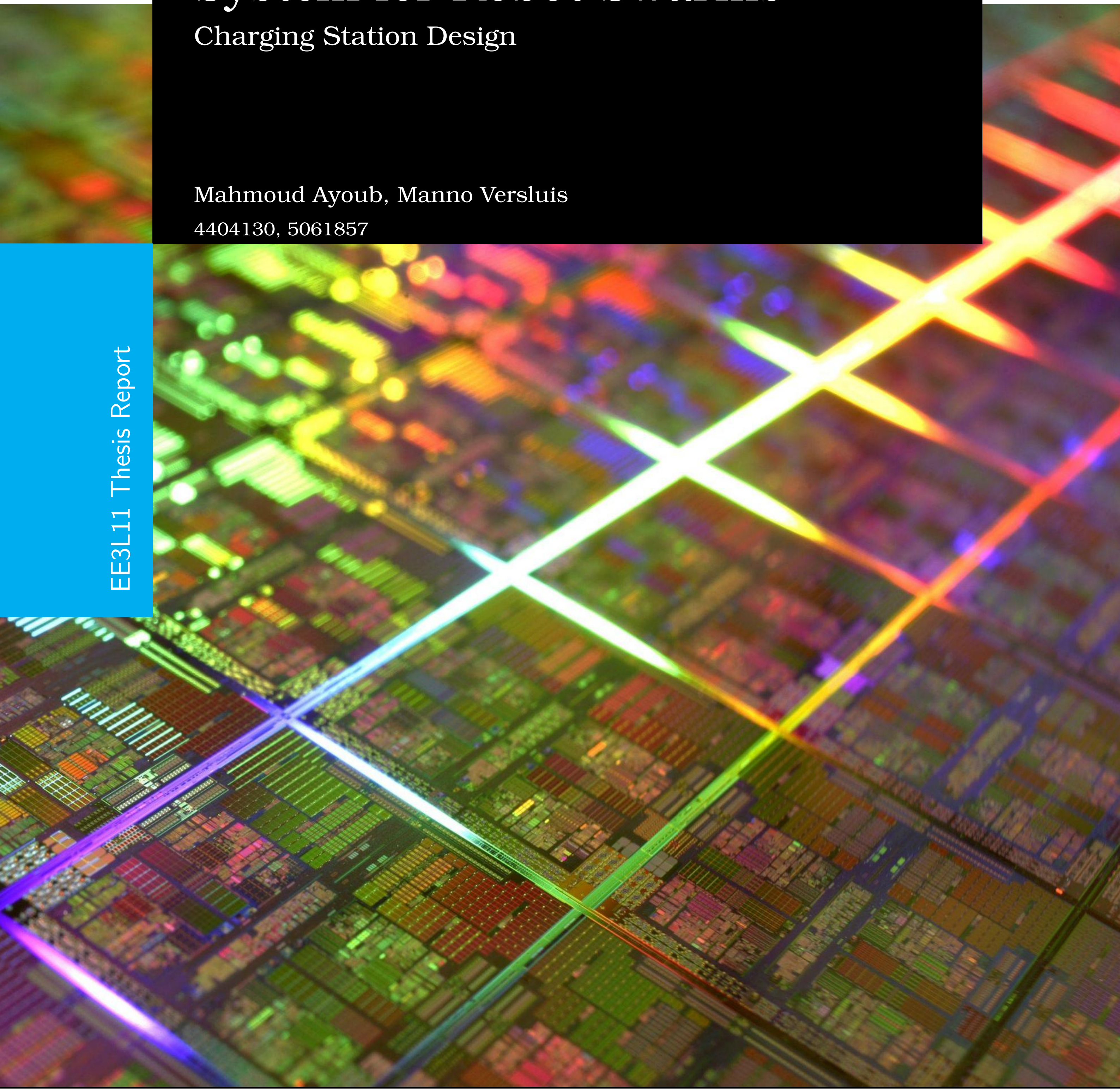
Autonomous Wireless Charging System for Robot Swarms

Charging Station Design

Mahmoud Ayoub, Manno Versluis
4404130, 5061857

EE3L11 Thesis Report



Delft University of Technology

Autonomous Wireless Charging System for Robot Swarms

Charging Station Design

EE3L11 Thesis Report

Mahmoud Ayoub, Manno Versluis
4404130, 5061857

June 20, 2023

Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS)
Delft University of Technology

TU Delft Delft University of Technology

Delft University of Technology

The undersigned hereby certify that they have read and recommend to the Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS) for acceptance a thesis entitled

Autonomous Wireless Charging System for Robot Swarms

by

Mahmoud Ayoub, Manno Versluis

in partial fulfillment of the requirements for the degree of

EE3L11 Thesis Report

Dated: June 20, 2023

Supervisor(s): ______________________

Reader(s): ______________________

Abstract

This project focuses on building a wireless charging station for the Lunar Zebro robots and building upon the existing functionalities by matching the interfacing voltage and current requirements for the wireless power transceiver.

Table of Contents

Introduction

1-1 Robot charging

In order to stay functional, robots need to stay charged. To optimise productivity, it is desirable to keep robots charged and thus operating, as much as possible. Since a lot of robots can't connect or disconnect themselves to or from a charger, or go to and from their charging station on their own, someone else needs to do this for the robots. Having someone move robots to a charging station and then connect and disconnect the robots to and from the charging station will cost time and money. It will also lead to inefficiencies, since it will take time to find out that a robot needs to be charged and the robots will not be removed from the charging station the moment they are recharged, which is undesirable.

A possible solution to this is to have the robots navigate to the charging station on their own and then charge wirelessly, thus not needing to connect or disconnect to the charging station. This way, no one needs to do this for the robots, saving time and money. It also leads to reduced inefficiencies, since the robots can go to and from the charging station when needed instead of having to wait for a human to arrive and then do so for the robot.

Designing this possible solution for the Duckiebot, the robot shown in figure 1-1, is the goal of the Bachelor Final Project this thesis is a part of. The problem was split into three parts for the three different subgroups:

Figure 1-1: A Duckiebot robot that was used for the project

1. The Wireless Charging Hardware subgroup: The hardware needed to perform the wireless charging of the Duckiebots on both the Duckiebots themselves and on the charging pads in the charging station that the Duckiebots can charge from.

2. The Charging Park Design subgroup: The traffic control inside the charging station needed for the robots navigate to and from the charging pads and an orderly way as well as prioritisation in

case more Duckiebots want to charge at once than there are charging pads.

3. The Robot Control and Navigation subgroup: Controlling the robots and ensuring the robots are capable of navigating to and from the charging station and can navigate inside the charging station based on communication with the charging station.

This thesis is about the traffic control and prioritisation inside the charging station part of the Bachelor Final Project.

1-2 Document structure

Chapter 2 specifies the programme of requirements for the Charging Park Design subgroup. In chapter 3, the choices that were made are explained and supported. The theory involving some of those choices is explained in chapter 4. In chapter 5 the designed algorithms and the hardware needed to implement it is explained. In chapter 6, the developed algorithm was verified and the results were shown. Finally, in chapter 8 a conclusion is drawn and recommendation were given for future development.

Programme of requirements

In order for the charging station part of the project to be considered a success, a number of requirements must be satisfied. The requirements needed and the assumptions made will be listed here.

2-1 Assumptions

After discussion with other subgroups and consulting the Duckiebot manual [1] regarding the specification of the Duckiebot, the following assumptions were made:

1. The camera on the Duckiebot can accurately capture images from a distance of at least 2 meters.
2. Duckiebots can follow coloured lines on a road.
3. Duckiebots can use image recognition to recognise predefined objects.
4. Duckiebots can measure the distance to objects in front of it.

2-2 Mandatory requirements

These are criteria that need to be fulfilled for the minimum viable product. They can be divided into functional and non-functional requirements. Functional requirements are requirements the product must do and non-functional requirement are requirement the product must have.

2-2-1 Functional requirements

1. The traffic control must be able to guide the Duckiebots from the entrance of the charging station to a charging pad.
2. The traffic control must be able to guide the Duckiebots from the charging pad to the exit of the charging station.
3. The traffic control must prevent the Duckiebots from colliding inside the charging station.

2-2-2 Non-functional requirements

1. The charging of the Duckiebot must happen wirelessly.
2. Communicating with the Duckiebot must happen wirelessly.
3. The communication with a Duckiebot must be received by that Duckiebot inside the charging station.
4. The roads in the charging station must be large enough for the Duckiebot.

2-3 Trade-off requirements

There are criteria that are not vital, but do add significant value.

1. When a Duckiebot with a critically low SoC enter the charging station when all charging pads are occupied, the charging station will attempt to empty a pad
2. The charging station should receive the state of charge, SoC [2], and the minimum desired state of charge, mSoC, shown in figure 2-1 from the Duckiebots on the charging pads.
3. When asking Duckiebots to leave their charging pad, they should be asked to leave in order based on their SoC and mSoC.
4. The Duckiebots must function as independent robots with minimal centralised control.

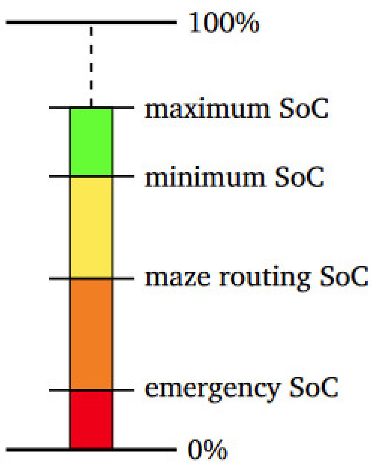


Figure 2-1: The SoC of the Duckiebot

Choice analysis

3-1 Duckiebot communication

Since the Duckiebot only has a camera on its front to receive information with, it was decided to send the information needed for the Duckiebot to navigate to and from a charging pad without colliding with other Duckiebots using traffic lights . The reason for this was the following, that the goal is for the Duckiebots to function as autonomous mechanical organisms. Cameras are the mechanical analogue to eyes in the animal kingdom. Cameras are relatively inexpensive, established and thus simple to incorporate into a vehicle, blend easily onto the vehicles and can distinguish shapes, colours and thus can are able to understand the meaning of objects such as road signs. Whereas LIDAR systems are expensive, bulky, impacted by weather influences, not as estabished as Camera technology and unable to distinguish between colours or Characters. Similarly RADAR struggles in the recognition and classification of objects. If the velocity of a static object is the same as the vehicle, then this is difficult to detect. Furthermore, detecting small objects is difficult for shorter wavelengths and finally, reaction times can be slower. All of these differences make the Camera for a more reliable system to utilise right out of the box and can be applied to other Duckiebots in the future.

3-1-1 PIR vs AIR

The Camera can be considered an one-way communication method, where the Charging station is the Transceiver and the Duckiebot the Receiver. This means that the Duckiebots can't communicate with one another on a higher more complex level, unless the Duckiebots are able to somehow interact with the charging station, influence it and have the charging station communicate those changes back to the other Duckiebots. The use of sensors rectifies that problem. The choice was made between Passive Infrared (PIR) detectors and Active Infrared in the shape of Photoelectric Beam detectors, also colloquially known as "Break-Beam Sensors". The manner in which PIR functions is that, PIR sensors are detecting the changes of the infrared energy levels that are caused by movement from objects (i.e. Humans, Animals, Falling leaves.... etc). PIR has some drawbacks due to the fact that PIR sensors are sensitive to the temperature changes of the environment, which can cause false positives. For AIR the functionality is different in that it consists of two parts namely, a light emitting diode (LED) and a photo electric receiver. This means that when an object is near the sensor, the infrared light from the LED reflects off of the object and is detected by the receiver.

Solution	Difficulty	Work needed	Potential danger	Chance of success
Multiple charging pads	−	-	−	++
Ordering new battery	-	-	-	+
Designing new battery	++	++	++	−

Table 3-1: Comparison of possible solutions for slow charging.

However, for Break-Beam Sensors the application is slightly different, as on one side lies an infrared light transmitter with opposite to it on the other side lies a photo electric receiver. The light transmitter continually sends out a beam to the receiver, so that when an object passes through the beam this results in the connection being broken, signalling that an object has entered the promixity. An additional difference is that PIR detects those differences in movement over a larger area than AIR, which is more precise due to the limited detection distance, so that random objects causing false positives is a lot less likely. For those reasons is why the choice went towards the Break-Beam Sensors.

3-2 Multiple charging pads

The battery of the Duckiebot can't charge with more than 10 W, so it takes around 5 hours to fully charge the battery, while the battery can become empty in only 2.5 hours.[1] In order to be able to keep multiple Duckiebots in the field, either the battery must be replaced with a different battery, decreasing the charging time, or the charging station must multiple charging pads, allowing multiple Duckiebots to charge at once. A comparison of possible solutions can be seen in figure 3-1. Designing a new battery is an unrealistic goal to do during this project, with a lot of work needed, high risks involved, such as the battery spontaneously combusting, with a low chance of success. Ordering a different battery that is compatible with the Duckiebot is much more realistic, though it will likely still be less safe than the battery of the Duckiebot. Making multiple charging pads is little added work, with no more risk than making just a single charging pad. It also makes the charging station expandable, by adding more charging pads. Due to multiple charging pads being the superior option, the charging station was designed with 3 charging pads, though more can be added with relatively small changes to the code and hardware.

3-3 Charging station layout

Due to there being multiple charging pads in the charging station, the layout of the charging station needs to be chosen. The charging pads can either be placed in series, or in parallel of each other, with the traffic through the charging station being either one or two directional. The advantage of parallel charging pads, shown in figure 3-1 is that the Duckiebots on the charging pads can enter and leave in any order independent of the other charging pads, while with charging pads in series, shown in figure 3-1, it works using a first in first out, FIFO, principle, which means that when a Duckiebot has been fully charged, it has to wait until the Duckiebot before it leaves until it can leave the charging station. It also means that when a Duckiebot with a critically low SoC wants to enter, the Duckiebot in front of the line needs to leave for the critical Duckiebot to enter. The advantage of charging pads in series is that it is easy to implement a waiting queue and handle the traffic control, due to there being only one way for the Duckiebots to go through the charging station.

