NAIT
Edmonton, Alberta

Arven

Personal Medication Delivery Robot

As a submission to

Mr. Kelly Shepherd, Instructor

English and Communications Department

&

Mr. Kevin Moore, Instructor

Computer Engineering Department

Submitted by

Kia Skretteberg, Student

Samantha Suitor, Student

&

Nubal Manhas, Student

CMPE2965

Computer Engineering Technology

April 27, 2023

NAIT
11762 – 106 Street NW
Edmonton, AB T5G 2R1

April 27, 2023

Mr. Kelly Shepherd, Mr. Kevin Moore
Instructors
NAIT
11762 – 106 Street NW
Edmonton, AB T5G 2R1

Dear Mr. Shepherd and Mr. Moore,

As per the requirements of CMPE2965, we are submitting the report titled *Arven, Personal Medication Delivery Robot* for evaluation. This report details the overall design and implementation of the autonomous robot project. Further outlined are the successes and roadblocks encountered throughout the course of the project. The report details the electronics components used and the various levels of programming required to get the robot functioning.

We would like to take the opportunity to thank our instructors who have helped us throughout our time at NAIT; their patience, and expertise served as huge help not only on this project but throughout our entire education. Learning has been a journey thanks to all of their hard work and dedication.

Sincerely,

Ms. Kia Skretteberg, Ms. Samantha Suitor, Mr. Nubal Manhas

CNT Students

Table of Contents

List of Figures and Tables

Figures

Tables

Statement of Contribution

This document was co-authored by Kia Skretteberg, Samantha Suitor, and Nubal Manhas. The list of contributions in the table below were reviewed for accuracy by all participants, and we view the division of effort to be approximately equivalent.

Section	**Author**	**Reviewer**
Title Page, Table of Contents, Cover Letter, List of Figures and Tables, General Formatting	Kia Skretteberg	Samantha Suitor
Abstract	Samantha Suitor	Kia Skretteberg
Introduction	Kia Skretteberg	Nubal Manhas
Overview	Samantha Suitor	Kia Skretteberg
Mechanical Design	Samantha Suitor	Kia Skretteberg
Electronics Design	Kia Skretteberg	Nubal Manhas & Samantha Suitor
Web Interface Design	Kia Skretteberg	Samantha Suitor
Database Design	Samantha Suitor	Nubal Manhas
Robot State Design	Kia Skretteberg	Nubal Manhas
Web Interface Implementation	Samantha Suitor	Nubal Manhas
Web API implementation	Kia Skretteberg	Samantha Suitor
Embedded Software Implementation	Nubal Manhas	Kia Skretteberg
Project Results	Nubal Manhas	Kia Skretteberg
Conclusion	Samantha Suitor	Nubal Manhas
References	Samantha Suitor	Kia Skretteberg
Appendices	Kia Skretteberg	Samantha Suitor

Abstract

Arven is an autonomous delivery robot that can carry a small payload within a dynamic environment while having a user interface controlled by a web application.

The robot was constructed using independent DC motors to drive the robot, ultrasonic sensors, IR sensors and bump sensors for obstacle detection, and UWB modules to navigate between two points. A Raspberry Pi Pico W acts as the logic controller, with an ATmega328P acting as an interface between the sensors and the Raspberry Pi Pico W. A custom web API was made to allow the robot to interface with the web application, which is hosted on a cloud server. The web app also allowed the user to alter their data with the database, also securely stored in the cloud and only accessible via an SSH tunnel, and allow access to the controls of the robot.

During the project there were some hardware issues with components getting damaged or having confusing or missing documentation making them impossible to work with. Throughout the project all aspects worked in isolation but by the end there were issues getting them all to work together. The most successful version was only missing the IR sensors and one of the motors could not respond to direction changes (forward and reverse). The portions of the web application required to control the robot were accomplished successfully though additional features had to be dropped.

1.0 Introduction

There are some people who are in need of a support animal who brings them medication, for various reasons. However, there are some people who are unable to have a support animal due to financial reasons, health reasons, or otherwise. The purpose of this project is to make a non-living replacement for this type of support animal in the form of a robot.

The robot will not be able to retrieve medication, it must be placed on the robot ahead of time. It will also not be able to dispense medication; the user will be responsible for taking their medication. Medication compliance will be tracked based on predefined user values, not accurate measurements. The robot's delivery zone will be limited to a single floor of a building and will not be able to open doors. A web application will be provided to manage medication and view a brief overview of the robot's operating information. The web application will be limited to a single device per user. The robot will only be able to deliver medication to a single tracking device.

The purpose of this report is to outline the experiences and decisions made during development of Arven, a personal medication delivery robot. It will discuss the design of the robot and web app, the implementation of the robot and web app, and the results of testing the robot and web app. The report will wrap up with conclusions based on the results of the testing and overall implementation and design process.

2.0 Overview

The project requires two main parts to allow proper use; the robot and a web app that acts as a user interface for the robot. While both sides are separate, they must correctly work in tandem for success of the project.

When it is time for a delivery of a medication, the robot will leave the home dock and start navigating to the user. It will use its sensors to detect then avoid any obstacles as it is navigating, automatically adjusting its path as necessary. Once the Robot is at the user and the user has taken the medication, the robot will log a successful delivery to the database through the custom web API, updating the necessary values. It will then return to its home dock, returning to awaiting state. On the chance of the robot getting stuck, the user will need to return the robot to the home dock and use the web application to reset it back to the waiting state.

The web application acts as a user interface for the robot, using its connection to the database to be able to add, alter, or delete any of their data almost instantly. This includes the medications and schedules, which will automatically be updated for the robot. The user can also request a one time delivery and the robot will start locating the user then travelling towards them, though delivery can take up to 15 minutes if there are many obstacles. Another thing the user is able to achieve is to manually log a dose, which will also instantly be updated in the database.

Both these aspects work together to bring a fully functioning autonomous robot to life and serve its purpose of safely delivering medication to its user.

3.1 Mechanical Design

3.1.1 Chassis Design

The main consideration for the project was navigation within a constantly changing space such as a home. Given that the main purpose is to deliver medication, doing so in a safe and secure manner was also a consideration. The other requirement would be simplicity due to the time constraints of the project being four months. In order to select components and program the robot, the team would first need to know what type of robot they would be working with.

Designing the chassis for the robot was a long process, with close to a month dedicated just to researching ideal designs and their mechanics in depth. With the requirements of navigating a changing environment, delivering a payload securely, and doing so in the simplest manner, two main chassis designs were presented:a simple box chassis with three wheels and a Rocker-Bogie style chassis with six wheels.

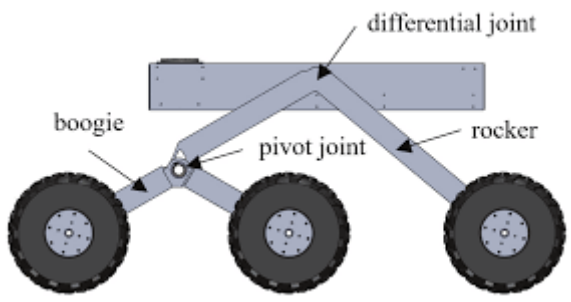
The team's reasons for being interested in a box chassis despite its generally simple design was for a few reasons. The first being that no matter the material, a simple box shape would be much easier to make in comparison to a more complex design, leaving fewer chances for mechanical malfunctions that would delay development. Due to its smaller size, it would be easier to transport between locations. Additionally, if it were to be damaged in any way, the basic shape would make repairs much easier due to fewer parts and easier access to those parts. For these reasons, the initial choice was made to make this design the testing grounds, to allow development of the hardware and software without the time delays it would take to create a fully operational rover. This would maximize the team's efficiency of development in the hopes that the project could be completed faster using this preliminary design as a stepping stone.

If time were to allow, the ideal design chosen for Arven was a Rocker-Bogie style chassis, as used in the Mars Curiosity Rover. Its dynamic design could effectively achieve the scope of the project while remaining simple enough to understand and complete. The general shape of this design supports "equal pressure on all six wheels" (Verma et al., 2017), allowing the robot to traverse over softer terrain, like carpet, as "excessive ground pressure can result in the vehicle

sinking into the driving surface" (Verma et al., 2017). In the case of a harder floor, like linoleum, "all six wheels will nominally remain in contact with the surface and under load, helping to propel the vehicle over the terrain" (Verma et al., 2017). This is also aided by the fact that each wheel is run by an independent motor and there are no axels connecting the left and right motors, allowing each side to go over different obstacles without getting stuck. In the event of stairs, this design has the ability to go over vertical obstacles that are "up to twice the wheel's diameter in size" (Verma et al., 2017), which would further open up the home for delivery with a reduced risk of becoming stuck during navigation

As mentioned earlier, both the left and right sides of the robot operate independently. This allows a freedom of movement that gives the robot a way to "maintain the average pitch angle of the body"(Verma et al., 2017) which allows the robot to "reduces the motion of the main… vehicle body by half compared to other suspension systems" (Verma et al., 2017). This keeps the medication compartment stable as it moves, preventing spills of its payload, which can have fatal consequences in an environment that could be accessed by children and pets.

Figure 1: Basic Design of a Rocker Bogie Chassis.

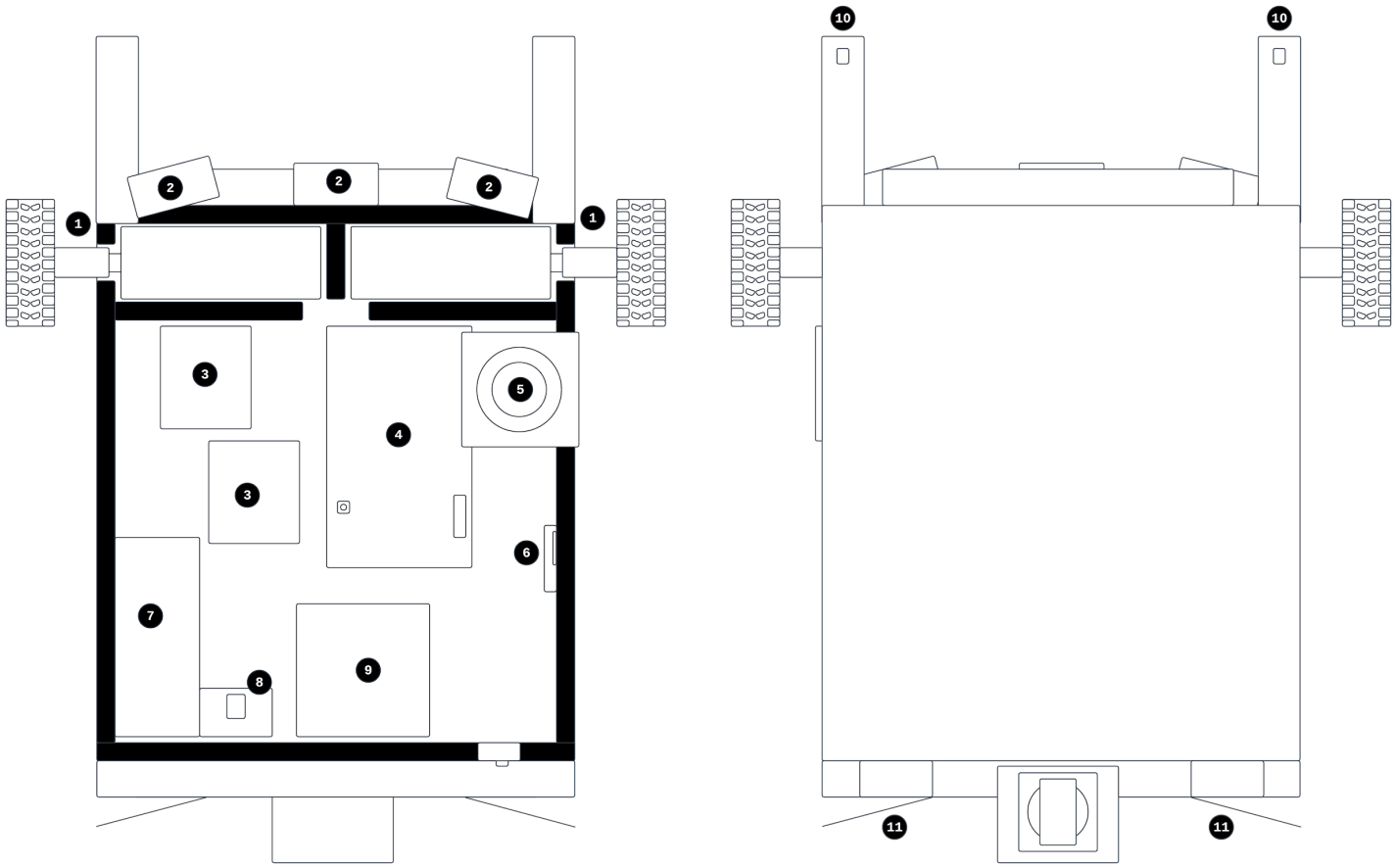


From Tomaszuk, P., Łukowska, A., Rećko, M., Dzierżek, K., & Straszyński, P. (2019, June 20). Active wheel speed control to avoid lifting the swingarms in rocker-bogie suspension. IEEExplore.

Unfortunately, due to the complexity of the project being underestimated, it was determined that a more convoluted design would not be necessary to meet the adjusted goals as the extra detail would cost more time which would be better spent on other aspects, so the preliminary design was promoted to the official one, keeping the design simple but effective for the new set of goals.

The final design of the robot can be seen in Figure 2 below. The figure shows the top of the robot on the left and the bottom of the robot on the right.

Figure 2: Line Drawing of Robot (Top [Left], Bottom [Right])



1. Motors and Wheels
2. Ultrasonic Sensors (HC-SR04)
3. Motor Controllers (DRI0002)
4. PCB with Raspberry Pi Pico W and ATmega328P
5. Medication Holder
6. UWB Module (DWM1001-DEV)
7. 12V Lead-Acid Battery
8. Power Jack (PJ-096H)
9. 12V Lead-Acid Battery Charge Module (DFR0580)
10. IR Sensor Modules (SEN0427)
11. Bump Sensors (Snap Action Switches)

Each of the parts contained within the robot will be discussed in more detail later in the report but the placement and positioning is shown in Figure 2 above. The two motorized wheels would be placed at the front of the robot for quick response. The third wheel is a caster wheel placed at the back for additional stability.

Since the robot would not be able to do stairs, avoiding stairs became a necessity. This was solved by placing two IR distance sensors facing downward at the front of the robot. These sensors were placed on arms that extend out from the robot to ensure that the drop would be detected with enough time for the robot to stop. The IR sensors are discussed further in depth in Section 3.2.1.

In order to avoid hitting obstacles and be able to navigate around them appropriately, three ultrasonic sensors were placed on the front of the robot as well. These sensors were placed in a three sensor array covering a full angle of 45 degrees to appropriately detect obstacles in front, to the right, and to the left of the robot. The ultrasonic sensors are discussed further in depth in Section 3.2.1.

Given that the robot would not back up frequently, the rear obstacle detection was to be handled by a couple snap action switches (McComb, 2018, p.21) which was for simplicity.

The main body of the robot was left open to house the rest of the necessary components such as the battery, battery charge module, UWB position module, motor controllers, and custom PCB.

Though less than ideal, the medication compartment was designed with simplicity in mind due to time constraints. This meant that the compartment was mounted on a pole sticking up just high enough to be above all of the hardware. On top of the pole was a multi-tiered platform designed to fit the exact dimensions of the most common prescription medication bottles. In the bottom of the platform was the sensor used for measuring the presence of the bottle to detect when medication was delivered.

3.1.2 Materials

The material for a chassis is a vital component and can affect all aspects of the robot. If a chassis is not properly suited to the task at hand, consequences could lead to damage to the robot and its components. When assessing the requirements and budget, three options presented themselves, wood, steel, and PVC.

The first material that was initially considered was wood. It is a widely available material with tools that require little training and can be found in family homes. It is also not conductive to electricity as "it lacks free electrons to pass an electric current" (Savitri, n.d.). This makes it a great insulator and prevents any electrical interference from any other parts or the outside. The downside of wood is that it offers no protection against any possible flames sparked by the circuitry in error. For an electrical spark, it can produce up to 1316 °C (Cafe, 2007) while wood, based on the type, can get ignited at 190 - 260 °C (Cafe, 2007).

The second material that was presented was steel. It would be strong and sturdy, able to take any abuse from the testing process and is non-combustible (*Steel Is a Non-Combustible Material and Doesn't Burn*, n.d.). Also, with a melting point of 1100 - 1600 °C (Cafe, 2007), a spark likely would only melt a small portion of it. Despite this, the material was rejected due to numerous cons.

The first con was that steel is a heavy material. Due to the small motors and estimated component weight, there was a high likelihood that a steel chassis would simply make the robot too heavy for the chosen motors, which we were unable to replace due to time and cost. The second reason is that it would require training to safely use steel cutting equipment which would waste time that could go elsewhere for the project. The third reason is that metal is known to impact wireless signals which the robot relied heavily on and could cause lots of issues. Lastly, it can conduct electricity, so there would need to be the extra step of insulating components to avoid this issue.

The final option for a chassis material was Polyvinyl chloride(PVC). This suggestion had many things that would make it ideal for a robotic chassis like "resistance to flame, electrical, water, chemical, and abrasion" (*PVC (Polyvinyl Chloride, Type I or Type II)*, n.d.) as well as being both lightweight and stable. These qualities would make it the ideal for the project but upon investigation, there were a few obstacles that prevented this. The first being that upon further research most available PVC sheets were corrugated which were not stable enough upon testing for the use of a robot. Another factor is there would be additional skills including plastic soldering

that would require time spent to learn as well as extraneous equipment for creation purposes, again, time that could be better spent elsewhere in development.

Once sufficient research was conducted on all the materials and all situations considered, wood was chosen as the main material for the chassis. The main reasons for this being the material is much easier to get, easy access to tools, and the much cheaper price. This coupled with the fact it is easier to work with, with no soldering required and a decent insulator, made it a great choice for the needs of the project.

3.1.3 Home Dock

The ideal for the home dock was an unmoving anchor point that would act as a "home" for the robot and provide it with necessary support to be fully autonomous in its functions and lessen the interactions the user would need to provide to keep the robot functional. As this development would need a nearly fully functional robot for accurate sizing for housing and navigation errors, this was left until near the end of the development process.

The first need that had been identified for the dock was an anchor point for the robotic navigation. This point would be unmoving and could easily be used as a starting reference point for the robot, allowing a distinct pathing between the dock and the user as well as assist in triangulating the position of the user.

The second function that it was required to have would be a shelter for the robot when not in use. The expected environment for this robot would be a user home, which can be a chaotic environment for a robot to exist in and where the delicate parts of a robot could break if not properly cared for. With a dock, this problematic scenario would be lessened, as the dock would provide a safe housing, protecting any protruding parts of the robot in its enclosed space, as well as being robust enough to withstand any force that a human could inflict in an accidental scenario of carelessness or poor placement of the home dock.

A third function that was designated as an ideal feature was an induction charging station within the dock so the user would not have to ensure the robot was charged enough to be operational. If it were hands off, there would be less user error to affect day to day operations and

the robot was likely to be fully charged at all times, reducing the chances of it running out of power on a delivery.

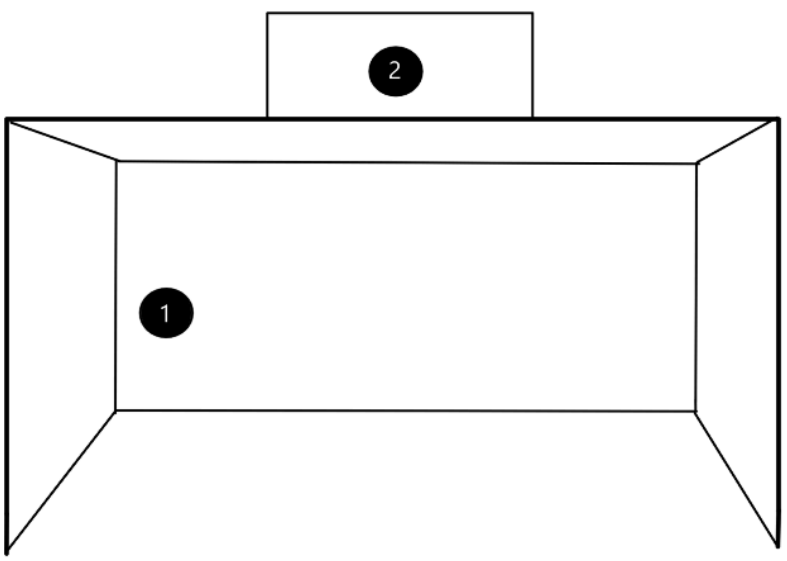
With these in mind, and once the dimensions of the robot were finalized to guide the size, the dock was to be built. At the point of starting, it was decided that induction charging would not be implemented for a variety of reasons. One reason was that navigation would need to be extremely precise to ensure there was appropriate alignment for the charging coils to function properly. Second, the induction charging circuit would have taken additional time to research and build which was not within the budget. Due to excluding the need for induction charging, the shape of the main housing could be a simple box shape as it just needed to provide a stable secure place for the robot to sit when idle.

Another choice for the dock that was made was for it to be made of wood. This material would be robust enough to withstand an impact from the robot without relieving damage and would not be hard enough to damage the robot itself in the likelihood of a navigation error during testing.

A final consideration was the navigation modules, which will be discussed later. They required a certain orientation to work correctly and with a signal that could be possibly blocked by the walls of the dock, the choice was made to have a containment unit for the module at the top of the dock, sitting at the center of the entrance for easier navigation for the robot itself.

Considering the circumstances, the design of the dock was decided to be a simple box shape, with containment on top. The main box would be made of a thicker plywood for its robust nature, added with the fact it was cheap and easy to supply. A top container for the navigation module would be made of a thinner wood with no top so the signal would not be impeded as much. This box would also have a plywood base to add extra support as well as give it a firm connection to the housing. The general design of the robot dock can be seen in Figure 3 below.

Figure 3: Line Drawing of Home Dock



1. Robot Enclosure
2. UWB Anchor Container

3.2 Electronics Design

3.2.1 Component Selection

Component selection began simply, but as the project progressed reactionary selection became necessary. At the start, research was conducted to determine which motors, wheels, and sensors would be needed to make the robot go. As time went on, difficulties with the original navigation modules chosen caused a pivot to different ones.

The first step when choosing components was to do some research, both theoretical and experimental. For the motors and wheels it was a bit of a combined research task as the size of the wheels impacted the power needed from the motors. The motors were also responsible for defining the maximum size of the robot. The calculations to determine minimum and maximum RPM were completed using the formula:

$$RPM = \frac{60 \bullet speed}{\pi \bullet wheel\ diameter}$$

(Joseph, 2015, Ch. 2, Sec. 2)

In this calculation, spare wheels that were pre-purchased were used. These wheels measured a diameter of 65 mm and an estimated speed based on a sample speed of 20 cm per second of a robot vacuum (Vorwerk, 2018). The same calculation was performed using values retrieved from experimental testing of walking speeds. With this, a speed range of 20 cm to 30 cm per second

was deemed reasonable. Using the speed and wheel diameter, an ideal RPM of 60 to 88 was deemed reasonable.

