




MARVigators

Group 3

Name and Surname	Student Number	Signature
Brendon de Beer	04842384	
Reinhardt von Brandis-Martini	20461632	
Samuel Dachs	20491685	

By signing this assignment we confirm that we have read and are aware of the University of Pretoria's policy on academic dishonesty and plagiarism and we declare that the work submitted in this assignment is our own as delimited by the mentioned policies. We explicitly declare that no parts of this assignment have been copied from current or previous students' work or any other sources (including the internet), whether copyrighted or not. We understand that we will be subjected to disciplinary actions should it be found that the work we submit here does not comply with the said policies.

December 21, 2022

1.1 REFERENCED DOCUMENTS

- [1] E. Ecsedi, H. Silaghi, E. Mihok, and V. Spoială, “The development of an autonomous maze robot,” 2019. [Online]. Available: <https://ieeexplore-ieee-org.uplib.idm.oclc.org/document/8795166>
- [2] S. Geetapriya, N. R. Pillai, C. K. Aswin, and M. Menon, “Graph-based algorithm for mobile robot navigation in a known environment,” 2019. [Online]. Available: <https://ieeexplore-ieee-org.uplib.idm.oclc.org/document/8862775>
- [3] M. Tartagni, *Electronic Sensor Design Principles*, 2021.
- [4] “Flutter documentation,” 2022. [Online]. Available: <https://docs.flutter.dev/>
- [5] imjeffparedes, “Add wifi to arduino uno.” [Online]. Available: <https://www.instructables.com/Add-WiFi-to-Arduino-UNO/>
- [6] “Liquidcrystal.” [Online]. Available: <https://www.arduino.cc/reference/en/libraries/liquidcrystal/>
- [7] “Liquidcrystal i2c.” [Online]. Available: <https://www.arduino.cc/reference/en/libraries/liquidcrystal-i2c/>
- [8] F. Cleary, “Capacitive touch sensor design guide.” [Online]. Available: <https://ww1.microchip.com/downloads/aemDocuments/documents/TXFG/ApplicationNotes/ApplicationNotes/Capacitive-Touch-Sensor-Design-Guide-DS00002934-B.pdf>
- [9] M. Elbanhawi and M. Simic, “Sampling-based robot motion planning: A review.” [Online]. Available: <https://ieeexplore-ieee-org.uplib.idm.oclc.org/document/6722915>
- [10] B. Ma, Y. Rui, C. Wang, C. Peng, H. Zhu, and X. Li, “Research on leg transmission mechanism of quadruped robot.” [Online]. Available: <https://ieeexplore-ieee-org.uplib.idm.oclc.org/document/9620134>
- [11] G.-H. Lin, C.-H. Chang, M.-C. Chung, and Y.-C. Fan, “Self-driving deep learning system based on depth image based rendering and lidar point cloud.” [Online]. Available: <https://ieeexplore-ieee-org.uplib.idm.oclc.org/document/9258010>

2 STATE AND NAVIGATION CONTROL SUBSYSTEM

2.1 State and Navigation Control (SNC)

Reinhardt von Brandis-Martini, 20461632

2.2 SNC NEEDS ANALYSIS

Briefly, the complete system requires some way to operate the robot (a human-machine-interface) and navigate a maze. It is these two issues that the SNC is aimed at addressing. There are a number of high-level tasks that the operator will need the robot to complete. The first of these is the calibration of sensors in the system, and the second is the autonomous navigation of a maze. To complement these tasks, the inclusion of an idle mode, in which the robot is simply doing nothing, and an emergency stop function would prove useful during testing and operation of the robot.

These needs can be split into two categories, namely, control of the MARV's state, and the control of the MARV during navigation of a maze. In other words, the operation of the robot and the algorithm that controls the way in which it navigates.

2.2.1 State Control and Human-Machine-Interface

The state control component of the SNC addresses the need of the human to operate the robot. The team requires some way to control the robot's operation. More specifically, the team needs to be able to address hardware and software bugs, calibrate the system, have it start maze navigation and end maze navigation. In summary, the need of this component is the need to have a way for a human to control the operation of the robot.

This means that the state control subsystem¹ requires ways to signal to the SS and MDPS that they must commence calibration, that navigation of the maze has begun, and that navigation of the maze has concluded. Additionally, the human interface not only serves as a method of operating the MARV, but also as a way for the operator to receive feedback on the state of the MARV's operation. Therefore, another critical need of the state control subsystem is a diagnostics display.

2.2.2 Navigation Control

The navigation control component of the SNC addresses the main functionality of the MARV itself, that is the navigation of a maze. The robot requires some way to find the path it needs to take in order to reach the end of the maze, and it is this problem that the navigation control subsystem of the SNC will address. This includes sending control commands to the MDPS to accomplish the required movement of the robot, as well as the gathering of sensor information from the SS, which is required in the navigation algorithm.

2.3 SNC CONCEPT EXPLORATION

2.3.1 Human-Machine-Interface

As stipulated in the A-Maze-Eng MARV practical guide, the human-machine-interface must consist of three components.

- A touch-activated sensor for the purpose of transitioning to the next system state.
- A clap/snap sensor for transitioning to the SOS state, or as it has been called above, an emergency stop.
- A critical diagnostics display.

Concepts for the above components are discussed separately.

2.3.1.1 Touch-Activated Sensor

In this context, a touch activated sensor is interpreted as any mode of touching. This includes all forms of capacitive sensing, thermal sensing as well as systems in which the activation signal is as a result of some independent form of touch, such as a wireless signal produced by a smartphone application in which the user presses a button on a touch screen.

¹Here, the SNC is considered as a system, rather than a subsystem. As such state control and navigation control are referred to as subsystems

2.3.1.2 Capacitive sensing methods [3]

The first form of capacitive sensing discussed in [3] is the capacitive fringing field, in which two metal plates are placed close to each other. The operator presses down on the assembly to decrease the distance between the plates, thereby increasing their capacitance. This variable capacitance can be placed in a transient RC circuit, and the variance in time constant can be used to detect a change in capacitance.

The second, more commonly used method, would be to simply place a conductive metal in parallel with a capacitor. The capacitance of the human body is added to the overall capacitance and the same method as above can be used to detect this change.

