over, but they provide enough functionality for our purposes.

For the timer handling the system timeout, we needed to choose a timeout duration that allowed enough time for the sprinter to hear the buzzer, start the race, and clear the height sensor before timing out. To be extremely conservative in our estimate, we selected a timeout of 6 seconds. Once we had our timeout duration, we had to perform timer math to set the prescaler and counter period to count the desired time. TIM16 was used for this timer. Since TIM16 is a 16MHz timer, a prescaler of 1600 – 1 was chosen. The notation 1600 – 1 is used because in the STM32Cube IDE the prescaler variable is zero-based. The counter period was not changed from its default value of 65536 – 1.

$$\frac{16M\ ticks}{1\ s} * \frac{1}{1600\ prescaler} = \frac{1\ tick}{100\ \mu s}$$

$$\frac{100\ \mu s}{1\ tick} * \frac{65536\ ticks}{1\ cycle} = 6.554\ s/cycle$$

This timer starts when the microcontroller UART interrupt detects an 'S' has been sent, indicating the start of the race. Once the timer expires, the microcontroller enters the HAL_TIM_PeriodElapsedCallback function (Fig. 68 in Appendix), where it stops the timer and sends the force values to the external device.

For the buzzer toggle timer, we needed a short duration timer to turn off the buzzer after the start of the race. We settled on a value of half a second for simplicity. TIM17 was configured with a prescaler of 256 – 1 and a counter period of 31250 – 1.

$$\frac{16M\ ticks}{1\ s} * \frac{1}{256\ prescaler} = \frac{62500\ ticks}{1\ s}$$

$$\frac{1\ s}{62500\ ticks} * \frac{31250\ ticks}{1\ cycle} = 0.5\ s/cycle$$

This timer starts at the same time as TIM16, but when it expires it simply stops the timer and toggles the buzzer GPIO pin.

When configuring these timers, it is important to enable both timer interrupts in the NVIC settings. Another quirk about these hardware timers is that an update event will trigger the callback function when the timer is first started so a blocking if-statement must be used within the callback function to prevent unintentional behavior. One final note; we initially designed the program to record the TIM16 time when a certain force threshold was exceeded in the force value readings to indicate the sprinter's reaction time. However, we discovered that we could accomplish this on the Python side of the software much easier, so we scrapped that piece of code.

GPIO System

General purpose input/output (GPIO) pins are customizable, and their functionality can be controlled entirely by software [18]. We used two GPIO pins to control both input and output events in the system. We configured pin PB4 as the height sensing laser's input to the STM32, and pin PA8 as the STM32's output pin to the buzzer.

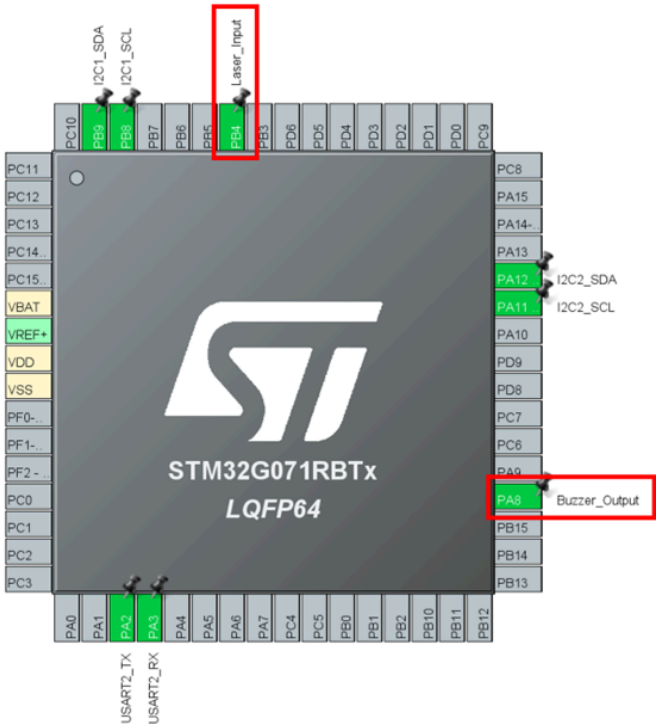


Fig. 30. Configuration of GPIO Interfaces in STM32 .ioc File

The height-sensing laser pin was configured as an interrupt-based GPIO pin because we needed an interrupt to trigger when a signal was detected on this pin. The laser is active high, so we configured the interrupt to trigger only on a falling edge. When the laser signal goes low, the interrupt triggers the HAL_GPIO_EXTI_FallingCallback function (Fig. 69 in Appendix), and the

height Boolean value is recorded. We decided to attach the height Boolean value to the most-significant-bit (MSB) of the first value in the forces array to save transmission space. Since the 14^{th} and 15^{th} bits of the force values are not used, this will not alter the force value data at all, since we extract that bit on the Python companion app side. When the interrupt is not triggered (good height), a 0 is in the MSB of the first force value array entry. When the interrupt is triggered (bad height) a 1 is written to the MSB.

The other GPIO pin functions as the output signal from the STM32 to the buzzer. This pin is toggled high when the STM32 receives the 'S' character from UART and toggled low when TIM17 expires. Additionally, we noticed a slight voltage leak on the buzzer causing it to emit a low hum if the GPIO pin is not intentionally pulled low at the start of the program. We remedied this issue by writing a 0 to the pin early in the main function of the program using HAL_GPIO_WritePin.

Initially, we had plans to use a third GPIO pin for the microphone. While the STM32 was configured correctly and the code was written correctly, we could not get the hardware to function as expected so we had to move on from this idea.

Transport Layer

Shown below is the packet format used for communication from the microcontroller to the laptop. This is relevant for understanding the transport layer and where the expected number of bytes comes from.

Packet Format:
[CRC-32 (4 bytes)][3000 left force samples (6000 bytes)][3000 right force samples (6000 bytes)]
Notes:

- CRC is an unsigned 32-bit value sent as 4 little-endian bytes.
- Each force value is an unsigned 16-bit value sent as 2 little-endian bytes. Two force values are sent at a time so 4 bytes total per chunk.
- Each chunk is 4 bytes, so 3,001 chunks are sent per packet.

Packet Length:

- Checksum: 4 bytes
- Data: 12,000 bytes
- Total: 12,004 bytes.

Owen and Garrett developed a simple transport layer on top of UART to ensure reliable communication between the microcontroller and the laptop. We drew on our knowledge from ECE 4457: Computer Networks where we learned in depth about all the layers in the OSI Model as well as the TCP/IP protocol stack. Referring to the OSI Model, in our system the physical layer (layer 1) is the Recommended Standard 232 (RS-232) [19]. The data link layer (layer 2) is

a universal asynchronous receiver/transmitter (UART). There is no network layer (layer 3) since the Smart Sprinter project is not connected to the Internet. The transport layer we developed will correct for data loss and corruption. Fig. 31 illustrates how the system works when the data is sent perfectly.

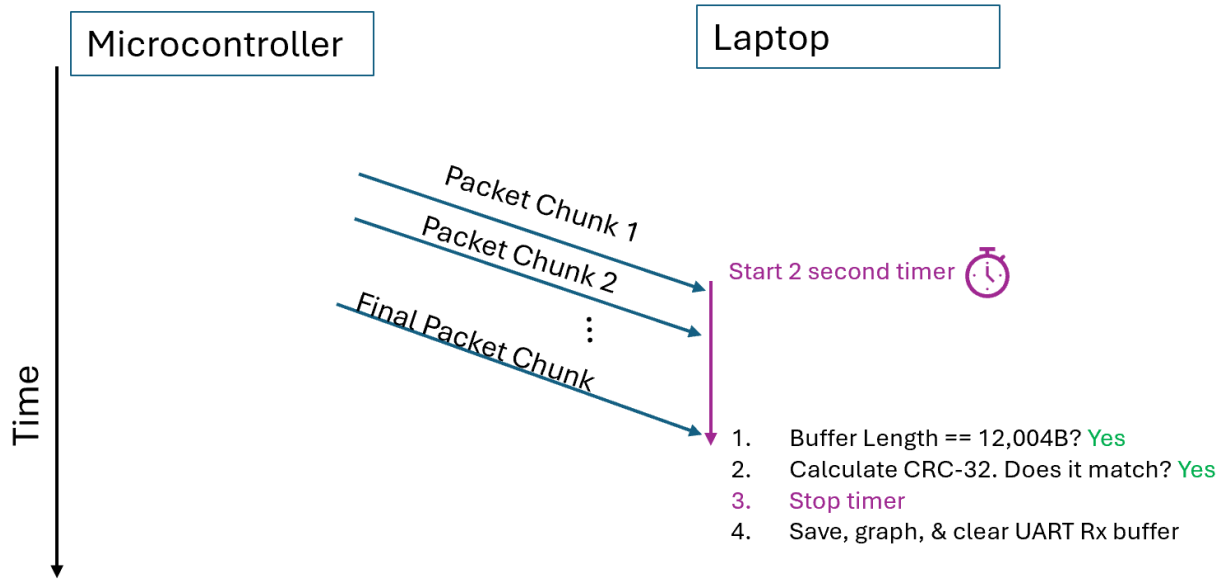


Fig. 31. Transport layer diagram with no data loss or corruption

Possible data corruption such as a random bit flip is detected with a cyclic redundancy check (CRC). This is a type of hash function applied to the data and sent at the beginning of our data packet. On the STM32 Nucleo microcontroller, the 32-bit CRC is calculated at the hardware level using the CRC calculation unit [19] allowing for fast computation. Fig. 32 shows how the transport layer recovers from data corruption. When the Python companion program on the laptop receives the packet data in 4B chunks. Upon reception of a new chunk, it appends the chunk to a UART Rx buffer, and then checks if the total length of the buffer is the 12,004 bytes expected. If so, it then computes the CRC-32 from the 12,000 bytes of data using the Python binascii module in the standard library. The Python program then checks if the CRC-32 matches the 32-bit or 4-Byte checksum at the beginning of the packet. If there is a difference, then the Python program clears the UART Rx buffer and sends a resend signal to the microcontroller. The whole process then repeats until a packet with no corruption or data loss comes in, which is assumed to be the next try in Fig. 32.

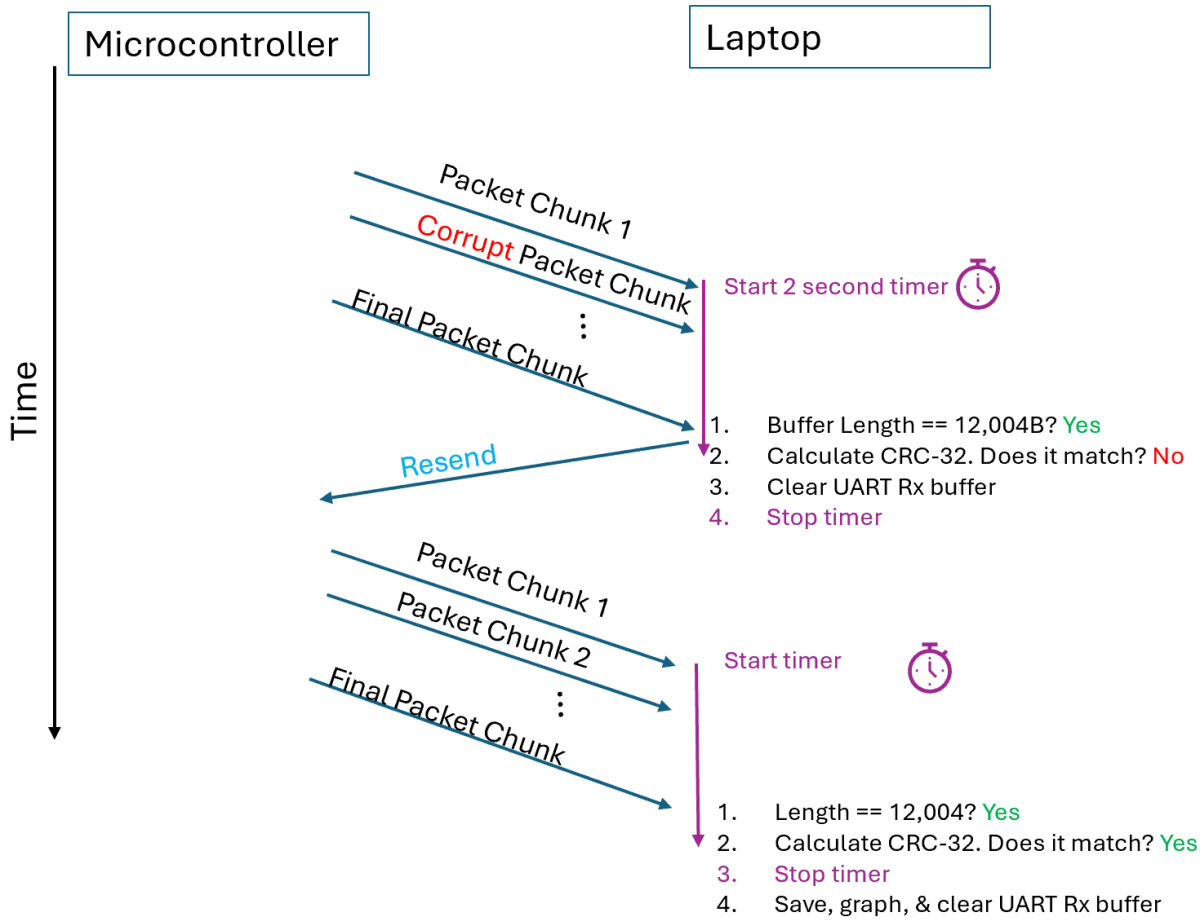


Fig. 32. Transport layer diagram showing recovery from corruption

We utilized a 2-second timer to detect possible data loss such as if one of the packet chunks got dropped during transmission. The agreed upon UART baud rate between the laptop and the microcontroller is set to 115,200 bits per second, the fastest option the STM32 offers. The calculation below shows that we can anticipate all the bytes to be transferred in about 0.83 seconds, assuming perfect conditions.

$$12,004\ Bytes\ *\ \frac{8\ bits}{Byte} * \frac{1s}{115{,}200\ bits} \approx 0.83\ s$$

Thus, we set the timeout to 2 seconds to be conservative and because we are using a 3-meter-long USB cable which might add latency. We utilized multithreading to implement the timer on the Python program, drawing on knowledge from CS 3130: Computer Systems and Organization 2 (CSO2). After the first chunk of data is received, the Python program spawns a new thread with a 2-second timer. After the 2-second timer expires, if the expected 12,004 bytes have not been received, the UART buffer is cleared and a resend code is sent to the microcontroller. Fig. 33 shows this process in action. When a correct packet is received, the timer is stopped.

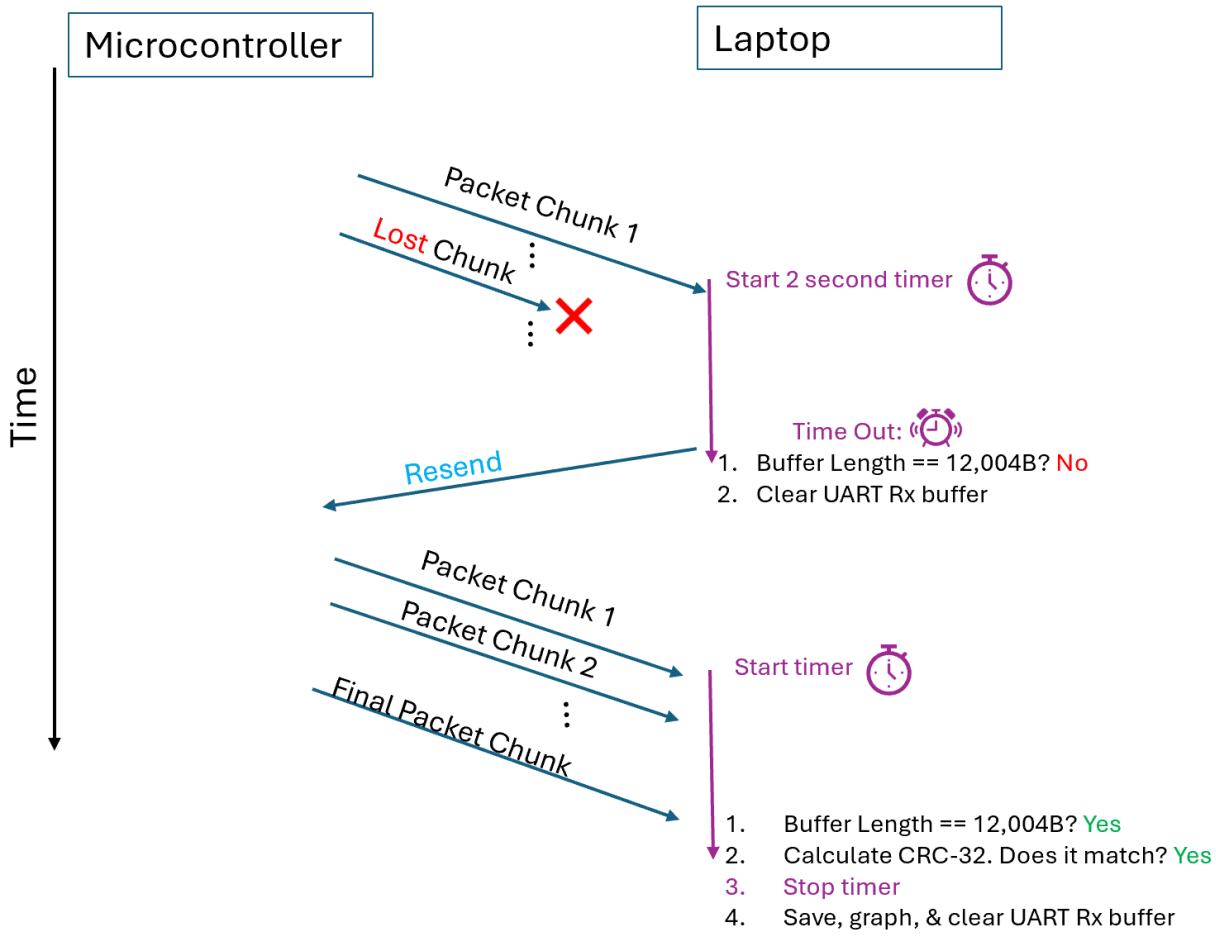


Fig. 33. Transport layer diagram showing recovery from data loss

GUI Companion Application

The graphical user interface (GUI) companion application was created in Python. The wxPython library [20] was used for the GUI framework and the pySerial library [21] is utilized for communication with the STM32 via UART.