Experimental tests were conducted to determine the coefficient of friction for the wheels to be between 0.3 and 0.4 using the formula for calculating the force of friction:

$\mu_s = \frac{F_f}{F_N}$ where $F_f = F_g \bullet cos\theta$ and $F_N = F_g \bullet sin\theta$ which simplifies to $\mu_s = tan\theta$

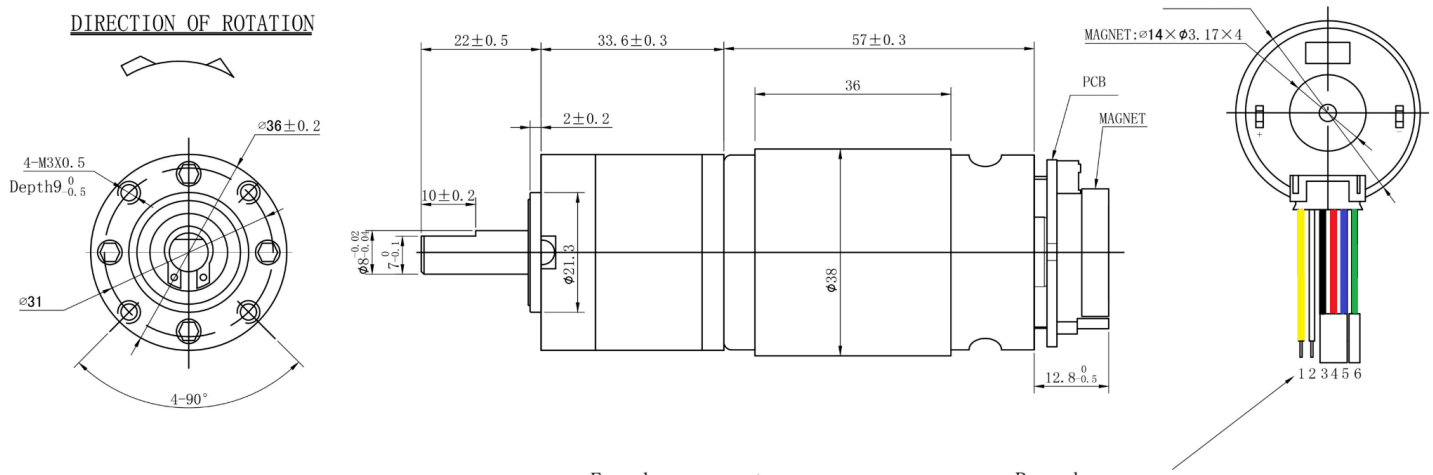
First the robot's weight was measured. Then the robot was placed on an inclined plane with a known angle. Using the angle (θ), a coefficient of friction (μ_s) was calculated and recorded for the specified weight/surface type. The calculated values varied depending on how rough the surface was and how dirty the wheels were. The coefficient of friction was then used in a formula to determine the minimum torque required:

$$T = \mu \bullet N \bullet r$$

(Joseph, 2015, , Ch. 2, Sec. 2)

This was calculated to be about 0.12 Nm for a 20 N mass with two powered wheels. The challenge was then to find a motor that could provide 0.12 Nm of torque while going 88 RPM at least. The challenge was an interesting one and was originally based on 30 N mass with six powered wheels but was capable of working with a 30 N mass with two powered wheels. Most motors that could achieve this were fairly expensive but a deal on some motors that could provide more than enough torque/speed was found from Robot Shop, made by Lynxmotion (Figure 4).

Figure 4: Motor/Encoder Datasheet



SPECIFICATIONS
For reference only

MODEL	36GP-555-27-EN
VOLTAGE	12V
AT NO LOAD	
SPEED	170rpm±10%
CURRENT	≤0.3A
AT RATED TORQUE	
SPEED	140rpm±10%
CURRENT	≤1.45A
TORQUE	5.5Kg.cm
PERMIT TORQUE	25kgcm
CURRENT	≤6.5A
TOTAL GEAR RATIO	1:27
STALL ALLOWED OR NOT	NO

The final specifications depend on the test result of samples.

Encoder parameter
1. Voltage: 5V
2. Signals: Square wave AB 90°
3. Temperature:-20℃-80℃
4. Pulse: 12PPR
5. Frequency: 800KHz

Remarks
Wire length:300±20mm
Motor Wire Gauge: 26#
Hall Wire Gauge: 26#
Terminal model: PH2.54-3P+1P

Wiring sequence
1. Yellow Motor +
2. White Motor -
3. Black Hall Sensor GND
4. Red Hall Sensor VCC
5. Blue Hall Sensor A Vout
6. Green Hall Sensor B Vout

(Robotshop, downloaded as part of order, Retrieved January 26, 2023)

One thing that was missed during the planning was the physical dimensions of the motors. Once the motors arrived, they were too long (over 100 mm) for the original concept of this project and quite heavy (2 kg each). This posed some interesting challenges to the design and almost required ordering new motors but the team found a way to make it work by designing a new chassis that could accommodate them.

After choosing the wheels and motors, choosing the sensors proved to be both easier and more complex. A variety of books were used to determine what sorts of sensors would be appropriate to use, but ultimately the ones chosen were the best options given this project's budget and time constraints were ultrasonic sensors for obstacle detection, IR sensors for drop detection, and some simple snap action switches for bump detection when reversing.

The team had obtained one ultrasonic sensor from a simple two-wheeled robot prior to the project start that was used for some initial testing. Nearly every book that was read for the research used ultrasonic sensors, and for good reason, as they are fairly easy to work with. In

order to use the sensors, set the trigger pin high for 10 µs, then low, and wait for the echo pin to go low in response (ELEC Freaks, n.d.). One of the major questions to be asked about the sensors, is that some of the books indicated you could not use multiple ultrasonic sensors together, which was exactly what was required, but most books implemented them in arrays so it turned out to not be a concern if done properly. The arrangement was based on a three sensor array (Cicolani, 2021, Ch. 7) that covers a total of 45 degrees (15 degrees each) in front of the robot. Despite the warnings about using them together, this three sensor array seemed to have met the requirements exactly.

While looking into the ultrasonic sensors, the same book had shown using IR sensors, specifically line following sensors, pointed at the ground (Cicolani, 2021, Ch. 8). While this robot specifically used the sensors for line following, one book that was used for research discussed detecting stairs to prevent the robot from falling down stairs (Govers, 2018, Avoiding Stairs). It was decided to combine the two ideas and use IR sensors to prevent the robot from falling down stairs. However, this meant that an IR sensor needed to be selected that could measure very short distances, less than 5 cm, while still being able to measure large enough distances to detect a stair specifically. The reason for this was that initially the Robot was eventually going to get to a point of being able to climb stairs so it needed to distinguish between a height that was safe versus one that wasn't. In the end, all that was required was detecting the difference between was 1 cm or less, and anything larger than. In the research, an IR sensor was found that could measure up to 200 mm at most (Adafruit Industries, n.d.), though the various modules that used the sensor disagreed and the specific module that was obtained only indicates 100 mm (DFRobot, 2021a). For the purposes of this project, this proved to be sufficient.

Determining what battery to use was a little out of the team's control as it was based on safety requirements and availability. The initial plan was to use Lithium-Ion batteries due to their well known capacity vs size and weight ratio. They are typically known to be small, light weight, but have a large capacity. Lithium-Polymer batteries were also considered but were quickly discarded due to the high risk of explosion if mishandled.

The limiting factor that ended up taking the decision out of the team's hands was that the project required 12 V to power the motors which also had high current demands. The base current required to run the full project was calculated around 200 - 300 mA but the inrush current for the motors was around 1.5 A per motor. These factors combined meant that the team needed a decent sized battery to be able to hold enough charge to run the robot for the required minimum duration of 30 minutes (15 minutes to get to the user, 15 minutes to get home).

When the team went to purchase the batteries, hoping to be able to make a battery pack out of 3.7 V Lithium-Ion cells, the salesperson indicated they would need to be cold welded to produce a battery pack. The business was unwilling to produce a battery pack and this was outside the realm of the team's expertise so the idea was quickly abandoned. On the salesperson's recommendation, the team switched to a single rechargeable 12 V Lead-Acid battery (Power Kingdom, n.d.) with a 1200 mAh capacity.

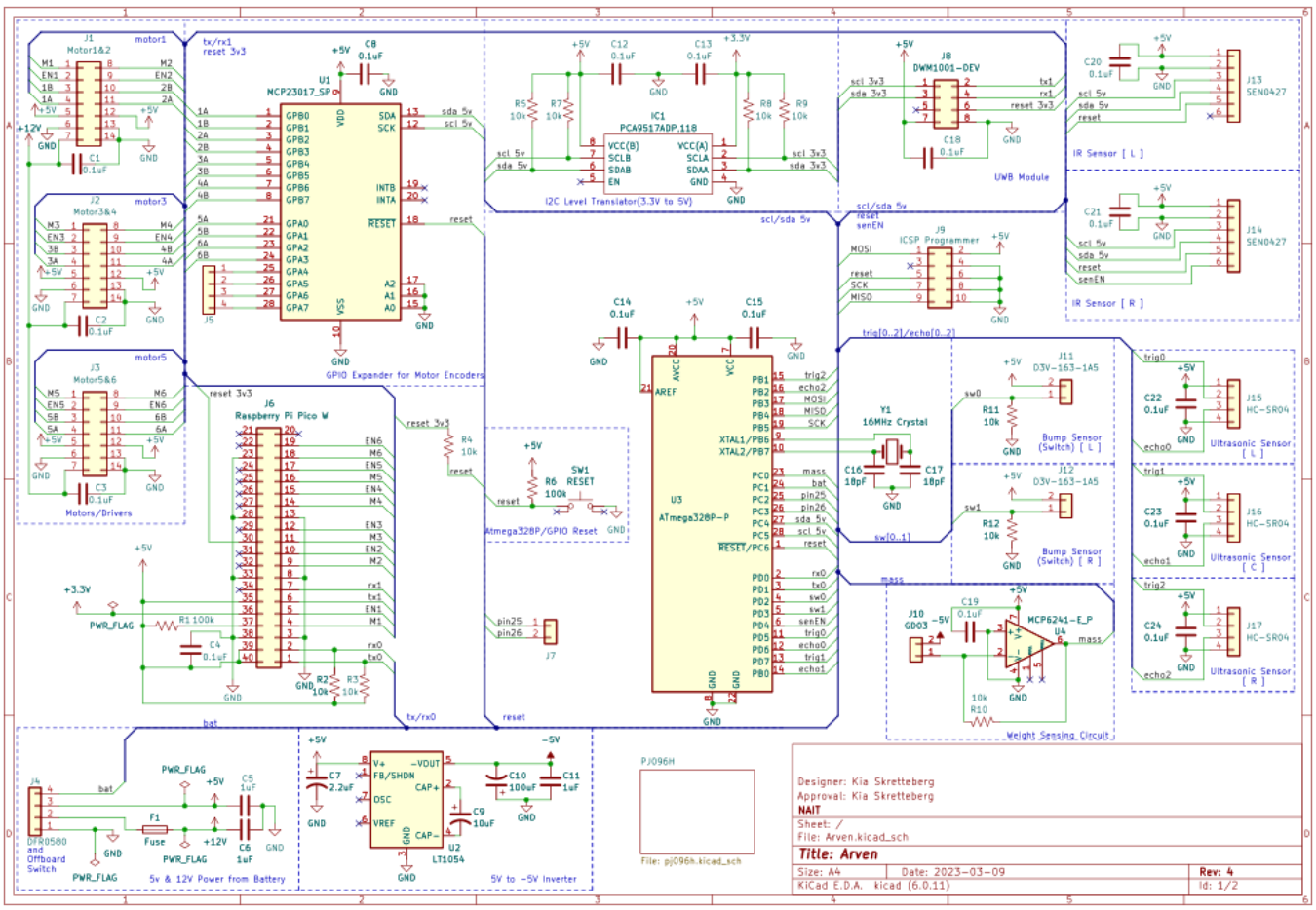
Prior to purchasing the battery, the team had been considering various charging modules for Lithium-Ion batteries and needed to do a quick pivot to support the new battery. The battery charging module needed to satisfy the following requirements. The charging module needed to be suited for the 12 V Lead-Acid battery chemistry, it needed to be easy to use, and it needed to have some way of charging it via a wall outlet. The module that was chosen was designed to work with solar power for charging the battery, but luckily it also had a mode for using a laptop charger with a voltage range of 15 - 24 V (DFRobot, n.d.) and a power requirement of 65 W. This worked out well as the team had an old laptop charger that met these exact requirements and would just need to purchase a separate power jack to connect the charger to the module. The added benefit the module provided was the ability to output both 5 V and 12 V with different current limits thus ensuring the separation between the power for the motors and the power for the microcontrollers.

3.2.2 Circuit Design

The circuit design was a complicated process that involved a lot of revisions as errors were noticed. The design (Figure 5, see **Appendix B** for larger version) was reviewed numerous times during the process by Ross Taylor, which informed the decision to use buses for better

organization and the placement of numerous capacitors at all power inputs to the various chips and modules.

Figure 5: Arven PCB Schematic



In addition to external review, the circuit was informed by numerous datasheets along the way. For J10, the GD03 module, the datasheet (Uneo, 2014) suggested an op amp, a negative source voltage, and a feedback resistor, to obtain linear output. For U2, the voltage inverter that generated the negative voltage for J10, the datasheet (Texas Instruments, 2015) detailed the various capacitors that were required, though it was a bit unclear for C7 what value it actually needed to be as the sheet suggested 2 µF which did not seem to be obtainable so the decision was made to use 2.2 µF as it was close enough. Most other datasheet requirements were handled by the modules themselves and any additional circuitry was added based on recommendations from various instructors, including Kevin Moore.

The original design planned for the potential to upgrade the robot to have six wheels and thus motor pin headers were added to accommodate six motors and encoders. This quantity of motors and encoders far exceeded the number of GPIO pins available on the ATmega328P (Atmel,

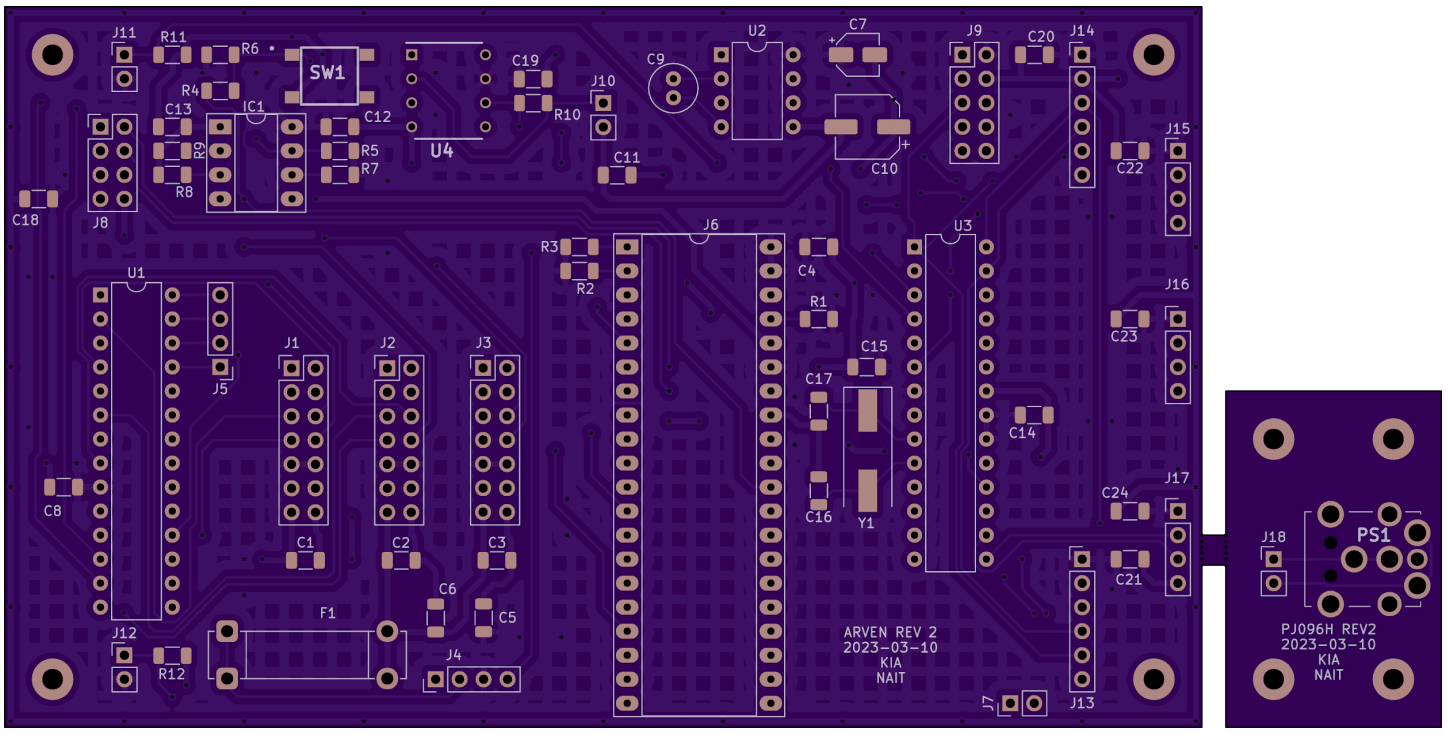
2009) and thus a sixteen pin GPIO expander was added to accommodate the encoders. The ATmega328P also did not have enough PWM enabled pins to be able to control all six motors so the PWM pins on the Raspberry Pi Pico W were used to control the motor speed.

The datasheet for the DFR0580 module (DFRobot, 2021a) indicated that it could be powered by an 18.5 V laptop charger which would need to be connected via a barrel jack. The jack that was chosen was PJ-096H (CUI Devices, 2023), which was best suited to be mounted on a separate PCB and thus required an additional circuit schematic, though a very simple one at that (see **Appendix B**).

3.2.3 PCB Design

When designing the PCB (Figure 6, see **Appendix C** for a larger version) from the circuit, the main considerations were the physical size of the board and the physical placement of the headers in relation to where the off board modules would be placed on the robot. The goal was to keep the board as small as possible to reduce costs while placing the components in appropriate places for the least wire crossovers. As can be seen in Figure 6, the ultrasonic sensors (J15, J16, and J17) and the IR sensors (J13 and J14) were placed on the right side of the board, which was intended to be the front of the robot.

Figure 6: Arven PCB and PJ-096H Daughter Board



The bump sensors (J11 and J12), were placed near the left side of the board as that was intended to be the back of the robot. The original design plans allowed for six motors with two at the front, two in the middle, and two at the back, so the motor pin headers (J1, J2, and J3) were placed roughly in the middle to accommodate the dispersed placement of the motors. The complexity of how these traces needed to be routed required heavily using the underside of the board (see **Appendix C**) in order to make all of the connections work.

After receiving the PCB and beginning assembly, a few flaws in the design were found. First, J13 (the left IR sensor), and J14 (the right IR sensor) were swapped (Figures 5 and 6); J13 should have been at the top and J14 at the bottom. Second, J14 has two pins senEN and reset (Figure 5) that should both be connected to the same external pin on the SEN0427 module but instead it was put in as a separate header pin. Third, there were a couple issues with J6 (the Raspberry Pi Pico DIP): the footprint was too small, and the tx0/rx0 pins were swapped. Fourth, the footprint for the PJ-096H connector was oriented backwards, putting the pin header in the way of the connector. Lastly, the orientation of the Raspberry Pi Pico resulted in the micro USB connector being placed facing inwards to the PCB which was less than ideal and resulted in the need to mount C9 on the bottom of the board, instead of the top to be able to use the port. See **Appendix C** for a 3D rendering of what the board was intended to look like when assembled.

3.3 Software Design

For the web application a few aspects needed a bit of design planning before we could build them. For the robot, there wasn't much design needed as a lot of it was experimentation of how to work with sensors. Late in the project the team realized a bit of design was indeed needed for the robot in order to keep the operational aspects organized. For the web application, the team did a set of UI mockups to guide the operation of the website which then informed the design of the database.

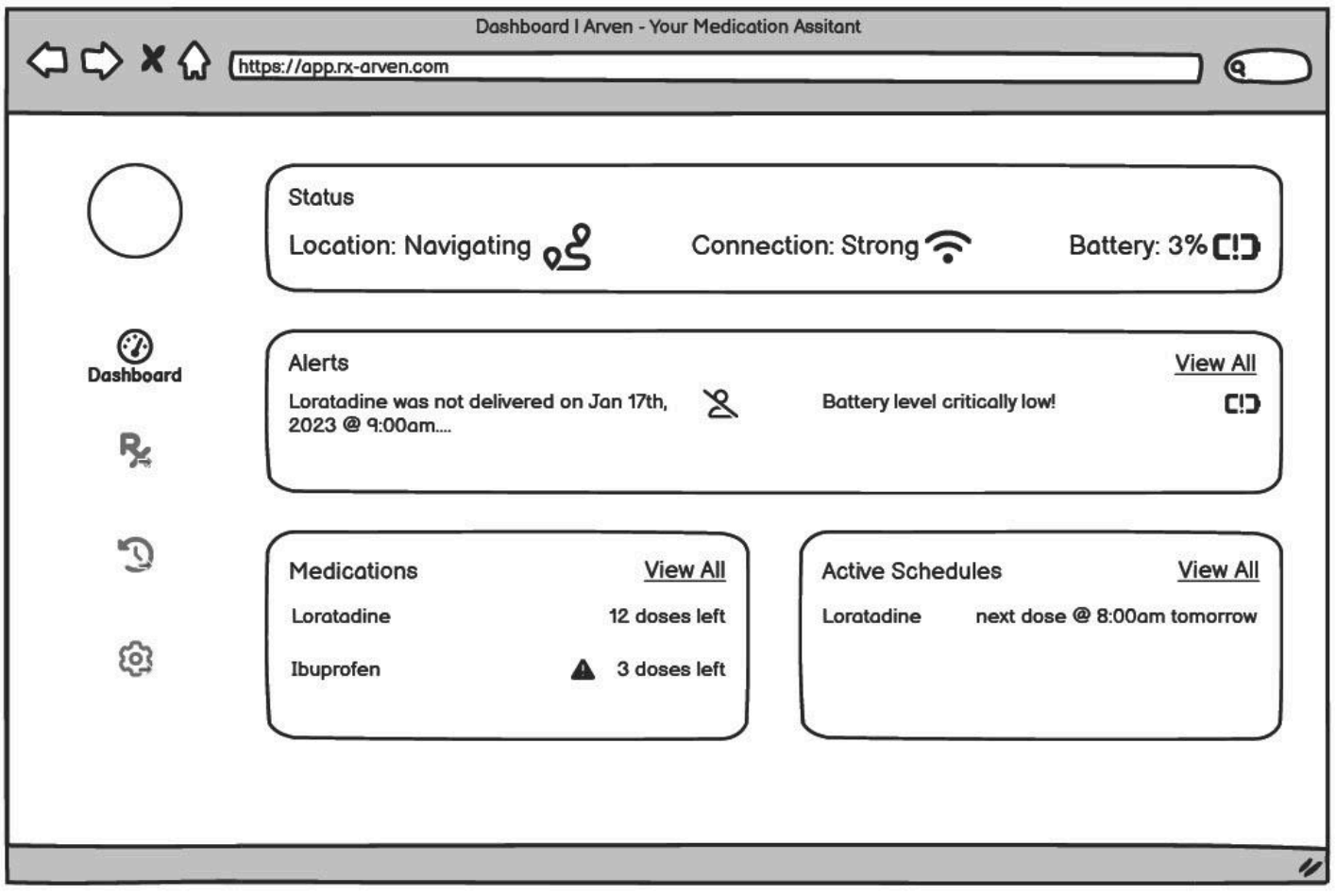
3.3.1 Web Interface Design

One of the first tasks that was accomplished was designing the web interface due to wanting to get it out of the way before things got more complicated. The team chose to use Balsamiq Cloud (Balsamiq Studios, n.d.) due to prior experience using the application for designing

web applications previously. The interface allowed for a quick and easy way to build a semi-functional prototype.