Both of these methods can be used to detect a touch. The detection can then be used to create a signal which will cause a transition to the next state in the MARV.

2.3.1.3 Other touch-activated methods

The main consideration here would be an app in tandem with a wifi module. Apps for all mobile operating systems can be created with Flutter [4]. I am familiar with this app development framework, therefore it would be the most logical choice. A microcontroller with a wifi module [5] can then be used as a way to signal that the user has touched a button on the app's interface to the microcontroller.

2.3.1.4 Critical diagnostics display

There are two feasible methods of displaying critical diagnostics. Firstly, An LCD, using either parallel communication with a microcontroller, or the I2C protocol. Secondly, displaying the information on the same app discussed above. Libraries for the Arduino IDE that allow control of an LCD in both ways exist, see [6] and [7]. This would make the integration of an LCD simple. The use of a WiFi module, requires a larger time investment, since a mobile app would have to be developed. Additionally, WiFi modules and microcontrollers with builtin WiFi are more costly.

2.3.2 Navigation Control

A number of studies done on autonomous maze navigation algorithms will now be explored.

2.3.2.1 Development of an Autonomous Maze Robot [1]

The aim of this robot was to navigate a maze similar to that of the MARV. However, the maze that it was designed for has raised walls. This means that instead of detecting colours on a mat, the robot detects distance from a wall and tries to find openings in the wall to find its way out of the maze. The robot has six proximity sensors that allow it to detect walls in front of and adjacent to it.

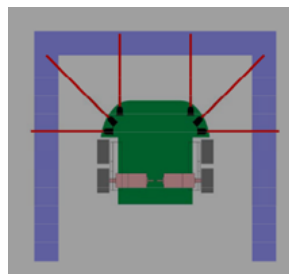


Figure 1: Placement of proximity sensors in [1]

In contrast to the proximity sensors, the MARV will use a colour sensor. However, the problem that the SNC has to address is the algorithm that navigates the robot through the maze. In this implementation, Eveline Ecsedi et al. used the following algorithm:

- if there is no wall on the right then go straight 40 mm, turn right and go straight 15 mm
- else if there is no wall in the front then go straight and stay centred on the lane
- else if there is no wall on the left then go straight 40mm, turn left and go straight 15 mm, else turn around 180

This robot considers dead ends as well, which is a scenario that should not occur in the MARV's case.

2.3.2.2 Graph-Based Algorithm For Mobile Robot Navigation In A Known Environment [2]

S Geetapriya et al. constructs a graph of the environment over the course of navigating a maze, with nodes in the graph representing regular square sections of the maze and edges being the paths between the nodes. For the MARV's SNC, nodes are analogous to the sections of the maze enclosed by lines, and the edges to the green crossings. If this method were used, the MARV would learn where the green lines are as it navigates the maze, and use that information to find its way out of the maze.

2.4 SNC CONCEPT DEFINITION: PLANNING

2.4.1 State Control and Human-Machine-Interface

The use of a capacitive sensor would pose the least risk, as I have implemented one before. This implies the use of an LCD, as a mobile application is not required for the implementation of a capacitive sensor. In addition, implementing an app would require much more time than using an LCD, and the libraries for controlling an LCD provided by the Arduino IDE are simple to use, in contrast to the Flutter framework [4]. This means that by using a capacitive sensor and an LCD, the risk involved in the implementation of the human-machine-interface is minimised.

Therefore, the following initial requirements for the state control component of the SNC will be used.

- A capacitive sensor with 90% accuracy.
- An LCD large enough to display all required system diagnostics.

These requirements apply and can be translated to the *human control Interface* and *information display* functions of the SNC.

The software functions which are parts of the state control component, will complement the human-machine-interface, and are as follows: *signal commencement of calibration*, *signal conclusion of calibration*, *signal end of maze navigation*, and *signal transition to idle mode*. They will allow the system to transition between its states by signalling when these transitions should occur, and will correspond to commands received by the human-machine-interface.

2.4.2 Navigation Control

The algorithm used in [1] is what one could call a trial and error approach to the navigation of a maze. The potential problem with this is that mazes with edge cases could cause the navigation algorithm to enter an infinite loop, in which case, the robot would simply arrive at the same dead end repeatedly without realising it.

In [2], the construction of a graph allows for the use of various AI algorithms to solve the maze. The literature survey done in [2] highlights some of these algorithms, with the simplest being a depth-first-search of the graph. This would allow the MARV to compute a solution to any maze.

The second implementation is most certainly more robust. However, an algorithm of this caliber is not necessary to satisfy the given specifications. This is because mazes that the MARV is required to

navigate will not include dead-ends. When encountering a black line (or in the case of [1], a wall), the MARV will always find the correct path either on its right or its left.

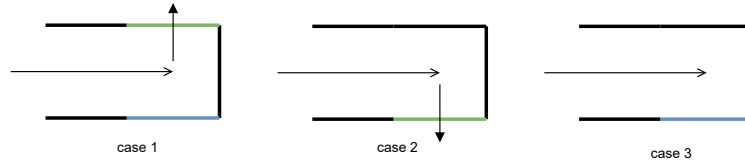


Figure 2: Generalised cases for a maze

This means that case 3 in Figure 2 will never occur. Therefore, it stands to reason that a trial and error method of navigating a maze satisfies the specifications. Furthermore, the specifications provided already detail an algorithm that can complete the task.

- If a blue or black line is encountered, reverse and rotate 90° clockwise
- If a blue or black line is encountered thereafter, reverse and rotate 180° clockwise

The first 90° rotation accomplishes a right turn, and in the case that the MARV actually needed to turn left, the second 180° rotation accomplishes a left turn. In this way, both possible cases of encountering a blue or black line are addressed. The above algorithm is however, a simplified version of the algorithm in the specifications, and does not consider errors in the angle of incidence with a line that may occur due to inaccuracies in the MDPS's operation. These intricacies are to be considered later in the systems engineering process.

After considering all the above information, the following initial operational requirements for the navigation control subsystem were defined.

- When a blue or black line is encountered, send the appropriate commands to the MPDS to accomplish a clockwise rotation of 90° .
- When a blue or black line is encountered directly after previously encountering a blue or black line, send the appropriate commands to the MPDS to accomplish a clockwise rotation of 180° .
- When a green line, or no line is encountered, send the appropriate commands to the MPDS to drive forward.