A GUI runs in a continuous event loop, it is like an infinite while loop. When the user clicks on a mouse button or presses a key, the hardware sends an interrupt to the kernel of the operating system. If the SmartSprinter GUI is in focus on the user's laptop, then it receives a signal from the kernel. These are the events for which the GUI continuously checks. Upon the reception of an event, the GUI app handles these signals with various callback functions, i.e. what to do when a certain button is clicked or key pressed. A pySerial receiver works in the same way, except checking for UART transmission via a computer's COM port. To do both of these things simultaneously, multithreading must be applied.

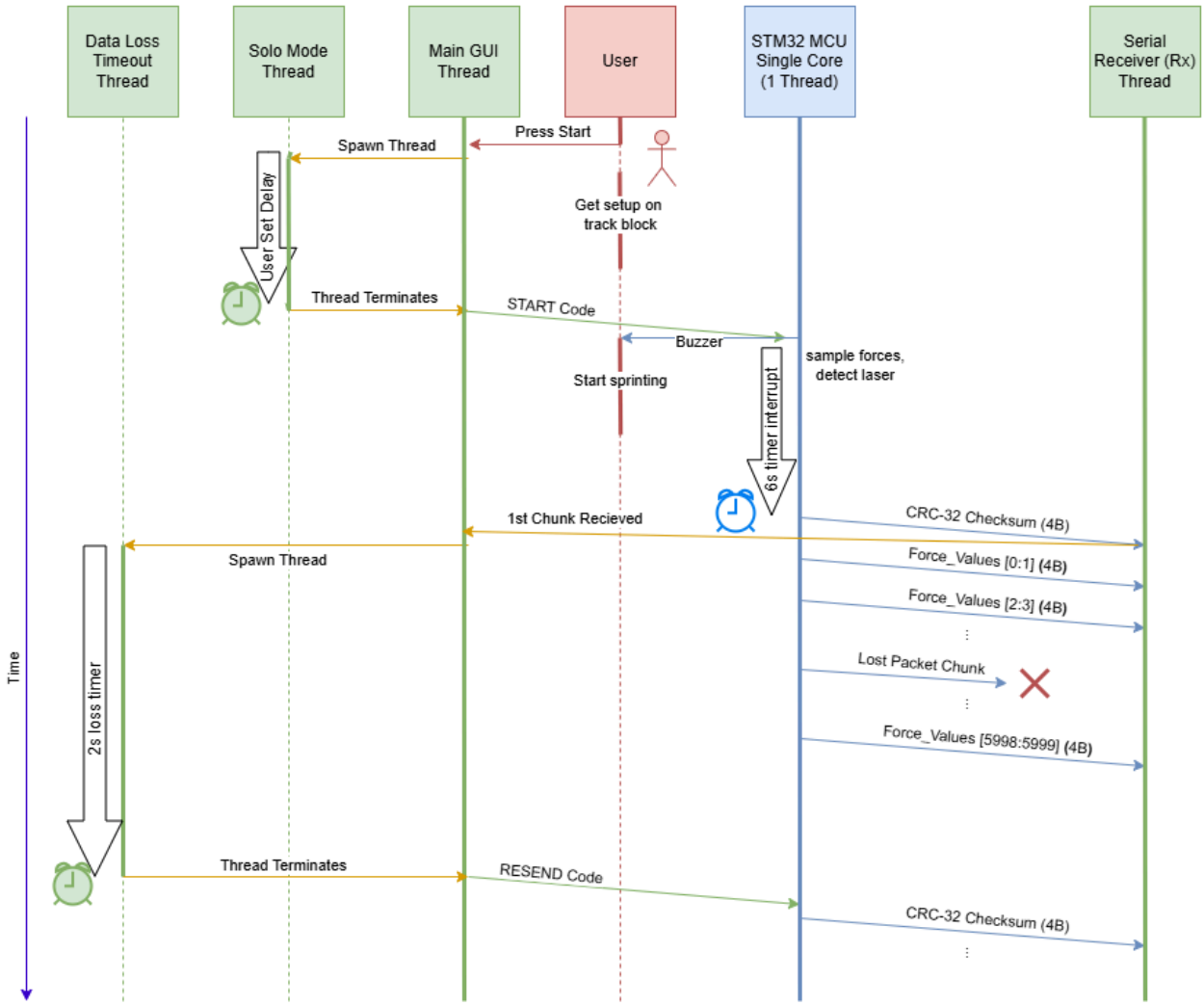


Fig. 34. GUI Thread Diagram showing solo mode and data loss

Fig. 34 shows all four possible threads of the Smart Sprinter application in action. Shown in green are threads of the Python companion application, blue is the microcontroller and red is the end user. This diagram represents how the four threads of the companion application interact, in the most complex scenario: solo mode enabled and data loss. For further explanations of the Transport Layer see that section above.

The main GUI thread spawns the serial thread on power up (not shown in Fig. 34) Both those threads remain alive until the program is closed. Solo mode is a feature we added to allow the end user to be able to use the product without needing a friend or coach to press start for them. If solo mode is disabled, then the start buzzer and force sampling occur immediately once the start button is pressed on the GUI. With solo mode enabled, the main GUI thread will spawn a solo mode thread timer based on a user set delay, which defaults to 10 seconds. This gives the user enough time to press start on the GUI and get set up on the track block and wait for the

buzzer to go off. After the solo mode thread terminates, the main GUI thread sends a start code (ASCII ‘S’) to the STM32 via a transmission function in the pySerial library.

Upon reception of the start code, the STM32 then emits the buzzer beep and samples force values for the first 3 seconds after the start buzzer. The STM32 waits for 6 seconds (an additional 3 seconds after force sampling) just in case the user is slow, and it takes a while for them to cross the height module. All data is stored in memory and collected before then being transmitted in 4 byte chunks. There is also a checksum as described in the Transport Layer section above. Upon reception of the first chunk, the serial receiver thread notifies the main GUI thread that transmission has begun. Then the main thread spawns a data loss timeout thread. After the data loss timeout thread terminates, if the correct number of bytes is not received a resend code is sent from the main GUI thread via pySerial transmission.

Python has a global interpreter lock (GIL), which is a mutex lock allowing only one thread to control the Python interpreter at any time [22]. In other words, all multithreaded python programs operate on a single core and achieve concurrency through context switches in the operating system’s thread scheduler. This is in contrast to true parallelism that can be achieved on a multicore processor in a language like C or C++. However, this bottleneck was not an issue due to the relative simplicity of our design. Additionally, Python’s Numpy library was used for all the number crunching like conversion from I2C values to Newtons, calculating the max force for each foot, and the block exit time. Numpy is mostly written in C so it can release the global interpreter lock allowing for faster number crunching [23].

For the final look of the entire GUI, see the Final Results section of the report. Shown in Fig. 35 is an example plot that can be generated through the File Export option in the GUI. Originally, I was planning on using Matplotlib for the plotting integration as stated in the Project Proposal and Midterm Design Review. However, the wxPython GUI framework already has a plotting library [24], so I used this as it was much easier to integrate.

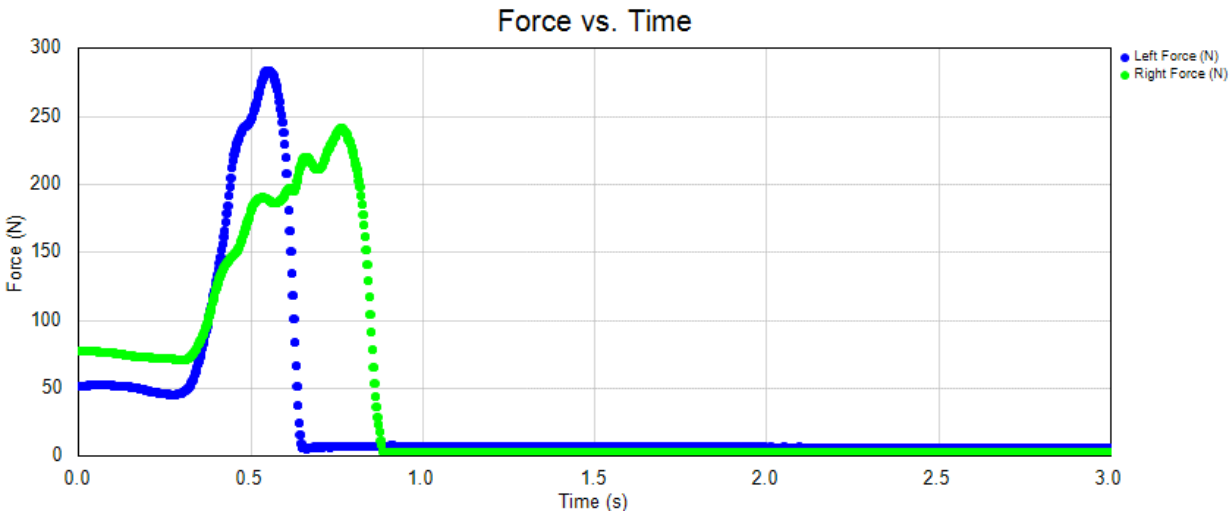


Fig. 35. Force Plot Diagram from the GUI

General Assembly

Fig. 36. Laser Stand with Magnets Attached

The Smart Sprinter product had three key miscellaneous tasks that needed to be built to have a fully functional working project. The first important one was the laser height stands. This was built through 2x4 wood beams and screws, standing 7ft tall and having four legs to stand on their own. In addition to the base structure, the laser height stands have approximately 2-3ft long metal tape that comes down from the top. The lasers are held by metal brackets and circular magnets. This allows each laser to attach easily to the wooden stands and slide up and down to easily adjust height and rotate to align the lasers. This is to account for different runner's height and form, and what height would be ideal to run below based on their practice. The magnet design is different from the original design of drilling holes, as it would've been too hard to align the lasers and measure where the holes would go. The magnet design allowed more flexibility with where you can place the stands and the height. When the height is found, the wires on the

back of the lasers need to be taped to the wooden stands as the weight of the wires causes the laser to rotate upwards. This can be seen in Fig. 36 above.

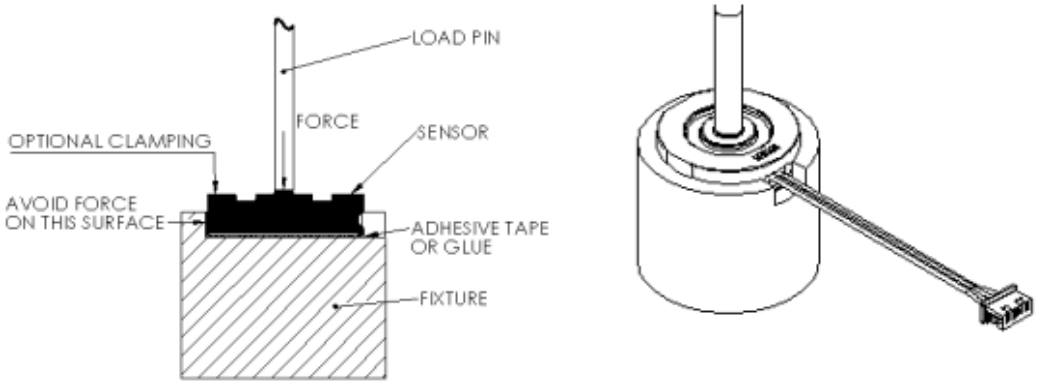


Fig. 37. Load pin of the force sensor [12]

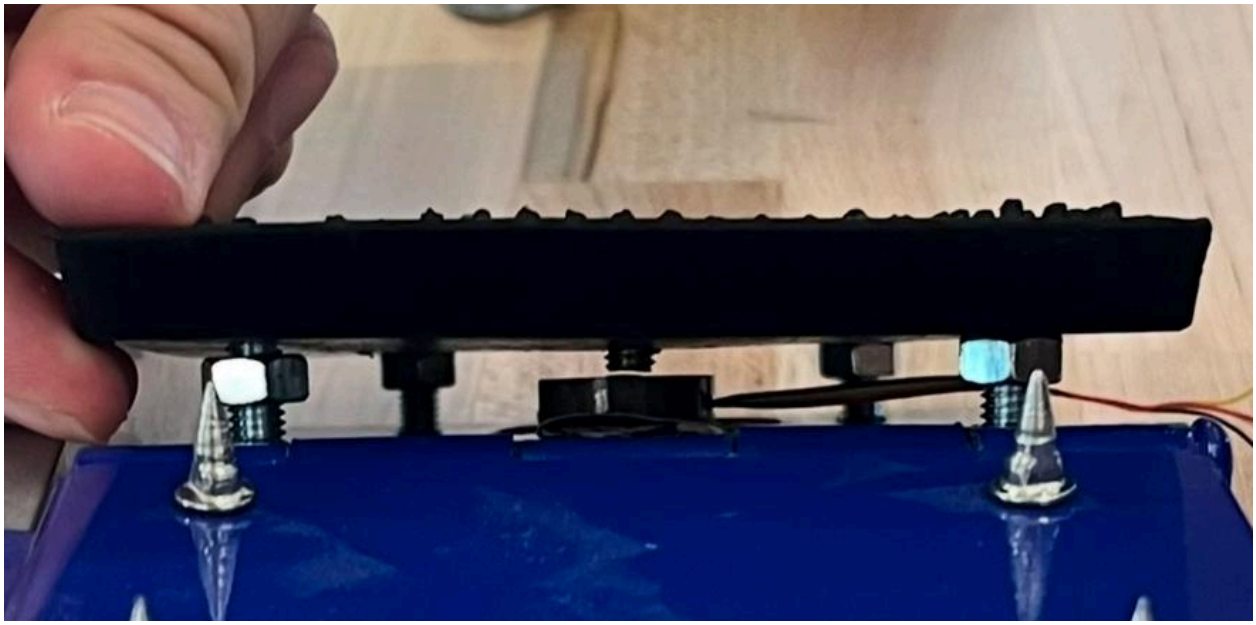


Fig. 38. Front view of bolt attachments, and pinpoint screw on the force sensor

Fig. 39. Side view of final force sensor assembly

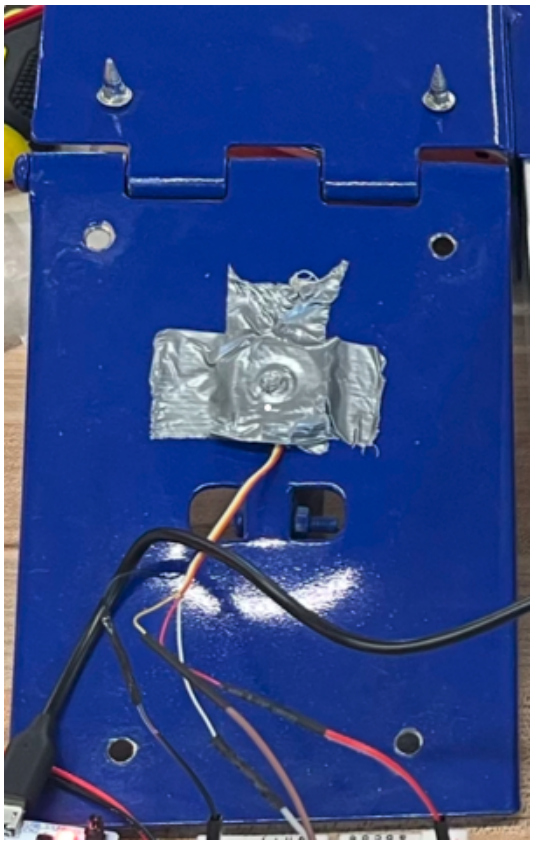


Fig. 40. Inside view of force sensor attachment, without cardboard

The second key assembly feature was to embed the force sensors. This task was completed through taking the rubber pads off the track block, and replacing the 4 corner bolts with longer bolts to create a space between the rubber pad and the metal stand, leaving room for the force sensor. The bolts also had nuts that tightened to the rubber pads that increased stability and rigidness of the rubber pads giving the runner a comfortable surface from which to launch. Finally, the group used the middle shorter bolt as a load pin for the force sensor. This is due to the data sheet saying the force sensor obtains more accurate readings when the force is concentrated on the center of the force sensor, as seen in Fig. 37. The middle bolt was held down with epoxy to make it move in unison with the rubber pad when the runner takes off (Fig. 38). The force sensors were also surrounded by cardboard to fill the space between the rubber pads and metal stands while also making it uniformly compressible and further increasing stability (Fig. 39). The force sensors were held down by duct tape and the cardboard, ensuring removability in case they need to be replaced (Fig. 40).