The first page designed was the dashboard as it was the most prominent page a user would visit on a regular basis and was the largest indicator of what functionality the team wanted the web application to provide. As can be seen in Figure 7, the web app was intended from day one to have an indication of where the robot currently is, what its connection status is, and what its battery level is.

Figure 7: Web Application UI Mockup of Dashboard



Additionally, the team knew the important pieces included "medications and schedules as well as displaying information about them. The mockups for these pages can be seen in **Appendix A**. Typical dashboards also include notifications of some variety so it seemed natural that a robot might have some notifications to display so the Alerts section was added. A full page for viewing the full list of notifications was also needed for historical purposes. The history page can also be seen in **Appendix A**.

In order for the web application to function appropriately the team determined it made sense to be user driven, requiring a user to create an account and login. In order to support this functionality, a login (see **Appendix A**) and account creation page were made. The main difference with the account system is the need to tie an account to a device so rather than simply a basic account creation page, it was treated as a device setup page (see **Appendix A**) that involved creating an account.

While it made sense to allow editing some of the information that was input during account creation, the focus of the application is to be more device centric so the last page within the application is the configuration page (see **Appendix A**). This page allows the user to edit some account information but also control the robot's state such as rebooting the device.

For the overall design of the website, the team took inspiration from the Google Material case study Rally (Google, n.d.) and just generally attempted to follow Material design principles due to being familiar with them. The biggest driving factor in this implementation was building the website on a layout structured around the same grid layout that Rally uses, which changes depending on the device the web app is being viewed on. Designing this way made the mobile design and build fairly simple as it was built in as a requirement from the ground up.

Early on the team decided on cohesive branding, in the form of colours and font, that would be used across the entire project, from the reports and documents, to the website. For this reason, a lot of thought went into the colours chosen. The intended user base of the website includes people of all walks of life with a larger focus on those who have some variety of disability. This meant that it was extremely important to the team that the colours meet all accessibility guidelines possible so as to ensure everyone would be able to use the web app without issues. Unfortunately, this posed a lot of challenges as choosing a colour palette that met all of the team's requirements was not easy.

The requirements the team had were that the colours must be AAA WCAG compliant, must follow colour theory for movies, and must be non-triggering for neurodivergent individuals. When choosing a colour that was WCAG compliant, which defines how visible the colour is for people

with colour blindness and various levels of vision loss, the team used ColorSafe (Rapp & Donielle Berge, n.d.) to generate colours that would be suitable. The colours were narrowed down according to greens and oranges based on a list of colours designed for people with Autism (PPG, n.d.). The final decision for colour came from colour theory for movies in which green is considered evil so orange was landed upon instead. With the main colour chosen to be orange, an orange colour was selected from the list of colours presented from the WCAG list. This colour was then plugged into an accessible colour palette generator (Venngage Inc, n.d.) which allowed the team to generate a collection of complimentary colours that didn't violate any of the previous requirements, avoiding any shades of green, or any primary colours. This resulted in a colour palette reminiscent of multivitamins which fit nicely with the medication theme of the project.

3.3.2 Database Design

As the database was a backbone for holding and organizing the data required to run the project, it was a main priority, being completed in the early stages so the other software could have an unchanging base to be built on. While the creation went smoothly overall, there were still a decent amount of challenges upon building it.

The database was hosted on a cloud server hosted by DigitalOcean (Digital Ocean, n.d.) and utilized the Secure Socket Protocol(SSH) as a login. In a real scenario, the database would hold sensitive information regarding the users, their location and private health details, so it was used in addition to passwords to add an extra level of security for this data. For the language used for construction of the database, MySQL was chosen for its familiarity and ease of use. Overall it is a simple language and makes the usually long task of creating a database much faster than if a different language was used.

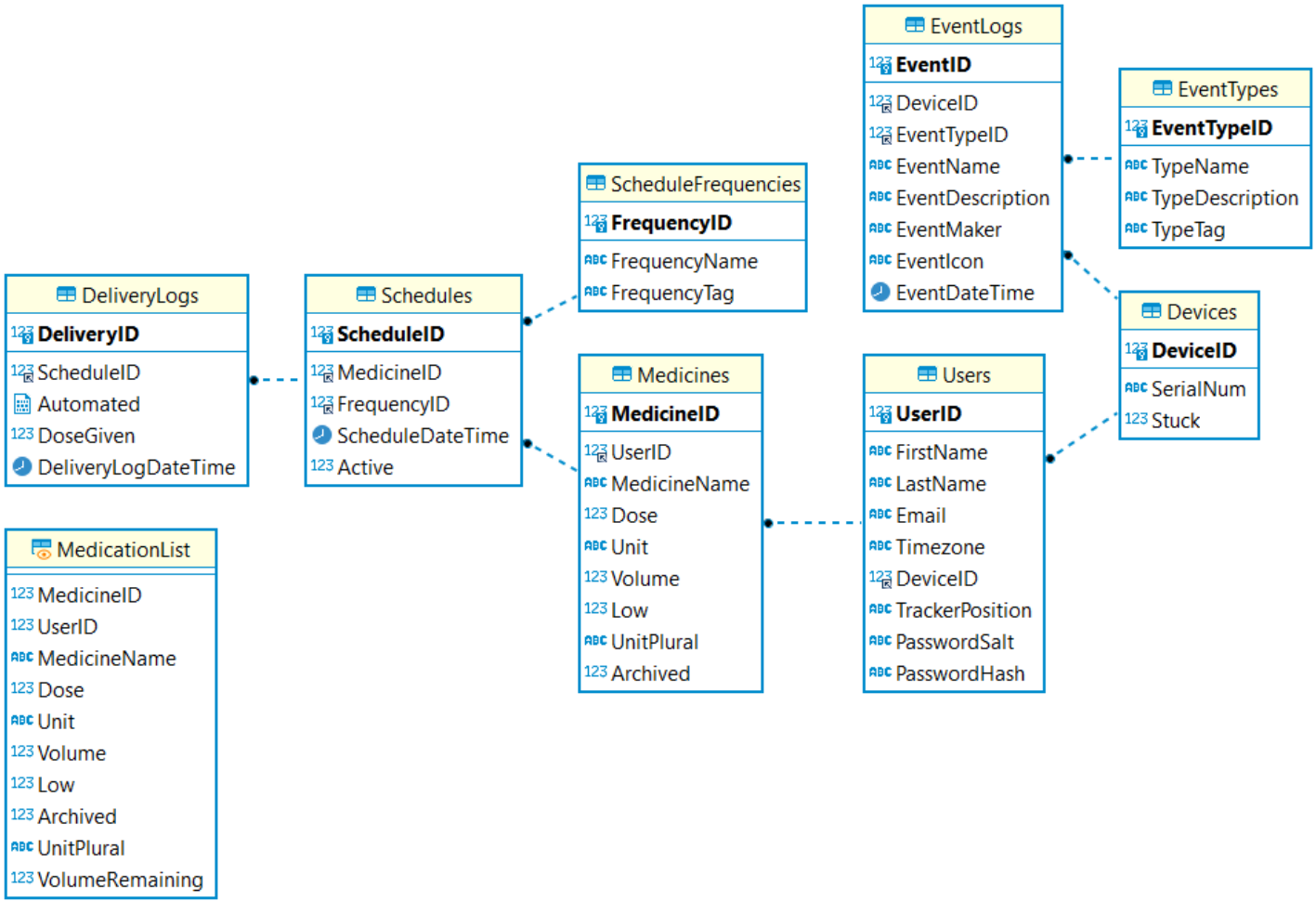
To access the database, the database management tool known as DBeaver was used, as it further streamlined creation and editing. Unfortunately due to it being free, the tradeoff was that it has unconventional controls to learn in order to make it faster to use. The layout and design of the application was not set up very well, but any issue that arose could be bypassed through manually typing in the given console. Thankfully, this did not hinder development too much as there were a good set of resources online to help.

The main issue that was involved was the normalization of the database. Throughout the project, especially in the first few weeks, the scope was changing every few days. It was quickly decided that the database would need to have a more dynamic layout to accommodate any changes further down the road. The first step to normalization was to decide the relationship between the devices and the users, as both would act as the backbone to the entire database. While the initial idea was having a composite key in a separate table, allowing multiple users to have multiple devices and vice versa, the problem of complexity immediately became a glaring issue. To combat this, the executive decision to have multiple users for only a single device was chosen with the reasoning being that a single household could have multiple people inside, but only a single robot would be ideal for navigation purposes, thus it was decided that the database and subsequent interfaces would be based off the user and not the device. This setup gave enough complexity to be interesting but not enough to hurt the development of any other parts.

There was also a second major problem that delayed development further. The robot would need to send events to the database, and have a way to store various event logs. The initial idea was to have all logs stored in a single table for ease of access. After discussion, this was changed so that the events are tracked through the device itself, rather than the tasks completed.

Since multiple renditions of the database were made to settle on the database layout, it took a couple weeks to finalize the design which is shown in Figure 8 below.

Figure 8: Database Final ER Diagram

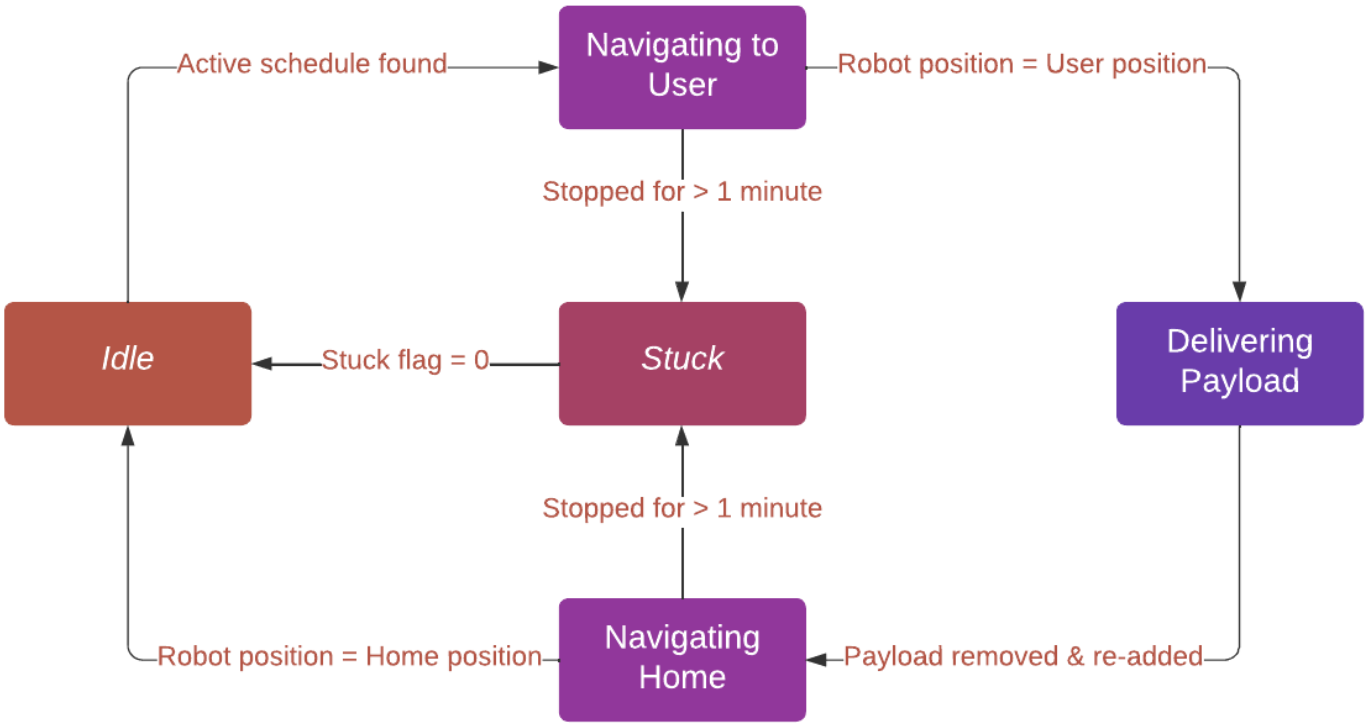


While this design is not complex enough to allow for multiple devices per user, it perfectly fits the scope of the project while having a simplistic design that has a linear progression from one table to the next. This makes calling information from the database much easier and more straightforward.

3.3.3 Robot State Design

Late in the project, the team realized that some preliminary design should have been done in the early stages, resulting in this design being one of the last things completed. It was determined that the robot would operate within a variety of states and perform different functionality within those states. These states are: Idle, Navigating to User, Delivering Payload, Navigating Home, and Stuck. The flow between these states can be seen in Figure 9.

Figure 9: Robot State Diagram



During the Idle state, the robot would check to see if there was an active schedule that needed to be acted upon. In the Navigating to User state, the robot would retrieve the user's location and navigate towards it. During the Delivering Payload state, the robot would wait for the user to remove the medication and place it back, thus indicating a successful delivery. During the Navigating Home state, the robot would navigate towards the home dock, which has a position of 0, 0, 0 in an x, y, z coordinate system. The last state is an optional error state in which the robot sits and waits for an indication that it is no longer stuck. How these states were implemented will be discussed more in the Raspberry Pi Pico (Logic) Execution section.

4.0 Implementation

4.1 Web Interface Execution

The web application was created to utilize the Model-View-Controller (MVC) pattern to give some experience in a widely used website design style. In this type of design, the view is what the user sees. It is simply a display and does not have any direct access to any sensitive data. The model controls all access to the database exclusively. The controller acts as an interface that connects both the model and the view. Most of the logic and backend work for the web application is here. It controls all calls and functions and can populate any variables needed to be accessed by the view from the model. While this seems elaborate for what was to be a simple application, it provides a great deal of organization and separation to each step of building the application, making it far easier to track where errors are coming from.

There were four languages used to build this web application: HTML, Javascript, CSS and PHP. They were all used in the standard way, except for PHP. Again, to gain experience, the PHP framework known as CodeIgniter 3 was used. While there is decent documentation of all the processes through the online user guide (*Welcome to CodeIgniter — CodeIgniter 3.1.13 Documentation*, 2022), it is mostly built using examples and little information on how things work exactly. This makes it a decent tool if the creator has a background in coding though it still makes it difficult to troubleshoot issues. The main difference between conventional PHP and CodeIgniter is that CodeIgniter is an object-oriented MVC (Model-View-Controller) framework that utilizes class hierarchy throughout.

4.1.1 Database Functions

The most important function for the website was the database calls that were controlled through the model. It provided the base for the application to work.

4.1.1.1 Get Function

The first call was the get function which took an array of data that would pass any required filters, if any, before pulling the information from the database and returning it to the user. If no filters were added, then it would return all possible data for processing manually.

Figure 10: Get Function from Users Model

```
function get($options = array(), $result = true)
{
        extract(filter_options(array('id', 'device', 'email'), $options));

        if($id) $this->db->where('UserID', $id);
        if($email) $this->db->where('Email', $email);

        if($device)
        {
               $this->db->join('Devices', 'Devices.DeviceID = Users.DeviceID');
               $this->db->where('Devices.SerialNum', $device);
        }
        $query = $this->db->get('Users');
        return $this->helper_functions->return_result($query, $result);
}
```

This function starts by filtering through the given options array for any filters that apply to the specific database table the model is connected to. If any are found, it will extract them into a variable by the same name for easy use in the function. Any information that is not applicable is discarded and not used.

From there a generic select all call is created within the system itself. For each variable gained through the filtering previously, a new clause is added automatically to the hidden query, waiting to be called. As shown in Figure 10, join is also a possible call though there is a need to manually set the connections between the databases. Once everything is filtered for, the function then calls a database get function which retrieves the data using the artificially constructed SQL call and returns it as an array of data. The model get function then calls a helper function to turn the result into easily accessible rows of data.

4.1.1.2 Save Function

The save call acts as both an add or update function for ease of use when calling it from a controller. The function determines which action to use by the presence of a unique identifier, represented in Figure 11 by the $id variable. If requested, the function will return the new set of data to use, if there is any, allowing the web application to react to the result and be updated accordingly. This information can then be used to inform the user of the status of their actions.

Figure 11: Save Function from Users Model

```
function save($data, $id = false, $return_user = false)
{
	$this->db->set($data);

	// UPDATE
	if($id !== false)
	{
		$this->db->where('UserID', $id);

		$this->db->update('Users');
	}
	//CREATE
	else
	{
		$this->db->insert('Users');
	}

	$user_id = $this->helper_functions->return_id($id);

	if(!$return_user || !$user_id)
	{
		return $user_id;
	}
	else
	{
		$options = array('id' => $user_id);

		return $this->get($options, false);
	}
}
```

4.1.1.3 Delete Functions

The delete calls work similarly to a get call, and due to the nature of deletion, must be able to cascade during deletion, thus it would require a specific ID. If the ID is not provided, then the call will not run, preventing any unintentional deletion. Once the call for deletion is made, the amount of rows altered needs to be checked to verify that the call was completed successfully and will return a bool value based on the result. In this instance specifically (Figure 12), the function checks if the user wants to simply render the medication unused or to be deleted from the database completely. If the choice is made to archive it, then it will simply call the update function and continue from there.

Figure 12: Delete Function from Medicines Model

```
function delete($options = array())
{
	extract(filter_options(array('id', 'archive'), $options));

	if(!$id) return false;

	if($archive)
	{
		$this->save(array("Archived" => 1), $id);
	}
	else
	{
		$this->db->where("MedicineID", $id);
		$this->db->delete("Medicines");
	}

	if($this->db->affected_rows() > 0)
	{
		return true;
	}

	return false;
}
```

4.1.2 Controller Functions

As stated previously, the controller acts as an interface between the view and the model. Using the index in the controller (Figure 13), specific variables within the view can be set using a set of data from the model. As the model cannot call other functions, everything that is needed in a view needs to be dealt with by the respective controller, whether by adding additional functions to the controller or storing information as variables.

Figure 13: Index Function from Medications Controller

```
public function index()
{
	$medications = $this->medicines_model->get_list(array("user_id" => $this->userID,
"include_schedules" => true));

	$this->set_view_data(array(
		'medications' => $medications
	));
}
```

The controller also controls any logic that is required for any of the pages. There is one thing to be aware of though, as "a[ny] public method in a controller is exposed as a controller

action”(Walther, 2022), meaning that any other file can call this. This is why the filtering is required in the model files.

4.2 Web API Execution

In order for the robot to communicate with the web application, and vice versa, the team decided a web API would be required. Given that the web application existed already, the web API existed on the same server and used the same codebase, just a different pathway. However, along the way a few hiccups occurred that ended up defining how the web API functioned.

The Raspberry Pi Pico W had been chosen specifically because of its wireless functionality so it was only logical that communication with the web application would occur from the Raspberry Pi Pico W. In order to achieve this functionality, the team followed a guide on creating a simple web client in C (Fairhead, 2022). The main issues that came about using this methodology is that it only supports GET requests, and it must be done via HTTP, rather than HTTPS. These both posed issues as the website wasn’t set up to support GET requests using query parameters, and the website was enforcing HTTPS. In order to get around these issues, it was determined that the web API would live on a separate url that did not enforce SSL and the requests would all be done using url segments such as “/check_schedule/device/RX-AR2023-0001” where “check_schedule” is the endpoint, “device” is a parameter, and “RX-AR2023-0001” is the value.

The other issue the team ran into while implementing the API functionality were struggles with string parsing in C. The team’s experience with string parsing in C up to this point was relatively limited and had wrongfully assumed they would have access to functionality similar to that available in C#. This meant that the team assumed building the url segment given above from variables would be simple, but pointers and string concatenation made it less than simple. Parsing content from a received string was not simple either. Splitting a string on a delimiter isn’t straight forward, so a received value of “UserID:5;ScheduleID:7” isn’t as easy as splitting on the semicolon and then splitting on the colons, once again due to how pointers to strings work. This meant that both requests and responses needed to be as simple as possible to work with.

With the issues discussed above, the API itself was accomplished in as simple a manner as possible. The main functionality that was implemented was the ability to check if there is an active schedule that needs to be delivered, set/get the user's position to aid in navigation, log when a successful delivery occurs, and log when a user can't be found. As was noted previously, the web API was created as part of the same code base as the web application, so it was written in PHP using CodeIgniter as a framework to make it faster and easier to work with. Some of the functions were written in a future looking manner that would allow support for multiple users, whereas some were written in a limited manner due to needing to limit the scope late in the process.

4.2.1 Check Schedule

The check schedule function is the most important function in the API as it is what drives the robot state machine. The purpose of the function is to return a schedule ID if there is a schedule to act on, or to return none if there's nothing to act on. This function (Figure 14) was written in a future looking manner though the result is not used in such a way.

Figure 14: Check Schedule API Function

```
// expect url format: "check_schedule/device/RX-AR2023-####"
public function check_schedule()
{
	$start_date = new DateTime("- 30 seconds"); // give 30 second leeway for one time
deliveries
	$end_date = new DateTime("+ $this->MAX_TRAVEL_TIME minutes");

	$schedules = $this->schedules_model->get(array(
		"device" => $this->device->DeviceID,
		"order_by" => "ScheduleDateTime",
		"order_dir" => "DESC",
		"include_once" =>  true,
		"time_after" => $start_date->format('Y-m-d H:i:s'),
		"time_before" => $end_date->format('Y-m-d H:i:s')
	));

	foreach($schedules as $schedule)
	{
		$next_schedule = determine_next_delivery(new
DateTime($schedule->ScheduleDateTime), $schedule->Frequency);

		// the next schedule is today, so it's relevant, we want to process the first
schedule we find
		if(strpos($next_schedule, "today") !== false)
		{
			echo "UserID:".$schedule->UserID.";ScheduleID:".$schedule->ScheduleID.";";
```

```
                    exit;
                }
            }

            // no schedules to process
            echo "none";exit;
}
```

The function starts by checking the database for any schedules that have a scheduled time between 30 seconds prior, and a configured amount of time that is intended to be the max delivery time in the future. The start limit being 30 seconds in the past allows for the code not running quite fast enough to immediately catch any manual requests for delivery. The configured end time is set to 15 minutes which is the max amount of time the robot is allowed to take to deliver the medication before it's considered an error. By looking 15 minutes into the future, this allows, as an example, being able to start a delivery for 8:00 AM prior to the delivery time, at 7:45 AM, in order to make it there by the intended time.