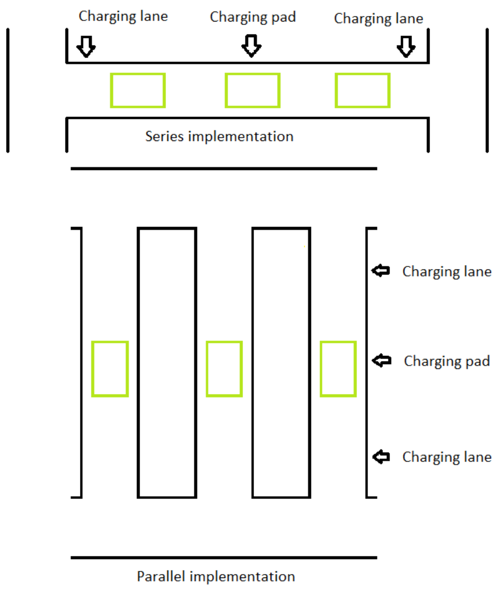


Figure 3-1: Charging pads series and parallel layout

Because a parallel layout has better results than a series layout, the parallel layout was implemented.
To make the charging station bidirectional, would require that the Duckiebots can exit the charging pads in both directions, regardless of the direction that the Duckiebots entered in. To do this, the Duckiebots must either be able drive backwards without collisions, or be able to turn around on the spot. Due to the facts that the Duckiebots only have a camera on the front and no camera on the back and thus can't drive backwards safely and that Duckiebots also can't turn around in place, two directional traffic can't be implemented in the charging station, leaving one directional traffic to be implemented.

3-4 Communication protocols

This chapter discusses several serial communication protocols to communicate between the charging pads and the charging station and explains the reason behind the choice for I^2C. Popular communication protocols such as SPI, CAN and UART are briefly introduced. Some protocols mentioned in the state-of-the-art analysis are not considered due to their cost complexity factor and are also becoming rare in recent times.

3-4-1 Serial Peripheral Interface

Serial Peripheral Interface (SPI) is a synchronous and a high speed (with speeds of usually up to 60 MHz) communication protocol with four bus lines [14]. Those four bus lines aid in the communication interfacing between the master and slave devices, where the master device both reads and writes the data. Furthermore, SPI supports two communication interfacing modes, namely standard mode and point-to-point mode. In standard mode, a single master device can communicate with multiple slave devices and in point-to-point mode the master device communicates with a singular slave device. The four bus lines are as follows: the Serial Clock (SCLK), the Master In Slave Out Line (MISO), the Master Out Slave In (MOSI) and the Slave Select (SS).

Slave Select has the same function as chip select and it is utilised instead of an addressing concept. The SS bus line is used to select a slave. In case of multiple slaves, additional SS lines are required. For instance running 5 devices one would require 8 lines, as the formula is 3 reserved lines (SCLK, MISO and MOSI) with n number of SS lines, one for each slave device. This would the make SPI not immensely scalable as one works with increasing number of devices. A benefit however is that the message size of SPI is arbitrary, dependent on the design of the application.

3-4-2 Controller Area Network

Controller Area Network (CAN) is an asynchronous, multi-master, robust serial communication bus that utilises multi-cast communication[12][20]. The way CAN operates is that each individual device, that is connected to the bus, is its own master and thus is capable of transmitting or request data to and from another device or multiples thereof. Depending on the purpose of the data transmission, the formatting can be done in different structures. On a physical level the way CAN works is that, two bus lines in twisted-pair make use of differential signalling between the two bus lines in order to transmit and receive messages. The reason for this is that the differential signalling and the twisted-pair configuration create noise immunity. Furthermore, depending on the bit timing being programmed CAN is able to operate at a data transmission rate of 20 kbit/s up to 1 Mbit/s and in CAN 2.0 or CAN FD (Flexible Data-rate) the speeds can go up to 5Mbit/s. In addition to this, CAN also has built-in error checking mechanisms, such as avoidance by arbitration on the message priority, the cyclical reduncancy check and finally the collision detection.

3-4-3 Universal Asynchronous Receiver/Transmitter

Universal Asynchronous Receiver/ Transmitter (UART) is an asynchronous, single-to-single, serial communication protocol in a microcontroller or an integrated circuit (IC) that is used to implement serial communication, with the main purpose being that of transmission and receiving serial data to and from devices in an embedded system[13]. The UART has two bus lines, where one is used for transmission (TX) and the other for receiving (RX), but is also unique in that it can operate on one bus line, when a pull-up resistor is introduced to the Transceiver side. These bus lines communicate through a digital pin 0 and a digital pin 1. The UART data is sent over the bus in the form of a packet, which consists of a start bit, data frame, a parity bit, and stop bits. The parity bit is utilised as an error check mechanism to help ensure the integrity of the data.

The UART protocol is considered to be “universal”, as the parameters including transfer speed and data speed are configurable by the developer. Furthermore, the UART supports bidirectional data transmission through 3 ways: simplex, half-duplex and full-duplex operations. As the UART is asynchronous this entails that the system doesn’t utilise a clock signal to synchronize the output bits from the transmitting UART to the sampling bits on the receiving UART. This means, that the receiving and transmitting UART need to be of the same bit rate or baud rate, as this is what allows the system to know where and when the bits have been clocked.

3-4-4 Inter-integrated Circuit

Inter-integrated Circuit (I^2C) is a synchronous, multimaster, serial communication bus. It is mainly used for many similarities between seemingly unrelated designs such as intelligent controls, general purpose circuits and application-oriented circuits[15]. I^2C has the following features, a bi-directional two-wire bus which allows all I^2C-bus compatible devices to communicate with each other. These two bus lines consist of a serial data line (SDA) and a serial clock line (SCL). The SDA is responsible for data transferal between the master and slave, whereas the SCL carries the clock signal. Each device connected to the bus have their own unique address and undergo simple master/slave relationships with other devices, both of which are software driven. And as it is a multi-master bus, the bus contains an error scheme that enables collision detection from transmission between the multiple devices, and prevents data corruption via arbitration. Furthermore, it has two directions of data transfers, bidirectional and unidirectional. The Bi-directional data transfer offers 2 modes of transmission, Quick mode with a maximum data rate up-to 400 kbit/s and High-Speed mode with a maximum data rate up to 3.4 Mbit/s. In Unidirectional data transfer the data rates go up to a maximum of 5 Mbit/s. Finally, I^2C even has on-chip filtering which prevents spikes to occur on the bus line in order to prevent data integrity.

3-4-5 Comparison

The described protocols for serial communication are compared on several aspects, which can be observed in Table 3-2.

To start off SPI and I^2C are synchronous because both use a clock line, while CAN and UART are asynchronous, as all of the devices have their own particular clocks, that are specifically

	SPI	CAN	UART	I^2C
Synchronicity	Synchronous	Asynchronous	Asynchronous	Synchronous
Noise Immunity	Low	High	Medium	Medium
Message Formatting	None	Multiple Structures	Multiple Structures	1
Maximum Speed	60Mbit/s	1.0 Mbit/s	1.5625 Mbit/s	3.4 Mbit/s
Number of Masters and Slaves	One master to many slaves	Multi-masters	Point-to-Point	Multi-masters
Wire Complexity	Increasingly complex when more devices implemented	Simple to implement new devices	Simple to implement new devices	Simple to implement new devices
Different Required Bus Lines	3 + n * SS	2	1	2
Error Detection and Handling	None	CSMA/CD+AMP Bit Monitoring CRC Frame Check	Frame Check Overrun Check	CSMA/CD Arbitration

Table 3-2: Differences Between the Serial Communication Protocols

programmed for their respective use cases. Additionally, UART and SPI are typically used in point-to-point communication and respectively require one bus line for UART (or just 1 line) and four different bus lines for SPI. The topology for SPI is relatively simple when utilised for a small number of nodes, however it will become increasingly complex as more devices enter the network, because an additional bus line is required for each additional slave. Furthermore, SPI has no standard message formatting, which means that the developer has to design a messaging format in order to achieve their requirements. This is different for UART, where only one bus line as long as a pull-up resistor is introduced to the Transceiver side and thus reduces the expenses due to the wiring and the complexity that is needed for it to operate correctly. SPI also does not have any error detection and handling mechanisms, whereas UART implements parity bit check for general error detection and correction purposes, making it more reliable and advantageous, than SPI under critical accuracy requirements. If data must be transferred at 'high speed', then SPI is the protocol of choice over UART, because SPI does not define any speed limit, as implementations often go over 10 Mbit/s.

The way CAN and I^2C function is via a Multi-master two-wire bus line topology. In comparison to SPI the "minimum number" of bus lines in CAN and I^2C reduces the expenses (comparatively to SPI) due to less wiring and thus the complexity itself. Devices can be easily added and removed to and from these buses due to their wiring configurations. Despite the increasing complexity of SPI, it has a significantly higher maximal data transfer speed compared to all the other protocols. I2C is faster than CAN, its maximal speed differs by a factor of three, whereas CAN has a higher noise immunity due to its twisted pair configuration and use of differential signals. This makes CAN more robust compared to SPI, UART and I^2C. Just like UART, CAN and I2C have their own implementations of error detection and handling, whereas SPI has none. This means that SPI requires additional programming in order to mitigate errors. When we compare CAN, UART and I^2C to oneanother, then the CAN protocol has a greater number of error detection and handling schemes. Despite that SPI and I^2C are comparatively low-cost to the other implimentations. I^2C is often considered a good choice for connecting short-distanced, low-speed devices like microcontrollers, EEPROMs, I/O interface, and other peripheral devices like sensors in an embedded system.

It is deemed that I^2C is the most suitable for the application of communication in the "Autonomous Wireless Charging System for Duckiebot Swarms" for a multitude of reasons, as when comparing I^2C to SPI, the flow control and error handling, makes it a more reliable protocol comparatively, as I^2C uses a two-wire interface where slave devices share the data and clock lines. This means that adding multiple devices to the bus is simple and reduces complexity of the circuit Furthermore, I^2C can support multi-masters in a configuration, while SPI can only support one master. So while SPI has a speed advantage, it is more difficult and costlier to add multiple slave devices to the bus. This is because each slave needs its own slave select line, so the number of wires needed to communicate increases with each device. Unlike communication protocols like I2C and SPI, UART is a physical circuit. This means that while SPI and I2C use a master/slave paradigm to control devices and send data, UART communication need to implement two UART devices to send and receive the data. It also doesn't operate using a clock so it is necessary for the baud rates of each UART to be within 10% of each other to prevent data loss. While different in these aspects, UART[19] is similar to I2C and SPI certain ways. For instance, both I2C and UART implement a two-wire interface to send and receive data and is often ideal for low-speed data transmission. As mentioned, UART and I2C also both make use of error checking mechanisms to help ensure data integrity, as I2C uses an ACK/NACK bit and UART uses a parity bit to distinguish any changes in data during transmission. Additionally, both UART and SPI support full-duplex communication and cannot support a multi-master configuration. To conclude, the choice for I^2C is due to the weighing of it being relatively low-cost, faster than the other protocols (besides SPI), making use of error checking schemes, built-in message formatting, CSMA/CD, relative simple implementation and the Multi-master configuration.

3-5 Switches

3-5-1 Multiplexer/Demultiplexer

A Multiplexer (Mux)[18] is used to select one of several outputs based on some fewer number of inputs representing a binary number. For example, an address bus on a CPU is connected the inputs of a "3-to-8 multiplexer" to address lines A11, A12, and A13. Since the 3-input binary value has its lowest bit connected to address line A11 there are 11 address lines before it that can all change to any binary value from 0-2047 (address lines A0-A10). Once an address shows up >= 2048 (2KB) the first output pin (1 of 8) will go HIGH and the rest will remain LOW. The lower address bits can continue to count up and the first output will stay high for any address from 2048 - 4095. Once the address gets high enough and the address lines set address line A11 HIGH and lowers A11, the first output bit will change to LOW and the second output bit(2 of 8) will go HIGH.

Conversely, a Demultiplexer (Demux)[16] is used to select one of several outputs based on some fewer number of inputs representing a binary number. The way this works is that one can see the GPIO outputs as already being multiplexed and using by a demultiplexer one is able to separate the three GPIO outputs (inputs) into eight output signals.

3-5-2 Shift Register

A shift register [17] is a number of flip-flops (generally speaking 8) that are connected in series. Each flip-flop can store and output one singular bit. When a 1 or 0 is placed on the 'Data In' pin and the 'Clock' pin on the shift register is taken HIGH (or LOW, some shift registers' clock is active HIGH and some are active LOW) the shift register will grab the 1 or 0 on the 'Data In' pin and shift it into the first flip-flop. All flip-flips are connected to the clock pin so they each grab their input and latch it at the same time. Since the first flip-flop output is connected to the second flip-flop input and so on, the shift register will "shift" all of the bits over one position for each clock pulse. Lastly, each output of each flip-flop is also connected to an output pin. So if the LEDs to the outputs and shifted in "11011011" those bits would show up on the 8 outputs.

There are several flavors of shift register: Serial to Parallel shift registers will take in a serial input and expand it to 8 parallel outputs as described before. This is often used to expand the number of OUTPUT pins on a micro controller. A Parallel to Serial shift register works the other way as it has 8 parallel inputs and 1 serial output and that can be use to get the value of 8 different inputs using one input pin connected to Serial Out and one output pin for the clock line. Lastly, there are shift registers that have both Serial and Parallel inputs and Serial and Parallel outputs. Using an additional pin to tell it which way to shift (in or out) and this shift register can do everything the other two can do combined. This is often used to expand the number of input pins on the micro controller if the system runs out. Or it makes an easy way to interface with something like a 4-row 4-column keypad matrix using only a few pins on the micro controller.