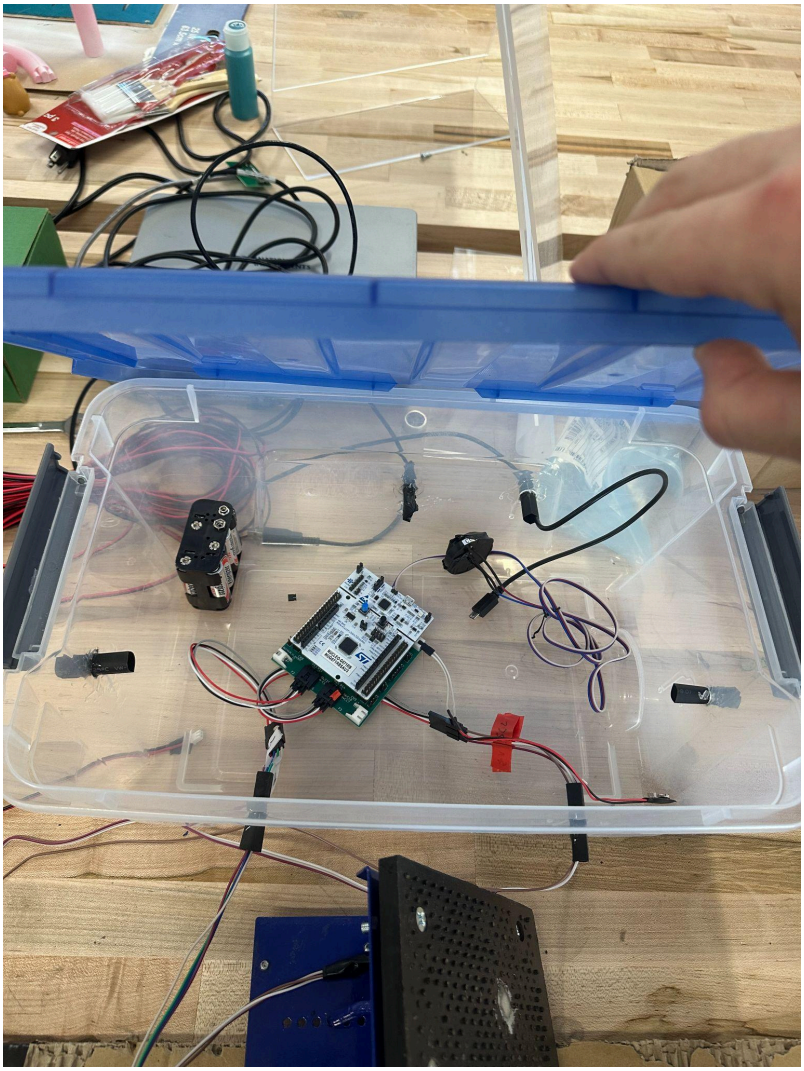


Fig. 41. Picture of overall encasing

Finally, the group needed an encasing for the PCB, microcontroller, battery pack, and starting buzzer. Ideally, the group needed this encasing to be lightweight, waterproof, and have a removable lid. The encasing needed to be lightweight to be easily transportable to the track, or other outdoor locations. It needed to be waterproof in case of rain when testing. It needed a removable lid to plug in the battery pack, plug in the proper wires into the PCB/STM32, and for any other debugging purposes. This led to the design shown in Fig. 41, which accomplished all three of these key requirements and prioritized functionality over aesthetics.

Test Plan

As soon as the board was fully soldered together, the first thing done was to check for cold solder joints. This was achieved by taking a multimeter connectivity test and making sure component's pins were connected to expected nodes. Specifically, we made sure that the output I2C lines from the pressure sensor were properly connected to the correct pins on the STM32. We also tested the speaker with connectivity checks through the multimeter. Immediately after checking for connectivity, we looked at the power rails. We made sure all voltages on the rails were within tolerance and reaching the correct nodes. Next, we slowly added each subsystem individually and made sure all their pins had expected values. For example, we used an oscilloscope on the NI VirtualBench to make sure that the output from the laser receiver had a proper falling edge and correct voltage height. Finally, we used test pin 5 to verify the signal integrity of the laser receiver.

To ensure proper functioning of the STM32 software, we needed to perform a series of tests on different sections of the program. First, we tested the power pins to ensure the board was being powered properly at 3.3V. We used a multimeter connected to the 3V3 pin to test this and determined that the board was sufficiently powered. Next, we needed to test that the board successfully communicated with the laptop using UART. We tested communication to and from the laptop using both "dummy data values" and real data values gathered from the force sensors. After confirming UART communication, we confirmed GPIO signal integrity by testing the GPIO output pins and measuring the voltage across them during a trigger event with a multimeter. Finally, we used a stopwatch to test and verify that the timers lasted for the expected period. This concluded the subsystem testing of the STM32. Since the STM32 relied on much of the other subsystems to function correctly, most of our time was spent debugging the program rather than testing the board hardware. To find these bugs we ran full system tests until a bug was discovered. At this point, we formulated a solution and implemented a solution to the bug. Most of our bugs in the STM were simple pointer errors, array length mismatches, or misconfigurations in the .ioc file.

Fig. 42. Unblocked laser and LaserOut voltage

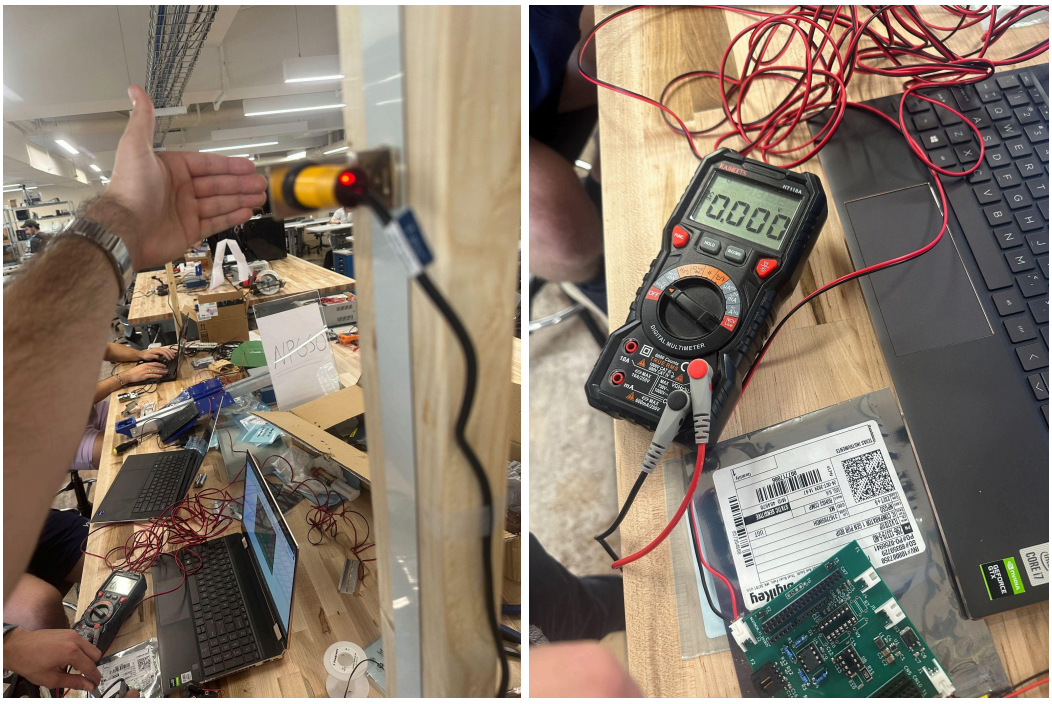


Fig. 43. Blocked laser and LaserOut voltage

Fig. 42 and Fig. 43 above show the test results for the laser system. The group used a multimeter and placed the probes on test connector T5 (bottom and middle pins as shown in Fig. 17 & Fig. 60 in Appendix). The group found that when the lasers are aligned and not blocked, the output that goes into the STM32 was around 3.3V, as seen in Fig. 42. When

blocked, such as when the runner is too high, or in this case by a group member's hand, the output that goes into the STM32 drops to 0V, as seen in Fig. 43. This falling edge is the trigger to signify that the height of the runner was bad, which was also tested through the software and shown in the GUI. This test confirmed that the hardware worked as intended.

Physical Constraints

We used the same microcontroller from ECE 3430: Intro to Embedded Computer Systems, the STM32 Nucleo-G071RB. We decided to use this microcontroller for three reasons. First, we already knew how to use it, and we did not need to spend any time learning a new tool. Second, it is relatively cheap at about $11, which helped with our constrained budget of $500 total. Third, it had all the necessary technical specifications we needed for the Smart Sprinter. The microcontroller has 128 kB of nonvolatile flash for instructions and 36 kB of random access memory (RAM) for data [25]. When recording the start of the race, we sampled the two force sensors embedded into left and right footpads at the track block at 1kHz for 3 seconds. Each sensor provides a 16-bit or 2-Byte value for the I2C force reading [12].

$$2\,sensors\ *\ 2\,Bytes/sensor\ *\ 1kHz\ *\ 3s\ =\ 12000Bytes$$

$$12000Bytes\ *\frac{1kB}{1024Bytes}\ \approx\ 11.72kB$$

All the data is collected and stored in an array in RAM on the microcontroller before being transmitted to the laptop via UART. Thus with 11.72 kB of force data, 36 kB of RAM on the microcontroller fit our needs just fine. On the next trial of the smart sensor, the force data is overwritten, since the previous data has been transmitted to the laptop where long-term storage occurs. The microcontroller supports Inter-Integrated Circuit (I2C) synchronous serial communication which is the protocol used to talk to the digital force sensors [26]. The STM32 also has external interrupts which we use to detect height and hardware timers for force sampling and emitting the buzzer sound.

Various software tools were utilized in the project on both the hardware and software side of the project. The STM32CubeIDE [27] was used for embedded software development in C. All group members knew how to use the integrated development environment (IDE) from ECE 3430 for compilation, debugging, and deployment onto the microcontroller. Pycharm IDE was used for the laptop companion application's development in Python, which we knew from CS: 1110 Intro to Programming. LTspice [28] was used for circuit design and simulation as well as KiCad [29] for printed circuit board (PCB) layout. These two hardware tools were new to the group, and PCB designer Shah Zaib had to learn them to complete his portion. However, since we all were familiar with NI Multisim and Ultiboard from the Fundamentals (FUN) series, we possessed the skills and just needed to learn the new tools. We had to learn how to design a PCB completely from scratch, since in the FUN classes we were given a solid starting point with most of the components. We used the tool and website SnapEDA [9] to get the schematic symbols and PCB footprints for many of our components. All software used in the project was provided free of charge, which helped with our $500 budget.

Two of the biggest physical constraints in our project were the force sensors and track block. The force sensors we ordered had both a 500 Newton maximum rating version and a 1000N version. We were forced to order the 500N sensor due to a two month long lead time on all 1000N sensors. This was initially concerning because we were unsure if the sensors would be able to accurately measure the magnitude of force produced by the average sprinter. Thankfully, in our testing we have never seen force values that come close to maxing out the sensors. The track block was a constraint because the high cost of high-quality track blocks forced us to purchase a low quality block. Ultimately the block we purchased worked for our purposes, but if we were to design a production version of the prototype, we would need to invest in a nicer track block or design a proprietary block specifically for force measurements.

Furthermore, if we made a production version of the Smart Sprinter, we would need to focus heavily on improving the aesthetics of the design. Currently our design prioritizes functionality over aesthetic. This is necessary when prototyping, but would cause our design to fail in the marketplace. The three areas where our design needs the most aesthetic improvement are the cable management, height sensing laser poles, and encasing. Currently our design has many long cables that are not as neatly arranged as possible. One solution to this would be to create our own cables with specific length measurements and bundle all the related cables into a single wrapping. To improve the height sensing laser poles we could design a collapsible tripod system. This would make the poles more portable, and they would look better than the current 2x4 lumber construction. Finally, while the current encasing fulfilled our functional requirements of being weatherproof, lightweight, and having a removable cover, it does not look the most pleasing to the eye. In our production design, we would use a CAD tool to create a housing wherein the PCB and STM32 could be mounted and all cables entering the housing would be organized. This housing itself would be designed to mount directly onto the track block to reduce the number of items in the product. If these three features were aesthetically improved, we believe our product would succeed in a production environment.

Societal Impact

For this project, the overall goal is of course to target track sprinters to make their start better. However, it is not just isolated to the sprinter alone. When optimal, shaving block exit time and making sure the force is pure off the blocks can be a massive difference for a sprinter. This of course can have an impact on the track program itself. If the sprinters get better with this data, track programs can recruit better through winning more, such as the UVA program. At such a high level, fractions of a second can make or break a race, so a runner knowing this data can be monumental. With a better program, a school such as UVA can attract more money, increasing student enrollment and impacting all aspects of university life.

The project can also help get youth into running. It adds an additional competition component with the start that kids can use, increasing their involvement. We had hoped to see this in action on demo day, where kids can try out our project and get more involved with

running, but the weather and setup of demo day prevented this. Overall, this project will assist not just track sprinters, but the sport itself, and ultimately assist in athletic involvement.

External Standards

The applicable standards for this project are UM10204, TIA/EIA-232-F, and RS-232. The UM10204 standard outlines criteria for implementing I^2C communications [26] which will be used for interfacing between the force sensor module and the microcontroller. The TIA/EIA-232-F [30] and RS-232 [31] standards outline the control characteristics and voltage levels involved in UART/USART communications which will be used for interfacing between the laptop and microcontroller.

Intellectual Property Issues

The first patent is a reaction time measurement system [32]. It is a “battery-powered accelerometer module attached to a starting block or platform to detect acceleration when a contestant moves” [32]. Its primary independent claim that the group analyzed was “a detection unit attachable to a fixed block or platform against which an athlete bears in a pre-start position, wherein the detection unit includes an accelerometer mounted in said unit to move with said fixed block or platform and produce an accelerometer signal indicative of motion thereof, and a processor for processing said accelerometer signal to detect the athlete's starting reaction time” [32]. Looking at this, there are a few key differences between our project and this patent. First, they gather reaction time, not block exit time. However, our device is capable of manually analyzing reaction time, therefore I feel it encompasses relevant material to our project. The way this patent gathers the time is radically different from our device. Smart Sprinter utilizes force sensors and analyzes the time of the peak of the force to get block exit time, while this patent uses an accelerometer. Therefore, the group feels this patent wouldn’t prevent our device from being patented.

The second patent [33] initially looks as though it would prevent Smart Sprinter from being patented. It states that it includes “a force detection sensor connected to a processing circuit, and two starting blocks for competitor’s feet” [33], which is similar material to the group's project. However, further analyzing the primary independent claim makes this patent different from Smart Sprinter. It is:

> “A starting device for a competitor in a sports competition, the device including a base for fixedly holding the device on a competition surface, a detection sensor connected to an electronic processing circuit, and a block for at least one foot, which is connected to the base and which activates the detection sensor to determine the response time of the competitor at the start of the competition, wherein the block is connected to an H-shaped structure, which includes two spaced apart strips each having a first end and a second opposite end, wherein the

ends of said two strips are rigidly connected to an immobile portion of the device, while a central bar connecting the two strips can be pushed in a preferred direction via the bending of the two strips to activate the detection sensor when a force is applied against the block by at least one foot of a competitor" [33].

A key difference in this claim is the way the block activates the force sensor. This device combines both feet into a central bar that activates two detection sensors. Smart Sprinter has no central bar based on the embedding of the sensors and has two force sensors directly over each foot that allows the user to analyze both feet. Therefore, the group feels the way this patent attached the force sensors does not affect the patentability of Smart Sprinter.

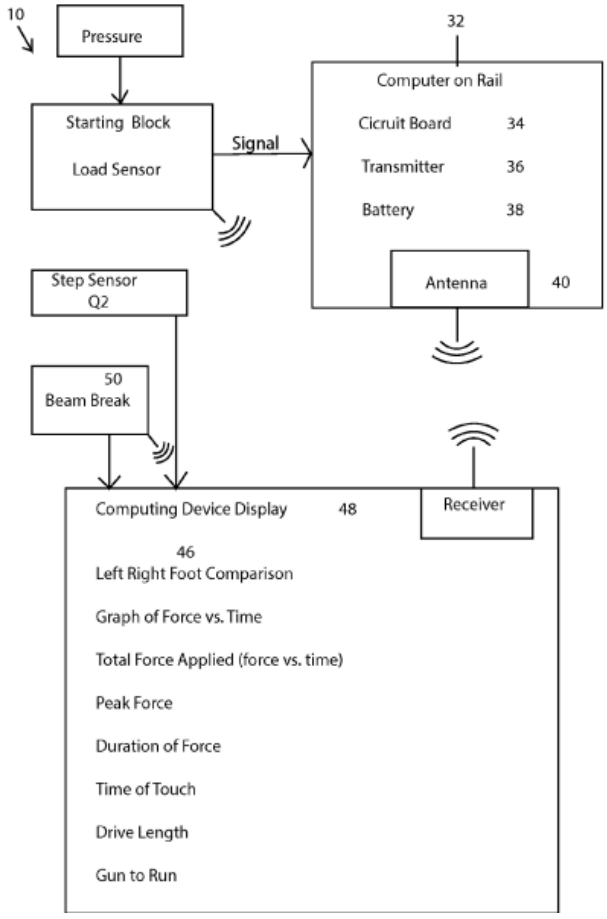


Fig. 44. Antenna and receiver [34]

The last patent [34] is the most similar to our project. Its abstract/primary independent claim is that it is "A force feedback starting block and system for using it. The technology includes starting blocks with sensors for pressure, a step sensor for determining where the runner's first step out of the starting blocks occurs, a laser ruler, and a beam break for

determining when a runner crosses a finish line. Timing and pressure information is displayed on a computer display" [34]. It has timing pressure and information displayed on a computer display, like ours, and has a step sensor to determine force and when a runner first exits the block, which is like our block exit time. However, a key difference is analyzing the finer details of the project/dependent claims. This patent transmits the data from the circuit board to the computer via an antenna and receiver, as seen in Fig. 44, which is vastly different from what our project does. Its system, while gathering similar data to ours, is not identical, such as this patent not gathering whether the height of the runner was optimal or not. Given these three patents, the group believes Smart Sprinter is patentable and has enough differentiation from the prior art research.