After grabbing schedules by time, the code then checks each schedule to see if it's supposed to occur today or not, based on the frequency set for the schedule and the current time. For example, if the schedule is set to weekly on Monday at 8:00 AM and it is currently Sunday at 7:50 AM, then the next delivery will be tomorrow so it is skipped. However, if the schedule is daily at 8:00 AM, then the next delivery would be today, so it would not be skipped.

Once the code determines if the schedule is today or not, the user ID associated with the schedule and the schedule ID are both output in a way that will be easily parsable by the C code on the Raspberry Pi Pico. If no schedule was found, "none" is output instead just for simple distinction as it does not follow the proper syntax of a valid response.

While the code does output the user ID, the Raspberry Pi Pico ignores this as the rest of the API does not require any user specific functionality due to scope limitations to a single device and a single user.

4.2.2 Set/Get User Location

The original intention was for the robot to be able to read the user tracker's location but this proved to not be how the UWB modules work so the team changed to instead store the user

tracker's location in the database for retrieval on demand. This required a two stage process, one where the user tracker posts its location up to the server (Figure 15), and one where the robot pulls down the user's location (Figure 16).

Figure 15: Set User Location Function

```
// expect url format: "set_user_location/x/######/y/######/z/######"
public function set_user_location()
{
	$user_id = 22; // hardcoded for ease of testing to the "User" user

	$this->users_model->save(array("TrackerPosition" => json_encode(array("x" => $x, "y" =>
$y, "z" => $z))), $user_id);
}
```

Figure 16: Get User Location Function

```
// expect url format: "get_user_location"
public function get_user_location()
{
	$user_id = 22; // hardcoded for ease of testing to the "User" user

	$user = $this->users_model->get(array("id" => $user_id), false);

	$position = json_decode($user->TrackerPosition);
	echo "x:".$position->x.";y:".$position->y.";z:".$position->z.";";
}
```

Both of these functions assume a user ID of 22 as this was determined to be the team's test user for demo purposes. In an ideal situation, this would take in a user ID to be able to set/get the appropriate user's information. One limiting factor in the short time the team had was how to associate a user id with a specific user tracker. Given the team only had one user tracker device physically, spending any time to figure out how to handle the situation did not seem worth it but would need to be evaluated for future development.

4.2.3 Log Delivery

The next most important function in the API is the log delivery function. This is used for tracking of dosage statistics for a medication. When a delivery is logged, the remaining medication is updated. This information also feeds into statistics showing when the last delivery occurred for a medication. Without this function, the user may as well only manage a schedule as the medication information is effectively useless. This function (Figure 17) only handles successful deliveries, unsuccessful ones are handled by the Log Missing User function (Figure 18).

Figure 17: Log Delivery Function

```
public function log_delivery()
{
	extract(filter_options(array("schedule_id"), $this->params));

	$medicine = $this->medicines_model->get(array("schedule_id" => $schedule_id), false);

	$this->deliveries_model->log_delivery(array(
		"ScheduleID" => $schedule_id,
		"Automated" => true,
		"DoseGiven" => $medicine->Dose
	));
}
```

The unique aspect to this function is the parameter parsing that happens immediately. Unlike the other functions, a schedule ID is required when logging a delivery as the code needs to know which medication this delivery was for in order to update the stats properly. The schedule ID is used to retrieve the medication associated with this schedule. Deliveries are tied to schedules but the dosage given as part of a delivery is tied to the medication as it does not vary between schedules.

4.2.4 Log Missing User

Logging a missed dose is just as important as logging a successful one but needs to be done in a slightly different way which is why the functionality is handled differently. One of the biggest differences is that successful deliveries go into a table that is used to generate a calculated column in the database for how much of a medication is remaining. If missed doses went into this table it would cause conflicts when trying to calculate how much medication has been taken and when the last delivery occurred. As can be seen in Figure 18, missed doses are instead logged into the event table so they can be displayed in the alerts on the dashboard or in other interfaces for easier debugging.

Figure 18: Log Missing User Function

```
public function log_missing_user()
{
	extract(filter_options(array("schedule_id"), $this->params));

	$schedule = $this->schedules_model->get(array("id" => $schedule_id, "include_device_id"
=> true), false);
	$event_type = $this->events_model->get_types(array("tag" => "system_err"), false);
	$schedule_time = determine_delivery_stamp(new DateTime($schedule->ScheduleDateTime));
```

```
        $this->events_model->log_event(array(
                "DeviceID" => $schedule->DeviceID,
                "EventName" => "User Not Found",
                "EventTypeID" => $event_type->EventTypeID,
                "EventDescription" => $schedule->MedicineName . " was not delivered " .
str_replace("Daily: ", "@ ", $schedule_time).".",
                "EventMaker" => "Arven",
                "EventIcon" => "user-slash"
        ));
}
```

Similar to the Log Delivery function, the Log Missing User function (Figure 18) needs to grab the schedule using the schedule ID passed in. The missing user event is considered a system error so the appropriate error type must be retrieved to be able to log the event properly. The last piece required to log the event is what time this schedule was supposed to have occurred at as a nicely formatted string so that it can be easily picked out of an event log.

4.3 Embedded Execution

In order to get the robot to function, the team decided to have one simple microprocessor, the ATmega328P, handle sensor interfacing, and a more powerful microprocessor, the Raspberry Pi Pico W, handle the logic processing of what to do with the sensor information. In order to facilitate communication between the two, the team designed a custom frame implementation that would be transferred via UART. The sections below will discuss the ATmega328P sensor execution, the custom frame execution, and the Raspberry Pi Pico W logic execution.

4.3.1 ATmega328P Software (Sensors) Execution

The software development for the sensors required for this project had many ups and downs. This caused time being lost and some changes in component selection, but the overall functionality of each sensor was implemented to an acceptable level. Each sensor had an independent library created to separate their functionalities.

Software was developed using C via Microchip Studio (Microchip Technology Inc, n.d.), which allows the ATmega328P (Atmel, 2009) to program the required sensors. Microchip Studio was selected as the IDE due to the team's familiarity with it to program the ATmega328P in other courses.

4.3.1.1 Ultrasonic Sensor Implementation

To begin with, the HC-SR04 Ultrasonic Sensor (ELEC Freaks, n.d.) would be one of the first devices to configure. It was decided that this device would rely upon interrupts for the sake of accuracy in distance measurement. Using the interrupts, the measurement is taken as soon as an echo is returned to the device without delay. The function and definitions within the code block in Figure 19 assisted in setting up the interrupts for the correct pins of the ATmega328P (Atmel, 2015).

Figure 19: HC-SR04 Init

```
typedef enum
{
    HCSR04_L = 0, // left HC-SR04 sensor    -- PD5/PD6
    HCSR04_C = 1, // center HC-SR04 sensor  -- PD7/PB0
    HCSR04_R = 2, // right HC-SR04 sensor   -- PB1/PB2
    HCSR04_None = 10
} HCSR04_Device;

//L = Left, R = Right, C = Centre
#define HCSR04_L_Trig 0b00100000 // PORTD
#define HCSR04_L_Echo 0b01000000 // PORTD
#define HCSR04_C_Trig 0b10000000 // PORTD
#define HCSR04_C_Echo 0b00000001 // PORTB
#define HCSR04_R_Trig 0b00000010 // PORTB
#define HCSR04_R_Echo 0b00000100 // PORTB


void HCSR04_InitDevice(HCSR04_Device device)
{
    switch(device)
    {
        case HCSR04_L:
            DDRD |= HCSR04_L_Trig; //output
            DDRD &= ~HCSR04_L_Echo; //input

            PCMSK2 |= HCSR04_L_Echo; // turn on PCINT22 pin mask (enable interrupts)
            PCICR |= 0b00000100; // turn on interrupts for group 2
            break;
        case HCSR04_C:
            DDRD |= HCSR04_C_Trig; //output
            DDRB &= ~HCSR04_C_Echo; //input

            PCMSK0 |= HCSR04_C_Echo; // turn on PCINT0 pin mask (enable interrupts)
            PCICR |= 0b00000001; // turn on interrupts for group 0
            break;
        case HCSR04_R:
            DDRB |= HCSR04_R_Trig; //output
            DDRB &= ~HCSR04_R_Echo; //input

            PCMSK0 |= HCSR04_R_Echo; // turn on PCINT2 pin mask (enable interrupts)
            PCICR |= 0b00000001; // turn on interrupts for group 0
            break;
```

```
            default:
                break;
        }
    }
```

The defines above are what each pin of the specified Ultrasonic Sensor (US) corresponds to. The HCSR04_L_Trig (left Ultrasonic Sensor trigger), for example, within the HCSR04_InitDevice function has its value OR'ed (|=) onto PORTD via DDRD (Atmel, 2015, p. 73) in order to set the bit PD5 to high to indicate it's an output. Similarly, the echo pin is set, but since it needs to be set as an input, the inverse of the value is AND'ed (&=) on instead. After this, the interrupts are configured by turning on the pin mask PCMSK0 (Atmel, 2015, p. 57) by using the specified echo pin and enabling interrupts on the interrupt control register: PCICR (Atmel, 2015, p. 56). This process is repeated for all three Ultrasonic Sensors, the only difference being the pin numbers that the sensor corresponds to. The pins chosen go in sequential order around the device (10, 11, 12, 13, 14, and 15) but these pins are located in two different registers, PORTD (PD5, PD6, PD7) and PORTB (PB0, PB1, PB2), which accounts for the difference in bit masks and registers used for configuring the pins. Since the Ultrasonic Sensors could now be configured as interrupts, the ISR was created. The ISR is defined as follows:

Figure 20: HC-SR04 ISR Function

```
volatile long echoTimeStart = 0;
volatile long echoTimeEnd = 0;

void HCSR04_ISR()
{

    // Only perform the check if there's an active device
    if(activeDevice != HCSR04_None)
    {
        PORTC ^= 0b00000100;
        int condition = 0;
        // determine the high condition for the active device
        switch(activeDevice)
        {
            case HCSR04_L:
                condition = PIND & HCSR04_L_Echo;
                break;
            case HCSR04_C:
                condition = PINB & HCSR04_C_Echo;
                break;
            case HCSR04_R:
                condition = PINB & HCSR04_R_Echo;
                break;
```

```
            default:
                break;
        }

        // When the echo starts, track current TCNT value
        if(condition)
        {
            echoTimeStart = TCNT1;
        }
        // When echo ends, track the new TCNT value and indicate no device is active
        else
        {
            echoTimeEnd = TCNT1;
            activeDevice = HCSR04_None;
        }
    }
}
```

The ISR checks for any active devices initially, ignoring any change that was triggered if no device is active. If there is an active device, it will respond to the active device's echo pin. When this pin goes high for the first time, a snapshot of the TCNT1 value is taken to compare against later. Once the ISR fires again but for a low signal on the pin the TCNT1 value is captured again in order to calculate the duration that the pin was high for. Now that the interrupts and ISR were set up, the code for the functionality of the Ultrasonic Sensors began. In order to trigger a device to check the distance, the following function was created:

Figure 21: HC-SR04 Trigger Function

```
int trigger(HCSR04_Device device)
{
    // ensure there is no active device
    if(activeDevice == HCSR04_None)
    {
        int pin;

        // set the device to be active
        activeDevice = device;

        // Determine which pin needs to be toggled based on the device
        switch(activeDevice)
        {
            case HCSR04_L:
                pin = HCSR04_L_Trig;
                break;
            case HCSR04_C:
                pin = HCSR04_C_Trig;
                break;
            case HCSR04_R:
                pin = HCSR04_R_Trig;
                break;
            default:
                break;
```

```
        }
        switch(activeDevice)
        {
            case HCSR04_L:
            case HCSR04_C:
                // set pin low for 2 us to ensure we're starting with a fresh pulse
                PORTD &= ~pin;
                _delay_us(2);
                // set the pin high for a minimum of 10us to ensure the 8 pulses are sent
                PORTD |= pin;
                _delay_us(10);
                PORTD &= ~pin;
                break;
            case HCSR04_R:
                // set pin low for 2 us to ensure we're starting with a fresh pulse
                PORTB &= ~pin;
                _delay_us(2);
                // set the pin high for a minimum of 10us to ensure the 8 pulses are sent
                PORTB |= pin;
                _delay_us(10);
                PORTB &= ~pin;
                break;
            default:
                break;
        }
        // successful trigger
        return 1;
    }
}
```

If there is currently no active Ultrasonic Sensor taking a measurement, determine what pin needs to be toggled by looking at the corresponding trigger pin value. According to the datasheet (ELEC Freaks, n.d.), the trigger pin must be set to high for a minimum of 10 µs to ensure that eight pulses are sent. Before this, the pin is set to low by using the inverse for the bit value of the pin and the &= operand on the trigger port (PORTD or PORTB). With this function, the trigger pin for a specified US can now be toggled as needed. To get the time it takes for an echo to return once the trigger is toggled though, an echo wait function had to be created first which can be seen below:

Figure 22: HC-SR04 Wait For Echo Function

```
long waitForEcho(HCSR04_Device device)
{
    char buff[20];
    long diff;
    // ensure this is the active device before waiting for echo
    if(activeDevice == device)
    {
        while(activeDevice == device);
        if( echoTimeEnd >= echoTimeStart){
            diff = echoTimeEnd - echoTimeStart;
```

```
        } else{
            diff = 65535 - echoTimeStart + echoTimeEnd;
        }
        return (diff)/2;
    }

    // active device is not this device, return invalid duration
    return -1;
}
```

The above function essentially waits for the specified Ultrasonic Sensor to become inactive, which is set by the ISR, and then calculates the difference between the time the echo started and the time the echo ended, based on the previously mentioned ISR. This difference is then halved due to the ISR running every half a microsecond, this conversion would lead to the difference in just microseconds, which makes it simpler for later duration calculations. Now finally, the trigger and echo functionalities can be combined into one simple function that can grab the total duration the echo took once a trigger has been toggled:

Figure 23: HC-SR04 Get Echo Duration Function

```
long HCSR04_GetEchoDuration(HCSR04_Device device)
{
    long duration = 0;

    echoTimeStart = 0;
    echoTimeEnd = 0;
    if(trigger(device))
    {
        duration = waitForEcho(device);
    }
    return duration / 2; // divide 2 in order to convert to uS
}
```

This function simply calls the trigger function to try and toggle the trigger. If successful, the duration of the echo is taken using the previously mentioned waitForEcho function. The duration is halved again because the duration returned is a round trip duration, to the obstacle and back, and we only care about the one direction when calculating distance.

The duration is then passed along to the Raspberry Pi Pico W for interpretation in terms of actual distance represented and what action to take based on that distance. The implementation of HC-SR04 Ultrasonic Sensor was an overall success, having all of the functionality required for the scope of this project.

4.3.1.2 Bump Sensor Implementation

The next sensor to begin software implementation was the D3V-163-1A5 SPDT Snap Action Switch (Omron, 2021). This sensor was fairly simple to implement and there were no struggles while doing so.

Since the D3V is a switch there are only two assumed states to consider: the switch has been activated due to an object bumping into it, or it hasn't. Given that a switch being active means we have already hit something we wanted to make sure the response was as immediate as possible. For this reason, it was decided that it would be best to implement this sensor with interrupts. The configuration of the interrupt can be seen below.

Figure 24: Backup Sensor Init Sensor Function

```
#define Back_Sens_L 0b00000100 // PORTD
#define Back_Sens_R 0b01001000 // PORTD
void Back_Sens_InitSens(int sens)
{
    DDRD &= ~sens; //input

    PCMSK2 |= sens; // turn on PCINT18/19 pin mask (enable interrupts) (12.2.6)
    PCICR |= 0b00000100; // turn on interrupts for group 2 (12.2.4)
}
```

This implementation is fairly similar to the Ultrasonic Sensor interrupt configuration seen in section 4.3.1.1, the key difference being the pin that the interrupt is on within the ATmega328P (Atmel, 2015). As indicated by the Back_Sens_L and Back_Sens_R definitions, these sensors are to be configured on PORTD. Within the initialization function, Back_sens_InitSens, one of the two definitions mentioned will be provided to enable the corresponding pin as an input on PORTD. Similar to the Ultrasonic Sensor, the pin mask (Atmel, 2015, p. 57) and PCICR control register (Atmel, 2015, p. 56) must be enabled as well. Now that the interrupts can be configured, the ISR was created as seen in the function in Figure 25, below.

Figure 25: Backup Sensor ISR Function

```
char Back_Sens_ISR(int pin)
{
    // check if this sensor was hit
    if(PIND & pin)
    {
        return 1;
    }
    return 0;
}
```

The goal of the ISR above is to check when the specified Bump Sensor is high. If it is high, simply return 1, otherwise it is assumed that the sensor is still in a low state. As previously discussed, these sensors are configured as inputs on PORTD, so the Port D Input Pin Address (Atmel, 2015, p. 73) is used alongside the & operand with the desired sensor's definition to check its state.

The state of the bump sensors is passed along to the Raspberry Pi Pico W in order to be interpreted just as the Ultrasonic Sensors were. To conclude, the Bump Sensor implementation in terms of software was quite simple. It was configured quite similarly to the Ultrasonic Sensor (Section 4.3.1.1) in terms of interrupts, so the team had no issues in successfully implementing this sensor.

4.3.1.3 Weight Sensor Implementation

Next, the GD03 Force Sensing Resistor (Uneo, 2014) library was created. This sensor was easily implemented and made functional. Little to no issues occurred while configuring this sensor's software.

The purpose of this sensor was to detect the weight of the pill bottle that the user has put on the Robot. It was decided that using the 10-bit Analog to Digital (A/D) module within the ATmega328P (Atmel, 2015, p. 5) would be best for measuring the changes in voltage as the weight changes once pills are added or removed. Fortunately, a library provided by Simon Walker had the A/D configuration within it, so the team simply had to implement code that would read the A/D value. The functions used for this can be seen below:

Figure 26: GD03 Functions

```
void GD03_Init(void)
{
    AtoD_Init(AtoD_Channel_0); // pin 23
}

int GD03_CaptureAtoDVal(void)
{
    unsigned char AD_low = ADCL; // must be read first
    unsigned char AD_high = ADCH;
    unsigned int atodval = (AD_low + AD_high * 256);
    return atodval;
}
```

Within the initialization function, GD03_Init, the AtoD_Init function is called in order to initialize channel 0 on the A/D module (Atmel, 2015, p. 69) which corresponds to pin 23. According to the ATmega328P datasheet (Atmel, 2015, p. 206), the 10-bit result of the A/D module is presented in the ADC data registers, ADCH and ADCL. The GD03_CaptureAtoDVal function captures these data registers within two variables: AD_low and AD_high.

The ADCL register must be read before the ADCH register to ensure that the content belongs to the same conversion (Atmel, 2015, p. 206). These two values will be summed to get the full 10-bit result. However, in the AtoD_Init function, the ADMUX register has the ADLAR bit set to low (Atmel, 2015, p. 206). This means that the result presented from the ADCH and ADCL registers will be right adjusted, thus meaning an 8-bit value is being read. This is why the atodval variable (Figure 26) within the GD03_CaptureAtoDVal function captures the sum of the two registers and multiples the value by 256, since the maximum for an 8-bit value is 256.

The raw AtoD value measured by the device is then passed along to the Raspberry Pi Pico for conversion to a weight value and interpretation. The implementation of the GD03 was quite simple. The goals of this library were to capture the change in voltage as weight was detected and turn that into a readable value. This goal was successfully implemented with the help of Simon Walker A/D library overall.

4.3.1.3 Infrared Sensor Implementation

For drop detection, the SEN0427 (DFRobot, 2021a) Infrared Sensor modules were implemented next. These sensors rely upon the I^2C protocol to be implemented for the

ATmega328P (Atmel, 2015). The team utilized Simon Walker's I^2C to facilitate communication between the devices. Due to some difficulties interpreting the various datasheets for the underlying IR sensor, the VL6180x, even the ones that had code samples (STMElectronics, 2018), the team resorted to utilizing the code of an Arduino library (DFRobot, 2021b) created by the manufacturer, DFRobot. This allowed the team to get the code functioning properly, with a few minor changes.

Overall, these sensors were all implemented to an adequate level, and the GitHub repository (Skretteberg & Manhas, 2023a) for this section can be used if more details are required in terms of code.

4.3.2 Custom Frame Execution

Once the sensors were functional using the ATmega328P (Atmel, 2015), the next step was to come up with a way to communicate between the ATmega328P and Raspberry Pi Pico (Raspberry Pi Ltd, 2023a). After discussing with Simon Walker, it was determined that creating a Frame to send from the ATmega328P to the Pico would be an appropriate way to facilitate this.

In order to create this frame, the team had to decide how to break down each sensor value being sent from the ATmega into different segments with varying ranges in hexadecimal. An example of the full frame can be seen below:

Figure 27: Full Communication Frame

$FFFFF1FFFF1FFFF1FFFFA3FF1^

In order to break down this frame further a table was created, which can be seen below in Table 1 below.

Table 1: Frame Translation

Segment #	1	2	3	4	5	6	7	8	9
Hex Range	00 To FF	00 To FF	00 To FF	00000 To FFFFF	00000 To FFFFF	00000 To FFFFF	A To D	000 To 3FF	0 To 1
# of Characters	2	2	2	5	5	5	1	3	1
Corresponding Sensor	N/A	Left Infrared Sensor	Right Infrared Sensor	Left Ultra sonic	Centre Ultra sonic	Right Ultra sonic	Bump Sensor	Weight Sensor	Battery Level

The frame seen in Figure 27 can be divided into 9 different segments, which are broken down in Table 1. The frame must be read from left to right in order to correspond the segment number to where it belongs in the frame. To begin with, the frame must have a "$" at the start of it and a "^" at the end of it, as shown by Figure 27. This is to keep the length of the frame consistent and help keep it distinct from any other data that may be sent. This allows for easy detection of frame start and frame end when receiving the data.

To further break the frame down, segment one in Table 1 is a single byte represented by two characters that indicates what segments have changed. Essentially, if this value is non-zero, a change has occurred based on the data being sent from the sensors. Each bit of this byte represents one of the segments, being set to 1 if it changed since the last frame, 0 if it hasn't. Segments two and three correspond to the two SEN0427 IR Sensors (DFRobot, 2021a), which represent distance in millimeters (mm) converted to hex. Segments four to six represent the number of microsends taken for the echo of each ultrasonic sensor (US) to send and receive, once again converted to hex.

The next segment, segment seven, indicates the bumper sensor detection. For simplicity, it was decided that using the hex values A through D would be simplest due to how the lower two bits of each of these values appear in binary. A visual representation of this can be seen in Table 2 below:

Table 2: Bump Sensor Frame Translation

Hex	A	B	C	D
Binary	1010	1011	1100	1101
Lower 2 Bits	10	11	00	01

As seen in Table 2, the lower two bits for each of these are distinct. Since two bump sensors were used for this project, we can correspond each of the lower two bits to each sensor. The leftmost bit of the lower two bits represents the left Bump Sensor and the rightmost represents the right Bump Sensor. If one of these bits are set to 1, that corresponding sensor must have been activated.