These three requirements can be translated to the function of the subsystem, namely *computation of navigation control outputs*.

2.4.3 Additional External Interactions

The communication mode to be used as stipulated in the A-Maze-Eng MARV practical guide is the UART protocol. This translates to a *digital communications interface* function, which will handle input to the SNC, as well as output to the MDPS and SS. Another consideration is that one of the functions of the MDPS is to provide the other subsystems with power. This power will need to be distributed to the various components of the SNC. As such, a *distribute electricity* function will be required.

Finally, after exploring various concepts for each component, and settling on the ones that make most sense, the functions of the SNC and their interactions can be defined, as in the following functional block diagram.

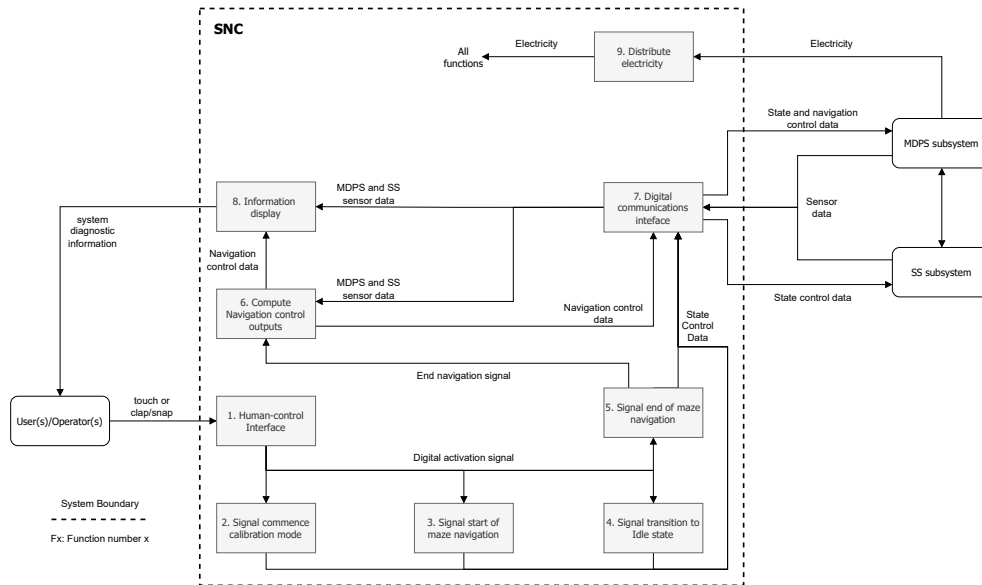


Figure 3: Functional Block Diagram for the SNC

Architectural components were derived from the above functions, and the following architectural block diagram was produced.

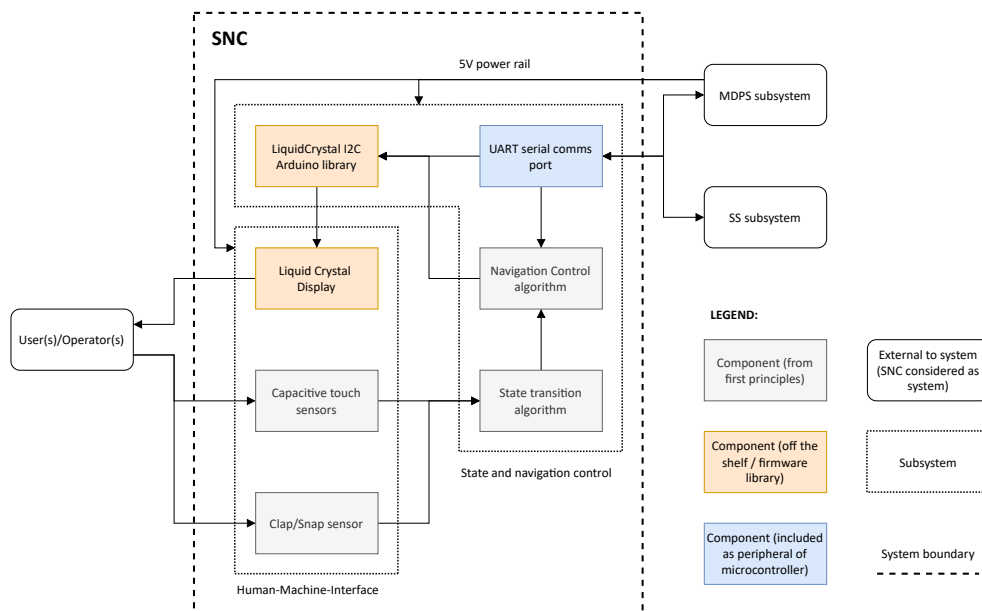


Figure 4: Architectural block diagram for the SNC

2.5 SNC ENGINEERING DESIGN

2.5.1 Tools and Methods

2.5.1.1 Engineering tools

The tools used during the design and development of the SNC can be divided into those used to design hardware components of the subsystem and those used to design firmware/software components of the subsystem.

Tools used for Hardware Components

The tools used for development of hardware components include LTSpice and Arduino libraries. LTSpice was used for confirming initial designs, and Arduino libraries for testing prototypes. The reason for using Arduino libraries was to test hardware more rapidly. The libraries allowed the hardware to be tested with fewer lines of software, and so, testing was done more frequently.

Tools used for Development of Firmware Components

Tools used for the development of software components include the use of a higher-level programming language to prototype functionality. The language used was Rust. A custom serial terminal was also created in Rust for early testing of serial communications. The Arduino Serial Monitor and Termite were also used for this purpose. The reason that the language Rust was used, is because it has first class support for Serial communications via the computer's USB, and it produces binaries that run up to 200 times faster than Python. This combination allowed for the rapid development of the aforementioned testing software, as well as more rapid feedback.

2.5.1.2 Engineering methods

Each component in Figure 4 was assigned functions from Figure 3, such that design of each function was done in the context of the integrated subsystem. A system's engineering approach was then applied to each component.