3-5-3 comparison

The switches will be briefly recapped and compared. A multiplexer switches between inputs, whereas the reverse is true for a demultiplexer, where the switching occurs between the outputs and as for a shift register it takes a parallel word and shifts it out as a sequence of bits. The mux/demux is faster than a shift register and also come in analogue and digital variants. However, the mux/demux use more pins than a shift register e.g. for 30 sensors one would need at least 6 pins, whereas a shift register would only need to 3 pins, for any number of bits. Furthermore, shift registers are cheaper than multiplexers/demultiplexers. In conclusion, the shift register was chosen for the reasons that the implementation is cheaper and can be daisy chained for a large number of sensors.

Theory

This chapter describes the comprehensive theory behind the chosen protocols that are necessary to implement the algorithms and hardware solution.

4-1 I^2C

I^2C (Inter-Integrated Circuit) is a synchronous and single-ended communication bus that enables packet switching, and multi-master/multi-slave (controller/target) communication. It is broadly used for short, intra-board communication between lower-speed auxiliary integrated circuits and microcontrollers or processors.

4-2 Serial Lines

The communication bus consists of two wires, the Serial Data Line (SDA) and the Serial Clock Line (SCL), where the SDA is responsible for the data transfer (sending and receiving) between the master and slave devices and the SCL carrying the clock signal generated by the master.

At the physical layer, both the busdrivers (SCL and SDA lines) are of an open-drain (MOSFET) or open-collector (BJT) bus design. This means that the busdrivers can pull the corresponding signal line low, but cannot drive it high. Thus, there can be no bus contention where one device is trying to drive the line high while another tries to pull it low. meaning that they can pull the corresponding signal line low, but cannot drive it high. Each signal line has a pull-up resistor on it, to restore the signal to high when no device is asserting it low.

By pulling the line to the ground the result is a logic "0" as an output, and by letting the line float (the output having a high impedance) a logic "1" is obtained as an output, as this causes the pull-up resistor to pull it high. The lines are never actively driven high. This enables the wires to allow for several nodes to connect to the bus without having to short the circuit due to signal contention. With systems, where speed is critical, such as with High-speed systems,a current source may be utilised instead of a pull-up only resistor configuration to pull up the SCL or both the SCL and the SDA, as to enable faster rise times and to accommodate for higher bus capacitance.

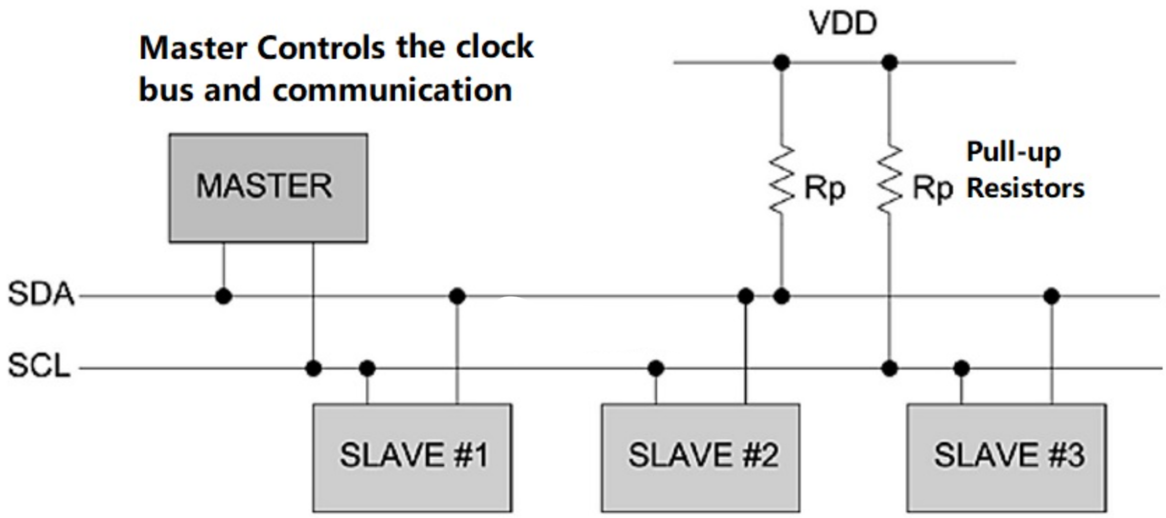


Figure 4-1: I^2C serial lines

The crucial consequence of this is that now multiple nodes can drive the lines simultaneously. What happens when any node is driving the line low, is that the line will be low as well. Nodes that are trying to transmit a logical "1" (by letting the line float high causing a high impedance) can detect this difference and thus conclude that another node is active on the line at the same time. If this principle is used on the SCL, then this can be utilised as a flow-control mechanism for targets and is also known as "clock stretching", whereas When this principle is utilised on SDA, this is called arbitration and ensures that there is only one transmitter at a time.

In the situation that the system is idle, then both lines are high. To resume transmission once more, the SDA is pulled to a low value whilst the SCL remains the previous high value. Next up is for the SCL value to be pulled down to a low value. It is considered "illegal" to transmit a stop marker by releasing SDA to float high again before pulling the the SCL to a low value, although this is usually considered harmless. The only time the SDA line changes its value is when the clock value is low. The exception to this are the start and stop signals. The manner of how a data bit is transmitted is by pulsing the SCL line high while the SDA line is steady at the level of desire. Next up is for the transmitter to set the SDA to the value of their desire whilst the SCL is low, then after a small delay for value propagation, the SCL is set to float high. Now the controller waits for the value of the SCL to actually go to high. This process can be delayed by the parasitic capacitance of the bus line, the RC time constant of the Pull-up resistor in the circuit and finally due to the clock stretching of the target.

As the SCL is now on a high value, the controller waits a minimum time frame, which is usually around 4 μs for standard-speed I^2C mode as to ensure that the receiver has seen the bit, then finally pulls the bus low again. This concludes the transmission of one bit. Now after every 8 data bits in (any particular) direction, an bit for the acknowledgement will be transmitted to the opposite side. After the transmitter and receiver both perform this action for one bit, is when the initial receiver ends up transmitting a single "0" bit (ACK) back.

In the event that the transmitter witnesses a "1" bit (NACK) instead, it will understand that the target (the controller being the initial transmitter) is incapable of accepting the data for a multitude of reason such as, that the message isn't clear, that there aren't any (viable) targets, or that the target is simply unable to accept more data. In the event that the target is transmitting a message to the controller, then this means that the controller is requesting the data transmission to cease after the current data byte. It needs to be noted that control over the SCL bus line is always maintained by the controller and that only the direction in the SDA bus line changes when the acknowledge bits are being sent. Finally, after the acknowledge process the clock line is now set on low, which gives the controller three options: 1. Sending a stop message, which releases the I^2C bus, 2. a repeated start message, which starts the transmission over the bus without release the bus, and 3. the transfer of another message.

Implementation

5-1 Algorithms

In order to correctly implement the traffic lights and allow the Duckiebots to get charged, a number of algorithms were used to control the various traffic lights[3][4]. There algorithms will be explained in this chapter. In figure 5-1 the various ways that a Duckiebot can go through the charging station are shown.

5-1-1 Traffic lights

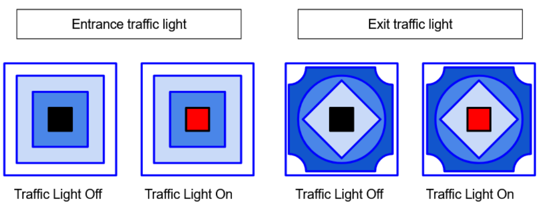


Figure 5-3: Two different kinds of traffic lights

There are four different kinds of traffic lights used in the charging station. The Duckiebots distinguish these traffic lights from one another based on the different coloured borders around the LED in the traffic light, an example of which is shown in figure 5-3.
The way that these traffic lights work is that the traffic control subgroup made the Duckiebot able to follow coloured lines on the road. At every intersection where the Duckiebot can go to multiple different directions, there is a differently coloured line for each of the different directions. When the Duckiebot decides what direction it wants to go, based on the status of the traffic light, it will then start following the corresponding coloured line to go in the correct direction.
One kind of traffic light is used at the entrance of the charging station to show the Duckiebots if there are empty charging pads, or if all charging pads are occupied. Another kind of traffic light is used at the entrances of the charging pads to show if the Duckiebot should go to that charging pad or not. The third kind of traffic light is used to ask Duckiebots to leave their charging pad when a Duckiebot with a critically low SoC enters the charging station and there are no empty charging pads. The last kind of traffic light is used to prevent collisions at the intersections that Duckiebots can go to from multiple different directions, for example the intersections at the exits of the charging pads, or at the charging station exit. The layout of the charging station can be seen in figure 5-2.

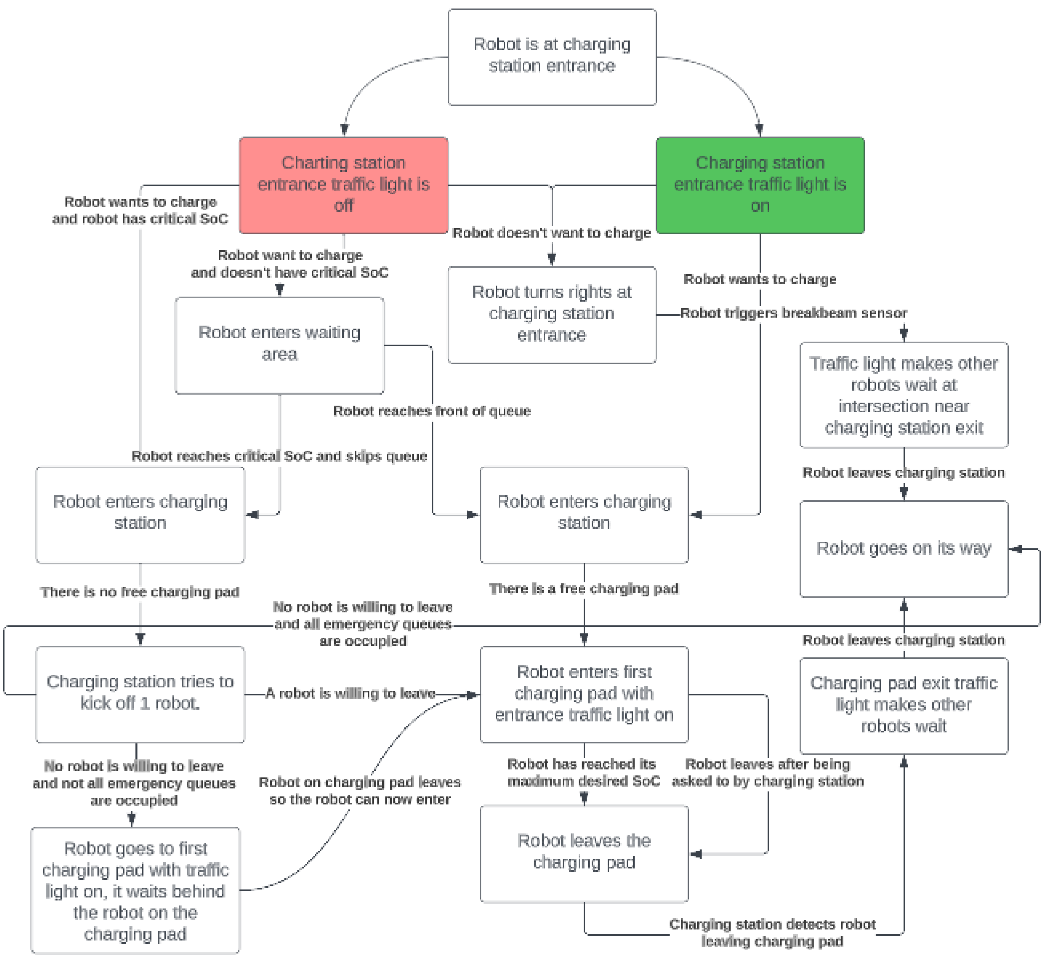


Figure 5-1: Duckiebot flowchart

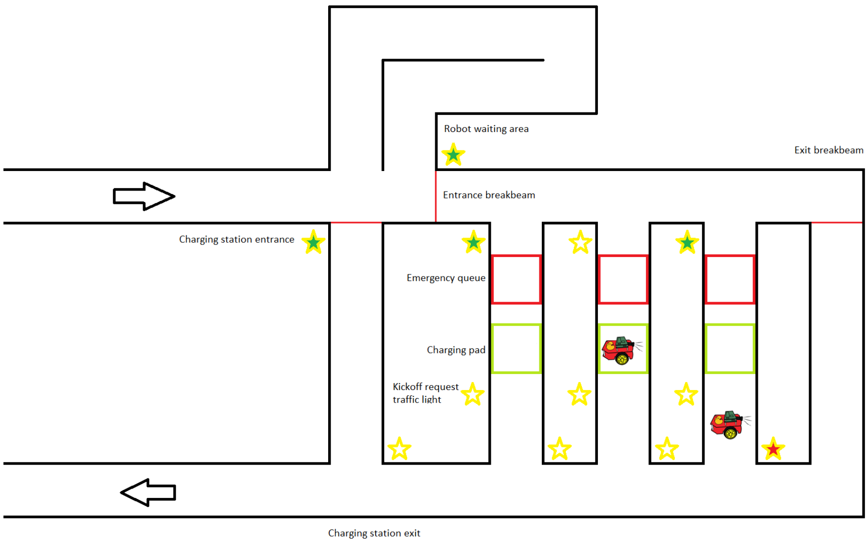


Figure 5-2: Charging station layout

5-1-2 Charging station entrance traffic light

The traffic light at the entrance of the charging station will be on if there are less Duckiebots in the charging station, then there are charging pads and it will be off if the amount of Duckiebots inside the charging station is equal or larger than the amount of charging pads. This way, when the entrance light is on, incoming Duckiebots will know that there are free charging pads they can charge on, while if the entrance light is off, Duckiebots will know there are no empty charging pads, so they can either enter the waiting area, look for another charging station, or enter anyways if they have a critically low SoC. A Duckiebot decides for themselves if their SoC is critically low. This is shown in figure 5-4.