Segment eight in Table 1 represents the GD03 (Uneo, 2014) Weight Sensor. As stated in Section 4.3.1.3, the library for this sensor takes a 10-bit raw A/D value representing a voltage from 0V to 5V (Uneo, 2014). Once this value is converted to hex it is limited to a Hex Range of 000 to 3FF, as indicated by Table 1.

The last segment of the frame is the Battery Level Indication, but due to time constraints this was not implemented fully. Despite this however, the overall frame covered all of the required sensors as needed.

After this, the communication between the ATmega328P (Atmel, 2015) and Raspberry Pi Pico (Raspberry Pi Ltd, 2023a) was ready to be implemented in code. In order to start the code implementation of this frame, it was decided that struct collection would be the best way to represent the frame. An example of this can be seen below:

Figure 28: PicoFrame Struct

```
struct PicoFrame
{
    unsigned char IR_L_Distance;    //measured in mm
    unsigned char IR_R_Distance;    //measured in mm

    long Ultrasonic_L_Duration;     //measured in us
    long Ultrasonic_C_Duration;     //measured in us
    long Ultrasonic_R_Duration;     //measured in us

    char Bump_L;                    // 1 if there's an object
```

```
    char Bump_R;                    // 1 if there's an object
    int  Weight;                    // AD value measured (5V ref)
    char Battery_Low;               // 1 if battery low
    char Motor_FL_Direction;        // 1 if forward
    unsigned char Motor_FL_Speed;   // measured in RPM
    char Motor_FR_Direction;        // 1 if forward
    unsigned char Motor_FR_Speed;   // measured in RPM
}
```

As seen in Figure 28, the struct used contains all of the necessary sensor values. Referring back to Table 1, each segment of the frame has different ranging values, hence why different types were used depending on the segment. For example, the Ultrasonic Duration variables are all the long type, this is in order to store the full value due to the hex range for these sensors ranging from 00000 to FFFFF (Table 1). An example of how this is used can be seen below:

Figure 29: PicoFrame Example

```
struct PicoFrame frame;
frame.Bump_R = 0;
frame.Bump_L = 0;
frame.Ultrasonic_C_Duration = 0;
frame.Ultrasonic_R_Duration = 0;
frame.Ultrasonic_L_Duration = 0;
frame.IR_L_Distance = 0;
frame.IR_R_Distance = 0;
frame.Motor_FL_Direction = 0;
frame.Motor_FR_Direction = 0;
frame.Motor_FL_Speed = 0;
frame.Motor_FR_Speed = 0;
frame.Battery_Low = 0;
frame.Weight = 0;

while(1)
{
    sleep_cpu();
    if(_Ticks > timerEventCount){
        _Ticks = 0;
        frame.Weight = GD03_CaptureAtoDVal();
        frame.Ultrasonic_L_Duration = HCSR04_GetEchoDuration(HCSR04_L);
        frame.Bump_L = bump_L;
        frame.Bump_R = bump_R;
        frame.Ultrasonic_C_Duration = HCSR04_GetEchoDuration(HCSR04_C);
        frame.Ultrasonic_R_Duration = HCSR04_GetEchoDuration(HCSR04_R);
        frame.IR_L_Distance = SEN0427_CaptureDistance(SEN0427_L);
        frame.IR_R_Distance = SEN0427_CaptureDistance(SEN0427_R);
        Pico_SendData(frame);
    }

}
```

From Figure 29, the struct is created and each variable within it must be set to zero initially to ensure a fresh frame is being sent once the application is built. Then, once the timer tick count is appropriate, the frame is populated with real data. Using the various functions to grab the

changed data of each sensor, as explained in Section 4.3.1, each variable within the struct is assigned appropriately using the corresponding function. The struct is then used with the Pico_SendData function, which utilizes the SCI capabilities of the ATmega328P (Atmel, 2015) via Simon Walker's library in order to break down the struct given and send the frame to the Pico.

To conclude this section overall, the frame was created and adapted into code form using structs. Using the struct format allowed the code to be organized and assisted in creating the overall foundation for the communication aspect of this project.

4.3.3 Raspberry Pi (Logic) Execution

The next step in the process of the Embedded Execution was to determine the logic of the Robot, for this a Raspberry Pi Pico W (Raspberry Pi Ltd, 2023a) was used. This was essential in order to transform the State Diagram (Figure 9) into code.

For this, it was decided after discussing with AJ Armstrong, that using C to program this device would be best. In order to start the process, the *Getting started with Raspberry Pi Pico guide* (Raspberry Pi Ltd, 2023a) was used. This guide provides instructions to set up Visual Studio Code (Microsoft, n.d.) to build programs onto the Pico using CMake.

Now that a platform was selected and the Pico could be built to, each state was then executed into code form. To do this, an enumeration within the main.c file of the Pico code was created as follows:

Figure 30: RobotState Enumeration

```
typedef enum
{
    RobotState_Idle,
    RobotState_NavigatingToUser,
    RobotState_Stuck,
    RobotState_DeliveringPayload,
    RobotState_NavigatingHome
} RobotState;
```

As seen in Figure 30, each state seen in Figure 9 has its own corresponding variable. This is then used to determine the state of the Robot depending on what happens throughout the

program. The enumeration is then used within a switch statement in order to determine what action to take depending on its status. An example of this can be seen below:

Figure 31: Initial RobotState Switch

```
RobotState robotState = RobotState_Idle;

switch(robotState)
{
    case RobotState_Idle:
        break;
    case RobotState_NavigatingToUser:
        break;
    case RobotState_DeliveringPayload:
        break;
    case RobotState_NavigatingHome:
        break;
}
```

As seen in Figure 31, each state is represented by the RobotState enumeration as defined in Figure 30. The switch is then used in order to isolate each state in order to separate actions depending on the current value of the robotState variable. The details of the actions taken in each state will be explained in the following subsections.

4.3.3.1 Idle

As stated in Section 3.3.3, the Idle state is the state in which the Robot is waiting for an active schedule for delivery. This is considered to be the initial state of the Robot and it will change once a schedule is received. To do this, an idle function was created:

Figure 32: Idle Function

```
int idle(void)
{
    web_request_check_schedule();
    // capture the schedule id once the web request has finished
    return web_response_check_schedule();
}
```

The above function executes a web request in order to get a schedule id from the Web Interface. Based on the value of the schedule id, the state will change or remain in the idle state. Once integrated into the main Robot code, state diagram in code form looks like so:

Figure 33: Main Switch With Idle State

```
switch(robotState)
{
    case RobotState_Idle:
        scheduleId = idle();
        if(scheduleId != -1)
            robotState = RobotState_NavigatingToUser;
        break;
    case RobotState_NavigatingToUser:
        break;
    case RobotState_DeliveringPayload:
        break;
    case RobotState_NavigatingHome:
        break;
}
```

As seen above, if the value of the schedule id returned from the web request is a -1, the schedule is considered inactive, thus meaning that the Robot will remain in its idle state. If the schedule id is valid, the robotState is changed into the RobotState_NavigatingToUser state (Figure 30) meaning that navigation may begin.

4.3.3.2 Navigating to User

In the Navigating to User state (Figure 9), as indicated by the name, the Robot begins navigating to the user based off of the UWB positioning. Before this can be done however, a way to check the current state of motion of the Robot had to be created. Alongside this, the motors must move depending on that state, then once combined, navigation code can begin. The motion state is setup as follows:

Figure 34: MotionState Example

```
typedef enum
{
    MotionState_Forward,
    MotionState_Reverse,
    MotionState_Stop,
    MotionState_TurnRight,
    MotionState_TurnLeft,
    MotionState_ToBeDetermined
} MotionState;

void act_on_motion_state(MotionState action)
{
    switch(action)
    {
        case MotionState_Forward:
            go_forward();
            break;
        case MotionState_Reverse:
```

```
            go_backward();
            break;
        case MotionState_TurnRight:
            turn_right();
            break;
        case MotionState_TurnLeft:
            turn_left();
            break;
    }
}
```

From Figure 34 above, the state of motion of the Robot is quite self-explanatory with the use of the MotionState enumeration. The Robot can be moving forward, reverse, right, or left in terms of motion. The Robot can also be stopped, or in a To be Determined state in which the Robot has yet to determine how to move. This is then used within the act_on_motion_state function in which a switch case using the previously stated enumeration determines what direction the motors turn. In order to check the current state of the navigation itself however, another enumeration was created:

Figure 35: NavigationResult Enumeration

```
typedef enum
{
    NavigationResult_Complete,
    NavigationResult_Incomplete,
    NavigationResult_Stuck
} NavigationResult;
```

As seen in Figure 35, there are three different results in navigation. These are: Complete, Incomplete, and Stuck. These results depend on the current state of navigation, meaning that many factors come into play. The end result of this is a rather robust function used for complete navigation, which can be seen in **Appendix E.** This function relies upon the DWM1001 (Qorvo, n.d.) positioning of the Robot and the user.

To add this in to the state switch in the code for the Pico (Raspberry Pi Ltd, 2023b), it will look like so:

Figure 36: Main Switch With NavigatingToUser State

```
switch(robotState)
{
    case RobotState_Idle:
        scheduleId = idle();
        if(scheduleId != -1)
            robotState = RobotState_NavigatingToUser;
        break;
    case RobotState_NavigatingToUser:
        result = navigating_to_user(sensorValues);
        if(result == NavigationResult_Complete)
            robotState = RobotState_DeliveringPayload;
        break;
    case RobotState_DeliveringPayload:
        break;
    case RobotState_NavigatingHome:
        break;
}
```

From the Figure above, there are now two states that have actions: the Idle state as explained in Section 3.3.1, and the Navigating to User state. If the result of the navigating_to_user function is considered Complete (Figure 35), the next state may begin. Otherwise, if the Robot is stopped for a length of greater than one minute (**Appendix E**), the Robot is considered stuck.

4.3.3.3 Delivering Payload

Once the Robot has successfully navigated to the user (Section 4.3.3.2), it must allow the user to pick up the medication off of the platform and put it back on. This means that the dose was taken by the user and the schedule on the Web Interface can now be updated.

In order to execute this, more states were required, hence the use of an enumeration. This can be seen below:

Figure 37: DeliveryState Enumeration

```
typedef enum
{
    DeliveryState_WaitingRemoval,
    DeliveryState_Removed,
    DeliveryState_Complete
} DeliveryState;
```

From Figure 37, it was deemed that three states would be required to be considered a full delivery: Waiting, Remove, and Complete. The Waiting state occurs simply when the Robot has

arrived at the user and is awaiting for the medication bottle to be lifted. Once the bottle is lifted the delivery state is considered Removed, and once returned back onto the platform it is considered a full delivery. The Robot must remain in one of these three states for at least five seconds for the state to be considered set, for example the medication must remain off of the platform for up to at least five seconds to be considered in a Remove state. This duration can be changed however by using a variable within the main Pico (Raspberry Pi Ltd, 2023a) code.

With this concept in mind, a delivering function was created, which is detailed in **Appendix F**. The function was then used in the main switch statement like so:

Figure 38: Main Switch With DeliveringPayload State

```
switch(robotState)
{
    case RobotState_Idle:
        scheduleId = idle();
        if(scheduleId != -1)
            robotState = RobotState_NavigatingToUser;
        break;
    case RobotState_NavigatingToUser:
        result = navigating_to_user(sensorValues);
        if(result == NavigationResult_Complete)
            robotState = RobotState_DeliveringPayload;
        break;
    case RobotState_DeliveringPayload:
        if(delivering_payload(sensorValues, scheduleId))
            robotState = RobotState_NavigatingHome;
        break;
    case RobotState_NavigatingHome:
        break;
}
```

With this in mind, the Robot would now go from an idle state to a navigating state, then wait for the medication to be picked up and put back down for a full delivery completion.

4.3.3.4 Navigating Home

Once the Robot has successfully delivered the medication, it is then considered to be in the Navigating Home state based off of the state diagram (Figure 9). Within this state, the Robot will navigate back to its origin, which can be thought of as a grid. This means that the X position, Y position, and Z position of the Robot must be set back to 0 to be considered "home". This means that the home is assumed to be at the origin for the sake of simplicity.

To execute this, the previously mentioned navigate function, featured in **Appendix E,** must be used. The primary stipulation to this however is that a new DWM (Qorvo, n.d.) position must be created and set as the origin, then the Robot must travel back to that spot. The following function was used to execute this concept:

Figure 39: Navigating Home Function

```
NavigationResult navigating_home(struct AtmegaSensorValues sensorValues)
{
    struct DWM1001_Position homePosition;
    homePosition.x = 0;
    homePosition.y = 0;
    homePosition.z = 0;
    homePosition.set = 1;
    return navigate(sensorValues, homePosition);
}
```

The above function creates a new DWM1001 (Qorvo, n.d.) position struct, which contains the X, Y, and Z coordinates of the module in millimetres. In this case, the struct is being used in order to send a desired position to travel to via the navigate function (**Appendix E).** As mentioned previously, the home position is considered to be at the origin, hence why the coordinates are set to 0. With this function, the Robot will travel back to its home position and enter the idle state once again as indicated by the state diagram (Figure 9).

To implement this final functional state, the main switch statement was updated as seen below:

Figure 40: Main Switch With NavigatingHome State

```
switch(robotState)
{
    case RobotState_Idle:
        scheduleId = idle();
        if(scheduleId != -1)
            robotState = RobotState_NavigatingToUser;
        break;
    case RobotState_NavigatingToUser:
        result = navigating_to_user(sensorValues);
        if(result == NavigationResult_Complete)
            robotState = RobotState_DeliveringPayload;
        break;
    case RobotState_DeliveringPayload:
        if(delivering_payload(sensorValues, scheduleId))
            robotState = RobotState_NavigatingHome;
        break;
```

```
    case RobotState_NavigatingHome:
        result = navigating_home(sensorValues);
        if(result == NavigationResult_Complete)
            robotState = RobotState_Idle;
        break;
}
```

With this switch statement, all of the previously mentioned states were combined into one state machine in accordance with the state diagram (Figure 9). The Robot would now enter an idle state initially, begin navigating to the user at a scheduled time, wait for a complete delivery once arriving to the user, and finally travel back to its home position and reenter the idle state waiting for a new scheduled time.

4.3.3.5 Stuck

The final state, Stuck, was considered as a sub-state, in that it is not required for the Robot to perform its main functionality. This state is useful for when any errors occur while navigating, as indicated in Section 3.3.3, since navigation can be unpredictable depending on the environment.

An example of this is used can be seen within the following section of code in the navigation function (**Appendix E**):

Figure 41: Stuck State Example

```
if(state == MotionState_Stop)
{
    // if this is the first time we stopped, capture the current time for comparison later
    if(stoppedSnapshot == 0)
        stoppedSnapshot = time_us_64();

    stop();
    // if the amount of time passed since we first snapped the stopped state has reached our
cutoff duration, we're stuck!
    if(has_duration_passed(stoppedSnapshot, STUCK_DURATION))
        result = NavigationResult_Stuck;
}
```

As seen in the code above, if the Robot has stopped a snapshot of the current time is taken using a timer function and a stop command is sent. If the Robot has stopped for a certain duration of time, determined by the STUCK_DURATION variable used in the has _duration_passed function,

it is considered to be in the Stuck state. The has_duration_passed function determines the time the Robot has been stopped for like so:

Figure 42: Has Duration Passed Function

```
bool has_duration_passed(uint64_t snapshot, uint64_t duration)
{
    uint64_t difference = (time_us_64() - snapshot) / 1000; // divided by 1000 to convert to ms
    return difference >= duration;
}
```

The function above simply takes the current time and subtracts it by the time taken at the snapshot given, this is then stored in the difference variable. It must be provided the duration as well, which is the amount of time that this difference is considered to be unacceptable. It then returns true or false depending on if the difference is equal to or greater than the unacceptable duration.

The Stuck state was then implemented within the main switch statement:

Figure 43: Main Switch With Stuck State

```
switch(robotState)
{
    case RobotState_Idle:
        scheduleId = idle();
        if(scheduleId != -1)
            robotState = RobotState_NavigatingToUser;
        break;
    case RobotState_NavigatingToUser:
        result = navigating_to_user(sensorValues);
        if(result == NavigationResult_Complete)
            robotState = RobotState_DeliveringPayload;
        else if (result == NavigationResult_Stuck)
            robotState = RobotState_Stuck;
        break;
    case RobotState_DeliveringPayload:
        if(delivering_payload(sensorValues, scheduleId))
            robotState = RobotState_NavigatingHome;
        break;
    case RobotState_NavigatingHome:
        result = navigating_home(sensorValues);
        if(result == NavigationResult_Complete)
            robotState = RobotState_Idle;
        else if (result == NavigationResult_Stuck)
            robotState = RobotState_Stuck;
        break;
}
```

To summarize all of the states, the Robot is able to execute all of the functionality required for this project from a software perspective with an error failsafe if it ends up getting Stuck. The GitHub repository (Skretteberg & Manhas, 2023b) for the navigation controls provides more details in regards to the code related to these states if more information is required.

5.0 Project Results

Once the overall project was complete, the majority of the functionality was implemented successfully, with some exceptions not being fully implemented. The final result of the Robot can be seen in **Appendix G.**

The navigation portion of the Robot was implemented fully using the main switch case, as seen in section 4.3.3, and the different states were tested individually.

Next, for obstacle detection, the HC-SR04 sensors (ELEC Freaks, n.d.) were all implemented fully. In combination with the navigation states, the Robot was able to successfully stop and navigate around obstacles as necessary.

One of the motor controllers (DFRobot, 2012) was implemented successfully as well, the other one that was intended to be used was not due to a miswiring issue. This caused the component to be replaced, but this did not fix the issue. It was decided to rely on one motor controller instead, which prevented the motors from reversing properly.

The bump sensors for backup object detection were also implemented successfully. If an object is detected behind, the Robot is able to respond appropriately by stopping.

One of the core sensors not implemented within the final design of the Robot were the two Infrared Sensors (DFRobot, 2021a). There were issues with soldering initially, but these issues were solved fairly hastily. Another issue with this sensor was that there was a misunderstanding in terms of the voltage required to run it, and the previously mentioned wiring issue for the motor controllers potentially caused one of the sensors to stop working all together. This is why only one Infrared Sensor is actually connected to the Robot. Despite this issue being resolved however, this sensor still had issues with its initialization. The I^2C read command sent to this sensor is not able to detect the device correctly, the exact reason behind this wasn't fully found due to time constraints.

To conclude, the basic functionality of this Robot was implemented to a slightly inadequate level, in that the Infrared Sensors were not fully implemented. Despite this, the Robot could travel from its starting position to the user at a scheduled time, wait for the user to pick up their medication and put it back on to the weight sensor, then travel back to its starting position.

6.0 Conclusion

Developing a robot takes a lot of separate pieces to come to fruition. If one part is not working, the robot will not work as intended, and this can have unforeseen consequences. Even if all parts work separately, if there is interference of communication, nothing may work at all. It takes a lot of time, effort and planning to make sure processes are working properly and effectively.

The report focuses on the development of an autonomous robot and connecting it to a functional web application. It first went over the designing process, giving an in depth look at all the considerations required for parts. It then followed up with a discussion of the implementation of all the software required to allow the robot to work. It also discusses challenges faced and breaks down the process of how things were developed.

Despite many challenges arising, the consensus of the development was that given a short amount of time to correct some software bugs and a couple hardware issues, the robot could easily become fully complete and functioning as a replacement for a support animal.