The method used was to revisit functional analysis, this time paying more attention to interfaces between components, as well as their affects on the interfaces with the other two subsystems. Thereafter, each component was designed with the resulting requirements in mind, using flow diagrams in the case of firmware components and circuit diagrams for hardware components. Simulations were then constructed from the designs to confirm that requirements would be met with said designs. Finally, prototypes were created and testing was done by comparing their real-world performance against simulations. Results were then discussed with the rest of the team. This was done in preparation for the integration of the subsystems, as the nature of the information received by the SNC would occasionally change, and interfaces therefore required adjustment. In this case, another prototype was developed for a second round of testing in order to identify the affect these adjustments, if any, had on the functional performance of the SNC.

Due to the nature of the SNC, most of the engineering development of the subsystem was done for firmware components. As, discussed in 2.5.1.1, the logic for the navigational control was implemented in Rust. A large list of navigation control inputs and outputs was then created (available at NAVCON Simulation). This allowed for an agile approach to software development to be used, in which planning, design, coding and testing were done repeatedly until all requirements and specifications were met. This was most utilised during integration, as testing produced drastic changes in requirements, mainly due to external interactions that were only discovered during integration testing. An agile approach allowed quick changes to the SNC's logic, as it emphasises modularity of code.

2.5.2 Selected design details

The following sub-components will be discussed:

- Navigation control algorithm
- State transition algorithm
- Capacitive touch sensor
- Clap/snap sensor
- Critical diagnostics display

2.5.2.1 Statements of requirements

1. Navigation control algorithm

- When a blue or black line is encountered, send the appropriate commands to the MPDS to accomplish a clockwise rotation of 90°.
- When a blue or black line is encountered directly after previously encountering a blue or black line, send the appropriate commands to the MPDS to accomplish a clockwise rotation of 180°.
- When a green line, or no line is encountered, send the appropriate commands to the MPDS to drive forward.

2. State transition algorithm

- transition each subsystem to next state upon detection of a touch.
- transition each subsystem to the SOS state upon detection of a clap/snap.

3. Capacitive touch sensor

produce an increase in period of an RC oscillator when touched

4. Clap/snap sensor

Identify a clap/snap from a distance of at least 5 m, by producing a higher voltage when someone claps or snaps

5. Critical diagnostics display

display the received colour sensor, angle of incidence, distance, and speed readings

2.5.2.2 Development

1. Navigational control algorithm (NAVCON)

The first step taken in development of the NAVCON, was functional analysis. To do this, the inputs and outputs of the component were defined. Inputs include distance, and rotation from the MDPS, as well as the colours sensed and angle of incidence from the SS. The output of the system is a navigation action, namely forward, reverse, stop, right rotation and left rotation. In the case of a rotation, the angle of the rotation also has to be sent to the MDPS.

With this in mind, the following transformative functions are performed by the NAVCON: produce a correction angle from the angle of incidence, and translate the colour sensed to the correct navigation action. This information was then used to produce a functional block diagram for the NAVCON, shown below.

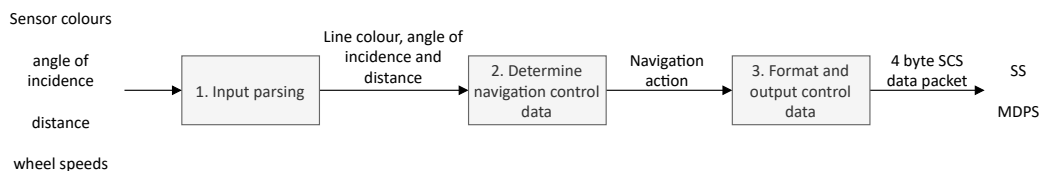


Figure 5: Functional block diagram for the NAVCON

The bulk of the NAVCON's work will be done in function 2 (Figure 5). Using the requirements, a flow diagram was produced to model the logic that had to be performed by the NAVCON is shown below, with output state shown as rectangular white blocks, and data in red text.

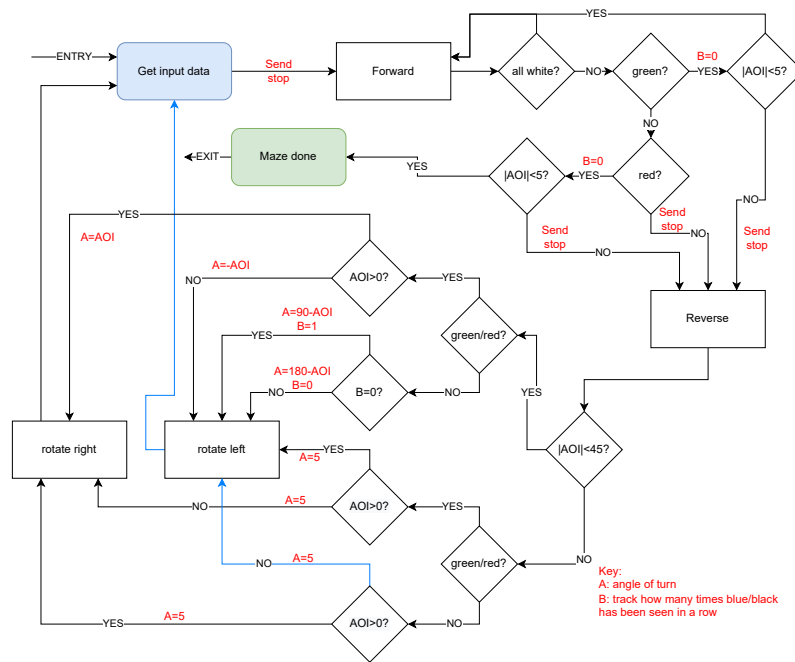


Figure 6: Flow diagram for the NAVCON

It was decided that implementing a state machine that performed the logic in Figure 6, would be the best solution. This is due to the fact that the MARV can only ever be in one of a number of states during maze navigation, namely, forward, reverse, left rotation, right rotation, and stop. These states correspond with the navigation instructions, and each state need only be exited if certain conditions are met. For example, in the flow diagram above, if the MARV is going forward, it can continue going forward until it encounters a green/red line at an angle of incidence greater than 5° , or a blue/black line is encountered. The following state machine was produced.