Timeline

Prior to beginning the project, we created a Gantt chart to outline a working timeline. This would ensure that we met all deadlines and had milestones in place for our work. While our adherence to the initial Gantt chart was not perfect, our work timeline aligned with our preliminary expectations more often than not.

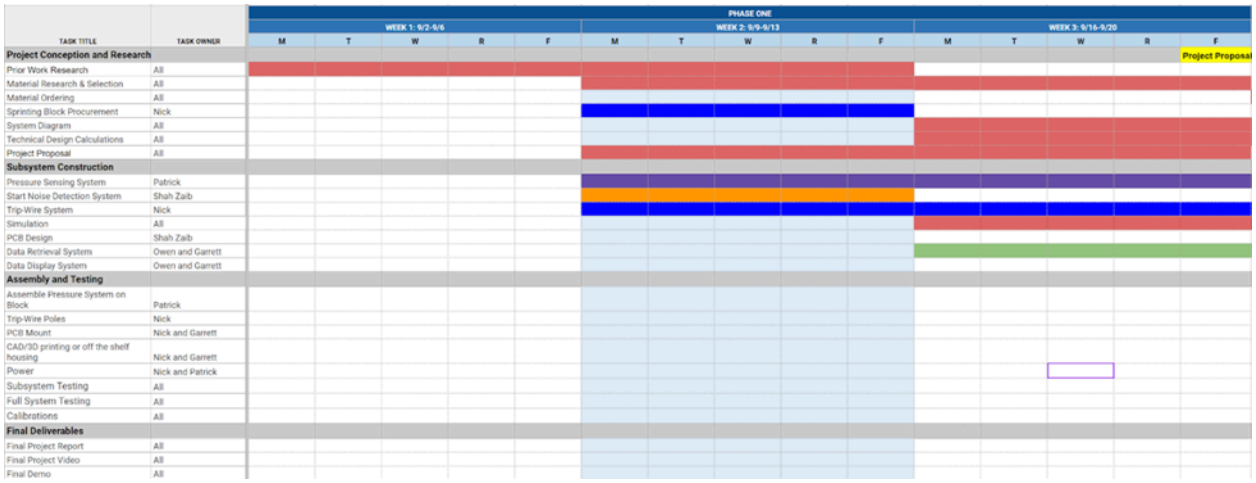


Fig. 45. Initial Gantt Chart Phase 1

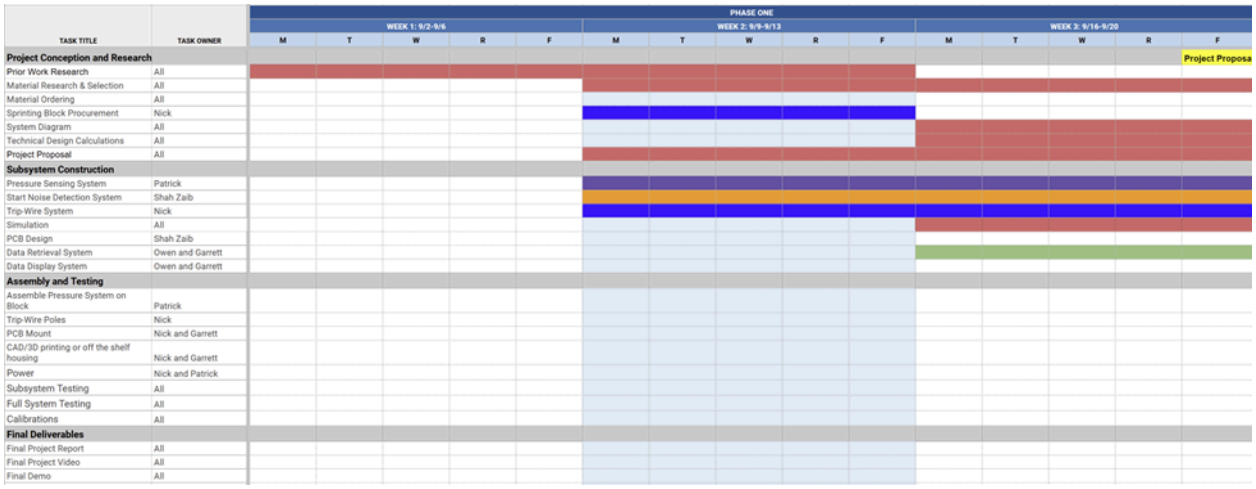


Fig. 46. Final Gantt Chart Phase 1

The first phase of development only exhibited a change in the construction of the "Start Noise Detection System." This was not due to any faults on the team, but rather just a naïve miscalculation in the time it would take to construct this system. This timeline extension did not affect the deadlines of other components.

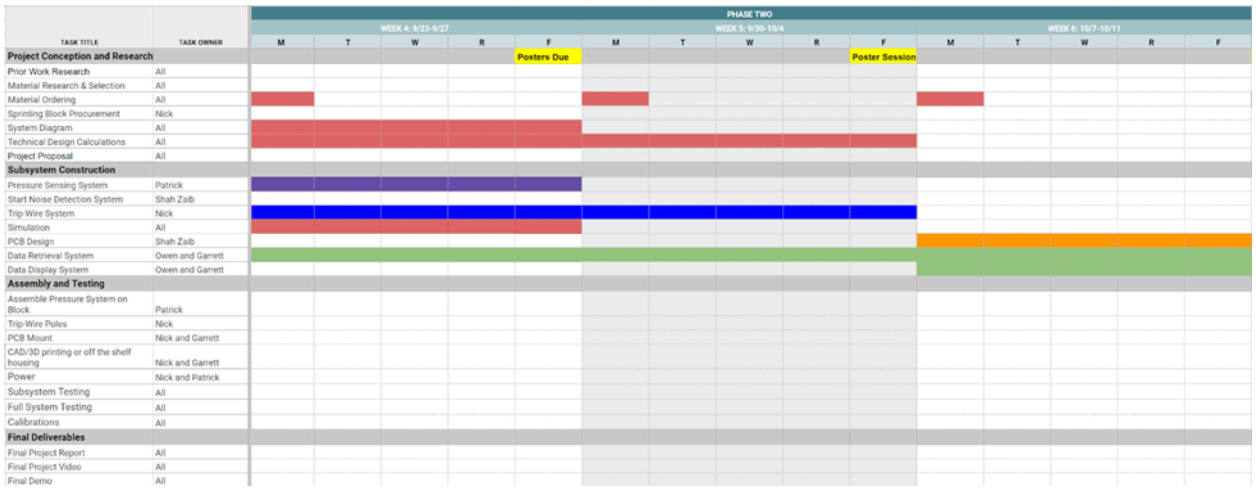


Fig. 47. Initial Gantt Chart Phase 2

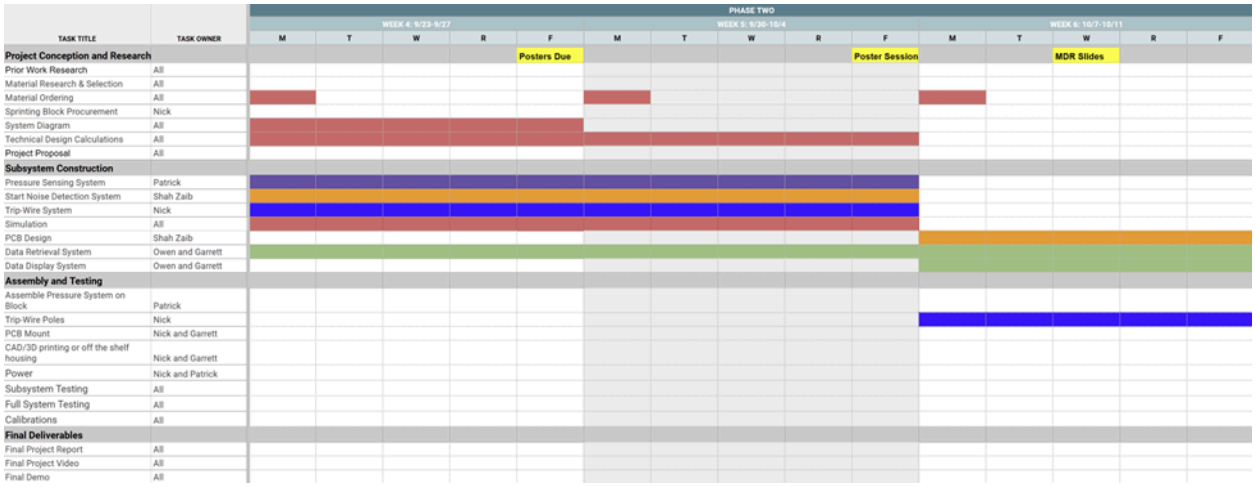


Fig. 48. Final Gantt Chart Phase 2

During the second phase of development there is a clear extension of the top four "Subsystem Construction" components. These extensions occurred for a variety of reasons including shipping delays and hardware compatibility struggles. Specifically, we had to solder jumper wires to the force sensor wires since the sensor did not come with any attached connector. This delay did not cause problems since we built in extra dead space as a buffer between the construction and assembly phases. Also note that the "Trip Wire Poles" were assembled in this phase as opposed to in the third phase as described by our initial Gantt chart.

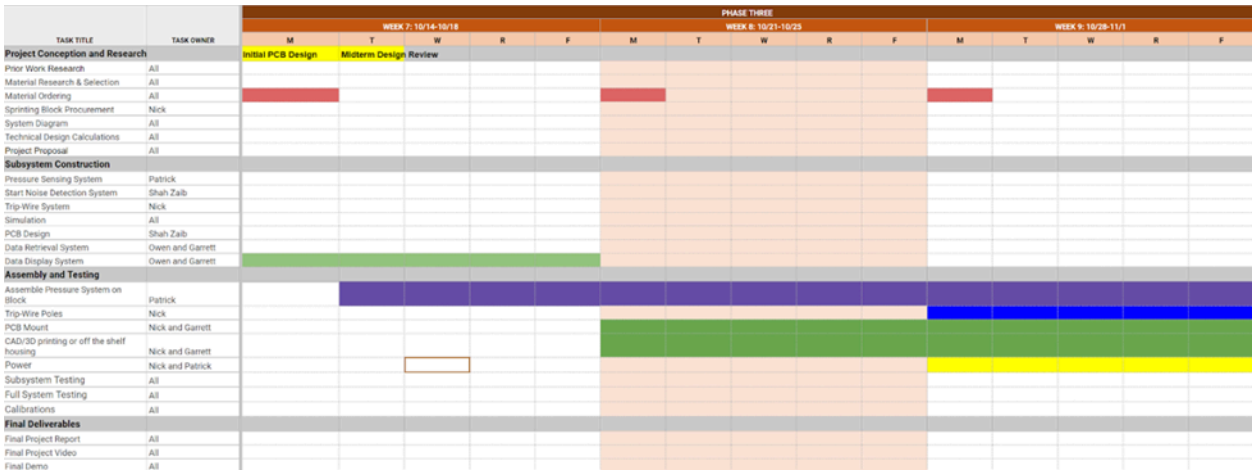


Fig. 49. Initial Gantt Chart Phase 3

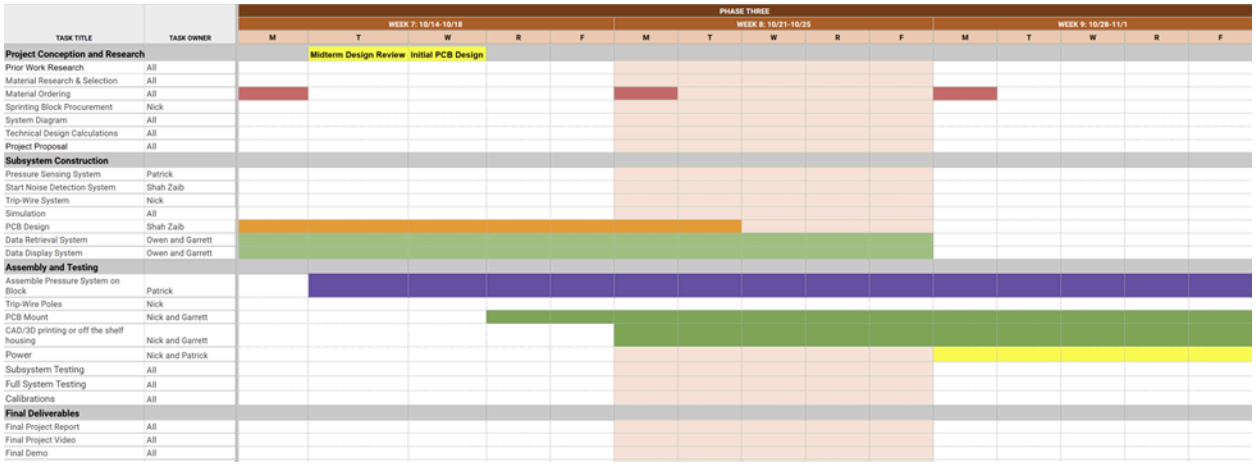


Fig. 50. Final Gantt Chart Phase 3

The third development phase saw the most consequential delay in our project. Our PCB design did not progress as quickly as we had anticipated due to a miscalculation of the complexity of our design. Additionally, we pursued a redesign to equip the PCB with a shield with which we could mount the STM32 directly onto the PCB. This caused a slight delay in the software design because the STM32 software could not be adequately tested without knowledge of how the STM32 would interface with the PCB. Despite this delay, we still met all deadlines on time. We also decided to use a preexisting plastic container as our PCB and STM32 housing instead of CAD/3D Printing as described in the initial Gantt chart.

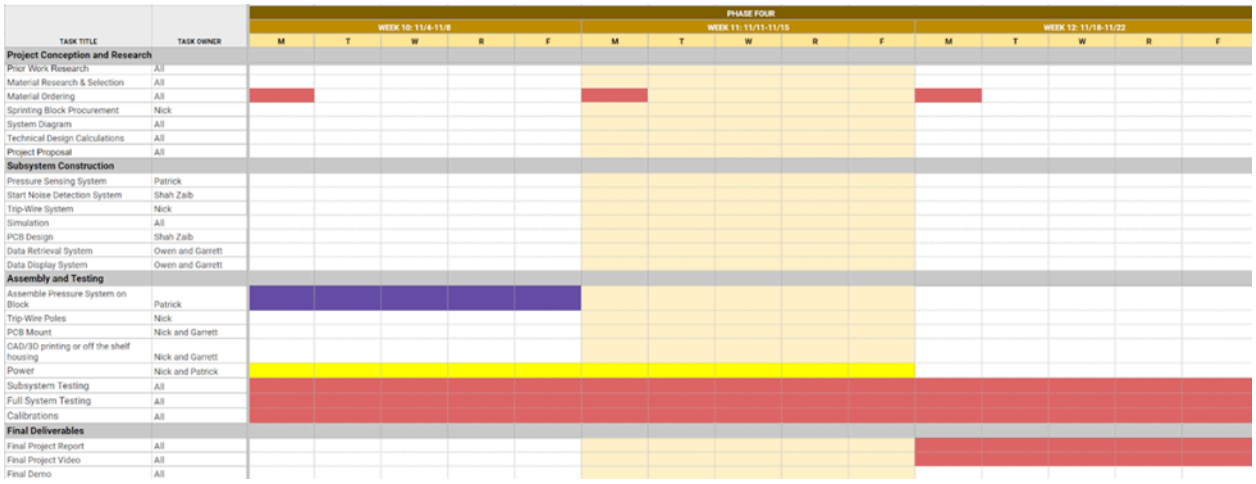


Fig. 51. Initial Gantt Chart Phase 4

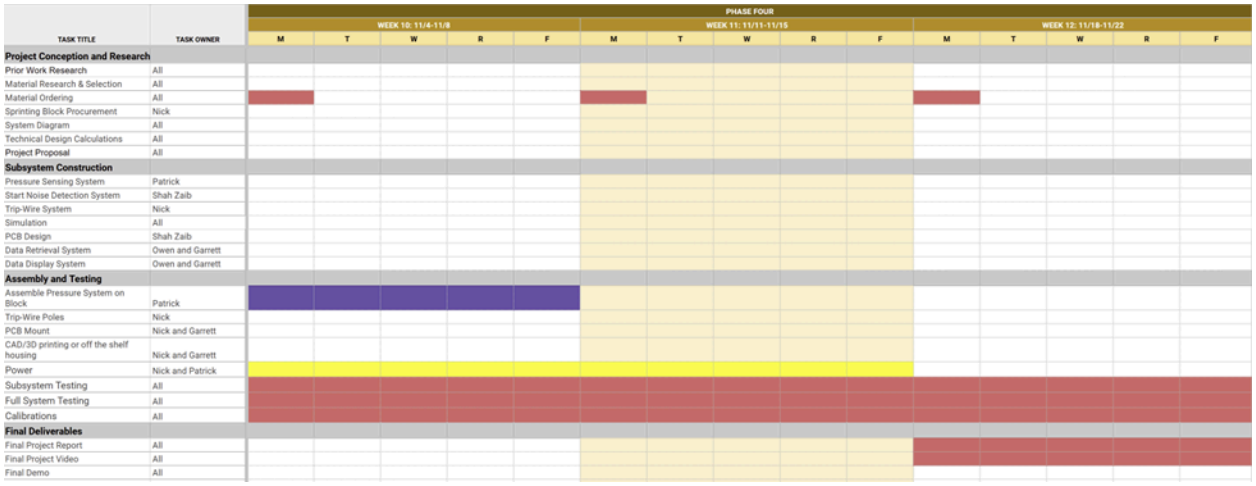


Fig. 52. Final Gantt Chart Phase 4

The penultimate fourth phase saw no alterations from the initial Gantt chart. We began all system testing on time and completed testing by the end of this phase.