References

Adafruit Industries. (n.d.). *Adafruit VL6180X Time of Flight Distance Ranging Sensor (VL6180)*. Adafruit Industries. Retrieved April 17, 2023, from https://www.adafruit.com/product/3316

Atmel. (2009, February). *8-bit Microcontroller with 4/8/16/32K Bytes In-System Programmable Flash* [ATmega328P pinout datasheet]. Retrieved April 17, 2023, from https://www.sparkfun.com/datasheets/Components/SMD/ATMega328.pdf

Atmel. (2015). *8-bit AVR Microcontroller with 32K Bytes In-System Programmable Flash* [ATmega328P datasheet]. Retrieved April 10, 2023, from https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf

Balsamiq Studios. (n.d.). Balsamiq Cloud. https://balsamiq.cloud/

Cafe, T. (2007, June 17). *T.C. Forensic: Article 10 - PHYSICAL CONSTANTS FOR INVESTIGATORS*. TC Forensic. Retrieved April 25, 2023, from https://www.tcforensic.com.au/docs/article10.html

Canadian Sheet Steel Building Institute. (n.d.). *Steel is a non-combustible material and doesn't burn*. CSSBI. Retrieved April 25, 2023, from https://cssbi.ca/mid-rise-construction/fire-safety

Cicolani, J. (2021). *Beginning Robotics with Raspberry Pi and Arduino: Using Python and OpenCV* (2nd ed.). Apress. https://learning.oreilly.com/library/view/beginning-robotics-with/9781484268919/?sso_link=yes&sso_link_from=NAIT

CodeIgniter Foundation. (2022, March 3). *Welcome to CodeIgniter — CodeIgniter 3.1.13 documentation*. CodeIgniter. Retrieved April 25, 2023, from https://codeigniter.com/userguide3/general/welcome.html

CUI Devices. (2023, January 18). *PJ-096H Datasheet - Jacks | Dc Power Connectors*. Retrieved April 17, 2023, from https://www.cuidevices.com/product/resource/pj-096h.pdf

DFRobot. (n.d.). *Solar Power Manager for 12V Lead-Acid Battery V1.0* [DFR0580 datasheet]. Retrieved April 10, 2023, from https://wiki.dfrobot.com/Solar_Power_Manager_For_12V_Lead-Acid_Battery_SKU__DFR0580

DFRobot. (2012). *MD1.3 2A Dual Motor Controller* [DRI0002 datasheet]. Retrieved April 26, 2023, from https://wiki.dfrobot.com/MD1.3_2A_Dual_Motor_Controller_SKU_DRI0002

DFRobot. (2021a). *Fermion: VL6180X ToF Distance Ranging Sensor Breakout Wiki* [SEN0427 datasheet]. Retrieved April 10, 2023, from https://wiki.dfrobot.com/DFRobot_VL6180X_TOF_Distance_Ranging_Sensor_Breakout_Board_SKU_SEN0427

DFRobot. (2021b, September 8). *DFRobot_VL6180X* [SEN0427 Arduino library for VL6180X]. https://github.com/DFRobot/DFRobot_VL6180X

Digital Ocean. (n.d.). DigitalOcean | The Cloud for Builders. https://www.digitalocean.com/

ELEC Freaks. (n.d.). *Ultrasonic Ranging Module HC - SR04* [HC-SR04 datasheet]. Retrieved April 10, 2023, from https://cdn.sparkfun.com/datasheets/Sensors/Proximity/HCSR04.pdf

Fairhead, H. (2022). Simple Web Client. In *Programming the Raspberry Pi Pico in C*. I/O Press. https://www.i-programmer.info/programming/hardware/15838-the-picow-in-c-simple-web-client.html

Google. (n.d.). *Rally - Material Design*. Material Design 2. Retrieved April 25, 2023, from https://m2.material.io/design/material-studies/rally.html

Govers, F. X. (2018). *Artificial Intelligence for Robotics*. Packt Publishing. https://learning.oreilly.com/library/view/artificial-intelligence-for/9781788835442/

Joseph, L. (2015). *Learning Robotics Using Python* (1st ed.). Packt Publishing. https://learning.oreilly.com/library/view/learning-robotics-using/9781783287536

Kitware, Inc. and Contributors. (2023). *Step 2: Adding a Library*. CMake. Retrieved April 10, 2023, from https://cmake.org/cmake/help/latest/guide/tutorial/Adding%20a%20Library.html

McComb, G. (2018). *How to make a robot* (First ed.). Maker Media. https://learning.oreilly.com/library/view/how-to-make/9781680455175/

Microchip Technology Inc. (n.d.). *Microchip Studio for AVR® and SAM Devices*. Microchip Technology. Retrieved April 18, 2023, from https://www.microchip.com/en-us/tools-resources/develop/microchip-studio

Microchip Technology Inc. (2022, June 22). *MCP23017/MCP23S17 - 16-Bit I/O Expander with Serial Interface* [MCP23017 datasheet]. Retrieved April 20, 2023, from https://ww1.microchip.com/downloads/aemDocuments/documents/APID/ProductDocuments/DataSheets/MCP23017-Data-Sheet-DS20001952.pdf

Microsoft. (n.d.). *Visual Studio Code*. https://code.visualstudio.com/

Omron. (2021). *D3V Miniature Basic Switch Reliable Basic Switch with External Lever* [D3V-163-1A5 datasheet]. Retrieved April 10, 2023, from https://www.mouser.ca/datasheet/2/307/en_d3v_01-595238.pdf

Plastics International. (n.d.). *PVC (Polyvinyl chloride, Type I or Type II)*. Plastics International. Retrieved April 25, 2023, from https://www.plasticsintl.com/shop-by-material/pvc

Power Kingdom. (n.d.). *Sealed Lead Acid Battery* [12 V Lead-Acid Battery datasheet]. Retrieved April 26, 2023, from http://www.mantech.co.za/Datasheets/Products/PS_SERIES.pdf

PPG. (n.d.). *Paint Colors For Autism*. PPG Paints. https://www.ppgpaints.com/paint-colors-for-autism

Qorvo. (n.d.). *DWM1001-DEV Ultra-Wideband (UWB) Transceiver Development Board*. https://www.qorvo.com/products/p/DWM1001-DEV#documents

Rapp, A., & Donielle Berge. (n.d.). ColorSafe. http://colorsafe.co/

Raspberry Pi Ltd. (2023a, March 2). *Getting started with Raspberry Pi Pico* [C/C++ development with Raspberry Pi Pico and other RP2040-based microcontroller boards]. Retrieved April 10, 2023, from https://datasheets.raspberrypi.com/pico/getting-started-with-pico.pdf

Raspberry Pi Ltd. (2023b, March 2). *Raspberry Pi Pico W Datasheet* [An RP2040-based microcontroller board with wireless]. Retrieved April 10, 2023, from https://datasheets.raspberrypi.com/picow/pico-w-datasheet.pdf

Savitri. (n.d.). *Does Wood Conduct Electricity?* Techiescientist. Retrieved April 25, 2023, from https://techiescientist.com/does-wood-conduct-electricity/

Skretteberg, K., & Manhas, N. (2023a, March 10). *arven-sensor-controls*. https://github.com/KiaSkretteberg/arven-sensor-controls

Skretteberg, K., & Manhas, N. (2023b, March 29). *arven-navigation-controls*. https://github.com/KiaSkretteberg/arven-navigation-controls

Skretteberg, K., & Suitor, S. (2023, April 24). *arven-app*. Github. https://github.com/KiaSkretteberg/arven-app

STMElectronics. (2018, July 26). *VL6180X basic ranging application note*. https://www.st.com/resource/en/application_note/an4545-vl6180x-basic-ranging-application-note-stmicroelectronics.pdf

Texas Instruments. (2015, July). *LT1054 Switched-Capacitor Voltage Converters With Regulators* [LT1054 datasheet]. Retrieved April 17, 2023, from https://www.ti.com/lit/ds/symlink/lt1054.pdf

Tomaszuk, P., Łukowska, A., Rećko, M., Dzierżek, K., & Straszyński, P. (2019, June 20). *Active wheel speed control to avoid lifting the swingarms in rocker-bogie suspension*. IEEExplore. Retrieved April 25, 2023, from https://ieeexplore.ieee.org/document/8740821

Uneo. (2014, July). *General Purpose 0.3cm-diameter Ultra-thin Flexible Pressure Sensor* [GD03 datasheet]. Retrieved April 10, 2023, from https://www.uneotech.com/uploads/product_download/tw/GD03-10N%20ENG.pdf

Venngage Inc. (n.d.). *Accessible Color Palette Generator | WCAG Compliant*. Venngage. Retrieved April 25, 2023, from https://venngage.com/tools/accessible-color-palette-generator

Verma, A., Yadav, C., Singh, B., Gupta,, A., Mishra, J., & Saxena, A. (2017, May). *Design of Rocker-Bogie Mechanism*. | International Journal of Innovative Science and Research Technology. Retrieved April 25, 2023, from https://ijisrt.com/wp-content/uploads/2017/05/Design-of-Rocker-Bogie-Mechanism-1.pdf

Vorwerk. (2018, May 14). *How fast does the robot vacuum cleaner travel?* Retrieved April 10, 2023, from https://support.vorwerk.com/hc/en-us/articles/360000896605-How-fast-does-the-robot-vacuum-cleaner-travel-

Walther, S. (2022, July 11). *Understanding Models, Views, and Controllers (C#)*. Microsoft Learn. Retrieved April 26, 2023, from https://learn.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/overview/understanding-models-views-and-controllers-cs