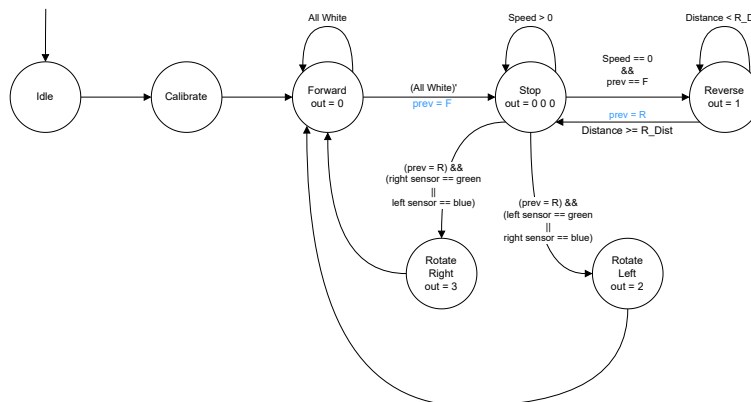


Figure 7: NAVCON state machine

Encountering a line can be categorised by the colour of the the line encountered and the angle of incidence at which it is encountered. Here, it is important to recognise that encountering a green line is handled in the same way as encountering a red line, and the same it true for blue and black. Hereafter, an encounter with a green or red line, is referred to as a green encounter and for a blue or black line, a blue encounter.

A green encounter can be further broken down by angle of incidence with the line. If the angle is less than 5 degrees, the MARV must continue going forward, for an angle greater than 5 and

less than 45 degrees, the MARV must reverse and rotate toward the line, and for an angle greater than 45 degrees, a small rotation of 5 degrees toward the line must be performed.

A blue encounter necessitates a right rotation such that when the MARV drives forward again, it is parallel with the encountered line, and if another blue encounter occurs thereafter, the MARV must turn around, or rotate 180 degrees. A tree which shows how an encounter with a line is broken down is shown below.

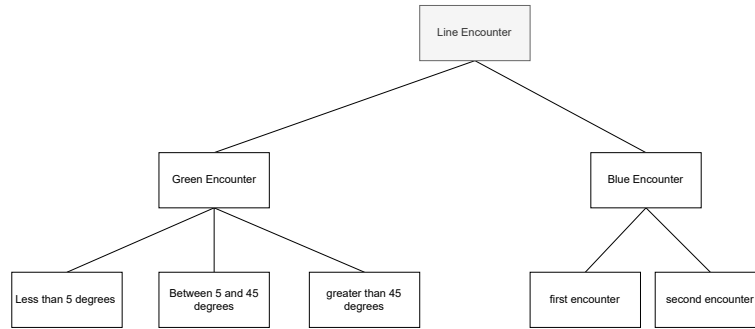


Figure 8: Cases for a line encounter

Blue and green encounters are split into two different functions where the NAVCON state is changed based on angle of incidence. The required logic was, again, modelled using flow diagrams.

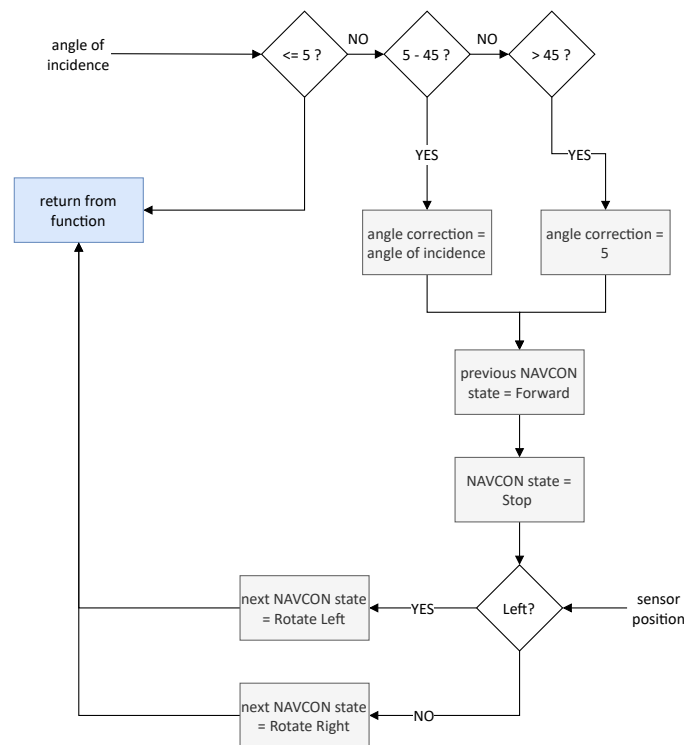


Figure 9: Flow diagram for a green encounter

The logic for a blue encounter is similar to that of a green encounter, however the correction angle has to be adjusted to accomplish a 90 or 180° rotation.

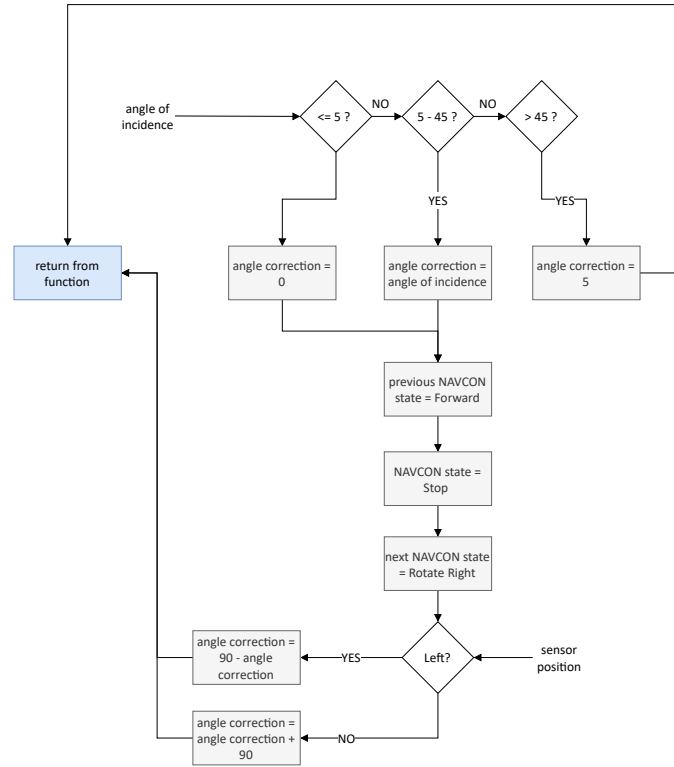


Figure 10: Flow diagram for a blue encounter

2. State transition algorithm

Transitioning to the next state occurs every time the capacitive touch sensor detects a touch. The states will always be activated in the same order, and the system can cycle through them. The only exception being the SOS state, which can only be transitioned to during Maze state with a clap/snap. A flow diagram was created to model the code.