Fig. 53. Initial Gantt Chart Phase 5

Fig. 54. Final Gantt Chart Phase 5

The only change made in the fifth phase of development was the extension of the final project video through the end of Week 14. In this phase our prototype was fully functional and our main focus was on aesthetic improvements and writing the final report.

Costs

The overall cost of our project was $391.69 as shown in Table 1 in the Appendix. This is more than $100 under our budget of $500. The big-ticket items were the force sensors, track block, laser sensor, laser relays, metal tape, lumber, and PCBs. We only purchased one force sensor initially because we wanted to ensure the sensor worked before making another $40+ purchase. However, when it came time to buy more force sensors, we found a supplier with the same sensors listed at merely $16.25. We bought the last 3 sensors in stock from this supplier since the price was so low. We were unable to use cost reduction methods on any of the other big-ticket items.

In addition to the big-ticket items, we had a plethora of small electronic components to purchase. These components came almost exclusively from Digikey and included resistors, capacitors, inductors, integrated circuits (ICs), and connector headers. These components were ordered in such small quantities that we could not take advantage of bulk cost reduction. There were also several items we purchased that we did not use in the final prototype. These items were the breadboards, laser relays, whistle, 32-36 AWG crimps, and analog microphone. If we were to produce a second iteration of this design, these items would not be purchased.

If we produced 10,000 units using the same purchase and cost structure we used in this prototype, the total cost would be $391.69*10000 = $3,916,900 (Table 2 in the Appendix). However, there are a number of cost saving strategies that we could employ to reduce this massive expense. First, most of the Digikey items have the option to order in bulk at a reduced price. If we ordered all Digikey in bulk, the total cost would be around $3,533,357.52 (Table 2 in the Appendix). This is nearly $400,000 less than if the components were not ordered in bulk. In addition to cutting costs by buying in bulk, we could automate some of the assembly to save time costs. The easiest assembly tasks to automate would be the PCB soldering, STM32 flashing, and laser pole manufacturing. We could also explore designing a proprietary track block for easy assembly of the force sensors. While all the programming work has already been done, we would need to consider how to distribute the companion Python GUI in a single application package. All these methods would reduce the cost of producing 10,000 units of our smart sprinting block.

Final Results

Overall, our device hits and exceeds most of the criteria our team put forward in our proposal. From the force sensors we do not just obtain max force values, we create a graph of the user's force on each pedal over a period of 3 seconds, which is shown in Fig. 55. This allows the user to obtain more information than originally proposed.

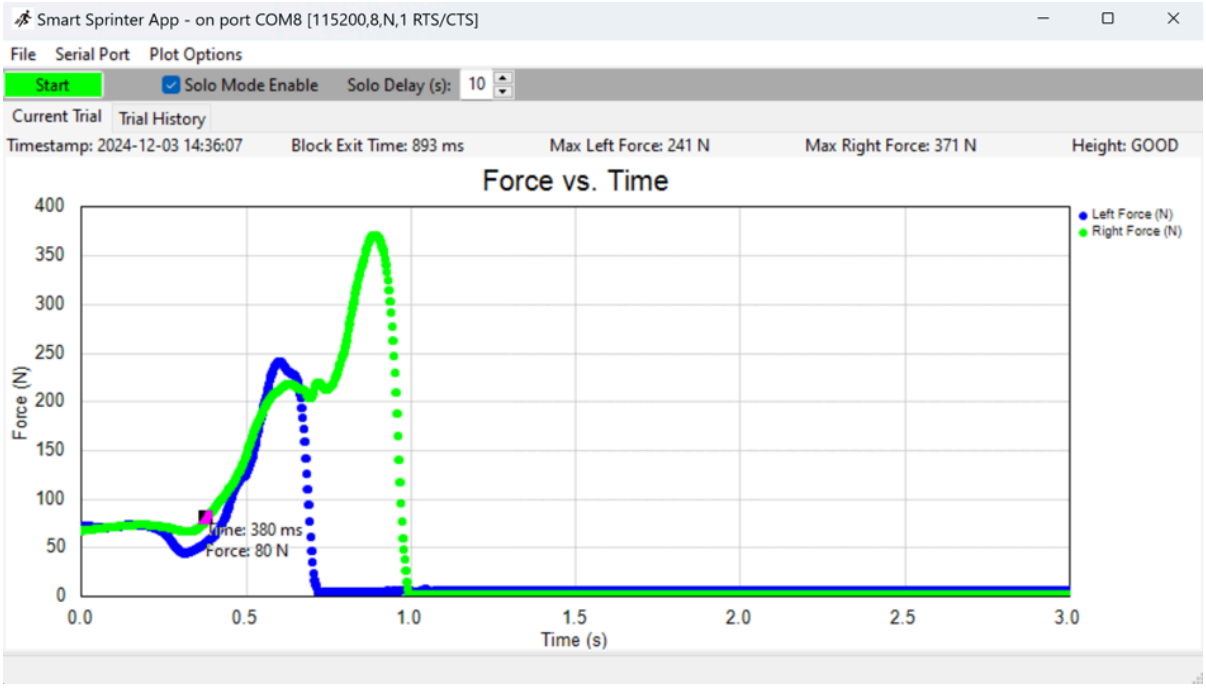
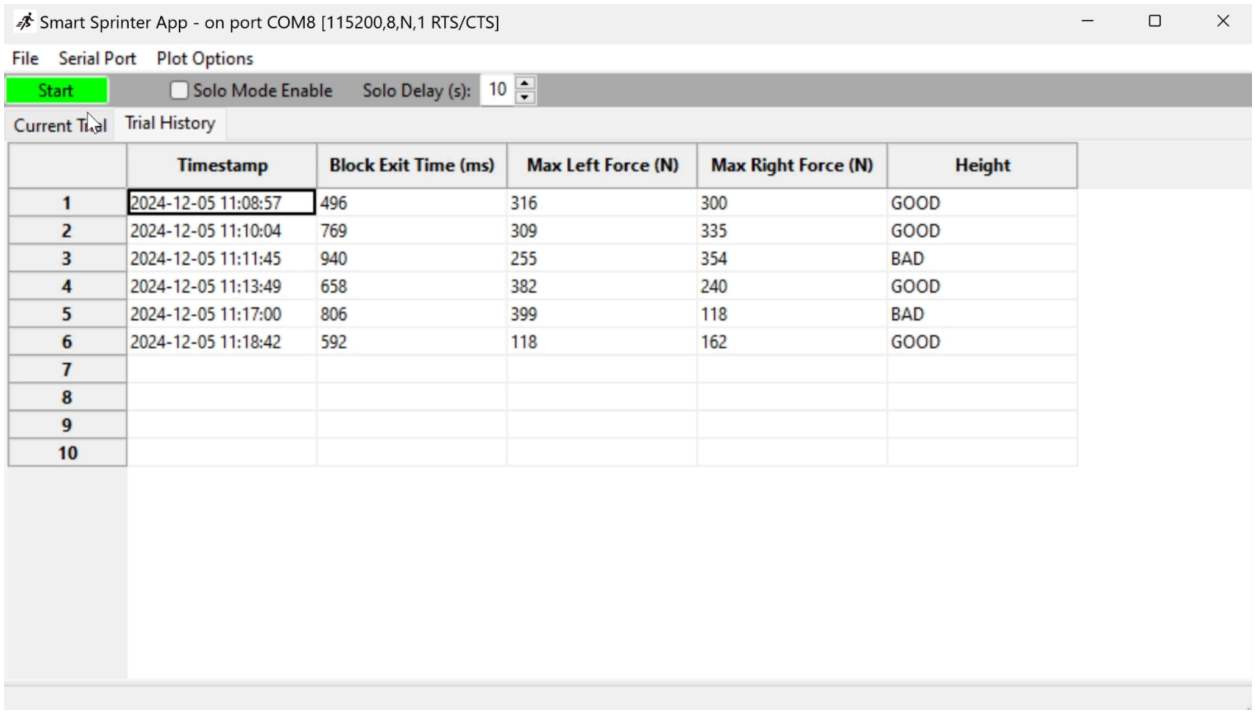


Fig. 55. GUI Data after a Track Start

For example, the user can obtain data on the time it takes for the sprinter to leave the track block, the amount of force the user places on each pedal over time and the maximum force. The GUI allows the end user to hover their mouse over the force plot and it will give the time and force reading at the point closest to their mouse. In Fig. 55 the user had a reaction time before the force spike of about 380 ms. This can be observed manually using the mouse. The block exit time is automatically calculated by grabbing the time of the later foot peak, 893 ms in Fig. 55, reported above the plot. In Fig. 55, the runner is right foot dominant. All this information is useful because it allows the user to see whether he is using his entire body in his track start and how long it takes him to begin the sprint. By optimizing these metrics, a sprinter will be able to greatly increase their ability to start races. There is also a tabular view that allows the sprinter to compare the key metrics of their previous trial shown in Fig. 56.



	Timestamp	Block Exit Time (ms)	Max Left Force (N)	Max Right Force (N)	Height
1	2024-12-05 11:08:57	496	316	300	GOOD
2	2024-12-05 11:10:04	769	309	335	GOOD
3	2024-12-05 11:11:45	940	255	354	BAD
4	2024-12-05 11:13:49	658	382	240	GOOD
5	2024-12-05 11:17:00	806	399	118	BAD
6	2024-12-05 11:18:42	592	118	162	GOOD
7					
8					
9					
10					

Fig. 56. GUI Tabular view of previous runs

In addition, the height sensor, shown in Fig. 57, works well. Long wires to the microcontroller and the movable stands allow the user to place the sensor where required for their use.

Fig. 57. Adjustable Height Stands

As a result, users can customize height and the duration of their low sprinting position. Third, the GUI created by our team shows all the data to the user in easy-to-understand format. The graph is shown on the bottom while specific individual data points, such as maximum force on each foot, height parameter, and block exit time, are shown above. The simple structure allows the user to immediately obtain their performance metrics and make changes to their track start. Finally, the buzzer can generate a noise that is loud enough for the runner to hear but not too loud to cause hearing loss. We added an option that allows the user to solo start the program by giving the system an ability to delay the start sound and data acquisition for a customizable amount of time.

While the buzzer circuit within the start module works exactly as intended, the microphone circuit was unstable and ultimately unable to provide a meaningful readout. The microphone was supposed to be able to detect when a sound peaked over a threshold, such as the loud impulse of a whistle or track start gun, and use this detection to signal a start to the microcontroller. The intermediate voltages of the op amps would drift towards the positive rail and never settle back down. This would occur slowly over anywhere between 30 seconds to 2 minutes depending on the sound present in the environment. After much testing and inspection we found that the flaw was in the configuration of the op amps. There is some kind of feedback between the non inverting amplifier and peak detector that continuously pushes the voltage up. The nature of this problem is such that it can't be simply fixed by changing component values, the op amp circuit would have to be fully redesigned. As such, we abandoned the idea due to time constraints. Our final product is still fully functional, as the buzzer still enables the system to start and produce the three metrics we set out to deliver to the runner.

Overall, the design was a success. We are successfully able to obtain sprint start performance metrics and display them to the user in an easy-to-read format. We are able to do this without impeding the runner and without required outside infrastructure, such as internet or wall power.

Engineering Insights

An important lesson we learned was that on a big project, integration between different subsystems, especially when designed by separate team members, will take longer than you anticipate. There are bound to be bugs and edge cases that require thorough and patient debugging. For example, when getting the STM32 microcontroller to talk to the laptop python program, there was a frustrating bug that took days for us to figure out. Now and then, a few of the bytes out of the 12,000 transferred would get lost. Sometimes we would go ten or more trials without a byte ever getting dropped. An observation was that we only ever saw bytes get dropped when force was applied to the foot pads. At the time we were unsure if this was a coincidence or not.

Originally for the prototype, we used a simple packet format to send the height, block exit time, peak left and right force as comma-separated ASCII text. On the laptop side, we split

and parsed the data by the comma. Later we switched to sending the entire force array for the left and right feet at 1kHz of 3 seconds, corresponding to 6,000 values total or 12,000 bytes. We then switched to sending the data as raw binary rather than ASCII. There was some boilerplate code from the pySerial library related to ASCII printability. Different operating systems use varying characters to represent the newline character. Possible options include carriage return (CR) '\r', line feed (LF) '\n', or both (CRLF) '\r\n'. The newline setting from the pySerial library was set to NEWLINE_CRLF. Thus, every time the bytes '\r' and '\n' were next to each other a byte would get dropped. Since '\r' = 0x0D and \n = 0x0A, anytime 2 bytes were next to each other in the form 0x0D0A, a byte would get dropped. Since the data is sent little-endian, this would correspond to the value 0x0A0D = 2573. We created a packet integrity test where we iterated through transmitting all possible unsigned 16-bit values [0, 2^{16}-1=65535]. Surely enough the value 2573 would get lost every time. After removing the ASCII newline transformation code, all values were sent correctly in the packet integrity test. It also makes sense why the byte loss never occurs when no force is applied, as our sensors output ~1000 then, never 2573.

So, we probably never needed the transport layer Owen and Garrett added with a CRC for corruption and timeout for loss. However, it was a good learning experience. We added some corruption by messing with the data a bit after the CRC is calculated on the STM before transmission and our recovery system works just in case it ever occurs. It also makes sense why the system would never recover since it was making this same mistake in preprocessing the newline character every time.

Another important insight we gained was the importance of proper labeling (Fig. 58). We did not spend enough time working out the proper labeling schemes of our designs and paid a price for this ignorance. During testing we had begun assembling the full system and were trying to connect the force sensors to the PCB. However, we could not remember the pinout of the PCB header connector so when we plugged the sensors in, the positive and negative voltages were switched around. We did not notice this mistake until we tested the sensors and could not get a good reading from either of them. This led to a day's worth of debugging and ultimately coming to the conclusion that we had broken two of our force sensors. Thankfully we had two sensors in reserve, but the time spent debugging this issue was time we should have been spending on full system testing. We also had difficulty when trying to find test pins on the PCB because there were no labels on the board for which pins corresponded to which subsystem. This labeling issue did not cause any mishaps, but did slow down testing occasionally. As a result of these issues, we created labeling diagrams and made sure to label all cables with their proper endpoints.

Fig. 58. Labeled Force Sensor Wires

Future Work

For the future, the main improvements would be on the aesthetic of the project. For the sake of functionality, time, and money, the general assembly of the project is very “DIY”. Future groups could make a more pleasing encasing that still hits the three key functionality points. Groups can also make laser height stands that are cleaner looking, and generally more stable to make alignment slightly easier. The track block itself could also be of higher quality, and the attachment of the force sensors could be more permanent and cleaner.

Aside from physical looks, the GUI could include more features. Eventually it could possibly automate a reaction time, as the group found this difficult. With this, it can calculate the rate of change of force, giving an indication of their jerk. The GUI could also have multiple user accounts and save the graphs, so multiple runners can practice off one laptop, and save their manual data to analyze what automated data cannot.

For the future, groups could also find a solution to the microphone problem explained in this report. This would allow for an additional starting mechanism, and one that is more authentic to a track meet. It is just another nice feature to add to have an accurate simulation of a track start. In addition, there could be an extra laser sensor at waist level that will detect when the runner passes the height stands. This is so the data for the runner is condensed and only includes the necessary information, instead of waiting six seconds so it could gather if the runner crossed the laser height sensor or not, and having the GUI contain force data past a certain point that isn't useful for the runner as they have already left the block a while ago. This extra laser sensor that is always triggered can also give information on the velocity at that set distance, and the acceleration to get there, providing further information for the runner in the GUI.