Appendix A

Web Application UI Mockups

Figure A1: Web Application UI Mockup of Setup Page

Figure A2: Web Application UI Mockup of Login Page

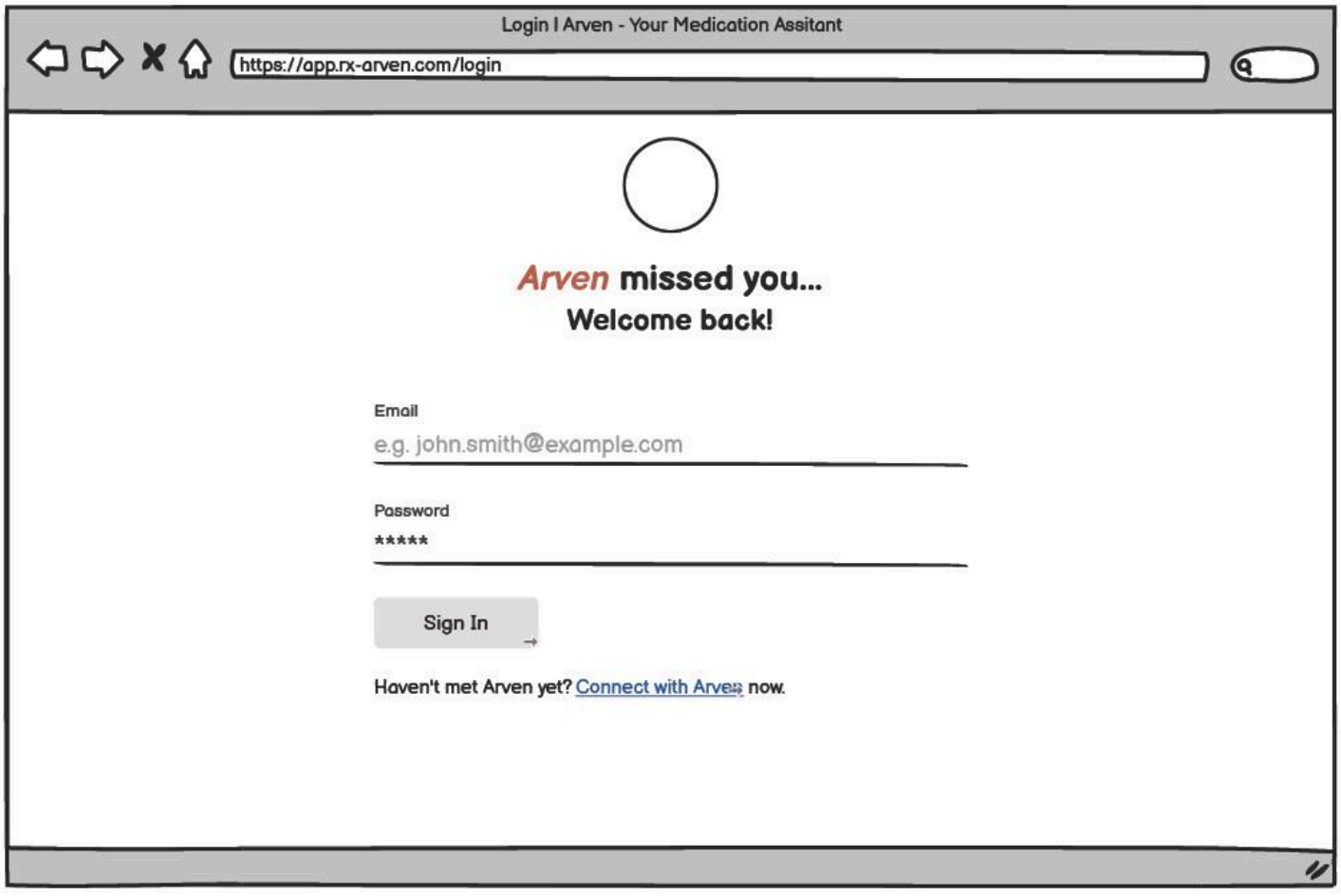


Figure A3: Web Application UI Mockup of Dashboard Page

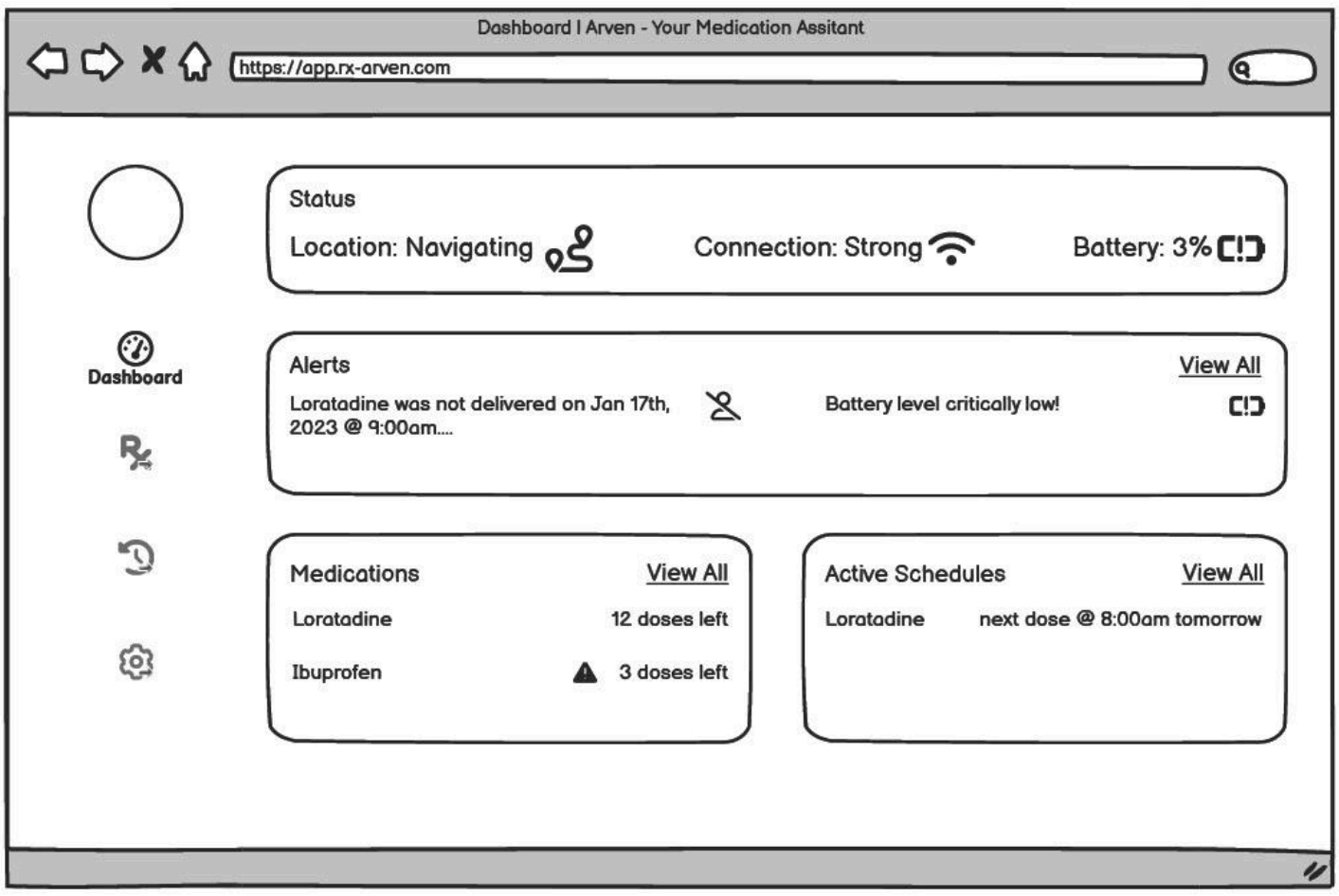


Figure A4: Web Application UI Mockup of Configuration Page

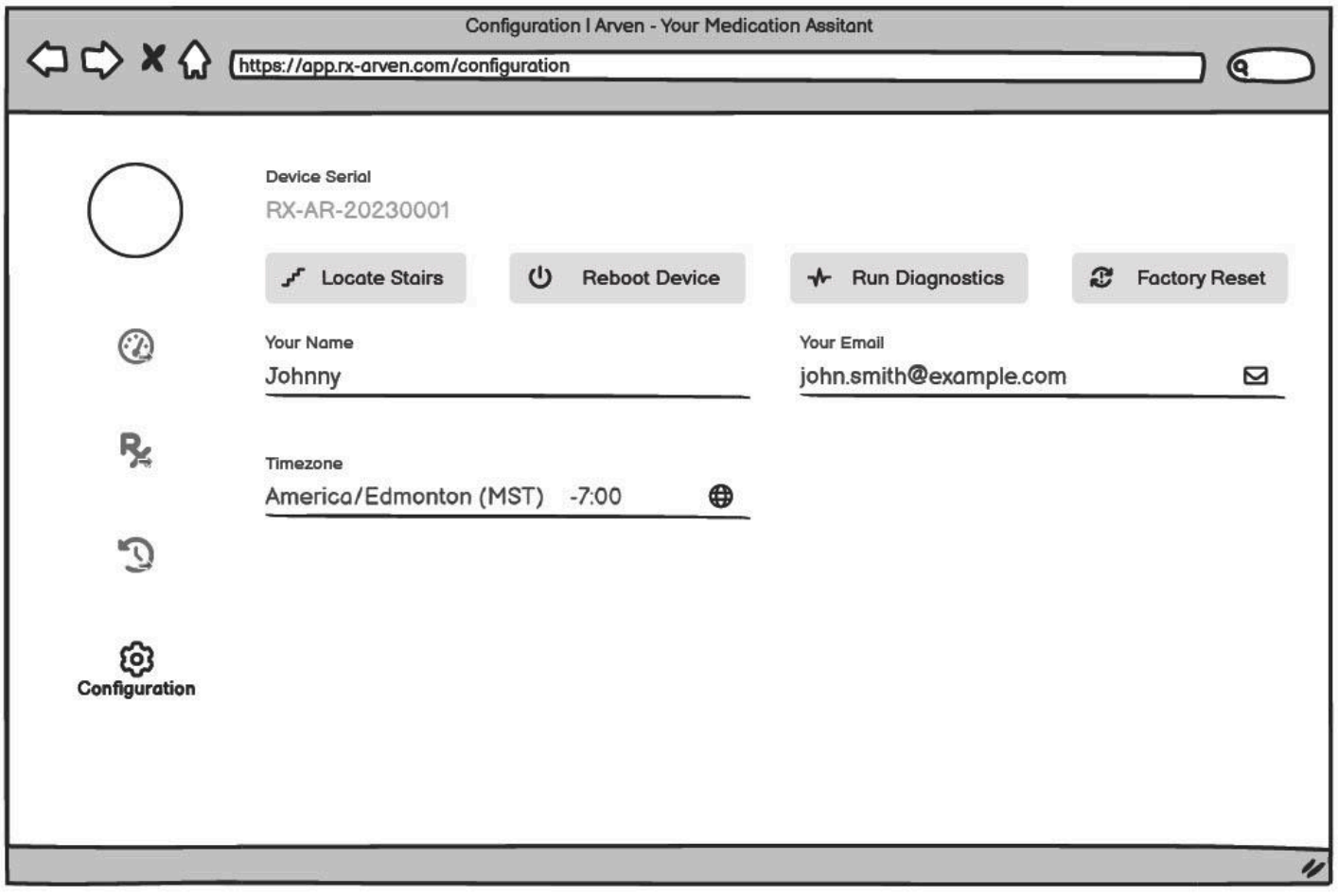


Figure A5: Web Application UI Mockup of History Page

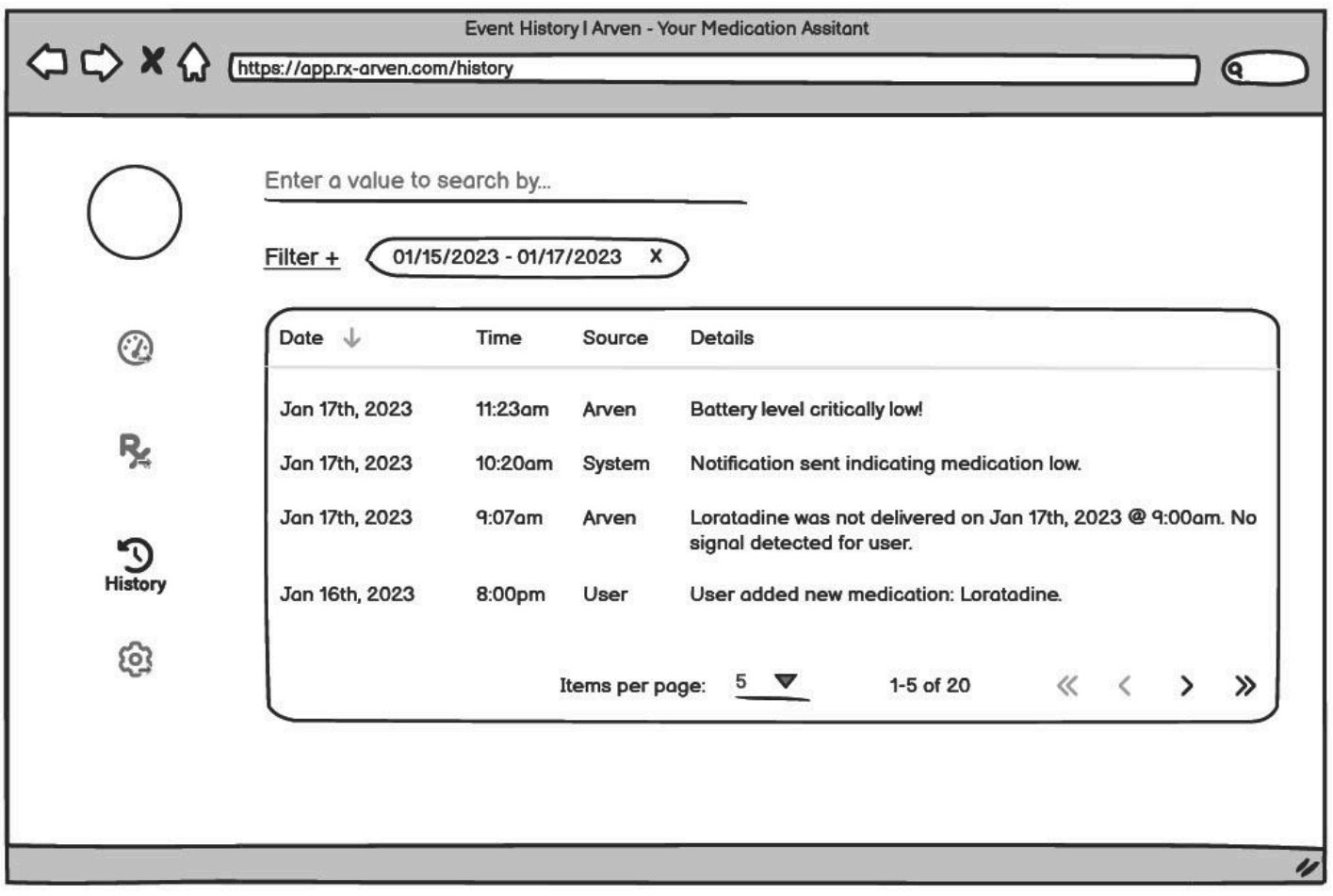


Figure A6: Web Application UI Mockup of Medications Page

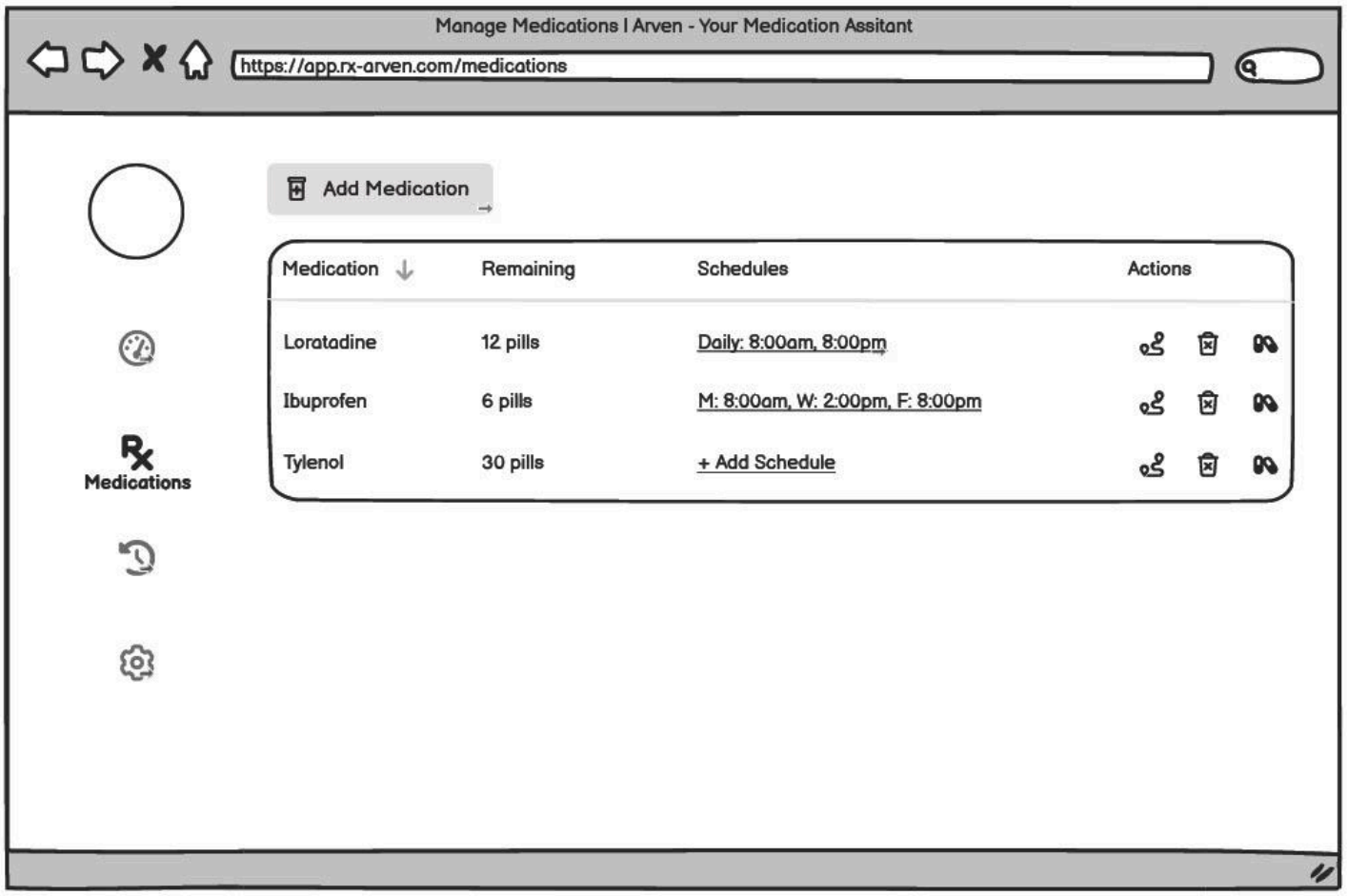


Figure A7: Web Application UI Mockup of Schedules Modal

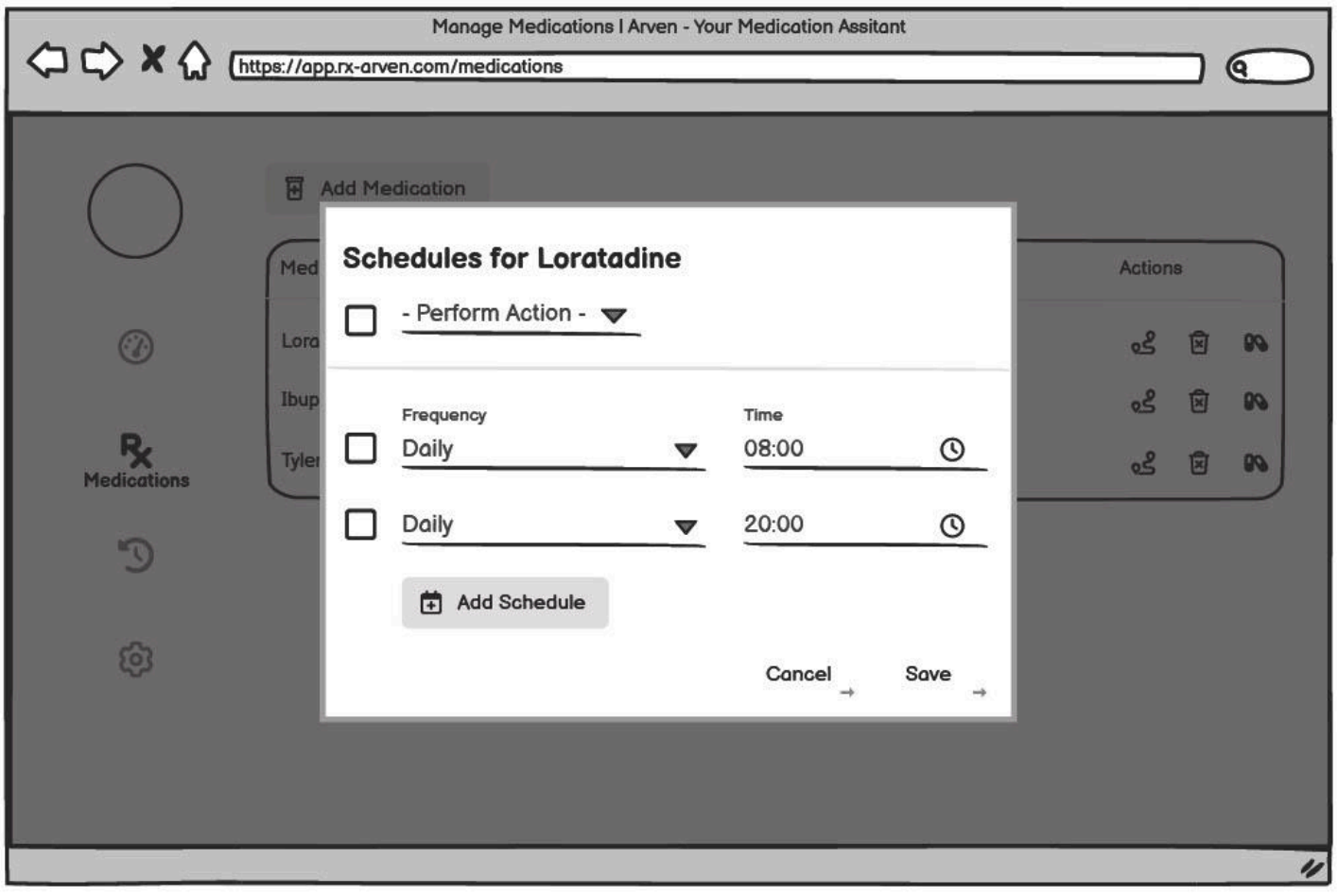
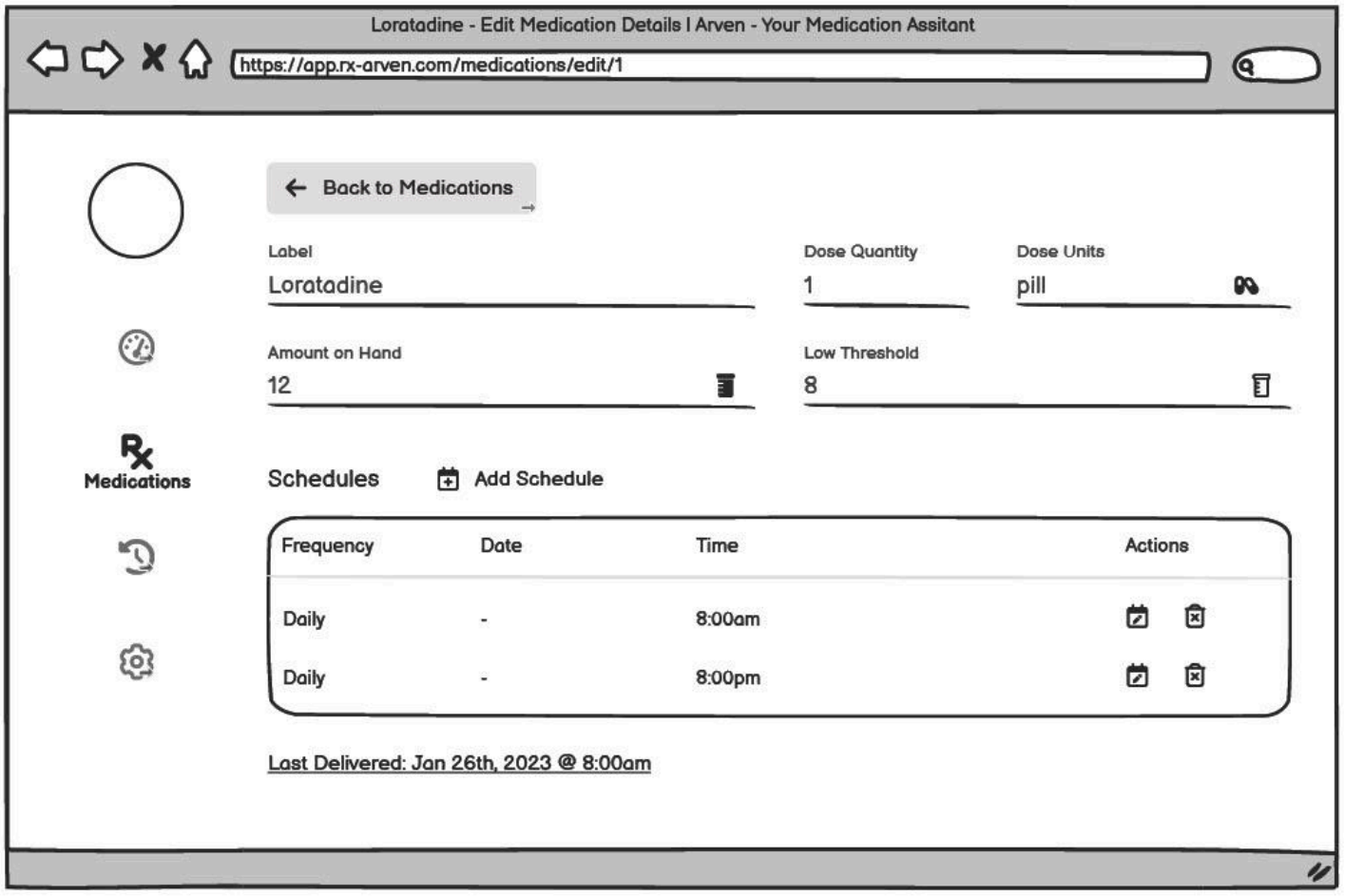


Figure A8: Web Application UI Mockup of Add/Edit Medication Page



Appendix B

Circuit Schematics

Figure B1: Arven PCB Schematic

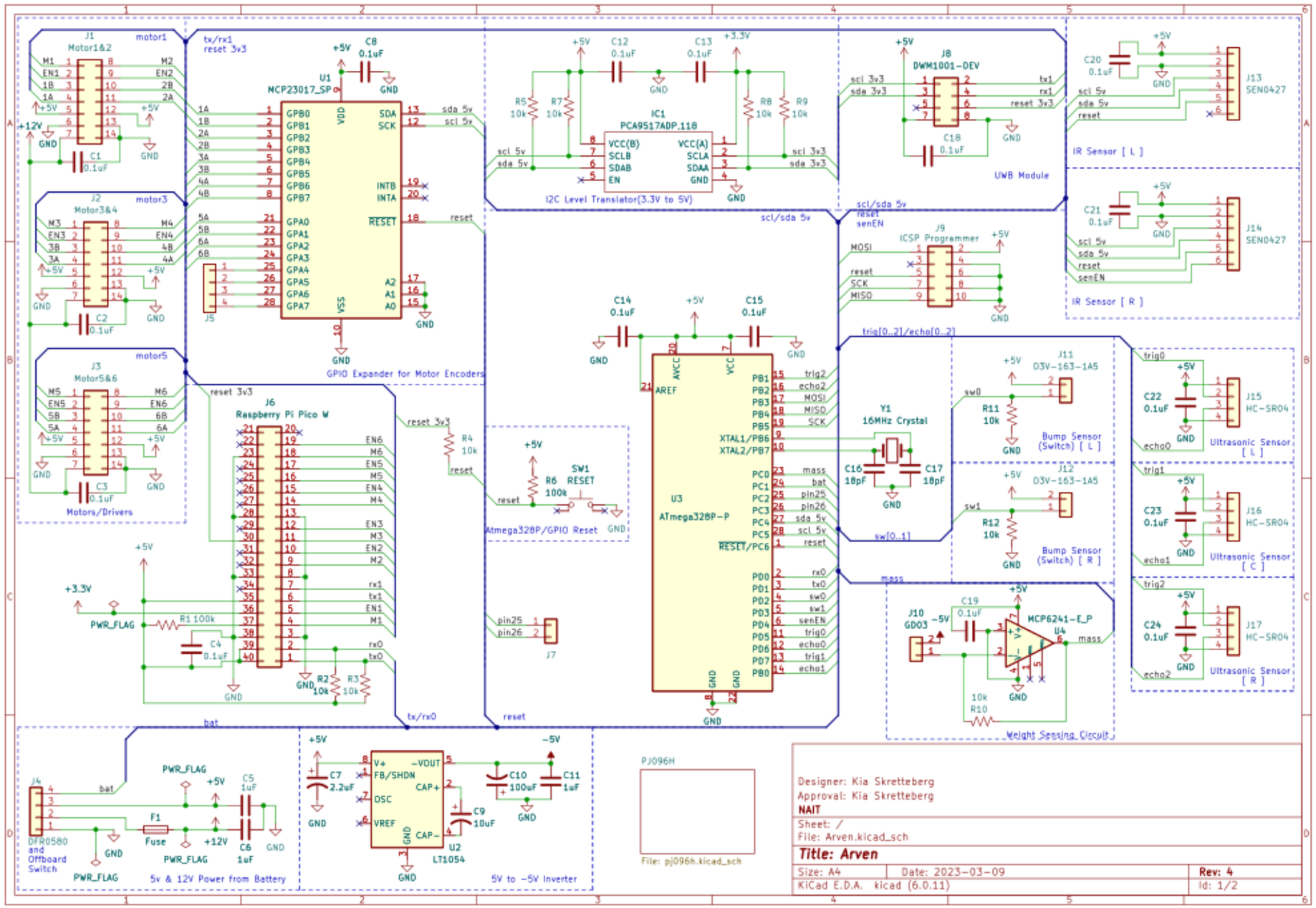
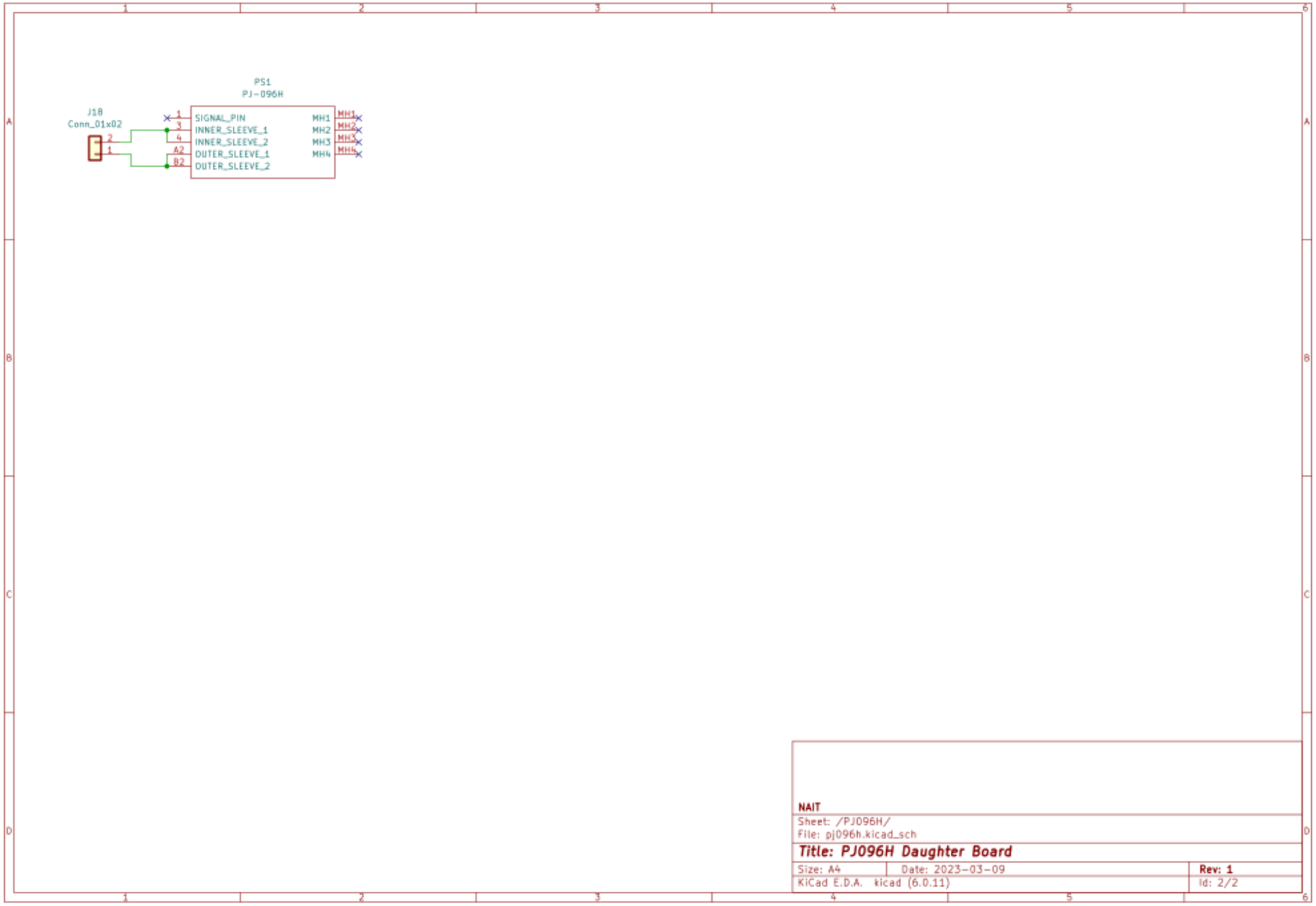