5-1-3 Charging pad entrance traffic lights

When there there is no Duckiebot on a charging pad, the traffic light at the entrance will be enabled, while if there is a Duckiebot on a charging pad, the traffic light will be disabled. This way Duckiebots that are passing by see the entrance traffic light and know if a charging pad is occupied or not.

If all charging pads are occupied, it may happen that a Duckiebot with a critical SoC arrives and no Duckiebot is willing to leave their charging pad. To solves this, of the charging pads with an empty emergency queue, which is shown in figure 5-2, the charging pad where the SoC minus the minimum desired SoC is the largest, the charging pad whose Duckiebot will be willing to leave the earliest, will have their traffic light turn on. This way, the Duckiebot with a critical SoC can wait there for the Duckiebot on the charging pad to leave. This is shown in figure 5-5.

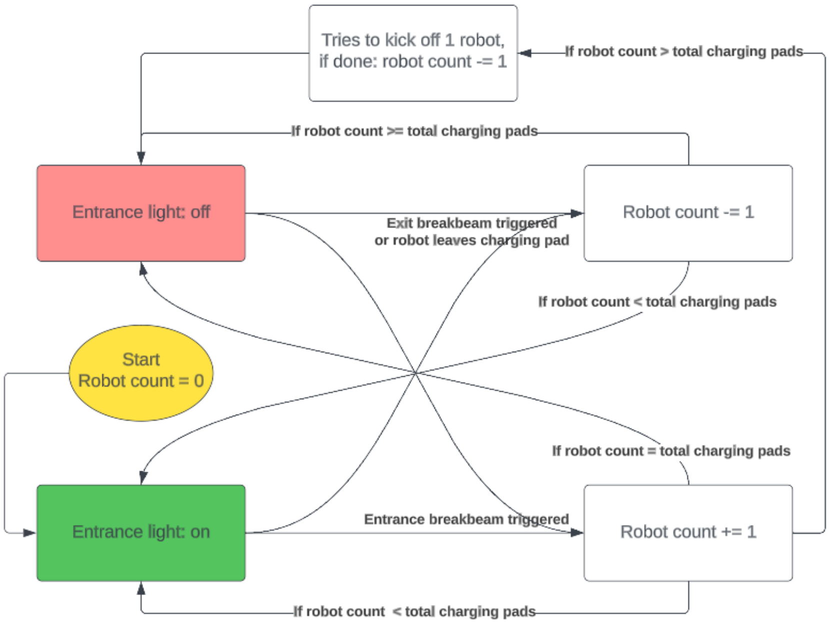


Figure 5-4: Charging station entrance light

5-1-4 Charging pad exit traffic lights

Since Duckiebots can only detect other Duckiebots using the camera on the front, they might not be able to detect other Duckiebots in time to prevent a collision on an intersection. To prevent such collisions, it was decided to give Duckiebots exiting the charging pads priority, while making other Duckiebots wait at the intersections for the Duckiebot exiting the charging station. This is because while it is known when a Duckiebot exits the charging pad, it can't be easily determined when a Duckiebot passes through such an intersection from the other direction without adding more sensors. This is implemented by using a timer that turns on the exit traffic light when a Duckiebot leaves their charging pad, until a few seconds later. This is also used for Duckiebots that turn right at the charging station entrance, only with an added breakbeam sensor to detect incoming Duckiebots. This is shown in figure 5-5.

5-1-5 Charging pad kickoff request traffic lights

Since the communication between the Duckiebot and the charging pad is unidirectional, with the Duckiebot sending information and the charging pad receiving information, traffic lights were added to ask the Duckiebots to leave their charging pad.

The traffic lights have two ways of asking a Duckiebot to leave: low priority and high priority. For both high and low priority, the traffic light will go on and then off again, with the traffic light staying on longer with high priority. The chance for a Duckiebot to leave with high priority is higher than with low priority. In both cases, the Duckiebot can decides if it stays on the charging pad, or if it leaves.

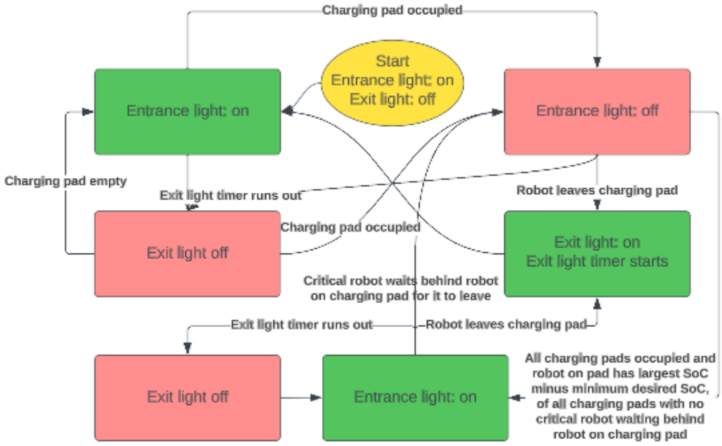


Figure 5-5: Charging pad entrance and exit traffic lights

When a Duckiebot with a critical SoC enters the charging station when all charging pads are already occupied, the charging station will try to get a Duckiebot on a charging pad to leave the charging station with low priority. It will also ask the Duckiebots that are on the charging pads with another Duckiebot in the emergency queue to leave with low priority as well. If a charging pad with an empty emergency queue becomes empty, the Duckiebot will go there. If not, the Duckiebot will enter the empty emergency queue behind the Duckiebot that is expected to leave the earliest.

If all charging pads and emergency queues are occupied, it will ask all Duckiebots on the charging pads to leave with high priority in order to prevent the Duckiebot with the critical SoC from leaving the charging station again without charging, which would likely lead to that Duckiebot "dying" due to a lack of charge. This is shown in figure 5-6.

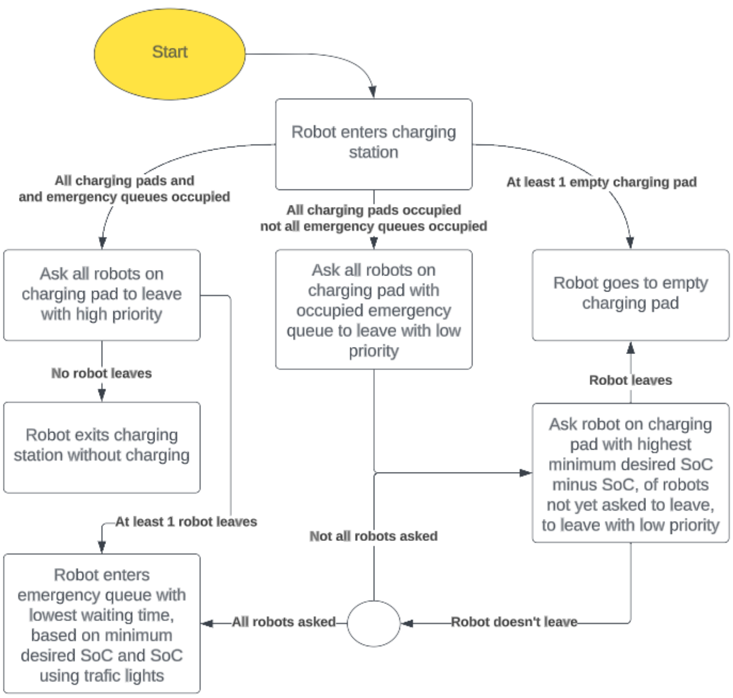


Figure 5-6: Charging pad kickoff requests

5-2 Hardware

In order to correctly execute the developed algorithms, making sure the Duckiebot can get charged, hardware is needed to execute the algorithms. The hardware used will be explained in this chapter. A block diagram of the hardware used can be seen in figure 5-7. For easy of viewing, only a single charging pad and breakbeam were shown. In reality, 3 similarly connected charging pads and breakbeams are used.

5-2-1 Microcontroller

For the microcontroller, the TeensyLC [5] was used. It was chosen for a number of reasons: it can be programmed and powered using a USB connection, not needing some kind of transformer or converter to give the microcontroller the correct input voltage and current. The TeencyLC is also compatible with Arduino, allowing for easy coding the programs needed. The bootloader on de microcontroller is stored on a different chip, preventing the bootloader from being erased when programming the chip. Finally, the in and output voltages of the pins of the microcontroller, 3.3V and one 5V pin, are compatible with the other components.

5-2-2 Breakbeam sensor

The breakbeam sensor consists of an IR emitter and an IR receiver, there specifications of which can be seen in [6]. The IR emitter is powered by the microcontroller and send out an

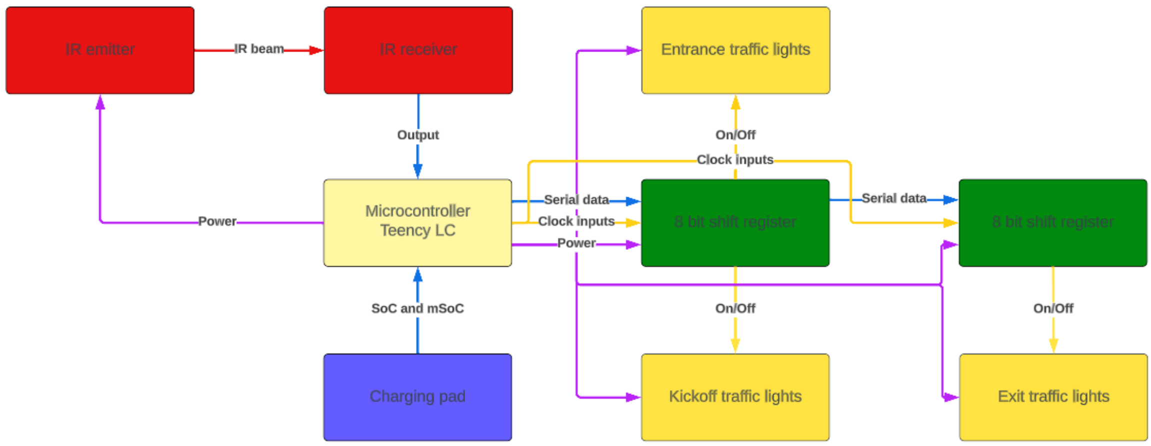


Figure 5-7: Charging station block schematic.

IR beam. When the IR receiver receives this IR beam, it causes a voltage over the input pin of the microcontroller. When a Duckiebot drives through a breakbeam, it will prevent the IR receiver from receiving the breakbeam, resulting in no voltage over the input pin of the microcontroller, thus detecting the Duckiebot.

5-2-3 Shift registers

The shift registers[7] were used to control the traffic lights, because although the microcontroller used has enough pins to control, power and communicate with all other components needed for three charging pads, this way the amount of charging pads can be expanded without running out of pins.

Since the charging station uses 12 traffic lights, 2 8-bit shift registers were used. These shift registers are powered and controlled by the microcontroller, with the serial data being the desired outputs of the traffic lights and the clock inputs updating the shift register and storage register. When data is shifted through the shift registers, the last bit of the first shift register is the input of the second shift register.

5-2-4 Traffic lights

For the traffic lights LED's were used. These LED's are controlled by the shift registers. The LED's are powered by the microcontroller, due to the shift registers having an open drain output. A high output on the shift register results in the pin being grounded, thus powering the LED's from the microcontroller and a low output results in the pin being floating, and thus the LED not being powered.

Verification and results

This chapter describes how the algorithm and hardware used are tested and how the algorithms and hardware will be integrated into the results of the other subgroups. Due to the necessary hardware not being delivered on time, it was unfortunately not possible to integrate the results of the different subgroups and test the system in the real world.

6-1 Simulation

In order to be able to simulate the algorithms used, a number of assumptions have been made:

1. The Duckiebots will enter the first charging pad they come across with an enabled entrance light. Implementing this is the responsibility of the robot control and navigation subgroup.
2. The Duckiebots will measure the distance to a mark on the road placed a set distance behind the charging pads, in order to stop on the charging pads. Implementing this is the responsibility of the robot control and navigation subgroup.
3. The breakbeam sensors will detect the Duckiebots correctly. This assumption was made due to the breakbeam sensors not being delivered on time.
4. The IR communication between the Duckiebots and the charging pads and the I2C communication between the charging station and the charging pads will work correctly. This assumption was made due to the charging pads not being finished in time.

Due to these needed assumptions, using the code of the verification simulation in appendix A-1, a number of Duckiebots were simulated to enter the charging station at predefined times. It was also predefined to which charging pad or exit the Duckiebots went, according to the assumptions and the algorithms used.
The results of this simulation, an example of which can be seen in 6-1 are that provided that the assumptions that were made are indeed correct, the algorithms do indeed work correctly, though there was a potential problems. The algorithms allow the Duckiebots to move to and from the charging pads, with prioritization of Duckiebots with a critically low state of charge and with an emergency queue in case there is a critical Duckiebot, but none of the robots on the charging pads are willing to leave.

```
Duckiebot enters charging station.
There are currently: 1 Duckiebots in the charging station.
Duckiebot turns right at charging station entrance.
Duckiebot entered charging pad 1.
Disable charging pad 1 entrance traffic light.
Duckiebot enters charging station.
There are currently: 2 Duckiebots in the charging station.
Disable charging station exit traffic light.
Duckiebot entered charging pad 2.
Disable charging pad 2 entrance traffic light.
Duckiebot enters charging station.
Disable waiting area exit traffic light.
Disable charging station entrance traffic light.
There are currently: 3 Duckiebots in the charging station.
Duckiebot entered charging pad 2.
Enable charging pad 1 entrance traffic light.
Disable charging pad 3 entrance traffic light.
Duckiebot enters charging station.
Ask the Duckiebot on charging pad 1 to leave.
There are currently: 4 Duckiebots in the charging station.
Duckiebot leaves charging pad 1.
Enable charging pad 1 exit traffic light.
There are currently: 3 Duckiebots in the charging station.
Duckiebot entered charging pad 1.
Disable charging pad 1 entrance traffic light.
Enable charging pad 3 entrance traffic light.
```

Figure 6-1: Part of the results of the verification simulation.