Finally, future groups are recommended to have better cable management. The laser cables that attach to the PCB are very long and easy to tangle, so sleeves and a roll up system would be very ideal. In addition, the connectors for the lasers and buzzer could be upgraded, as sometimes throughout testing they'd fall out of their crimps. Also, having each pin individually attached to the PCB can allow for the wrong cable being plugged in, which happened and led the group to label the wires as there was no time to buy better connectors. Overall, a more aesthetic product, a working microphone start system, a more fleshed out and usable GUI, and more stable cable management to hinder user error would be the next steps for Smart Sprinter.

References

[1] A. Srivastava, A. Chaudhary, D. Gupta, and A. Rana, "Usage of Analytics in the World of Sports," in *2021 9th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*, Sep. 2021, pp. 1–7. doi: 10.1109/ICRITO51393.2021.9596466.

[2] M. Čoh, B. Jošt, B. Škof, K. Tomažin, and A. Dolenec, "Kinematic and Kinetic Parameters of the Sprint Start and Start Acceleration Model of Top Sprinters," *Gymnica*, vol. 28, 1998, [Online]. Available: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=0b837b883ad5005ff4a1d8d87523db056fb13dc4

[3] R. Nagahara and S. Gleadhill, "Catapult start likely improves sprint start performance," *Int. J. Sports Sci. Coach.*, vol. 17, no. 1, pp. 114–122, Feb. 2022, doi: 10.1177/17479541211015123.

[4] H. Subhashana, C. Bandara, I. Bandara, A. Devindi, K. N, and T. Dharmasena, "Novel Sprinter Assistive Smart Agent for Continuous Performance Improvement," in *2021 International Conference on Advances in Electrical, Computing, Communication and Sustainable Technologies (ICAECT)*, Feb. 2021, pp. 1–6. doi: 10.1109/ICAECT49130.2021.9392395.

[5] M. J. Harland and J. R. Steele, "Biomechanics of the Sprint Start," *Sports Med.*, vol. 23, no. 1, pp. 11–20, Jan. 1997, doi: 10.2165/00007256-199723010-00002.

[6] Same Sky, "CPS-4013-110T Datasheet - Audio Transducers," sameskydevices. Accessed: Sep. 20, 2024. [Online]. Available: https://www.sameskydevices.com/product/resource/cps-4013-110t.pdf

[7] D. K. Meinke *et al.*, "Impulse noise generated by starter pistols," *Int. J. Audiol.*, vol. 52, no. 0 1, pp. S9-19, Feb. 2013, doi: 10.3109/14992027.2012.745650.

[8] Same Sky, "CMEJ-0605-36-L030 Datasheet - Electret Condenser Microphones," sameskydevices. Accessed: Sep. 20, 2024. [Online]. Available: https://www.sameskydevices.com/product/resource/cmej-0605-36-l030.pdf

[9] "Design electronics in a snap. Download free symbols, footprints, & 3D models for millions of electronic components." Accessed: Nov. 28, 2024. [Online]. Available: https://www.snapeda.com

[10] STMicroelectronics, "UM2324 User manual," ST. Accessed: Sep. 20, 2024. [Online]. Available: https://www.st.com/resource/en/user_manual/um2324-stm32-nucleo64-boards-mb1360-stmicroelectronics.pdf

[11] "Taiss/ 1 Pair Photoelectric Sensor M18 Infrared Ray Through-Beam Reflection Optical Photoelectric Switch Sensor NPN NO 6-36VDC Proximity Switch Inductive Distance 5M with mounting Bracket E3F-5DN1: Amazon.com: Industrial & Scientific." Accessed: Sep. 20, 2024. [Online]. Available: https://www.amazon.com/Taiss-Through-Beam-Reflection-Photoelectric-E3F-5DN1-2Z/dp/B07PD9LCK1/ref=pd_bxgy_d_sccl_2/143-6186901-1606826?pd_rd_w=NLbA2&content-id=amzn1.sym.f7fa8b58-6436-47b8-8741-9e90c231669e&pf_rd_p=f7fa8b58-6436-47b8-8741-9e90c231669e&pf_rd_r=21Y6Y2DCPSX39EPR4YRN&pd_rd_wg=wkWAy&pd_rd_r=21f01e83-e644-4706-9296-efac1aaf44f5&pd_rd_i=B07PD9LCK1&th=1

[12] TE Connectivity, “FX29 Compact Compression Load Cell,” Mouser. Accessed: Sep. 20, 2024. [Online]. Available: https://www.mouser.com/datasheet/2/418/9/ENG_DS_FX29_A6-3356260.pdf
[13] J. Wu, “A Basic Guide to I2C,” 2022.
[14] STMicroelectronics, “Description of STM32F2 HAL and low-layer drivers,” ST. Accessed: Sep. 20, 2024. [Online]. Available: https://www.st.com/content/ccc/resource/technical/document/user_manual/56/32/53/cb/69/86/49/0e/DM00223149.pdf/files/DM00223149.pdf/jcr:content/translations/en.DM00223149.pdf
[15] E. Peňa and M. G. Legaspi, “UART: A Hardware Communication Protocol Understanding Universal Asynchronous Receiver/Transmitter,” *Analog Devices*, vol. 54, no. 4, Dec. 2020.
[16] Arm, “NUCLEO-G071RB | Mbed.” Accessed: Dec. 03, 2024. [Online]. Available: https://os.mbed.com/platforms/ST-Nucleo-G071RB/
[17] S. Hymel, “Getting Started with STM32 - Timers and Timer Interrupts,” DigiKey. Accessed: Dec. 03, 2024. [Online]. Available: https://www.digikey.com/en/maker/projects/getting-started-with-stm32-timers-and-timer-interrupts/d08e6493cefa486fb1e79c43c0b08cc6
[18] STMicroelectronics, “Getting started with GPIO - stm32mcu.” Accessed: Dec. 03, 2024. [Online]. Available: https://wiki.st.com/stm32mcu/wiki/Getting_started_with_GPIO
[19] STMicroelectronics, “RM0444 Reference manual: STM32G0x1 advanced Arm®-based 32-bit MCUs.” Nov. 2020. Accessed: Dec. 01, 2024. [Online]. Available: https://www.st.com/content/ccc/resource/technical/document/reference_manual/group0/2f/21/cb/33/78/80/42/64/DM00371828/files/DM00371828.pdf/jcr:content/translations/en.DM00371828.pdf
[20] Team, “Welcome to wxPython!,” wxPython. Accessed: Dec. 06, 2024. [Online]. Available: https://wxpython.org/index.html
[21] C. Liechti, “pySerial 3.4 documentation.” Accessed: Dec. 06, 2024. [Online]. Available: https://pyserial.readthedocs.io/en/latest/
[22] R. Ajitsaria, “What Is the Python Global Interpreter Lock (GIL)? – Real Python.” Accessed: Dec. 06, 2024. [Online]. Available: https://realpython.com/python-gil/
[23] “Thread Safety — NumPy v2.1 Manual.” Accessed: Dec. 06, 2024. [Online]. Available: https://numpy.org/doc/2.1/reference/thread_safety.html
[24] “wx.lib.plot — wxPython Phoenix 4.2.2 documentation.” Accessed: Dec. 06, 2024. [Online]. Available: https://docs.wxpython.org/wx.lib.plot.html
[25] STMicroelectronics, “Arm® Cortex®-M0+ 32-bit MCU, up to 64 KB Flash, 18 KB RAM, 2x USART, timers, ADC, DAC, comm. I/Fs, 1.7-3.6V.” Sep. 2021. [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32g071rb.html
[26] NXP Semiconductors, “I2C-bus specification and user manual.” 2021. [Online]. Available: https://www.nxp.com/docs/en/user-guide/UM10204.pdf
[27] STMicroelectronics, “STM32CubeIDE - Integrated Development Environment for STM32 - STMicroelectronics.” Accessed: Nov. 28, 2024. [Online]. Available: https://www.st.com/en/development-tools/stm32cubeide.html
[28] Analog Devices, “LTspice Information Center | Analog Devices.” Accessed: Nov. 28, 2024. [Online]. Available: https://www.analog.com/en/resources/design-tools-and-calculators/ltspice-simulator.html
[29] KiCad, “KiCad EDA: A Cross Platform and Open Source Electronics Design Automation Suite.” Accessed: Nov. 28, 2024. [Online]. Available: https://www.kicad.org/
[30] Texas Instruments, “Interface Circuits for TIA/EIA-232-F,” TI. Accessed: Sep. 20, 2024. [Online]. Available: https://www.ti.com/lit/an/slla037a/slla037a.pdf?ts=1726834649295
[31] X. Han and X. Kong, “The Designing of Serial Communication Based on RS232,” in *2010 First ACIS International Symposium on Cryptography, and Network Security, Data Mining and Knowledge Discovery, E-Commerce and Its Applications, and Embedded Systems*, Oct. 2010, pp. 382–384. doi: 10.1109/CDEE.2010.80.
[32] Lynx System Developers, “US Patent for Reaction time measurement system Patent (Patent # 6,002,336 issued December 14, 1999) - Justia Patents Search.” Accessed: Dec. 06, 2024. [Online].

Available: https://patents.justia.com/patent/6002336
[33] Swiss Timing, “US Patent for Starting device for a competitor in a sports competition Patent (Patent # 8,992,386 issued March 31, 2015) - Justia Patents Search.” Accessed: Dec. 06, 2024. [Online]. Available: https://patents.justia.com/patent/8992386#claims
[34] W. Lawrence, “Force feedback starting blocks,” US20140221159A1, Aug. 07, 2014 Accessed: Dec. 06, 2024. [Online]. Available: https://patents.google.com/patent/US20140221159A1/en

Appendix

Table 1. Costs

Part Name	Manufacturer Part Number	Supplier Part Number	Supplier	Qty	Per Unit Price	Cost
STM32	NUCLEO-G071RB	511-NUCLEO-G071RB	Mouser	1	10.98	10.98
FX29 Force Sensor	FX29K0-100A-0100-L	824-FX29K0-100A0100L	Mouser	1	40.74	40.74
Track Block		B07SRC3Y2T	Amazon	1	42	42
Analog Microphone	CMEJ-0605-36-L030	102-6549-ND	Digikey	1	0.74	0.74
Piezo Buzzer	CPS-4013-110T	102-CPS-4013-110T-ND	Digikey	1	2.31	2.31
Laser Sensor	E3F-5DN1	B07PD9LCK1	Amazon	1	19.99	19.99
Laser Relays	JQX-13FL-DC12V	B07QXXM1RV	Amazon	1	12.99	12.99
12V Battery (5-pack)	LW-23A-5	B06ZYRCP2B	Amazon	1	5.99	5.99
Breadboards	239	1528-2143-ND	Digikey	2	5.95	11.9
Whistle		B08YZ5LR15	Amazon	1	4.99	4.99
32-36 AWG Crimps	16020111	WM19444-ND	Digikey	50	0.1178	5.89
Piezoelectric Horn Driver	RE46C100E8F	RE46C100E8F-ND	Digikey	1	0.73	0.73
1MHz, Low-Power Op Amp	MCP6004-E/P	MCP6004-E/P-ND	Digikey	1	0.65	0.65
Magnets		B08K2KYW8K	Amazon	1	7.99	7.99
Metal Tape		B07JMY7QD4	Amazon	1	14.99	14.99
Laser Pointer		B09TFNQM7Z	Amazon	1	8.98	8.98
FX29 Force Sensor w/ Connector	FX29K0-040B-0100-L	40AH0471	Newark	3	16.25	48.75
I2C Cables (4-pack)	MFI2C-01	2234-MFI2C-01-ND	Digikey	1	7.95	7.95
38 Position Header Connector	PPPC192LFBN-RC	S7122-ND	Digikey	2	2.67	5.34
8 Position Header Connector	PPTC081LFBN-RC	S7006-ND	Digikey	2	0.75	1.5
6 Position Header Connector	PPTC061LFBN-RC	S7004-ND	Digikey	1	0.61	0.61
10 Position Header Connector	PPTC101LFBN-RC	S7008-ND	Digikey	1	0.83	0.83
2 Position Connector Header	S2B-XH-A-1	455-S2B-XH-A-1-ND	Digikey	3	0.21	0.63
3 Position Connector Header	S3B-XH-A-1	455-S3B-XH-A-1-ND	Digikey	2	0.26	0.52
14 Position DIP	1-2199298-3	A120348-ND	Digikey	1	0.24	0.24
8 Position DIP	1-2199298-2	A120347-ND	Digikey	2	0.23	0.46
Buck Switching Regulator IC 3.3V	AP63203WU-7	AP63203WU-7DITR-ND	Digikey	1	1.32	1.32
10uF 16V Ceramic Capacitor	CL21B106KOQNNNE	1276-2872-2-ND	Digikey	1	0.22	0.22

22uF 25V Ceramic Capacitor	CL32B226KAJNNNE	1276-3392-2-ND	Digikey	2	0.62	1.24
0.1uF 50V Ceramic Capacitor	CL10B104KB8NNNC	1276-1000-2-ND	Digikey	4	0.1	0.4
3.9uH Inductor	RLS-397	945-30004967-ND	Digikey	1	1	1
Voltage Regulator IC	TLV70220PDBVR	296-28449-2-ND	Digikey	1	0.3	0.3
1uF 50V Ceramic Capacitor	CL21B105KBFNNNE	1276-1029-2-ND	Digikey	9	0.11	0.99
Voltage Regulator IC	L78M09CDT-TR	497-1208-2-ND	Digikey	1	1.04	1.04
0.33uF 50V Ceramic Capacitor	CL21B334KBFNNNE	1276-1123-2-ND	Digikey	1	0.1	0.1
Comparator General Purpose CMOS	TLV3701IP	296-13379-5-ND	Digikey	1	1.74	1.74
Battery Holder	BH48AASF	BH48AASF-ND	Digikey	1	3.19	3.19
1.5V Alkaline Battery	ZEUS AA	2059-ZEUSAA-ND	Digikey	4	1.93	7.72
Battery Connector Strap	234	36-234-ND	Digikey	1	0.67	0.67
1000pF 50V Ceramic Capacitor	CL10B102KB8NNNC	1276-1018-2-ND	Digikey	1	0.1	0.1
4 Position Right Angle Connector Header	705550038	WM4177-ND	Digikey	2	1.05	2.1
2.2kOhm Resistor	CRCW08052K20FKEAHP	541-2.20KTTR-ND	Digikey	1	0.19	0.19
1.5MOhm Resistor	RC2512FK-071M5L	13-RC2512FK-071M5LTR-ND	Digikey	1	0.41	0.41
150kOhm Resistor	RC2512FK-07150KL	13-RC2512FK-07150KLTR-ND	Digikey	1	0.31	0.31
10kOhm Resistor	RC0603FR-0710KL	311-10.0KHRTR-ND	Digikey	4	0.1	0.4
30V Diode	BAT54LT1G	BAT54LT1GOSCT-ND	Digikey	1	0.1	0.1
USB-A to Micro-USB Cable	AK67421-3	AE10343-ND	Digikey	1	7.86	7.86
Vertical 2 Position Header Pin	61300211121	732-5315-ND	Digikey	4	0.13	0.52
Vertical 3 Position Header Pin	61300311121	732-5316-ND	Digikey	1	0.13	0.13
Vertical 5 Position Header Pin	61300511121	732-5318-ND	Digikey	2	0.26	0.52
22AWG Low Voltage Cable	9697T713	9697T713	McMaster	3	11	33
3 Rectangular Connectors	XHP-3	455-2219-ND	Digikey	2	0.11	0.22
2 Rectangular Connectors	XHP-2	455-2266-ND	Digikey	3	0.1	0.3
22AWG Crimps	SXH-001T-P0.6	455-1135-1-ND	Digikey	32	0.071	2.272
JBWeld			Walmart	1	6.54	6.54
Lumber			Lowe's	1	14.95	14.95
Hardware Bolts and Nuts			Lowe's	1	10.24	10.24
PCB			JLCPCB	5	6.582	32.91
					TOTAL COST	391.692