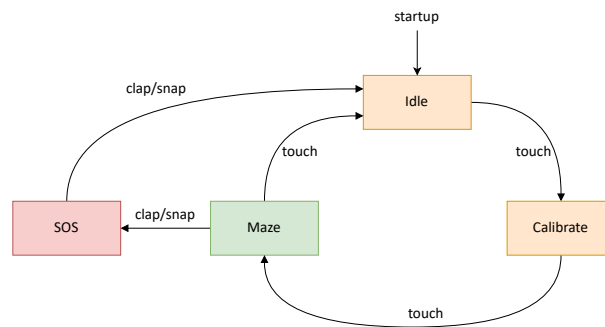


Figure 11: State flow for the MARV

3. Capacitive touch sensor

Development of a highly sensitive capacitive touch sensor means that the change in capacitance caused by a touch must create a change in the RC time constant that is as large as possible. The first consideration is the length of time required to charge the capacitor through the resistor. The Arduino nano uses a 16 MHz clock, which means each instruction cycle is $0.25\mu s$. If one is willing to delay for 200 cycles, then the capacitor will have $50\mu s$ to charge. For some safety, calculations were done as if the RC time constant with no touch was $25\mu s$.

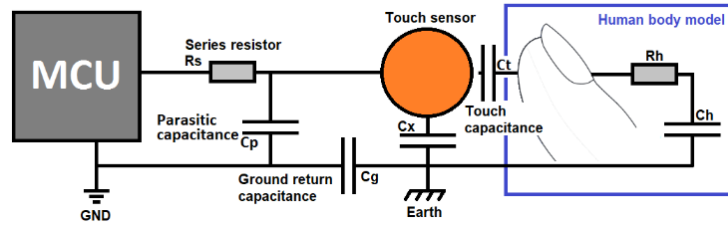


Figure 12: Capacitive touch model from [8]

For the worst case scenario, $C_H = 100\text{pF}$ and $C_g = 1\text{pF}$ [8]. It was decided that if the voltage across the RC circuit was measured after $25\mu\text{s}$, then there should be a 1 V difference between no touch and being touched. The following calculations were done to achieve this.

$$5e^{-\frac{\tau_1}{RC_n}} = 5e^{-\frac{\tau_1}{RC_t}} - 1$$

$$\text{where } C_t = C_n || (C_H + C_g) = C_n + 0.99\text{pF}$$

4. Clap/snap sensor

A clap/snap is a sound wave. The most logical component to use for detecting sound is an electronic microphone. A clap/snap produces sound within a certain range of frequencies, approximately between 2 kHz and 4 kHz. Off-the-shelf microphones produces very low amplitude sinusoidal waves. The signal will need amplification to be used in the Arduino's range of 0 - 5 V. The following functional block diagram was produced after functional analysis of the clap/snap sensor.



Figure 13: Functional block diagram for the SNC's clap/snap sensor

5. Critical diagnostics display

The critical diagnostics display simply has one function, namely display of critical diagnostics. critical diagnostics include the following.

- MARV state
- Sensor colours
- Angle of incidence
- Distance since last stop
- Wheel speeds in mm/s

To display all this information, an LCD of an adequate size will be required. Each state has a maximum of four letters, five colours require five characters, distance should not exceed 200 mm, angle of incidence will not exceed 90° , and wheel speeds will not exceed 150 mm / s. Taking all this into account, a minimum of 20 characters and with spaces in between information, 25 characters. For this reason, a 20 x 4 LCD was used for a total of 80 characters. This would allow information to be labelled as well, making the display more readable and therefore more effective.

2.5.2.3 Simulations

1. Software emulation

The NAVCON's logic was emulated using Rust. The language is high-level enough to allow for a quick implementation, and similar enough to C++ such that logic could simply be recoded in C++ with little

difference between the two.

Code was also used to generate all possible test cases. These were used as input to the emulation, and output saved to a text file for reference during development and testing. A code snippet showing how input data was generated is shown below.

```
1 let mut aois = Vec::new();
2
3 for i in -90..90 {
4     aois.push(i as i8);
5 }
6
7 let mut colours = Vec::new();
8
9 for col0 in Colours::iter() {
10     for col1 in Colours::iter() {
11         for col2 in Colours::iter() {
12             for col3 in Colours::iter() {
13                 for col4 in Colours::iter() {
14                     if (col0 == col1 || col1 == Colours::White || col0 == Colours::
15                         White) && (col2 == Colours::White && col3 == Colours::White && col4 == Colours::
16                         White) {
17                         let new_array = [col0, col1, col2, col3, col4];
18                         colours.push(new_array);
19                     }
20                 }
21             }
22         }
23     }
24 }
25
26 for angle in aois {
27     for col in &colours {
28         data.push(TestData::new(*col, angle));
29     }
30 }
```