The potential problem that was found is that when all charging pads are occupied, and none of the Duckiebots on a charging pad are willing to leave, Duckiebots entering the charging station have to go to an emergency charging pad, which are shown in figure 6-2. Since there were no breakbeam sensors available to detect if a Duckiebot has reached the emergency queue, due to the hardware not being delivered, this was instead implemented using a timer. This timer makes it so that it is assumed that when the Duckiebot doesn't exit the charging station within a certain amount of time, thus triggering the breakbeam sensor, it will have entered the emergency queue it is supposed to. When multiple critical Duckiebots enter the charging station very shortly after each other and none of the Duckiebots on the charging pads are willing to leave, the charging station won't direct the second Duckiebot to an emergency queue, either forcing it to exit the charging station, or making it enter an occupied emergency queue.
This problem can be solved by either using breakbeam sensors to detect when Duckiebots enter an emergency queue, or by making only a single emergency queue for multiple Duckiebots, thus leaving no doubt as to where the second Duckiebot is, or by making the Duckiebots hold a certain amount of distance from other Duckiebots when near or inside the charging station, thus preventing Duckiebots from entering right after each other.

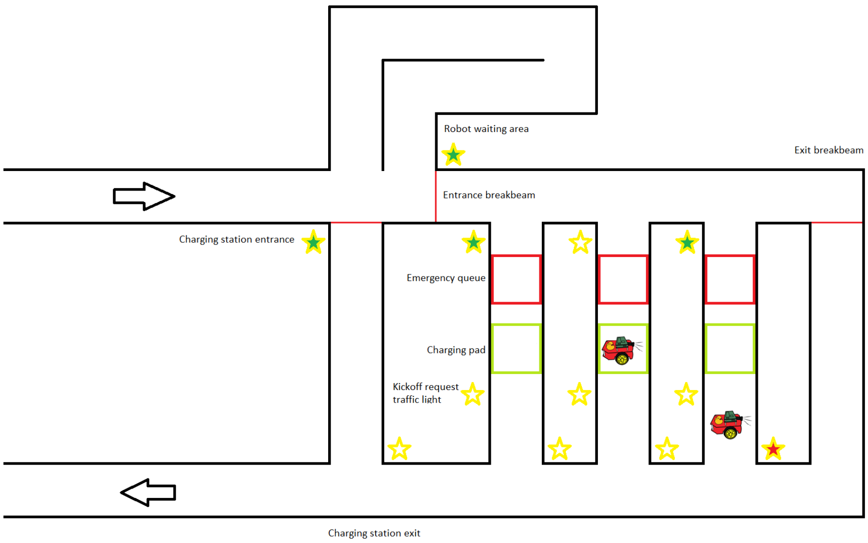


Figure 6-2: Charging station layout

6-2 Integration

6-2-1 Wireless charging hardware subgroup

The wireless charging hardware subgroup has made both a charging receiver on the Duckiebot and a wireless sender on the charging pad connected to a power supply in order to charge the Duckiebot [8]. The wireless sender and receiver will also negotiate the amount of power sent to the Duckiebot using IR communication for the Duckiebot to the charging pad. In order for a Duckiebot to be charged, it must be on the charging pad. This is achieved by the robot control and navigation subgroup and the charging station design subgroup.

6-2-2 The charging station design subgroup

The charging station design subgroup is responsible for handling the traffic control inside the charging station. This is done by communicating with the charging pads using I2C to know if there is a Duckiebot on that charging pad and if so, to know what it's state of charge and minimum desired state of charge are. There are also a number of breakbeam sensors inside the charging station, these are used to keep track of how many Duckiebots enter and exit the charging station. The traffic lights are used to communicate the state of the charging station to Duckiebots passing by, for example if there are free charging pads inside the charging station, or if a certain charging pad is empty. The traffic lights are also used to ask the Duckiebots to leave their charging pad in case a Duckiebot with a critically low state of charge enter the charging station.

6-2-3 The robot control and navigation subgroup

The robot control and navigation subgroup mas made the robot able to perform object recognition, object detection, distance measurement and made the robot capable of following a coloured line[9]. By having differently coloured lines going in every possible direction at an intersection and having the Duckiebot follow the coloured line of the desired colour based on the state of the detected traffic lights, the Duckiebot can navigate the charging station correctly. Using the distance measurement and a mark on the road at a predefined distance behind the charging pad, the robot can stop on the charging pad. The subgroup is also responsible for making the Duckiebots decide when they want to go to the charging station, what their minimum desired state of charge is: what state of charge the Duckiebot at least wants to obtain, and if the Duckiebot leaves when it is asked to leave by the charging station. If the Duckiebot decides to leave, it communicates this to the charging pad using the IR communication of the wireless charging hardware group, which will then stop charging the Duckiebot, after which the Duckiebot will leave.

6-2-4 Integrated subgroups

This way, with the implementations all of the different subgroups integrated, the Duckiebots should be able to go to the charging station when their battery begins to get low, where they will then be able to get charged, with prioritization of Duckiebots with a critically low state of charge, and leave when they have obtained their desired state of charge. This way the Duckiebots should be able to stay charged without human interaction.

Conclusion

This project serves as a study to improve the functionality of the Lunar Zebro Project as a whole, by introducing a system that enables the seemless facilitation between wireless charging mechanisms and traffic flow control protocols. This system was necessary, as the previously existing systems didn't account for the wireless charging of the Duckiebots and wireless communication between the charging station and the Duckiebots needed to satisfy the requirements. By working out the constraints, both on a hardware and software level, a model of the charging station was created. This model satisfies the program of requirements in that a charging queuing protocol is implemented, while still taking the autonomy of the Duckiebots as a mechanical organism into account, but also the cooperative and social aspects that you often see in successful species in nature, as the Duckiebots still will leave when one of their compatriots are critical depleted of energy (critically low SoC). All of the functional requirements have been met, as the traffic flow control protocols are able to guide the Duckiebots from the entrance of the charging station to a charging pad, from the charging pad to the exit of the charging station and prevent the Duckiebots from colliding with one another in the charging station area. Furthermore, the non-functional requirements have been met as well, as the roads in the charging station area are large enough to accommodate the Duckiebots, the charging of the Duckiebots occurs wirelessly and the communication with the Duckiebot occurs in a wireless fashion via the traffic lights, IR Breakbeam sensors and the IR communication over the I^2C serial bus protocol at the charging pads.

7-1 Future Work

Although the current system setup complies to the programme of requirements, there are still ways for the system to be improved:

- Look more in-depth into other sensor integrations, such as Radar or LIDAR, allowing for more robustness
- The Duckiebots could be altered to allow of radio communication[10][11] between them, allowing for more versatility.
- Explore the environmental impacts on the Duckiebot harder in space more in depth to show case any future complications.

Bibliography

[1] Duckietown-Team, "The Duckiebot operation manual," *online (pdf): https://docs.duckietown.com/daffy/opmanual-duckiebot/intro.html*, 2022.

[2] P. S. P. M. Iwan Wahyuddin and H. Sudibyo, "State of Charge (SoC) Analysis and Modeling Battery Discharging Parameters," *online (pdf): https://ieeexplore.ieee.org/document/8528631*, 2018.

[3] S. Zhang and J. J. Q. Yu, "Electric Vehicle Dynamic Wireless Charging System: Optimal Placement and Vehicle-to-Grid Scheduling," *online (pdf): https://ieeexplore.ieee.org/document/9528835*, 2021.

[4] J. P. Q. Tuan Nguyen Gia and T. Westerlund, "Exploiting LoRa, edge, and fog computing for traffic monitoring in smart cities," *online (pdf): https://www.researchgate.net/publication/340070758_Exploiting_LoRa_edge_and_fog_computing_for_traffic_monitoring_in_smart_cities*, 2020.

[5] PJRC, "Teensy® lc development board," 2023. [Online]. Available: https://www.pjrc.com/store/teensylc.html

[6] Adafruit-Industries, "Ir break beam sensor with premium wire header ends - 5mm leds," *adafruit industries blog RSS*, 2021. [Online]. Available: https://www.adafruit.com/product/2168

[7] N. B.V., "Power logic 8-bit shift register; open-drain outputs," *online (pdf): https://assets.nexperia.com/documents/data-sheet/NPIC6C596A_Q100.pdf*, 2020.

[8] O. Nezamuddin and E. C. dos Santos, "Vehicle-to-Vehicle In-Route Wireless Charging System," *online (pdf): https://ieeexplore.ieee.org/document/9161472*, 2020.

[9] J.-H. A. Sujin Youn and K. Park, "Entrance detection of a moving object using intensity average variation of subtraction images," *online (pdf): https://ieeexplore.ieee.org/document/4505600*, 2008.

[10] Y.-W. S. Jin-Shyan Lee and C.-C. Shen, "A Comparative Study of Wireless Protocols: Bluetooth, UWB, ZigBee, and Wi-Fi," *online (pdf): https://ieeexplore.ieee.org/document/4460126*, 2008.

[11] E. Ferro and F. Potorti, "Bluetooth and Wi-Fi wireless protocols: a survey and a comparison," *online (pdf): https://ieeexplore.ieee.org/document/1404569*, 2005.

[12] A. J. E. Hannah M. Boland, Morgan I. Burgett and R. M. S. III, "An Overview of CAN-BUS Development, Utilization, and Future Potential in Serial Network Messaging for Off-Road Mobile Equipment," *online (pdf): https://www.intechopen.com/chapters/77277*, 2021.

[13] A. K. Pranjal Sharma and N. Kumar, "Analysis of UART Communication Protocol," *online (pdf): https://ieeexplore.ieee.org/document/9936199*, 2022.

[14] I. K. Pia Kotiranta and M. Rouvala, "SPI Proceedings: Analysis of High-Speed Digital Interfaces in Flexible Interconnections," *online (pdf): https://ieeexplore.ieee.org/document/4069465*, 2007.

[15] NXP-Semiconductors, "I^{2}C-bus specification and user manual," *online (pdf): https://www.nxp.com/docs/en/user-guide/UM10204.pdf*, 2021.

[16] A. Dalal and A. Atri, "A General Overview of Multiplexer and Demultiplexer," *online (pdf): https://journals.pen2print.org/index.php/ijr/article/download/966/913*, 2014.

[17] B. Kjos-Hanssen, "Automatic complexity of shift register sequences," *online (pdf): https://www.sciencedirect.com/science/article/pii/S0012365X18301559*, 2018.

[18] S.-Y. R. Li and X. J. Tan, "Mux/Demux Queues, FIFO Queues, and Their Construction by Fiber Memories," *online (pdf): https://ieeexplore.ieee.org/abstract/document/5714255*, 2011.

[19] M. Zibayiwa, "A Review on The Inter-Processor Communication: I2C, UART, and SPI interfacing techniques," *online (pdf): https://www.researchgate.net/publication/356556000_A_Review_on_The_Inter-Processor_Communication_I2C_UART_and_SPI_interfacing_techniques*, 2021.

[20] H. S. I. Illgen and E. Schnieder, "Infrared CAN Interface – Principles of CAN Data Transmission using infrared light," *online (pdf): https://www.can-cia.org/fileadmin/resources/documents/proceedings/2000_illgen.pdf*, 2000.