Table 2. 10,00 Unit Costs

Part Name	Manufacturer Part Number	Supplier Part Number	Supplier	Largest Qty	Largest Quantity Per Unit Price	"10000-Unit" Cost
STM32	NUCLEO-G071RB	511-NUCLEO-G071RB	Mouser			109800
FX29 Force Sensor	FX29K0-100A-0100-L	824-FX29K0-100A0100L	Mouser			407400
Track Block		B07SRC3Y2T	Amazon			420000
Analog Microphone	CMEJ-0605-36-L030	102-6549-ND	Digikey	5000	0.30119	3011.9
Piezo Buzzer	CPS-4013-110T	102-CPS-4013-110T-ND	Digikey	100	1.575	15750
Laser Sensor	E3F-5DN1	B07PD9LCK1	Amazon			199900
Laser Relays	JQX-13FL-DC12V	B07QXXM1RV	Amazon			129900
12V Battery (5-pack)	LW-23A-5	B06ZYRCP2B	Amazon			59900
Breadboards	239	1528-2143-ND	Digikey			119000
Whistle		B08YZ5LR15	Amazon			49900
32-36 AWG Crimps	16020111	WM19444-ND	Digikey	2500	0.0769	3845
Piezoelectric Horn Driver	RE46C100E8F	RE46C100E8F-ND	Digikey	100	0.56	5600
1MHz, Low-Power Op Amp	MCP6004-E/P	MCP6004-E/P-ND	Digikey	100	0.5	5000
Magnets		B08K2KYW8K	Amazon			79900
Metal Tape		B07JMY7QD4	Amazon			149900
Laser Pointer		B09TFNQM7Z	Amazon			89800
FX29 Force Sensor w/ Connector	FX29K0-040B-0100-L	40AH0471	Newark			487500
I2C Cables (4-pack)	MFI2C-01	2234-MFI2C-01-ND	Digikey	101	6.36	63600
38 Position Header Connector	PPPC192LFBN-RC	S7122-ND	Digikey	440	1.209	24470.16
8 Position Header Connector	PPTC081LFBN-RC	S7006-ND	Digikey	5200	0.2925	6084
6 Position Header Connector	PPTC061LFBN-RC	S7004-ND	Digikey	5000	0.2327	2327
10 Position Header Connector	PPTC101LFBN-RC	S7008-ND	Digikey	5120	0.325	3328
2 Position Connector Header	S2B-XH-A-1	455-S2B-XH-A-1-ND	Digikey	10000	0.0675	2025
3 Position Connector Header	S3B-XH-A-1	455-S3B-XH-A-1-ND	Digikey	10000	0.08251	1650.2
14 Position DIP	1-2199298-3	A120348-ND	Digikey	2516	0.11412	1148.50368
8 Position DIP	1-2199298-2	A120347-ND	Digikey	2520	0.10864	2190.1824
Buck Switching Regulator IC 3.3V	AP63203WU-7	AP63203WU-7DITR-ND	Digikey	15000	0.2855	4282.5
10uF 16V Ceramic Capacitor	CL21B106KOQNNNE	1276-2872-2-ND	Digikey	10000	0.034	340
22uF 25V Ceramic Capacitor	CL32B226KAJNNNE	1276-3392-2-ND	Digikey	10000	0.13939	2787.8
0.1uF 50V Ceramic Capacitor	CL10B104KB8NNNC	1276-1000-2-ND	Digikey	12000	0.00303	145.44
3.9uH Inductor	RLS-397	945-30004967-ND	Digikey	500	0.71	7100
Voltage Regulator IC	TLV70220PDBVR	296-28449-2-ND	Digikey	15000	0.09385	1407.75
1uF 50V Ceramic Capacitor	CL21B105KBFNNNE	1276-1029-2-ND	Digikey	2000	0.0176	1584
Voltage Regulator IC	L78M09CDT-TR	497-1208-2-ND	Digikey	12500	0.22046	2755.75

0.33uF 50V Ceramic Capacitor	CL21B334KBFNNNE	1276-1123-2-ND	Digikey	10000	0.00957	95.7
Comparator General Purpose CMOS	TLV3701IP	296-13379-5-ND	Digikey	500	0.81832	8183.2
Battery Holder	BH48AASF	BH48AASF-ND	Digikey	5000	1.60443	16044.3
1.5V Alkaline Battery	ZEUS AA	2059-ZEUSAA-ND	Digikey	5040	0.26928	10857.3696
Battery Connector Strap	234	36-234-ND	Digikey	5000	0.33616	3361.6
1000pF 50V Ceramic Capacitor	CL10B102KB8NNNC	1276-1018-2-ND	Digikey	12000	0.0036	43.2
4 Position Right Angle Connector Header	705550038	WM4177-ND	Digikey	1026	0.51271	10520.8092
2.2kOhm Resistor	CRCW08052K20FKEAHP	541-2.20KTTR-ND	Digikey	10000	0.02424	242.4
1.5MOhm Resistor	RC2512FK-071M5L	13-RC2512FK-071M5LTR-ND	Digikey	12000	0.05406	648.72
150kOhm Resistor	RC2512FK-07150KL	13-RC2512FK-07150KLTR-ND	Digikey	12000	0.04947	593.64
10kOhm Resistor	RC0603FR-0710KL	311-10.0KHRTR-ND	Digikey	10000	0.00263	105.2
30V Diode	BAT54LT1G	BAT54LT1GOSCT-ND	Digikey	15000	0.01927	289.05
USB-A to Micro-USB Cable	AK67421-3	AE10343-ND	Digikey	1000	3.45375	34537.5
Vertical 2 Position Header Pin	61300211121	732-5315-ND	Digikey	1000	0.06	2400
Vertical 3 Position Header Pin	61300311121	732-5316-ND	Digikey	100	0.0688	688
Vertical 5 Position Header Pin	61300511121	732-5318-ND	Digikey	1000	0.11625	2325
22AWG Low Voltage Cable	9697T713	9697T713	McMaster			330000
3 Rectangular Connectors	XHP-3	455-2219-ND	Digikey	3000	0.03699	887.76
2 Rectangular Connectors	XHP-2	455-2266-ND	Digikey	3000	0.03366	1211.76
22AWG Crimps	SXH-001T-P0.6	455-1135-1-ND	Digikey	16000	0.01841	589.12
JBWeld			Walmart			65400
Lumber			Lowe's			149500
Hardware Bolts and Nuts			Lowe's			102400
PCB			JLCPCB			329100
					TOTAL "10000-Unit" Cost	3533357.515