Figure B2: PJ-096H Daughter Board Schematic



Appendix C

PCB Designs

Figure C1: Top of Arven PCB and PJ-096H Daughter Board

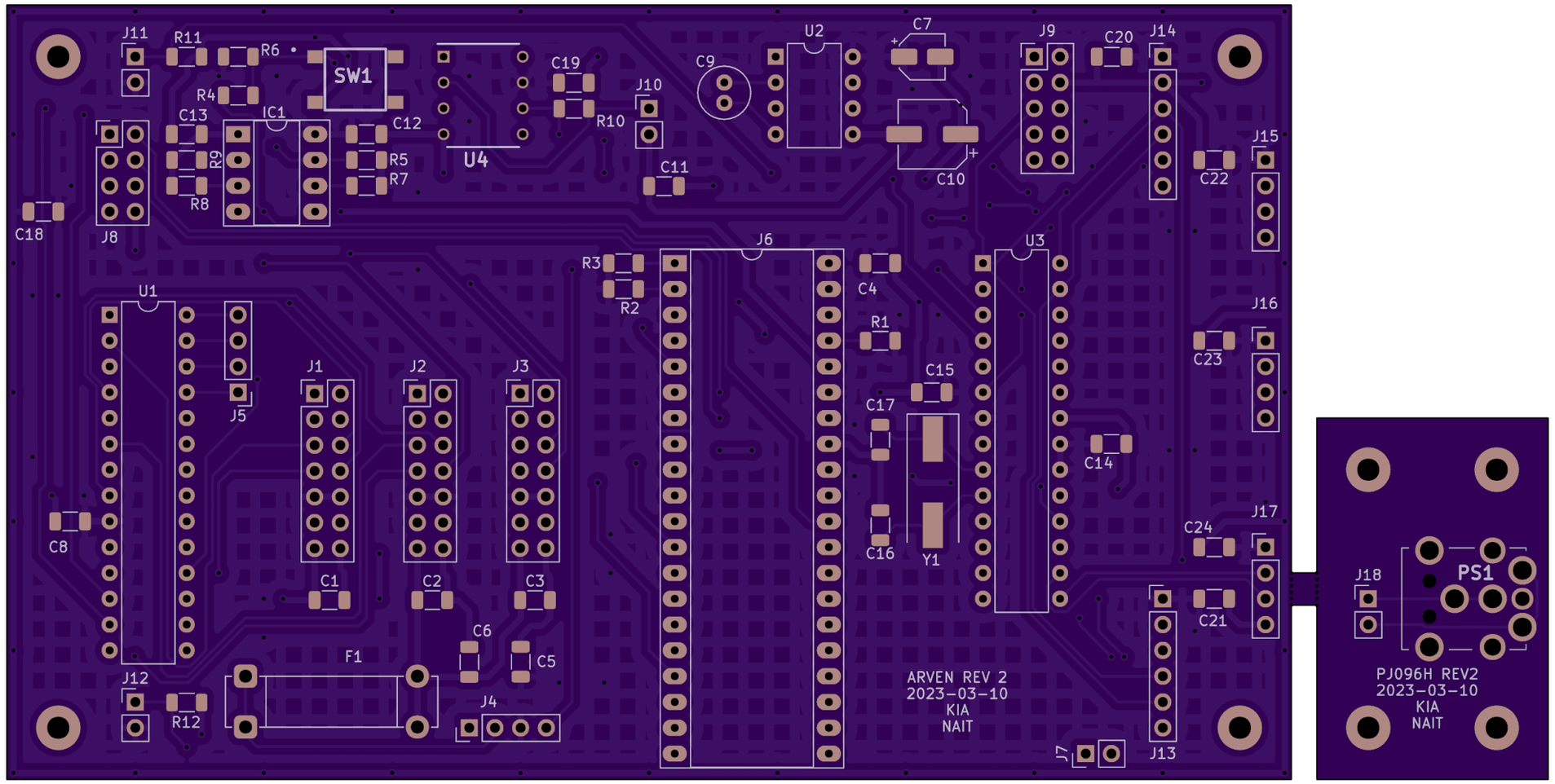


Figure C2: Bottom of Arven PCB and PJ-096H Daughter Board

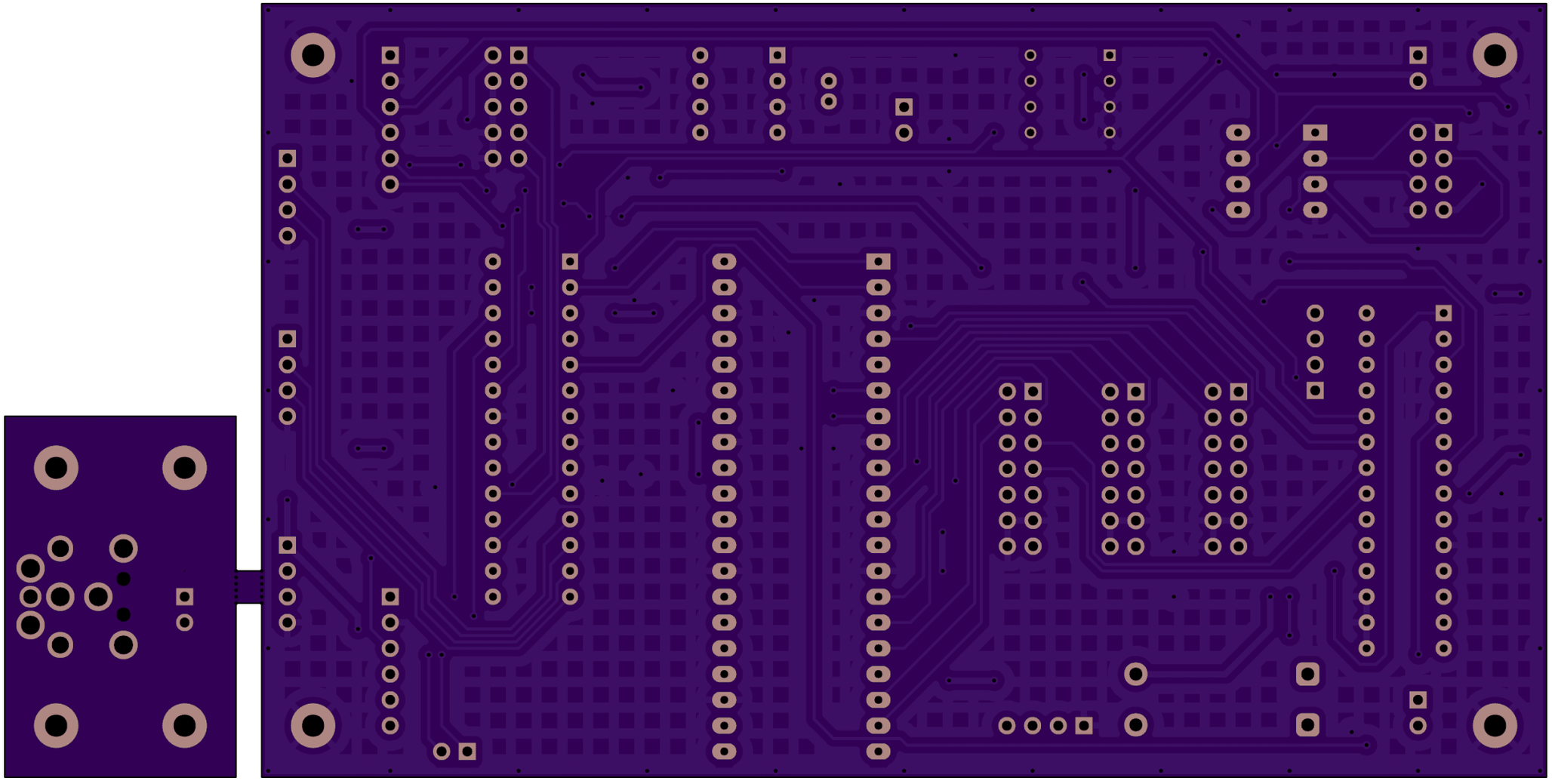
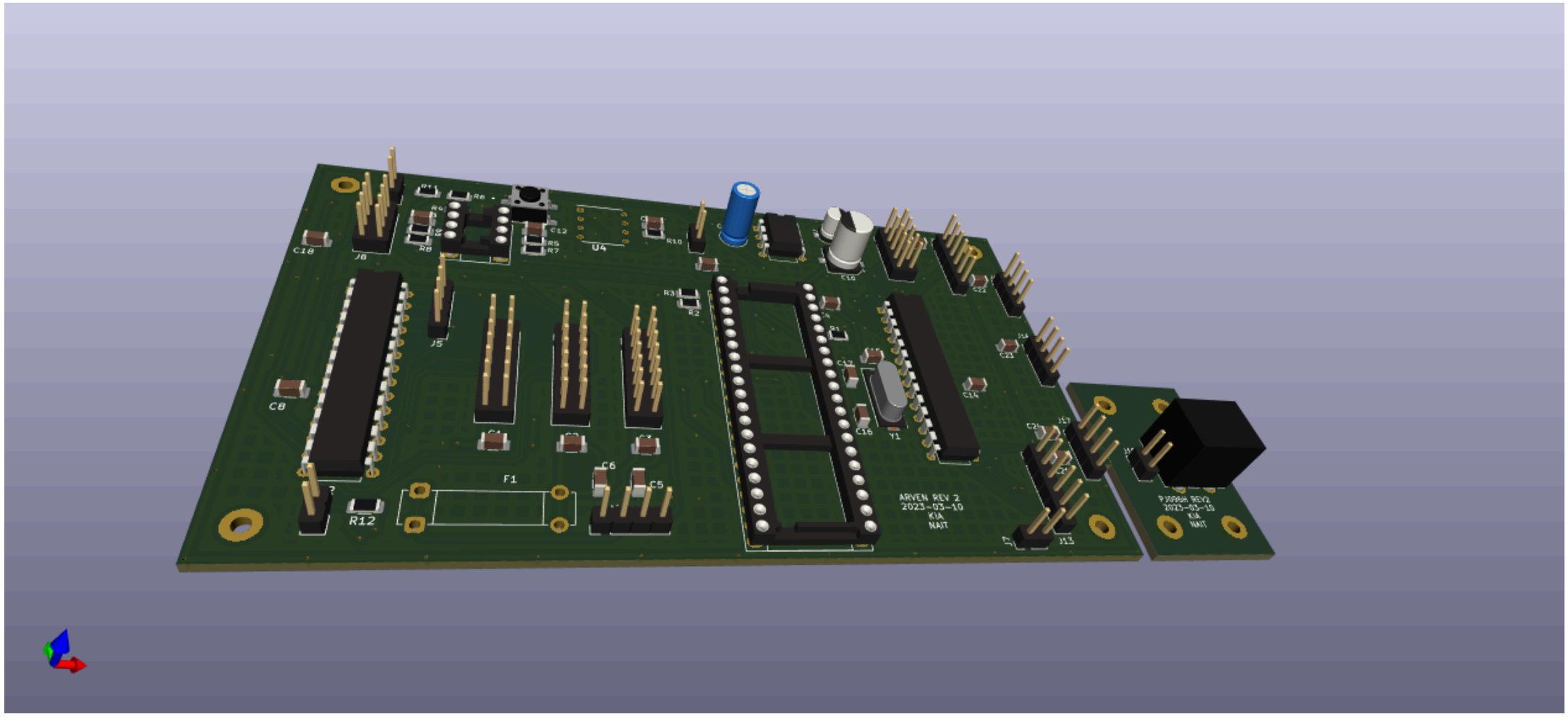


Figure C3: 3D Render of Arven PCB and PJ-096H Daughter Board



Appendix D

Bill of Materials

Item	Part	Qty	Ref Des	Description	Cost	Extended Cost
1	N/A	1	N/A	ARVEN PCB	$90.00	$90.00
2	C1206C104M5UAC	16	C1, C2, C3, C4, C8, C12, C13, C14, C15, C18, C19, C20, C21, C22, C23, C24	100 NF, CERAMIC, SMD1206, CAPACITOR	$0.14	$2.21
3	CL31B104KBCNNND	3	C5, C6, C11	1 UF, CERAMIC, SMD1206, CAPACITOR	$0.15	$0.46
4	C1206C180J5GAC	2	C16, C17	18 PF, CERAMIC, SMD1206, CAPACITOR	$0.17	$0.33
5	865060640002	1	C7	2.2 UF, ELECTROLYTIC, SMD, CAPACITOR	$0.25	$0.25
6	860010372001	1	C9	10 UF/63V, ELECTROLYTIC, RADIAL LEADS, CAPACITOR	$0.14	$0.14
7	865080142007	1	C10	100 UF, ELECTROLYTIC, SMD, CAPACITOR	$0.29	$0.29
8	CRGV1206J100K	2	R1, R6	100K OHM, 1/4W, 5%, 1206	$0.03	$0.05
9	CRGH1206F10K	10	R2, R3, R4, R5, R7, R8, R9, R10, R11, R12	10K OHM, 1/2W, 1%, 1206	$0.26	$2.62
10	PTS645SM43SMTR92 LFS	1	SW1	SMT SPST NO, 4 PIN	$0.41	$0.41
11	FOXSDLF/160-20	1	Y1	HC49-SD, SMD, CRYSTAL	$0.41	$0.41
12	01240041H	1	F1	FUSE HOLDER 5MM X 20MM, THT	$4.26	$4.26
13	BK1/S500-5-R	1	N/A	5A, 5MM X 20MM 250V FUSE	$0.51	$0.51
14	PCA9517ADP	1	IC1	I2C BUS REPEATER	$2.17	$2.17
17	MCP23017	1	U1	I2C I/O EXPANDER, 16 GPIO	$2.35	$2.35
16	LT1054	1	U2	SWITCHED-CAPACITOR VOLTAGE CONVERTER WITH REGULATOR	$3.81	$3.81
15	ATMEGA38-PU	1	U3	ATMEGA328P MICROCONTROLLER	$4.50	$4.50
18	MCP6241	1	U4	OP AMP	$0.61	$0.61
19	PH2-10-UA	3	J1, J2, J3, J8, J9	2 X 30 PIN HEADER, 2.54MM, VERTICAL	$0.25	$0.75
20	PH1-20-UA	3	J4, J5, J15, J16, J17, J7, J10, J11, J12, J18, J13, J14, N/A	1 x 20 PIN HEADER, 2.54MM, VERTICAL	$0.47	$1.41
22	SC0918	2	J6, N/A	RASPBERRY PI PICO W	$8.85	$17.70
26	PJ-096H	1	PS1	7.55MM x 5MM BARREL JACK	$2.68	$2.68
27	SEN0427	2	N/A	VL6180X IR SENSOR MODULE	$9.52	$19.04
28	GD03	1	N/A	10N FORCE SENSING RESISTOR	$9.90	$9.90

29	D3V-163-1A5	2	N/A	SPDT SNAP ACTION SWITCH, LONG LEVER ACTUATOR, CHASSIS MOUNT	$4.86	$9.72
30	DRF0580	1	N/A	12V LEAD ACID BATTERY CHARGING MODULE, 5V AND 12V OUTPUT	$27.46	$27.46
31	HC-SR04	3	N/A	ULTRASONIC SENSOR MODULE	$2.33	$6.99
32	DWM1001-DEV	5	N/A	DWM1001, UWB MODULE	$26.91	$11.65
33	DRI0002	2	N/A	6V - 12V, 2A, DUAL MOTOR CONTROLLER DRIVER	$23.53	$47.06
34	36GP-555-27-EN	6	N/A	36D, 12V, 220RPM PLANETARY GEARMOTOR W/ 2 CHANNEL ENCODER, 1:27 GEAR REDUCTION	$17.98	$107.88
35	JS202011CQN	1	N/A	DPST CHASSIS MOUNT SWITCH	$7.23	$7.23
36	970150321	27	N/A	M3 X 15MM ALUMINUM F-F STANDOFF	$0.51	$13.80
37	97791003111	27	N/A	M3 X 10MM NC MACHINE SCREW	$0.37	$10.07
38	12VLEAD-ACIDBATTERY	1	N/A	12V LEAD-ACID BATTERY	$13.65	$13.65
39	18650	5	N/A	3.7 V LITHIUM-ION BATTERY	$17.80	$88.99
40	1043	5	N/A	3.7 V LITHIUM-ION BATTERY HOLDER	$4.32	$21.60

Appendix E

Navigate Function

```
NavigationResult navigate(struct AtmegaSensorValues sensorValues, struct DWM1001_Position
destinationPosition)
{
    NavigationResult result = NavigationResult_Incomplete;
    static uint64_t stoppedSnapshot = 0;
    static long lastXDiff = 0;
    static long lastYDiff = 0;

    MotionState state = interpret_sensors(sensorValues);
    if(state == MotionState_Stop)
    {
        // if this is the first time we stopped, capture the current time for comparison later
        if(stoppedSnapshot == 0)
            stoppedSnapshot = time_us_64();

        stop();
        // if the amount of time passed since we first snapped the stopped state has reached our
cutoff duration, we're stuck!
        if(has_duration_passed(stoppedSnapshot, STUCK_DURATION))
            result = NavigationResult_Stuck;
    }
    else
    {
        //reset the stopped snapshot so it can be reinitialized later
        stoppedSnapshot = 0;
        if(!robotPosition.set || time_us_64() >= next_robot_request)
        {
            // rearm to request again
            next_robot_request = time_us_64() + ROBOT_REQUEST_DURATION;

            struct DWM1001_Position position = dwm1001_request_position();

            if(position.set)
            {
                robotPosition.x = position.x;
                robotPosition.y = position.y;
                robotPosition.z = position.z;
                robotPosition.set = position.set;

                printf("\nrobotPosition: x:%d y:%d z:%d", robotPosition.x, robotPosition.y,
robotPosition.z);
            }
        }

        if(robotPosition.set && destinationPosition.set)
        {
            long xDiff = robotPosition.x - destinationPosition.x;
            long yDiff = robotPosition.y - destinationPosition.y;
            printf("\nxDiff: %d, yDiff: %d", xDiff, yDiff);
            //long zDiff = robotPosition.z - destinationPosition.z;
            //basically we would want to find the difference between the two points, so
            //we would take the absolute value of the difference between the two since we
            //don't care about magnitude when it comes to how close they are to each other
            if (robotPosition.set && destinationPosition.set &&
```

```
    abs(xDiff) >= 300 || abs(yDiff) >= 300)
{
     switch(state)
     {
         case MotionState_Reverse:
         case MotionState_TurnRight:
         case MotionState_TurnLeft:
             act_on_motion_state(state);
             break;
         case MotionState_ToBeDetermined:
             //TODO: Decide what this should do
             if(xDiff < 0){
                 turn_right();
             } else{
                 turn_left();
             }
         case MotionState_Forward:
             // we got further away, so turn around
             if(xDiff > lastXDiff) {
                 // turn right for 5 ms
                 turn_right();
                 sleep_ms(5);
                 // then go forward
                 act_on_motion_state(state);
             // we got closer, so keep going
             } else if(xDiff < lastXDiff ) {
                 act_on_motion_state(state);
             }
             else
             {
                 act_on_motion_state(state);
             }
             break;
         }
         lastXDiff = xDiff;
         lastYDiff = yDiff;
     }
     else
     {
         result = NavigationResult_Complete;
         stop();
     }
    }
 }
 return result;
}
```

Appendix F

Delivery Method

```
bool delivering_payload(struct AtmegaSensorValues sensorValues, int scheduleId)
{
    static DeliveryState state = DeliveryState_WaitingRemoval;
    static uint64_t loadStateSnapshot = 0;
    bool complete = 0;
    // Check if we currently have something on the weight sensor
    Weight_LoadState loadState = Weight_CheckForLoad(sensorValues.Weight);

    // take asnapshot of the current time if the snapshot is 0
    if(loadStateSnapshot == 0)
        loadStateSnapshot = time_us_64();

    switch(state)
    {
        case DeliveryState_WaitingRemoval:
            if(loadState == Weight_LoadNotPresent)
            {
                if(has_duration_passed(loadStateSnapshot, WEIGHT_DURATION))
                {
                    state = DeliveryState_Removed;
                }
            }
            else
            {
                // reset the snapshot to re-check later
                loadStateSnapshot = 0;
            }
        break;
    case DeliveryState_Removed:
        if(loadState == Weight_LoadPresent)
        {
            if(has_duration_passed(loadStateSnapshot, WEIGHT_DURATION))
            {
                state = DeliveryState_Complete;
            }
        }
        else
        {
            // reset the snapshot to re-check later
            loadStateSnapshot = 0;
        }
        break;
    case DeliveryState_Complete:
        if(loadState == Weight_LoadPresent)
        {
            if(has_duration_passed(loadStateSnapshot, WEIGHT_DURATION))
            {
                printf("\ndose taken");
                state = DeliveryState_WaitingRemoval;
                web_request_log_delivery(scheduleId);
                complete = 1;
            }
        }
        else
        {
            // reset the snapshot to re-check later
```

```
                loadStateSnapshot = 0;
            }
            break;
    }

    return complete;
}
```

Appendix G

Final Robot Assembly

Figure G1: Front View of Robot

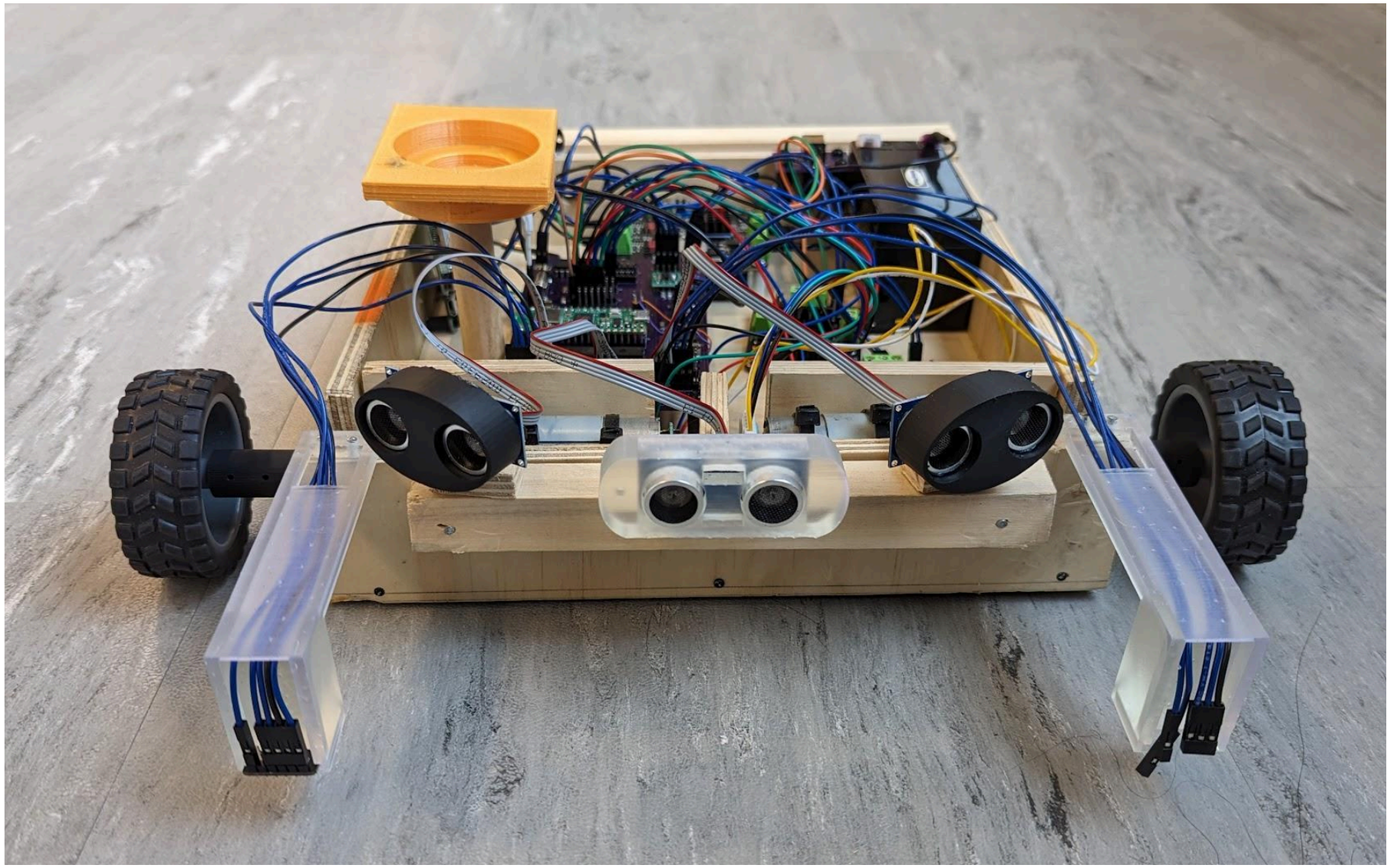


Figure G2: BackView of Robot

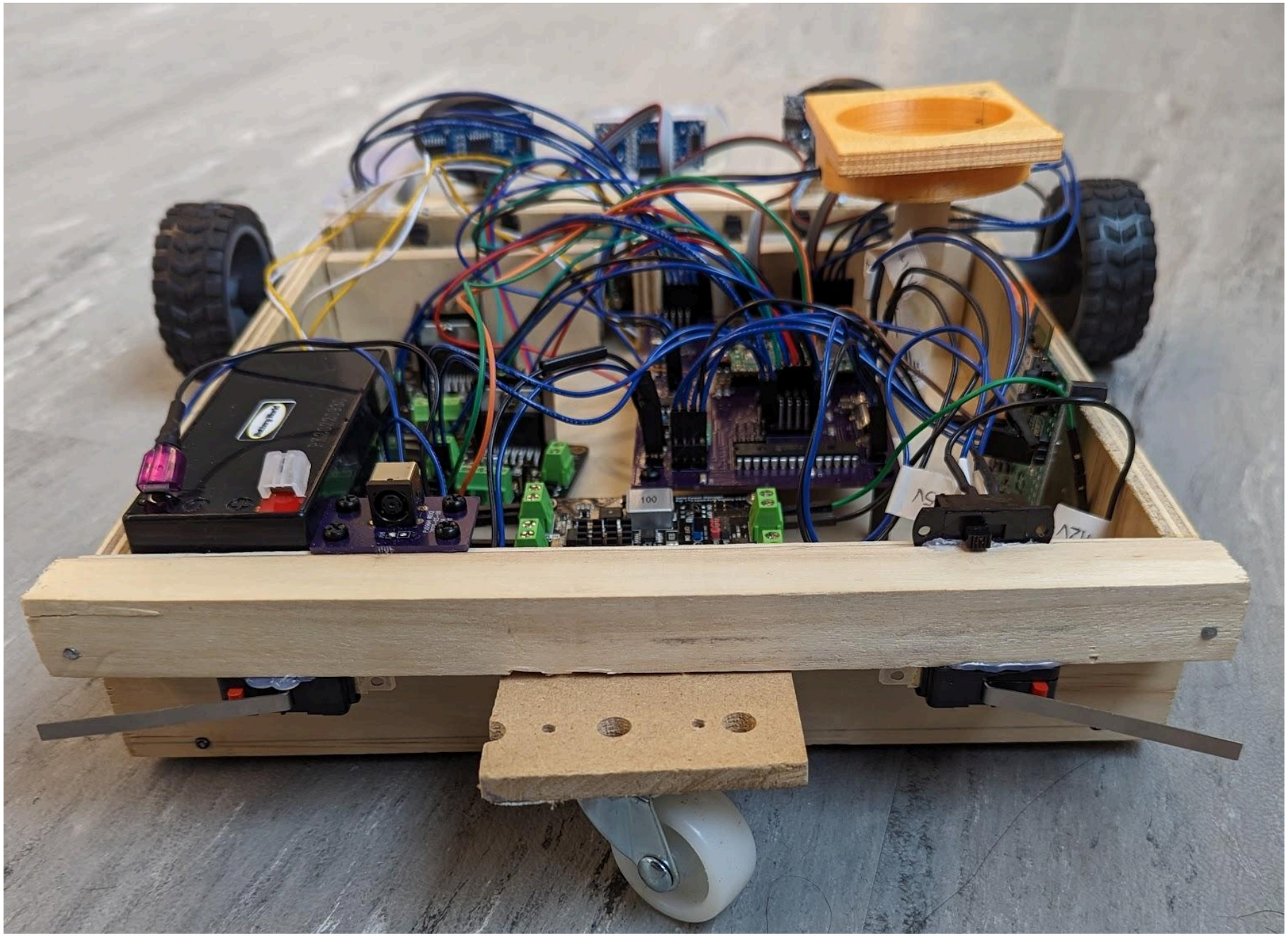


Figure G3: Top View of Robot