Appendix

A-1 Verification simulation code

This code can be uploaded to an Arduino compatible microcontroller, without needing additional hardware. It was made with and tested using the TeensyLC

```

int charging_pads = 3; // amount of charging pads
int robots = 0; // robots in charging station
int robots_old = 0;
bool pad_1_occ = false; // true if pad was occupied, false if not, updates ...
    based on pad_1_SoC
bool pad_2_occ = false;
bool pad_3_occ = false;
bool pad_1_lock = false; // keeps pad_1_occ enabled until pad_1_soc ...
    detects a robot
bool pad_2_lock = false;
bool pad_3_lock = false;
int pad_1_SoC = 0; // SoC of robot on pad, 7 bit, 000000 is no robot, else ...
    robot on pad, updates based on I2C (not implemented yet)
int pad_2_SoC = 0;
int pad_3_SoC = 0;
int pad_1_mSoC = 0; // minimum desired SoC of robot on pad, 7 bit?
int pad_2_mSoC = 0;
int pad_3_mSoC = 0;
bool pad_1_em = false; // is emergency queue occupied?
bool pad_2_em = false;
bool pad_3_em = false;
bool robot_to_em = false;
int pad_em_on = 0; // 0 if 0-2 pads occupied, or 6 pads occupied, pin of ...
    pad which has entrance light on
unsigned long exit_timer = 5000; // how long until the exit light goes off ...
    again in ms
unsigned long robot_enter = 0; // time when robot enters if 3 < x ≤ 6 pads ...
    occupied
unsigned long pad_1_exit = 0; // the time when the exit light goes on, ...
    used to disable exit light after ... ms.
unsigned long pad_2_exit = 0;
unsigned long pad_3_exit = 0;
unsigned long station_exit = 0;
```

```
unsigned long kickoff_high = 100; // how long the light is on for a high ...
    priority kick off request (all pads and emergency queues occupied)
unsigned long kickoff_low = 30; // how long the light is on for a low ...
    priority kick off request (all pads but not all emergency queues occupied)
unsigned long kickoff_start = 0; // when kickoff attempt starts
unsigned long waiting_out_time = 0; // the amount of time between the ...
    waiting area light and the entrance light activating (waiting first)
bool station_in_enable = false; // allows station_in to be on
int kickoff_tries = 0; // how many kick off requests this round
int kickoff_priority; // 1 = high, 0 = low
int kick_prev = 0; // last kick off attempt
int bb_in_state = 2; // 0 means no robot seen at last check, 1 means robot ...
    seen at last check, but not one before,
                     // 2 means robot seen at last check and at 1+ before, ...
                         if signal goes from 0 to 1 to 0, seen sensor error, ...
                         not car
int bb_out_state = 2;
int bb_in2_state = 2;
unsigned long old_m_1 = 0; // old value in milliseconds
unsigned long old_m_2 = 0;

// pin assignments
// traffic lights, update pin numbers to correct values
bool station_in = true;  // at station entrance
bool pad_1_in = true; // at pad 1 entrance
bool pad_2_in = true;
bool pad_3_in = true;
bool pad_1_out = false; // at pad 1 out
bool pad_2_out = false;
bool pad_3_out = false;
bool station_out = false; // at station exit
bool pad_1_kick = false; // at pad 1 kick off
bool pad_2_kick = false;
bool pad_3_kick = false;
bool waiting_out = true; // at waiting area out
// breakbeams
bool bb_in = false; // charging station entrance breakbeam
bool bb_out = false; // charging station exit breakbeam
bool bb_in2 = false; // breakbeam for robots that turn right at entrance

// I2C??? how to implement this, maybe use list for bits and then decode ...
    when done?

void setup() {
  // put your setup code here, to run once:
  // use the pin numbers from the board with pinMode(pin, input/output)
  // use digitalRead(pin) and digitalWrite(pin, high/low)
  Serial.begin(9600);

}

void loop() {
  // put your main code here, to run repeatedly:
  unsigned long curr_m = millis();
  if (curr_m - old_m_1 ≥ 10) { // runs this every 1000 ms
    old_m_1 = millis();
```

```
    robotCount(); // updates robots (value) if robots enter/exit and ...
        enables exit traffic lights upon exit of robots
    lightsControl(); // enables/disables entry traffic lights of pads and ...
        station
    kickOff();
    if (robot_to_em && (curr_m - robot_enter ≥ 5000) && pad_em_on != 0) {
      emergencyQueue(); // checks if emergency queue updates
    }
    if (curr_m - pad_1_exit ≥ exit_timer) { // disables charging pad exit ...
        lights if on for ... ms
      if (pad_1_out) {Serial.println("Disable charging pad 1 exit traffic ...
          light.");}
      pad_1_out = false;
    }
    if (curr_m - pad_2_exit ≥ exit_timer) {
      if (pad_2_out) {Serial.println("Disable charging pad 2 exit traffic ...
          light.");}
      pad_2_out = false;
    }
    if (curr_m - pad_3_exit ≥ exit_timer) {
      if (pad_3_out) {Serial.println("Disable charging pad 3 exit traffic ...
          light.");}
      pad_3_out = false;
    }
    if (curr_m - station_exit ≥ exit_timer) { // disable exit lights for ...
        going right a station entrance
      if (station_out) {Serial.println("Disable charging station exit ...
          traffic light.");}
      station_out = false;
    }
    if (curr_m - waiting_out_time ≥ 2000 && station_in_enable) {
      if (!station_in) {Serial.println("Enable charging station entrance ...
          traffic light.");}
      station_in = true;
      station_in_enable = false;
    }
    if (robots != robots_old) {
    Serial.print("There are currently: ");
    Serial.print(robots);
    Serial.println(" Duckiebots in the charging station.");
    robots_old = robots;
    }
  }
  testBench(); // to simulate various inputs
}

void testBench() {
  unsigned long curr_m = millis();
  // inputs: bb_in, bb_out, bb_in2 all at least 1 cycles on, robot in, ...
      out, turns right at entrance
  // inputs: pad 1, 2, 3 SoC, 0 means no robot on pad
  // inputs: pad 1, 2, 3 mSoC, minimum desired SoC, kick off requests ...
      based off lowest mSoC - SoC
  if (old_m_2 == 0) {old_m_2 = millis();} // testbench starts when this is ...
      executed
  if (curr_m - old_m_2 ≥ 1) {
    bb_in = true;
```

```
  }
  if (curr_m - old_m_2 ≥ 15) {
    bb_in2 = true;
  }
  if (curr_m - old_m_2 ≥ 30) { // 1 robot inside
    bb_in = false;
  }
  if (curr_m - old_m_2 ≥ 40 ) { // robot turned right at entrance
    bb_in2 = false;
  }
  if (curr_m - old_m_2 ≥ 3000) { // robot on pad 1
    pad_1_SoC = 50;
    pad_1_mSoC = 80;
  }
  if (curr_m - old_m_2 ≥ 3500) {
    bb_in = true;
  }
  if (curr_m - old_m_2 ≥ 3530) { // 2 robots inside
    bb_in = false;
  }
  if (curr_m - old_m_2 ≥ 6500) { // robot on pad 1 and 2
    pad_2_SoC = 10;
    pad_2_mSoC = 70;
  }
  if (curr_m - old_m_2 ≥ 6600) {
    bb_in = true;
  }
  if (curr_m - old_m_2 ≥ 6630) { // 3 robots inside
    bb_in = false;
  }
  if (curr_m - old_m_2 ≥ 9500) { // robot on pad 1, 2 and 3
    pad_3_SoC = 30;
    pad_3_mSoC = 60;
  }
  if (curr_m - old_m_2 ≥ 10000) {
    bb_in = true;
  }
  if (curr_m - old_m_2 ≥ 10030) { // 4 robots inside
    bb_in = false;
  }
  if (curr_m - old_m_2 ≥ 10150) { // robot on pad 1 left to make space for incoming robot, 3 robots inside
    pad_1_SoC = 0;
    pad_1_mSoC = 0;
  }
  if (curr_m - old_m_2 ≥ 13000) { // robot on pad 1, 2 and 3
    pad_1_SoC = 5;
    pad_1_mSoC = 55;
  }
  if (curr_m - old_m_2 ≥ 14000) {
    bb_in = true;
  }
  if (curr_m - old_m_2 ≥ 14030) { // 4 robots inside, robot will go to emergency queue of charging pad 3
    bb_in = false;
  }
  if (curr_m - old_m_2 ≥ 20000) { //
```

```
      bb_in = true;
    }
    if (curr_m - old_m_2 ≥ 20030) { // 5 robots inside, robot will go to ...
        emergency queue of charging pad 1
      bb_in = false;
    }
    if (curr_m - old_m_2 ≥ 26000) { //
      bb_in = true;
    }
    if (curr_m - old_m_2 ≥ 26030) { // 6 robots inside, robot will go to ...
        emergency queue of charging pad 2
      bb_in = false;
    }
    if (curr_m - old_m_2 ≥ 32000) { //
      bb_in = true;
    }
    if (curr_m - old_m_2 ≥ 32030) { // 7 robots inside
      bb_in = false;
    }
    if (curr_m - old_m_2 ≥ 36030) {
      bb_out = true;
    }
    if (curr_m - old_m_2 ≥ 36060) { // 6 robots inside, robot left through ...
        exit breakbeam
      bb_out = false;
    }
    if (curr_m - old_m_2 ≥ 37000) { //
      bb_in = true;
    }
    if (curr_m - old_m_2 ≥ 37030) { // 7 robots inside
      bb_in = false;
    }
    if (curr_m - old_m_2 ≥ 37330) { // robots on pad 1 and 3 left
      pad_1_SoC = 0;
      pad_1_mSoC = 0;
      pad_3_SoC = 0;
      pad_3_mSoC = 0;
    }
    if (curr_m - old_m_2 ≥ 38030) { // robots in emergency queue moved up, ...
        charging pads 1, 2 and 3 occupied, emergency queue on pads 1 and 3 ...
        now empty
                                    // robot will go to emergency queue of ...
                                        charging pad 1
      pad_1_SoC = 20;
      pad_1_mSoC = 80;
      pad_3_SoC = 15;
      pad_3_mSoC = 85;
    }
  }


  void kickOff() { // kicks off robots if needed
    unsigned long curr_m2 = millis();

    if ((kickoff_priority == 1) && (curr_m2 - kickoff_start > kickoff_high)) {
      pad_1_kick = false;
      pad_2_kick = false;
```

```
      pad_3_kick = false;
      //Serial.println("kick_high off");
    }
    else if ((kickoff_priority == 0) && (curr_m2 - kickoff_start > ...
        kickoff_low)) {
      pad_1_kick = false;
      pad_2_kick = false;
      pad_3_kick = false;
      //Serial.println("kick_low off");
    }
    if (robot_to_em){ // if all pads and emergency queues occupied and ...
        another robot enters, tries to kick off all robots on pads with high ...
        priority
      if (pad_1_em && pad_2_em && pad_3_em && kickoff_tries == 0) { // if ...
          all emergency queues occupied, no robot can enter anywhere
        pad_1_kick = true;
        pad_2_kick = true;
        pad_3_kick = true;
        kickoff_start = millis();
        kickoff_priority = 1;
        kickoff_tries = -1;
        Serial.println("Ask all Duckiebots on charging pads to leave.");
      }
      else { // else kick off robots with occupied em queue and try to get 1 ...
          free pad with low priority
        kickoff_priority = 0;
        if (kickoff_tries == 0) { // finds robot that will first leave
          //Serial.println("Kick off try 1");
          if (pad_1_em) {pad_1_kick = true;
            Serial.println("Ask the Duckiebot on charging pad 1 to leave.");}
          if (pad_2_em) {pad_2_kick = true;
            Serial.println("Ask the Duckiebot on charging pad 2 to leave.");}
          if (pad_3_em) {pad_3_kick = true;
            Serial.println("Ask the Duckiebot on charging pad 3 to leave.");}
          if (!pad_1_em && (pad_2_em || (pad_2_mSoC - pad_2_SoC ≥ pad_1_mSoC ...
              - pad_1_SoC)) && (pad_3_em || (pad_3_mSoC - pad_3_SoC ≥ ...
              pad_1_mSoC - pad_1_SoC))) {
            pad_1_kick = true;
            Serial.println("Ask the Duckiebot on charging pad 1 to leave.");
            kick_prev = 1;
          }
          else if (!pad_2_em && (pad_3_em || (pad_3_mSoC - pad_3_SoC ≥ ...
              pad_2_mSoC - pad_2_SoC))) {
            pad_2_kick = true;
            Serial.println("Ask the Duckiebot on charging pad 2 to leave.");
            kick_prev = 2;
          }
          else {
            pad_3_kick = true;
            Serial.println("Ask the Duckiebot on charging pad 3 to leave.");
            kick_prev = 3;
          }
          kickoff_tries += 1;
          kickoff_start = millis();
        }
        else if (kickoff_tries == 1 && (curr_m2 - kickoff_start ≥ 200)) { // ...
            finds next robot to leave
```

```
        //Serial.println("Kick off try 2");