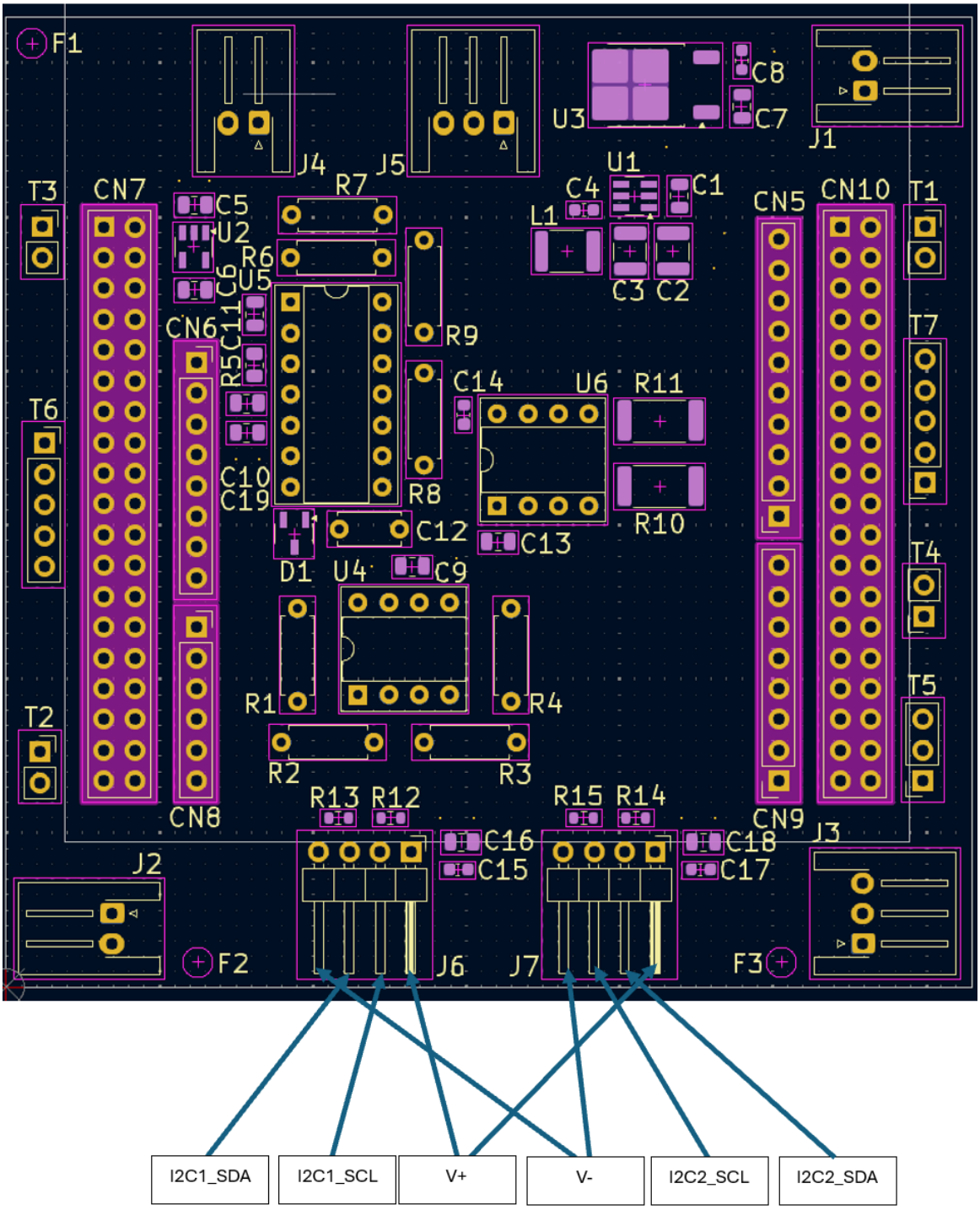


Fig. 59. I2C Pinout Diagram For PCB

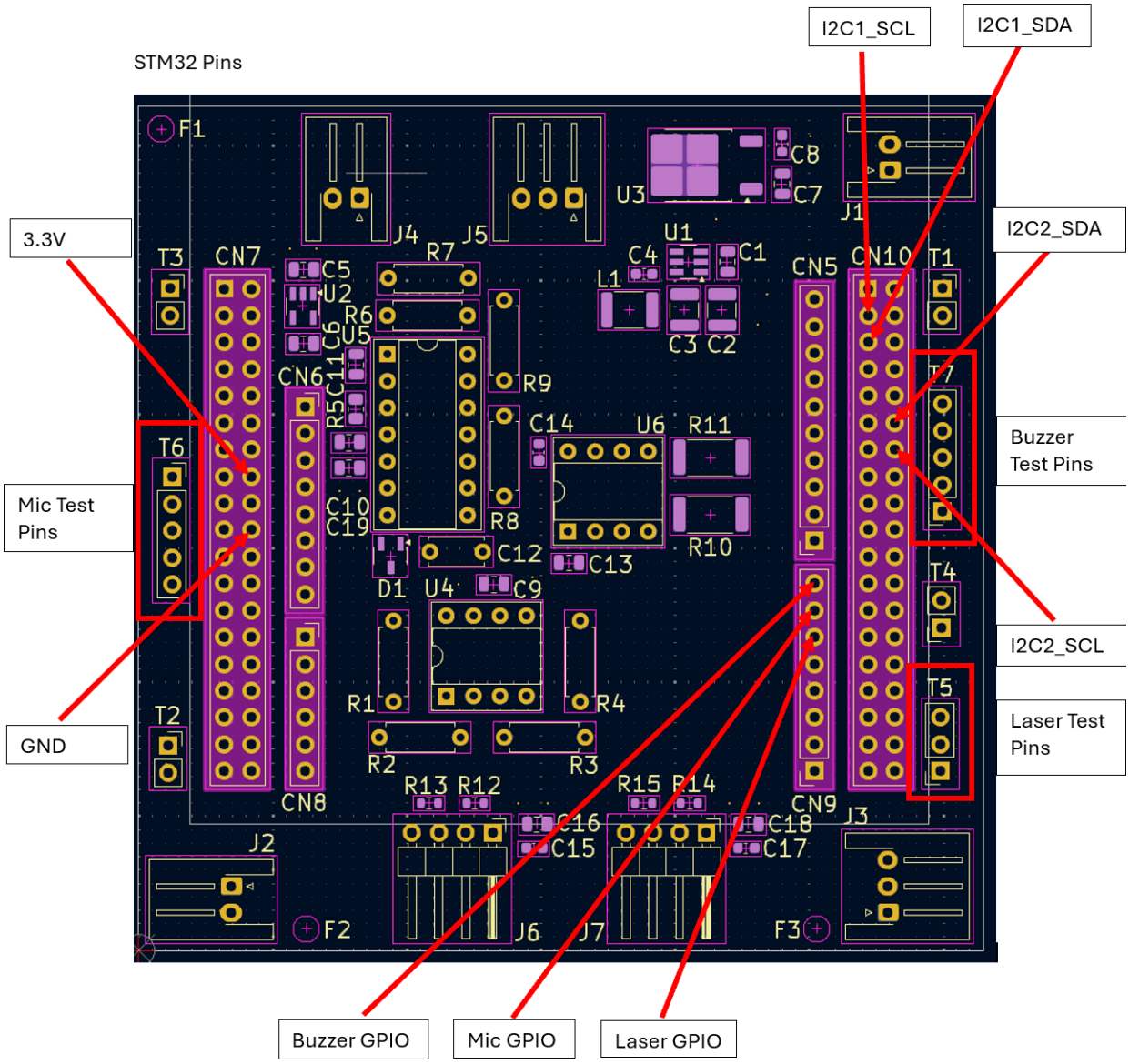


Fig. 60. Test Pin Diagram for PCB

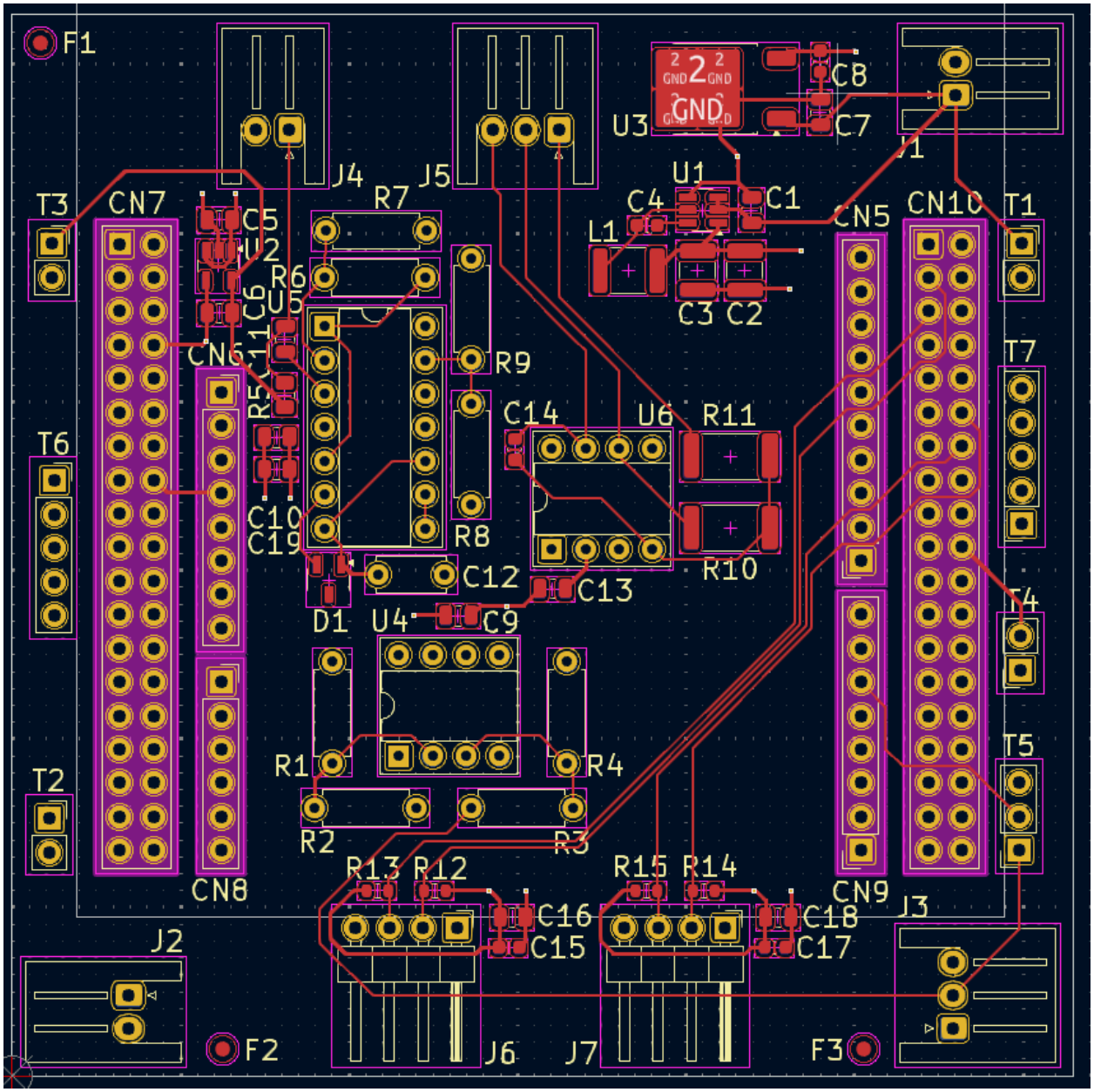


Fig. 61. PCB Layer 1 Surface Routing

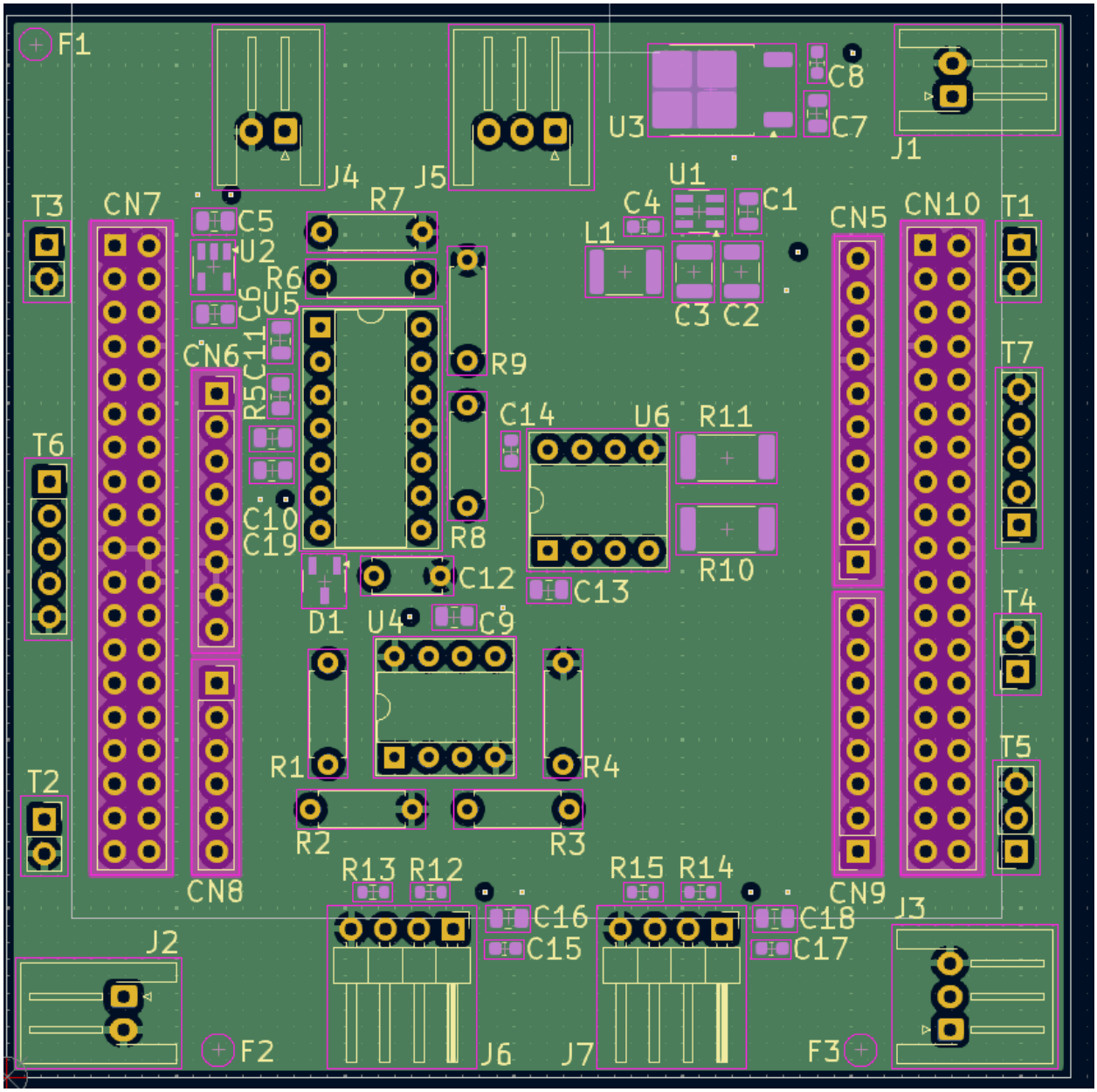


Fig. 62. PCB Layer 2 Ground Plane

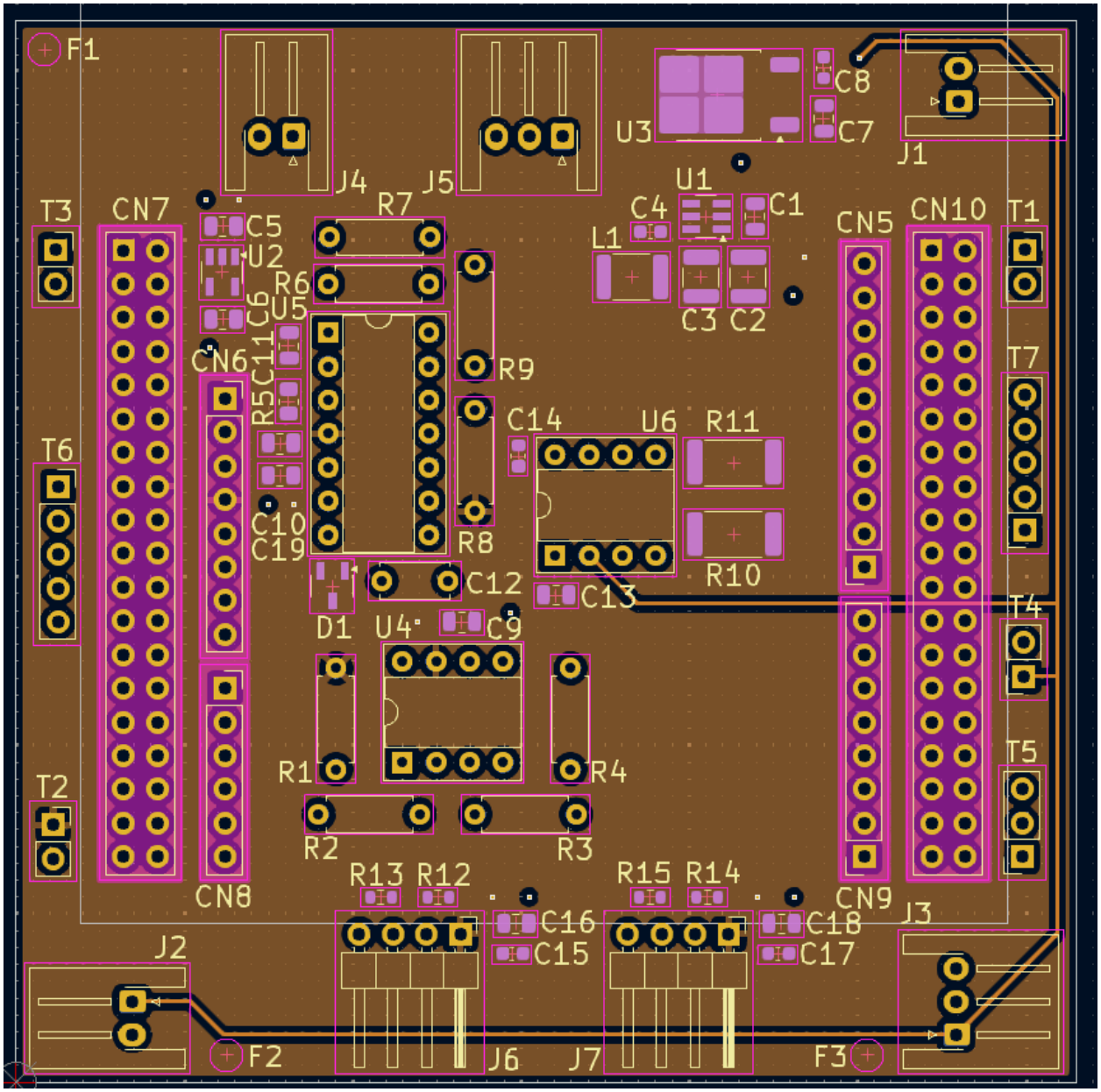


Fig. 63. PCB Layer 3 3.3V Power Plane

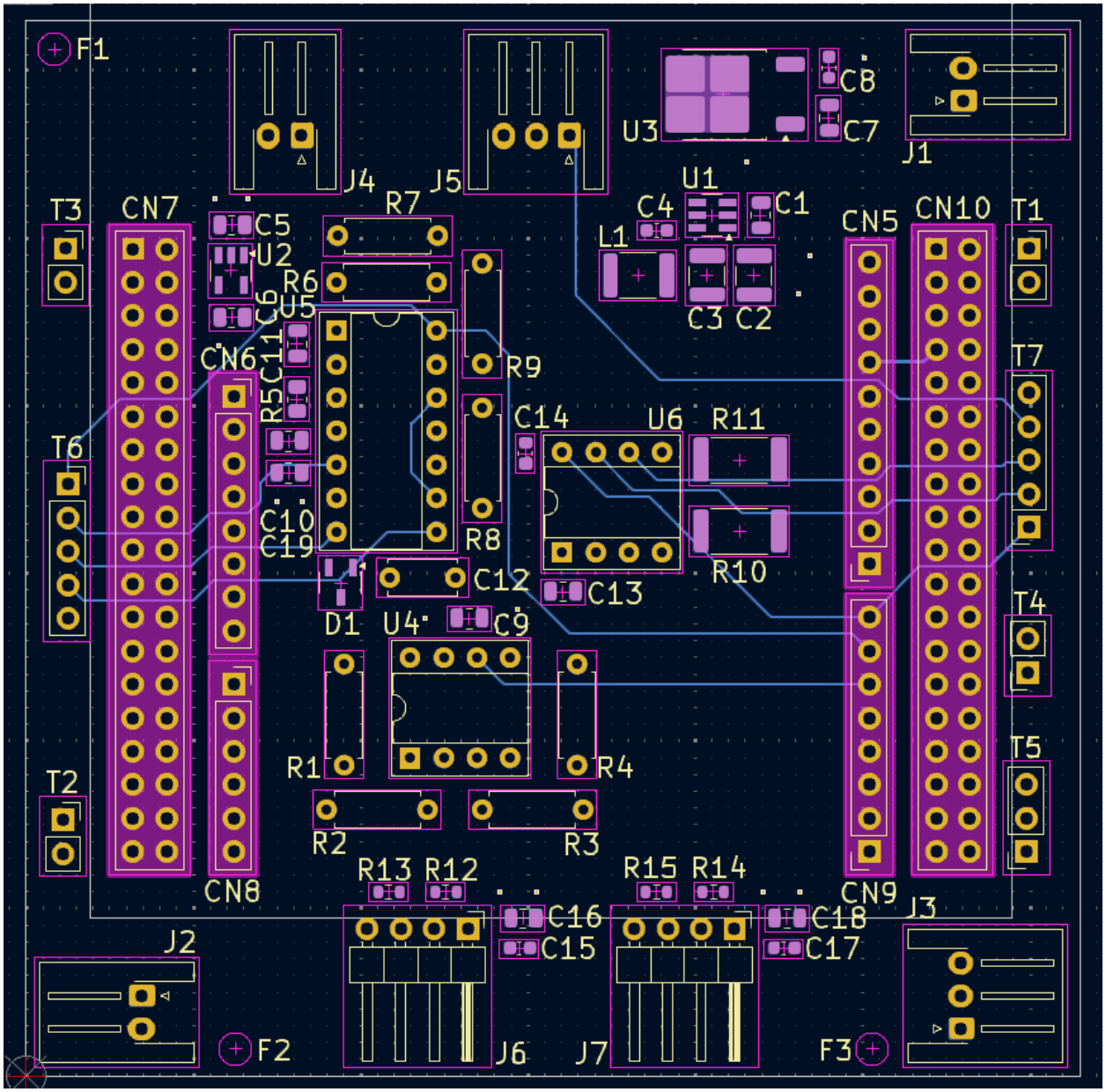


Fig. 64. PCB Layer 4 Back Routing

```c
int main(void)
{

  /* USER CODE BEGIN 1 */
    HAL_GPIO_WritePin(Buzzer_Output_GPIO_Port, Buzzer_Output_Pin, GPIO_PIN_RESET);
    HAL_StatusTypeDef ret_l;
    uint8_t buf_l[12];
//  int16_t val_l;
    HAL_StatusTypeDef ret_r;
    uint8_t buf_r[12];
//  int16_t val_r;
    int16_t threshold = 0x4B0; //Threshold = 1000

  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_TIM16_Init();
  MX_USART2_UART_Init();
  MX_I2C1_Init();
  MX_I2C2_Init();
  MX_CRC_Init();
  MX_TIM17_Init();
  /* USER CODE BEGIN 2 */
  // set button presses to 0
  button_presses = 0;
  timer_val = 1;
  timer_init = 0;
  timer17_init = 0;
  timer_started = 0;
//  height_good = 1;

  // Start UART_interrupt
  HAL_UART_Receive_IT(&huart2, &rx_data, 1);

  /* USER CODE END 2 */
```

Fig. 65. First half of main (ran once)

```c
  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {

      buf_r[0] = REG_DATA;
      buf_l[0] = REG_DATA;

      // TODO: change back to max delay?
      ret_r = HAL_I2C_Master_Receive(&hi2c1, FX29_ADDR, buf_r, 2, HAL_MAX_DELAY);


      ret_l = HAL_I2C_Master_Receive(&hi2c2, FX29_ADDR, buf_l, 2, HAL_MAX_DELAY);

      if ((ret_r != HAL_OK) || (ret_l != HAL_OK)) {
          uart_buf_len = sprintf(uart_buf, "Error Rx. Right: %d. Left: %d\r\n", ret_r, ret_l);
          HAL_UART_Transmit(&huart2, (uint8_t *)uart_buf, uart_buf_len, HAL_MAX_DELAY);
      }
      else {

          val_r = ((((uint16_t)buf_r[0])<<8)|(buf_r[1])) & 0x3FFF; // Get bridge data bits 13-0
          val_l = ((((uint16_t)buf_l[0])<<8)|(buf_l[1])) & 0x3FFF; // Get bridge data bits 13-0

          // Save val to force array
          if (forces_index < 3000){
              forces[forces_index] = val_l;
              forces[forces_index+3000] = val_r;
  //            forces_r[forces_index] = val_r;
              forces_index++;
          }


      }
      HAL_Delay(0); // Measure every millisecond, functions adds 1


    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
}
```

Fig. 66. Second half of main (infinite while loop)

```c
/* USER CODE BEGIN 4 */
void send_force_values(uint16_t *forces, uint16_t len_u16) {
    // calculate checksum
    uint32_t crc = ~HAL_CRC_Calculate(&hcrc, (uint32_t *)forces, FORCE_LENGTH);
    HAL_StatusTypeDef HALStatus = HAL_UART_Transmit(&huart2, &crc, 4, HAL_MAX_DELAY);
    if (HALStatus != HAL_OK) {
        Error_Handler();
    }

    // size wants num bytes so give 2*len since u16 are 2 bytes
    /*
    HALStatus = HAL_UART_Transmit(&huart2, forces, 2*len, HAL_MAX_DELAY);
    if (HALStatus != HAL_OK) {
        Error_Handler();
    }
    */
    for (int i=0; i<len_u16; i+=2) {
        // forces[i] = *(forces+i)
        HALStatus = HAL_UART_Transmit(&huart2, forces+i, 4, HAL_MAX_DELAY);
        if (HALStatus != HAL_OK) {
            Error_Handler();
        }
    }
}
```

Fig. 67. Send force values function

```c
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {
    if (htim->Instance == TIM16) {
        if (timer_init != 0) { // Deals with the update event that occurs the first
            // Stop timer
            HAL_TIM_Base_Stop_IT(&htim16);

            // Check if timer val = 0
            if (timer_val == 0) {
                // Send data to UART
                uart_buf_len = sprintf(uart_buf, "False Start\r\n");
                HAL_UART_Transmit(&huart2, (uint8_t *)uart_buf, uart_buf_len, 100);
            }
            else {
                // Send data to UART
                send_force_values(forces, FORCE_LENGTH);
            }
            // Set button_presses and timer_val to 0
            button_presses = 0;

        }
        timer_init = timer_init | 1;
        timer_started = 0;
    }

    if (htim->Instance == TIM17) {
        if (timer17_init != 0) { // Deals with the update event that occurs the firs
            // Stop timer
            HAL_TIM_Base_Stop_IT(&htim17);
            // Toggle buzzer
            HAL_GPIO_TogglePin(Buzzer_Output_GPIO_Port, Buzzer_Output_Pin);
        }
        timer17_init |= 1;
    }
}
```

Fig. 68. HAL Timer Period Elapsed Callback function

```c
// USR_BTN callback to record times & Laser callback to record height
void HAL_GPIO_EXTI_Falling_Callback(uint16_t GPIO_Pin) {

    if (GPIO_Pin == USR_BTN_Pin){ // This indicates the start of a race
        button_presses++;
        // Start timer if first button press (represents starting the race)
        if (button_presses == 1) {
            // Toggle the start buzzer
            HAL_GPIO_TogglePin(Buzzer_Output_GPIO_Port, Buzzer_Output_Pin);
            // Start buzzer hardware timer
            HAL_TIM_Base_Start_IT(&htim17);
            // Start Timer
            timer_started = 1;
            HAL_TIM_Base_Start_IT(&htim16);
            forces_index = 0;
        }
    }
    if (GPIO_Pin == Laser_Input_Pin) {
        // Set height value - bad
        forces[0] = forces[0] | 0x8000; // This means if height is bad, MSB
    }
}
```

Fig. 69. HAL GPIO External interrupt falling edge callback function

```
317 void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart) {
318     // if rx_data == 0x52 = 'R' -> resend data
319     if (rx_data == 'R') {
320         send_force_values(forces, FORCE_LENGTH);
321     }
322
323     // if rx_data == 0x53 = 'S' -> start timer
324     if (rx_data == 'S') {
325
326         // Toggle the start buzzer
327         HAL_GPIO_TogglePin(Buzzer_Output_GPIO_Port, Buzzer_Output_Pin);
328         HAL_TIM_Base_Start_IT(&htim17);
329         button_presses++;
330
331         // Start timer if GUI button press (represents starting the race)
332         if (button_presses == 1) {
333             // Start Timer
334             timer_started = 1;
335             HAL_TIM_Base_Start_IT(&htim16);
336             forces_index = 0;
337         }
338     }
339     HAL_UART_Receive_IT(&huart2, &rx_data, 1);
340 }
341
342 /* USER CODE END 4 */
```

Fig. 70. HAL UART Rx Callback function