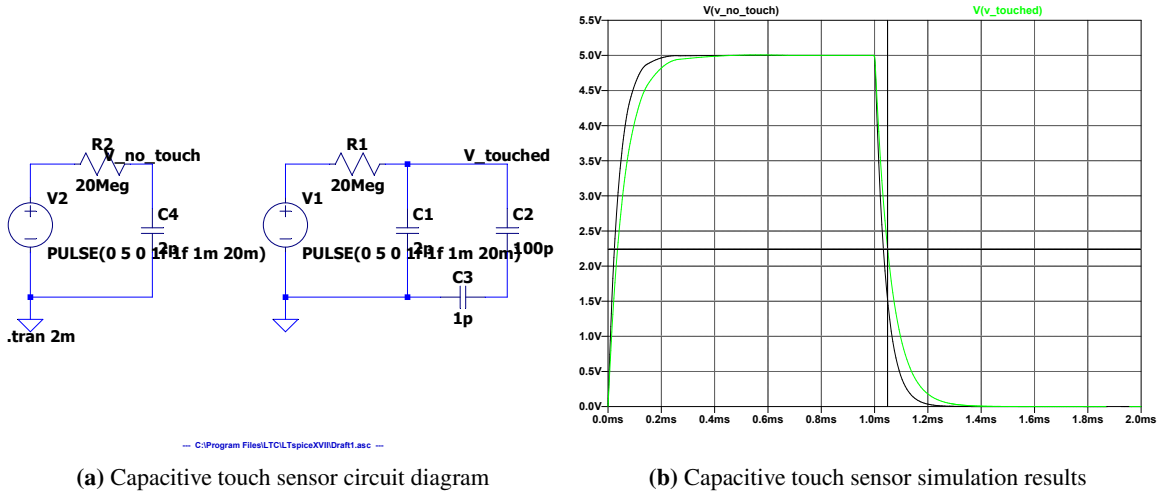
Samples of the generated test cases and the results obtained are shown below.

```
1 NAVCON Simulation/test bench:
2 Angle Of Incidence: 14 degrees
3 Colour Sensor: [Green, Green, White, White, White]
4 Distance since last stop: 0 mm
5 left wheel velocity: 0 mm/s
6 Right wheel velocity: 0 mm/s
7 Time since last stop: 0.0012557 s
8
9 Waiting for MARV to stop...
10
11 MARV reversing...
12
13 Distance since last stop: 0 mm
14 left wheel velocity: -10000 mm/s
15 Right wheel velocity: -10000 mm/s
16 Time since last stop: 0.0000013 s
17
18 Waiting for MARV to stop...
19
20 MARV rotating left!, 14
21 Distance since last stop: -14.661 mm
22 left wheel velocity: -10000 mm/s
23 Right wheel velocity: 10000 mm/s
24 Time since last stop: 0.0014662 s
```

All code and results are available at NAVCON Simulation.

2. Hardware

Hardware was simulated using LTSpice. The circuits were constructed using the software and simulated outputs were used as reference during development and testing. The first simulation done was for the capacitive touch sensor, with the circuit diagram shown below.



The same was done for the clap/snap sensor with the results shown below.

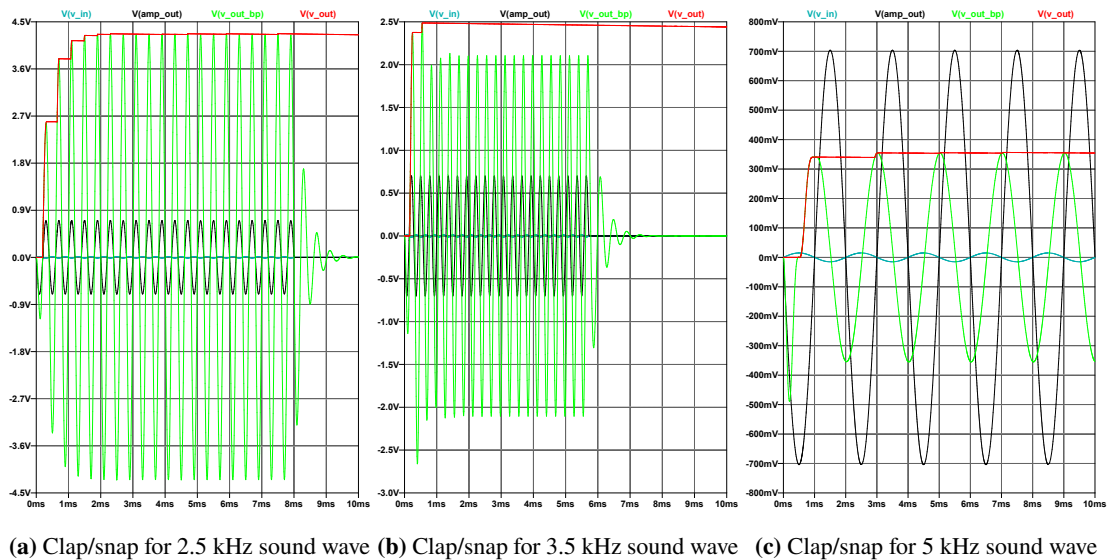


Figure 15: Clap/snap sensor simulation results

2.5.2.4 Approach to coding and testing

1. Coding

In order to keep track of all the data, a structure was used. This allowed all the input/output data, states and flags to be in the same place. Any new information that was necessitated by new requirements or novel system interactions could simply be placed in the structure. The following final structure was used.

```
1 struct NAVCON {
```

```

2     enum NavStates state; // current state
3     enum NavStates prev; // previous state
4     enum NavStates next; // next state (for stop)
5     bool outside_sensor; // has an outer sensor seen the line
6     enum SensorPosition first_sensor_side; // which side of the sensor array
    saw the line first
7     enum Colours colour; // the colour encountered
8     enum Colours prev_colour; // the previously encountered colour, for second
    blue/black
9     uint16_t reference_distance; // starting distance for > 45
10    uint16_t AOI_correction; // correction angle for rotations
11 };
12

```

Retrieval of input data (the blue block in Figure 6) occurs every time the NAVCON algorithm is executed. The type of data to be retrieved is determined by the first byte (control byte) of the data, and data can be either 8 bits or 16 bits in length. Below is a code snippet showing how data is retrieved in both cases.

```

1 // wait for packet to have expected control byte
2 while (packet_in.bytes[controlByte] != DESIRED_CONTROL_BYTE) {
3     packet_in = receive_packet();
4 }
5
6 // for saving 8 bit information
7 8bitData = (uint8_t)(in_packet.bytes[dat1]);
8
9 // for saving 16 bit information
10 16bitData = (uint16_t)((in_packet.bytes[dat1] << 8) + (in_packet.bytes[dat0]));