        if (!pad_1_em && (kick_prev != 1) && (pad_2_em || (pad_2_mSoC - ...
            pad_2_SoC ≥ pad_1_mSoC - pad_1_SoC) || kick_prev == 2) && ...
            (pad_3_em || (pad_3_mSoC - pad_3_SoC ≥ pad_1_mSoC - pad_1_SoC ...
            || kick_prev == 3))) {
          pad_1_kick = true;
          Serial.println("Ask the Duckiebot on charging pad 1 to leave.");
        }
        else if (!pad_2_em && (kick_prev != 2) && (pad_3_em || (pad_3_mSoC ...
            - pad_3_SoC ≥ pad_2_mSoC - pad_2_SoC))) {
          pad_2_kick = true;
          Serial.println("Ask the Duckiebot on charging pad 2 to leave.");
        }
        else if (!pad_3_em && (kick_prev != 3)) {
          pad_3_kick = true;
          Serial.println("Ask the Duckiebot on charging pad 3 to leave.");
        }
        kickoff_tries += 1;
        kickoff_start = millis();
      }
      else if (kickoff_tries == 2 && !pad_1_em && !pad_2_em && !pad_3_em ...
          && (curr_m2 - kickoff_start ≥ 200)) { // finds last robot to leave
        //Serial.println("Kick off try 3");
        if ((pad_2_mSoC - pad_2_SoC < pad_1_mSoC - pad_1_SoC) && ...
            (pad_3_mSoC - pad_3_SoC < pad_1_mSoC - pad_1_SoC)) {
          pad_1_kick = true;
          Serial.println("Ask the Duckiebot on charging pad 1 to leave.");
        }
        else if (pad_3_mSoC - pad_3_SoC < pad_2_mSoC - pad_2_SoC) {
          pad_2_kick = true;
          Serial.println("Ask the Duckiebot on charging pad 2 to leave.");
        }
        else {
          pad_3_kick = true;
          Serial.println("Ask the Duckiebot on charging pad 3 to leave.");
        }
        kickoff_tries = -1;
      }
    }
  }
}

void emergencyQueue() {
    robot_to_em = false;
    if (pad_em_on == 1) {
      if (!pad_1_em) {Serial.println("Charging pad 1 emergency queue ...
          occupied.");}
      pad_1_em = true;
    }
    else if (pad_em_on == 2) {
      if (!pad_2_em) {Serial.println("Charging pad 2 emergency queue ...
          occupied.");}
      pad_2_em = true;
    }
    else if (pad_em_on == 3) {
      if (!pad_3_em) {Serial.println("Charging pad 3 emergency queue ...
          occupied.");}
```

```
      pad_3_em = true;
    }
}

void lightsControl() {
  if (robots ≥ charging_pads) {
    if (waiting_out) {Serial.println("Disable waiting area exit traffic ...
        light.");}
    if (station_in) {Serial.println("Disable charging station entrance ...
        traffic light.");}
    waiting_out = false;
    station_in = false;
    station_in_enable = false;
  }
  else {
    if (!waiting_out) {Serial.println("Enable waiting area exit traffic ...
        light.");}
    waiting_out = true;
    if (!station_in_enable) { // to prevent it from resetting ...
        waiting_out_time constantly
      waiting_out_time = millis(); // station_in will be activated with a ...
          small delay to (hopefully) prevent collisions
    }
    station_in_enable = true;
    //Serial.println("enable station");
  }
  // if a robot enters the charging station before the previous robot has ...
      stopped, it might cause problems
  if (pad_1_occ && pad_2_occ && pad_3_occ) { // if all pads occupied, ...
      entrance light on/off depends on emergency queue
    if (pad_1_em && pad_2_em && pad_3_em) { // if all emergency queues ...
        occupied, no robot can enter anywhere
      if (pad_1_in) {Serial.println("Disable charging pad 1 entrance ...
          traffic light.");}
      if (pad_2_in) {Serial.println("Disable charging pad 2 entrance ...
          traffic light.");}
      if (pad_3_in) {Serial.println("Disable charging pad 3 entrance ...
          traffic light.");}
      pad_1_in = false;
      pad_2_in = false;
      pad_3_in = false;
      pad_em_on = 0;
    }
    else{
      // enable entrance light of charging pad with lowest mSoC - SoC and ...
          no robot in emergency queue, other entrance lights off
      if (!pad_1_em && (pad_2_em || (pad_2_mSoC - pad_2_SoC ≥ pad_1_mSoC - ...
          pad_1_SoC)) && (pad_3_em || (pad_3_mSoC - pad_3_SoC ≥ pad_1_mSoC ...
          - pad_1_SoC))) {
        if (!pad_1_in) {Serial.println("Enable charging pad 1 entrance ...
            traffic light.");}
        if (pad_2_in) {Serial.println("Disable charging pad 2 entrance ...
            traffic light.");}
        if (pad_3_in) {Serial.println("Disable charging pad 3 entrance ...
            traffic light.");}
        pad_1_in = true;
        pad_2_in = false;
```

```
          pad_3_in = false;
          pad_em_on = 1;
        }
        // light 1 if off, so between 2 and 3
        else if (!pad_2_em && (pad_3_em || (pad_3_mSoC - pad_3_SoC ≥ ...
            pad_2_mSoC - pad_2_SoC))) {
          if (pad_1_in) {Serial.println("Disable charging pad 1 entrance ...
              traffic light.");}
          if (!pad_2_in) {Serial.println("Enable charging pad 2 entrance ...
              traffic light.");}
          if (pad_3_in) {Serial.println("Disable charging pad 3 entrance ...
              traffic light.");}
          pad_1_in = false;
          pad_2_in = true;
          pad_3_in = false;
          pad_em_on = 2;
        }
        else { // lights 1 and 2 are off, so 3 is on
          if (pad_1_in) {Serial.println("Disable charging pad 1 entrance ...
              traffic light.");}
          if (pad_2_in) {Serial.println("Disable charging pad 2 entrance ...
              traffic light.");}
          if (!pad_3_in) {Serial.println("Enable charging pad 3 entrance ...
              traffic light.");}
          pad_1_in = false;
          pad_2_in = false;
          pad_3_in = true;
          pad_em_on = 3;
        }

      }
    }
    else{ // on/off depends on occupied
      if (pad_1_occ) {
        if (pad_1_in) {Serial.println("Disable charging pad 1 entrance ...
            traffic light.");}
        pad_1_in = false;}
      else {
        if (!pad_1_in) {Serial.println("Enable charging pad 1 entrance ...
            traffic light.");}
        pad_1_in = true;}
      if (pad_2_occ) {
        if (pad_2_in) {Serial.println("Disable charging pad 2 entrance ...
            traffic light.");}
        pad_2_in = false;}
      else {
        if (!pad_2_in) {Serial.println("Enable charging pad 2 entrance ...
            traffic light.");}
        pad_2_in = true;}
      if (pad_3_occ) {
        if (pad_3_in) {Serial.println("Disable charging pad 3 entrance ...
            traffic light.");}
        pad_3_in = false;}
      else {
        if (!pad_3_in) {Serial.println("Enable charging pad 3 entrance ...
            traffic light.");}
        pad_3_in = true;}
```

```
        pad_em_on = 0;
    }
}

void robotCount() {
  if (bb_in_state == 1) { // if robot blocks entrance breakbeam sensor for ...
      10-20 ms, it will count another robot entering the charging station
      bb_in_state = breakBeam_update(bb_in, bb_in_state);
      if (bb_in_state == 2) {
        robots += 1;
        Serial.println("Duckiebot enters charging station.");
        if (robots ≥ 4) { // if emergency queue used and robot enters
          robot_to_em = true;
          kickoff_tries = 0;
          robot_enter = millis();
        }
      }
    }
  else {
    bb_in_state = breakBeam_update(bb_in, bb_in_state);
  }
  if (bb_out_state == 1) { // if robot blocks exit breakbeam sensor for ...
      10-20 ms, it will count another robot exiting the charging station
    bb_out_state = breakBeam_update(bb_out, bb_out_state);
    if (bb_out_state == 2) {
      robots -= 1;
      Serial.println("Duckiebot exits charging station");
        if (robots ≥ 4) { // if emergency queue used and robot exits ...
            through exit breakbeam
          robot_to_em = false;
        }
    }
  }
  else {
    bb_out_state = breakBeam_update(bb_out, bb_out_state);
  }
  if (bb_in2_state == 1) { // if robot blocks breakbeam sensor to to the ...
      right of entrance for 10-20 ms, it will count activate station exit light
    bb_in2_state = breakBeam_update(bb_in2, bb_in2_state);
    if (bb_in2_state == 2) {
      station_out = true;
      station_exit = millis();
      Serial.println("Duckiebot turns right at charging station entrance.");
    }
  }
  else {
    bb_in2_state = breakBeam_update(bb_in2, bb_in2_state);
  }

  if (!pad_1_lock && pad_1_occ && pad_1_SoC == 0) { // robot left charging pad
    if (pad_1_em) { // robot in emergency queue now goes to charging pad
      pad_1_em = false;
      Serial.println("Duckiebot leaves charging pad 1, Duckiebot in ...
          emergency queue of charging pad 1 goes to charging pad 1.");
      pad_1_lock = true;}
    else {
      robot_to_em = false; // else possible incoming robot goes here instead
```

```
      Serial.println("Duckiebot leaves charging pad 1.");
      pad_1_occ = false;
    }
    robots -= 1;
    pad_1_out = true; // enable exit traffic light
    Serial.println("Enable charging pad 1 exit traffic light.");
    pad_1_exit = millis(); // save current time to disable exit traffic ...
        light after ... sec
  }
  else if ((!pad_1_occ || pad_1_lock) && pad_1_SoC != 0) { // robot ...
      entered charging pad
    pad_1_occ = true;
    Serial.println("Duckiebot entered charging pad 1.");
    pad_1_lock = false;
  }
  if (!pad_2_lock && pad_2_occ && pad_2_SoC == 0) { // robot left charging pad
    if (pad_2_em) { // robot in emergency queue now goes to charging pad
      pad_2_em = false;
      Serial.println("Duckiebot leaves charging pad 2, Duckiebot in ...
          emergency queue of charging pad 2 goes to charging pad 2.");
      pad_2_lock = true;}
    else {
      robot_to_em = false; // else possible incoming robot goes here instead
      Serial.println("Duckiebot leaves charging pad 2.");
      pad_2_occ = false;
    }
    robots -= 1;
    pad_2_out = true; // enable exit traffic light
    Serial.println("Enable charging pad 2 exit traffic light.");
    pad_2_exit = millis(); // save current time to disable exit traffic ...
        light after ... sec
  }
  else if ((!pad_2_occ || pad_2_lock) && pad_2_SoC != 0) { // robot ...
      entered charging pad
    pad_2_occ = true;
    Serial.println("Duckiebot entered charging pad 2.");
    pad_2_lock = false;
  }
  if (!pad_3_lock && pad_3_occ && pad_3_SoC == 0) { // robot left charging pad
    if (pad_3_em) { // robot in emergency queue now goes to charging pad
      pad_3_em = false;
      Serial.println("Duckiebot leaves charging pad 3, Duckiebot in ...
          emergency queue of charging pad 3 goes to charging pad 3.");
      pad_3_lock = true;}
    else {
      robot_to_em = false; // else possible incoming robot goes here instead
      Serial.println("Duckiebot leaves charging pad 3.");
      pad_3_occ = false;
      }
    robots -= 1;
    pad_3_out = true; // enable exit traffic light
    Serial.println("Enable charging pad 3 exit traffic light.");
    pad_3_exit = millis(); // save current time to disable exit traffic ...
        light after ... sec
  }
  else if ((!pad_3_occ || pad_3_lock) && pad_3_SoC != 0) { // robot ...
      entered charging pad
```

```
    pad_3_occ = true;
    Serial.println("Duckiebot entered charging pad 2.");
    pad_3_lock = false;
  }
}

int breakBeam_update(int beam_name, int beam_state) { // returns breakbeam ...
    state value: 0, 1 or 2, 0 means no robot seen at last check,
  // 1 means robot seen at last check, but not at one before that, 2 means ...
      robot seen at last check and at 1+ before
  // if signal goes from 0 to 1 to 0, seen sensor error, not car
  int beam_ret = 2;
  if (beam_name){ // signal not blocked, so no car
    beam_ret = 0;
  }
  else if (beam_state == 0) { // car blocks signal
    beam_ret = 1;
  }
  return beam_ret; // returns the new beam state
}
```

A-2 Algorithm implementation code

```

int charging_pads = 3; // amount of charging pads
int robots = 0; // robots in charging station
bool pad_1_occ = true; // true if pad was occupied, false if not, updates ...
    based on pad_1_SoC
bool pad_2_occ = true;
bool pad_3_occ = true;
bool pad_1_lock = false; // keeps pad_1_occ enabled until pad_1_soc ...
    detects a robot
bool pad_2_lock = false;
bool pad_3_lock = false;
int pad_1_SoC = 0; // SoC of robot on pad, 7 bit, 000000 is no robot, else ...
    robot on pad, updates based on I2C (not implemented yet)
int pad_2_SoC = 0;
int pad_3_SoC = 0;
int pad_1_mSoC = 0; // minimum desired SoC of robot on pad, 7 bit?
int pad_2_mSoC = 0;
int pad_3_mSoC = 0;
bool pad_1_em = false; // is emergency queue occupied?
bool pad_2_em = false;
bool pad_3_em = false;
bool robot_to_em = false;
int pad_em_on = 0; // 0 if 0-2 pads occupied, or 6 pads occupied, pin of ...
    pad which has entrance light on
unsigned long exit_timer = 5000; // how long until the exit light goes off ...
    again in ms
unsigned long robot_enter = 0; // time when robot enters if 3 < x ≤ 6 pads ...
    occupied
unsigned long pad_1_exit = 0; // the time when the exit light goes on, ...
    used to disable exit light after ... ms.
```

```
unsigned long pad_2_exit = 0;
unsigned long pad_3_exit = 0;
unsigned long station_exit = 0;
unsigned long kickoff_high = 100; // how long the light is on for a high ...
    priority kick off request (all pads and emergency queues occupied)
unsigned long kickoff_low = 30; // how long the light is on for a low ...
    priority kick off request (all pads but not all emergency queues occupied)
unsigned long kickoff_start = 0; // when kickoff attempt starts
unsigned long waiting_out_time = 0; // the amount of time between the ...
    waiting area light and the entrance light activating (waiting first)
bool station_in_enable = false; // allows shift_array[0] to be on
int kickoff_tries = 0; // how many kick off requests this round
int kickoff_priority; // 1 = high, 0 = low
int kick_prev = 0; // last kick off attempt
int bb_in_state = 2; // 0 means no robot seen at last check, 1 means robot ...
    seen at last check, but not one before,
                     // 2 means robot seen at last check and at 1+ before, ...
                         if signal goes from 0 to 1 to 0, seen sensor error, ...
                         not car
int bb_out_state = 2;
int bb_in2_state = 2;
unsigned long old_m_1 = 0; // old value in milliseconds
unsigned long old_m_3 = 0; // old value in ms
int storage_clock_counter = 0; // how many bits sent
int x = 0;

// pin assignments
// traffic lights, update pin numbers to correct values, no pins, shift ...
    register now
// since shift_array[0] enters the register first, it will be output from ...
    the last output bit of the register, not the first
bool shift_array[] = {false, false, false, false, false, false, false, ...
    false, false, false, false, false};
//bool shift_array[0] = false; // at station entrance
//bool shift_array[1] = false; // at pad 1 entrance
//bool shift_array[2] = false; // at pad 2 entrance
//bool shift_array[3] = false; // at pad 3 entrance
//bool shift_array[4] = false; // at pad 1 out
//bool shift_array[5] = false; // at pad 2 out
//bool shift_array[6] = false; // at pad 3 out
//bool shift_array[7] = false; // at station out
//bool shift_array[8] = false; // at pad 1 kick off
//bool shift_array[9] = false; // at pad 2 kick off
//bool shift_array[10] = false; // at pad 3 kick off
//bool shift_array[11] = false; // at waiting area out
// shift register pins
int data_serial = 0; // the input data
int storage_clock = 0; // updates output based on stored data
int shift_clock = 0; // stores current input data, moves all data 1 place ...
    further
// breakbeams
int bb_in = 12; // charging station entrance breakbeam
int bb_out = 13; // charging station exit breakbeam
int bb_in2 = 14; // breakbeam for robots that turn right at entrance

// I2C??? how to implement this, maybe use list for bits and then decode ...
    when done?
```

```

void setup() {
  // put your setup code here, to run once:
  // use the pin numbers from the board with pinMode(pin, input/output)
  // use digitalRead(pin) and digitalWrite(pin, high/low)
  pinMode(data_serial, OUTPUT);
  pinMode(storage_clock, OUTPUT);
  pinMode(shift_clock, OUTPUT);
  pinMode(bb_in, INPUT);
  pinMode(bb_out, INPUT);
  pinMode(bb_in2, INPUT);
  Serial.begin(9600);
}

void loop() {
  // put your main code here, to run repeatedly:
  unsigned long curr_m = millis();
  if (curr_m - old_m_1 ≥ 1000) { // runs this every 10 ms
    old_m_1 = millis();
    robotCount(); // updates robots (value) if robots enter/exit and ...
        enables exit traffic lights upon exit of robots
    lightsControl(); // enables/disables entry traffic lights of pads and ...
        station
    kickOff(); // tries to kick off robots if needed
    if (robot_to_em && (curr_m - robot_enter ≥ 5000) && pad_em_on != 0) {
      emergencyQueue(); // checks if emergency queue updates
    }
    if (curr_m - pad_1_exit ≥ exit_timer) { // disables charging pad exit ...
        lights if on for ... ms
      shift_array[4] = false;
    }
    if (curr_m - pad_2_exit ≥ exit_timer) {
      shift_array[5] = false;
    }
    if (curr_m - pad_3_exit ≥ exit_timer) {
      shift_array[6] = false;
    }
    if (curr_m - station_exit ≥ exit_timer) { // disable exit lights for ...
        going right a station entrance
      shift_array[7] = false;
    }
    if (curr_m - waiting_out_time ≥ 2000 && station_in_enable) {
      shift_array[0] = true;
      station_in_enable = false;
    }
    Serial.println(robots);
  }
  if (curr_m - old_m_3 ≥ 1) { // every 1 ms
    old_m_3 = millis();
    shiftReg();
  }
}

void shiftReg() { // sends data to shift register
  if (x ≤ 31) {
    if (x % 2 == 0) { // 2 8 bit shift register, so 16 bits sent
      digitalWrite(shift_clock, LOW); // shifts at rising edge
```

```
      if (x < 26) { // we only have 12 bits input and 2*12 < 26
        if (shift_array[x]){
          digitalWrite(data_serial, HIGH); // this bit is high
        }
        else {
          digitalWrite(data_serial, LOW); // this bit is low
        }
      }
      else {
        digitalWrite(data_serial, LOW); // not needed, so assumed low
      }
    }
    else { // shifts everything 1 bit further
      digitalWrite(shift_clock, HIGH); // shifts values 1 bit
    }
    x += 1;
  }
  else {
    digitalWrite(storage_clock, HIGH); // updates the output
    x = 0;
  }
}

void kickOff() { // kicks off robots if needed
  unsigned long curr_m2 = millis();

  if ((kickoff_priority == 1) && (curr_m2 - kickoff_start > kickoff_high)) {
    shift_array[8] = false;
    shift_array[9] = false;
    shift_array[10] = false;
    Serial.println("kick_high off");
  }
  else if ((kickoff_priority == 0) && (curr_m2 - kickoff_start > ...
      kickoff_low)) {
    shift_array[8] = false;
    shift_array[9] = false;
    shift_array[10] = false;
    Serial.println("kick_low off");
  }
  if (robot_to_em){ // if all pads and emergency queues occupied and ...
      another robot enters, tries to kick off all robots on pads with high ...
      priority
    if (pad_1_em && pad_2_em && pad_3_em && kickoff_tries == 0) { // if ...
        all emergency queues occupied, no robot can enter anywhere
    shift_array[8] = true;
    shift_array[9] = true;
    shift_array[10] = true;
      kickoff_start = millis();
      kickoff_priority = 1;
      kickoff_tries = -1;
    Serial.println("kick all em");
    }
    else { // else kick off robots with occupied em queue and try to get 1 ...
        free pad with low priority
      kickoff_priority = 0;
      if (kickoff_tries == 0) { // finds robot that will first leave
        Serial.println("kick try 1");
```

```
        if (pad_1_em) {shift_array[8] = true;}
        if (pad_2_em) {shift_array[9] = true;}
        if (pad_3_em) {shift_array[10] = true;}
        if (!pad_1_em && (pad_2_em || (pad_2_mSoC - pad_2_SoC ≥ pad_1_mSoC ...
            - pad_1_SoC)) && (pad_3_em || (pad_3_mSoC - pad_3_SoC ≥ ...
            pad_1_mSoC - pad_1_SoC))) {
          shift_array[8] = true;
          kick_prev = 1;
        }
        else if (!pad_2_em && (pad_3_em || (pad_3_mSoC - pad_3_SoC ≥ ...
            pad_2_mSoC - pad_2_SoC))) {
          shift_array[9] = true;
          kick_prev = 2;
        }
        else {
          shift_array[10] = true;
          kick_prev = 3;
        }
        kickoff_tries += 1;
        kickoff_start = millis();
      }
      else if (kickoff_tries == 1 && (curr_m2 - kickoff_start ≥ 200)) { // ...
          finds next robot to leave
        Serial.println("kick try 2");
        if (!pad_1_em && (kick_prev != 1) && (pad_2_em || (pad_2_mSoC - ...
            pad_2_SoC ≥ pad_1_mSoC - pad_1_SoC)) && (pad_3_em || ...
            (pad_3_mSoC - pad_3_SoC ≥ pad_1_mSoC - pad_1_SoC))) {
          shift_array[8] = true;
        }
        else if (!pad_2_em && (kick_prev != 2) && (pad_3_em || (pad_3_mSoC ...
            - pad_3_SoC ≥ pad_2_mSoC - pad_2_SoC))) {
          shift_array[9] = true;
        }
        else if (!pad_3_em && (kick_prev != 3)) {
          shift_array[10] = true;
        }
        kickoff_tries += 1;
        kickoff_start = millis();
      }
      else if (kickoff_tries == 2 && !pad_1_em && !pad_2_em && !pad_3_em ...
          && (curr_m2 - kickoff_start ≥ 200)) { // finds last robot to leave
        Serial.println("kick try 3");
        if ((pad_2_mSoC - pad_2_SoC < pad_1_mSoC - pad_1_SoC) && ...
            (pad_3_mSoC - pad_3_SoC < pad_1_mSoC - pad_1_SoC)) {
          shift_array[8] = true;
        }
        else if (pad_3_mSoC - pad_3_SoC < pad_2_mSoC - pad_2_SoC) {
          shift_array[9] = true;
        }
        else {
          shift_array[10] = true;
        }
        kickoff_tries = -1;
      }
    }
  }
}
```

```

void emergencyQueue() {
    robot_to_em = false;
    if (pad_em_on == 1) {
      pad_1_em = true;
    Serial.println("pad_1_em true");
    }
    else if (pad_em_on == 2) {
      pad_2_em = true;
    Serial.println("pad_2_em true");
    }
    else if (pad_em_on == 3) {
      pad_3_em = true;
    Serial.println("pad_3_em true");
    }
}

void lightsControl() {
  if (robots ≥ charging_pads) {
    shift_array[11] = false;
    shift_array[0] = false;
    station_in_enable = false;
    Serial.println("disable station");
  }
  else {
    shift_array[11] = true;
    if (!station_in_enable) { // to prevent it from resetting ...
        waiting_out_time constantly
      waiting_out_time = millis(); // shift_array[0] will be activated ...
          with a small delay to (hopefully) prevent collisions
    }
    station_in_enable = true;
    Serial.println("enable station");
  }
  // if a robot enters the charging station before the previous robot has ...
      stopped, it might cause problems
  if (pad_1_occ && pad_2_occ && pad_3_occ) { // if all pads occupied, ...
      entrance light on/off depends on emergency queue
    if (pad_1_em && pad_2_em && pad_3_em) { // if all emergency queues ...
        occupied, no robot can enter anywhere
      shift_array[1] = false;
      shift_array[2] = false;
      shift_array[3] = false;
      pad_em_on = 0;
    }
    else{
      // enable entrance light of charging pad with lowest mSoC - SoC and ...
          no robot in emergency queue, other entrance lights off
      if (!pad_1_em && (pad_2_em || (pad_2_mSoC - pad_2_SoC ≥ pad_1_mSoC - ...
          pad_1_SoC)) && (pad_3_em || (pad_3_mSoC - pad_3_SoC ≥ pad_1_mSoC ...
          - pad_1_SoC))) {
        shift_array[1] = true;
        shift_array[2] = false;
        shift_array[3] = false;
        pad_em_on = 1;
      }
      // light 1 if off, so between 2 and 3
```

```
      else if (!pad_2_em && (pad_3_em || (pad_3_mSoC - pad_3_SoC ≥ ...
          pad_2_mSoC - pad_2_SoC))) {
        shift_array[1] = false;
        shift_array[2] = true;
        shift_array[3] = false;
        pad_em_on = 2;
      }
      else { // lights 1 and 2 are off, so 3 is on
        shift_array[1] = false;
        shift_array[2] = false;
        shift_array[3] = true;
        pad_em_on = 3;
      }

    }
  }
  else{ // on/off depends on occupied
    if (pad_1_occ) {shift_array[1] = false;}
    else {shift_array[1] = true;}
    if (pad_2_occ) {shift_array[2] = false;}
    else {shift_array[2] = true;}
    if (pad_3_occ) {shift_array[3] = false;}
    else {shift_array[3] = true;}
        pad_em_on = 0;
  }
}

void robotCount() {
  if (bb_in_state == 1) { // if robot blocks entrance breakbeam sensor for ...
      10-20 ms, it will count another robot entering the charging station
      bb_in_state = breakBeam_update(bb_in, bb_in_state);
      if (bb_in_state == 2) {
        robots += 1;
        if (robots ≥ 4) { // if emergency queue used and robot enters
          robot_to_em = true;
          kickoff_tries = 0;
          robot_enter = millis();
        }
        Serial.println("bb_in on");
      }
    }
  else {
    bb_in_state = breakBeam_update(bb_in, bb_in_state);
  }
  if (bb_out_state == 1) { // if robot blocks exit breakbeam sensor for ...
      10-20 ms, it will count another robot exiting the charging station
    bb_out_state = breakBeam_update(bb_out, bb_out_state);
    if (bb_out_state == 2) {
      robots -= 1;
        if (robots ≥ 4) { // if emergency queue used and robot exits ...
            through exit breakbeam
          robot_to_em = false;
        }
      Serial.println("bb_out on");
    }
  }
  else {
```

```
    bb_out_state = breakBeam_update(bb_out, bb_out_state);
  }
  if (bb_in2_state == 1) { // if robot blocks breakbeam sensor to to the ...
      right of entrance for 10-20 ms, it will count activate station exit light
    bb_in2_state = breakBeam_update(bb_in2, bb_in2_state);
    if (bb_in2_state == 2) {
      shift_array[7] = true;
      station_exit = millis();
      Serial.println("bb_in_2 on");
    }
  }
  else {
    bb_in2_state = breakBeam_update(bb_in2, bb_in2_state);
  }

  if (!pad_1_lock && pad_1_occ && pad_1_SoC == 0) { // robot left charging pad
    if (pad_1_em) { // robot in emergency queue now goes to charging pad
      pad_1_em = false;
      pad_1_lock = true;}
    else {
      robot_to_em = false; // else possible incoming robot goes here instead
      pad_1_occ = false;
    }
    robots -= 1;
    shift_array[4] = true; // enable exit traffic light
    pad_1_exit = millis(); // save current time to disable exit traffic ...
        light after ... sec
  }
  else if ((!pad_1_occ || pad_1_lock) && pad_1_SoC != 0) { // robot ...
      entered charging pad
    pad_1_occ = true;
    pad_1_lock = false;
  }
  if (!pad_2_lock && pad_2_occ && pad_2_SoC == 0) { // robot left charging pad
    if (pad_2_em) { // robot in emergency queue now goes to charging pad
      pad_2_em = false;
      pad_2_lock = true;}
    else {
      robot_to_em = false; // else possible incoming robot goes here instead
      pad_2_occ = false;
    }
    robots -= 1;
    shift_array[5] = true; // enable exit traffic light
    pad_2_exit = millis(); // save current time to disable exit traffic ...
        light after ... sec
  }
  else if ((!pad_2_occ || pad_2_lock) && pad_2_SoC != 0) { // robot ...
      entered charging pad
    pad_2_occ = true;
    pad_2_lock = false;
  }
  if (!pad_2_lock && pad_3_occ && pad_3_SoC == 0) { // robot left charging pad
    if (pad_3_em) { // robot in emergency queue now goes to charging pad
      pad_3_em = false;
      pad_3_lock = true;}
    else {
      robot_to_em = false; // else possible incoming robot goes here instead
```

```
      pad_3_occ = false;
      }
    robots -= 1;
    shift_array[6] = true; // enable exit traffic light
    pad_3_exit = millis(); // save current time to disable exit traffic ...
        light after ... sec
  }
  else if ((!pad_3_occ || pad_3_lock) && pad_3_SoC != 0) { // robot ...
      entered charging pad
    pad_3_occ = true;
    pad_3_lock = false;
  }
}

int breakBeam_update(int beam_name, int beam_state) { // returns breakbeam ...
    state value: 0, 1 or 2, 0 means no robot seen at last check,
  // 1 means robot seen at last check, but not at one before that, 2 means ...
      robot seen at last check and at 1+ before
  // if signal goes from 0 to 1 to 0, seen sensor error, not car
  int beam_ret = 2;
  if (digitalRead(beam_name) == 1){ // signal not blocked, so no car
    beam_ret = 0;
  }
  else if (beam_state == 0) { // car blocks signal
    beam_ret = 1;
  }
  return beam_ret; // returns the new beam state
}
```