```

There is one exceptional case, the sensor colours, where each colour is encoded as a 3 bit value, all five of which are in a 16 bit string. To determine each colour, bit masking and shifting is used.

```

1 sensorSystem->sensor[0] = (enum Colours)((colours & 0b0111000000000000) >> 12);
2 sensorSystem->sensor[1] = (enum Colours)((colours & 0b0000111000000000) >> 9);
3 sensorSystem->sensor[2] = (enum Colours)((colours & 0b0000000111000000) >> 6);
4 sensorSystem->sensor[4] = (enum Colours)((colours & 0b0000000000000011) >> 0);
5 sensorSystem->sensor[3] = (enum Colours)((colours & 0b00000000000111000) >> 3);

```

The next consideration was the outputting the navigational control data in the correct format, based on the NAVCON's state. This sub-component of the SNC is shown below.

```

1 // will be outputting NAVCON data
2 packet_out.bytes[controlByte] = 147;
3
4 switch (navcon->state) {
5     case Forward: {
6         packet_out.bytes[dec] = 0;
7         packet_out.bytes[dat0] = 10;
8         packet_out.bytes[dat1] = 10;
9
10        break;
11    }
12    case Reverse: {
13        packet_out.bytes[dec] = 1;
14        packet_out.bytes[dat0] = 10;
15        packet_out.bytes[dat1] = 10;
16
17        break;
18    }
19    case RotateLeft: {
20        packet_out.bytes[dec] = 2;
21        packet_out.bytes[dat1] = (uint8_t)((navcon->AOI_correction & 0xFF00) >> 8);
22        packet_out.bytes[dat0] = (uint8_t)(navcon->AOI_correction & 0x00FF);
23    }

```



```

24     break;
25 }
26 case RotateRight: {
27     packet_out.bytes[dec] = 3;
28     packet_out.bytes[dat1] = (uint8_t)((navcon->AOI_correction & 0xFF00) >> 8);
29     packet_out.bytes[dat0] = (uint8_t)(navcon->AOI_correction & 0x00FF);
30
31     break;
32 }
33 case Stop: {
34     /* no need to change the packet. i.e. MARV must stop */
35 }
36 }
37

```

With both input and output functions out of the way, only the transformative functions remain. In order to write code that was as modular as possible, it was necessary to break every possible scenario into its fundamental parts. The first, and most obvious scenarios are not having encountered a line, and having encountered a line. The former occurs when all the sensors see white, and the latter when any sensor sees a colour that is not white. If all sensors see white, then the MARV can simply continue whatever operation it is carrying out. In any other case, the nature of the line encounter has to be considered. Below are the final two functions for the two different types of encounters, as discussed in section 2.5.2.2. The algorithms were created with close reference to figures 9 and 10.

```

1 void green_encounter(struct NAVCON* navcon, uint8_t incidence, enum SensorPosition
   position) {
2     // reset prev colour for second blue
3     navcon->prev_colour = Green;
4
5     if (incidence <= 5) {
6         // if the incidence is <= 5, continue forward
7         navcon->state = Forward;
8         return;
9     }
10    else if (incidence < 45) {
11        navcon->AOI_correction = (uint16_t)incidence;
12    }
13    else {
14        navcon->AOI_correction = 5;
15    }
16
17    navcon->prev = Forward; // navcon prev state now forward
18    navcon->state = Stop; // MARV should stop
19
20    // logic for which way to rotate
21    switch (position) {
22        case Left:
23            navcon->next = RotateLeft;
24            break;
25        case Right:
26            navcon->next = RotateRight;
27            break;
28    }
29 }
30
31 //=====
32
33 void blue_encounter(struct NAVCON* navcon, uint8_t incidence, enum SensorPosition
   position) {
34
35     // set the correction to the appropriate value for the input AOI
36     if (incidence <= 5) {

```

```

37     navcon->AOI_correction = 0;
38 }
39 else if (incidence < 45) {
40     navcon->AOI_correction = (uint16_t)incidence;
41 }
42
43 navcon->prev = Forward; // navcon prev state now forward
44 navcon->state = Stop; // MARV should stop
45 navcon->next = RotateRight; // always right for blue encounter
46
47 switch (position) {
48     case Left:
49         navcon->AOI_correction = 90 - navcon->AOI_correction;
50         break;
51     case Right:
52         navcon->AOI_correction += 90;
53         break;
54 }
55
56 if (navcon->prev_colour == Blue) {
57     // add an 90 if previous line was blue for 180
58     navcon->AOI_correction += 90;
59 }
60
61 if (incidence >= 45) {
62     navcon->AOI_correction = 5;
63 }
64
65 // set the prev colour for blue for second encounter
66 navcon->prev_colour = Blue;
67 }
68

```

These two functions address all possible cases of encountering a line. After testing them and ensuring they fulfilled all requirements, the state machine logic was next in the development cycle. The state machine was modelled using Figure 8 as a guide, as well as the flow in Figure 6. This state machine initially fulfilled all the requirements, however subsystem interactions that were only discovered during integration created the need for many additions. A watered down version of the state machine implemented for the Arduino is shown below.

```

1  void run_navcon(struct MDPS* motor_system, struct SS* sensor_system, struct
NAVCON* navcon) {
2      switch (navcon->state) {
3          case Forward:
4              if (navcon->outside_sensor == false && (sensor_system->sensor[0] != White
|| sensor_system->sensor[4] != White) && sensor_system->incidence == 0) {
5                  navcon->outside_sensor = true;
6                  navcon->reference_distance = motor_system->distance;
7                  if (sensor_system->sensor[0] != White) {
8                      navcon->colour = sensor_system->sensor[0];
9                      navcon->first_sensor_side = Left;
10                 }
11                 else {
12                     navcon->colour = sensor_system->sensor[4];
13                     navcon->first_sensor_side = Right;
14                 }
15             }
16
17             if (navcon->outside_sensor == true) {
18                 if ((motor_system->distance - navcon->reference_distance) > ISD) {
19                     greater_than_45(navcon, navcon->first_sensor_side);
20                     navcon->colour = White;
21                     navcon->outside_sensor = false;

```

```

22         }
23         else if (sensor_system->sensor[1] != White || sensor_system->sensor
[3] != White){
24             enum SensorPosition side;
25             if (sensor_system->sensor[1] != White) {
26                 navcon->colour = sensor_system->sensor[1];
27                 side = Left;
28             }
29             else if (sensor_system->sensor[3] != White) {
30                 navcon->colour = sensor_system->sensor[3];
31                 side = Right;
32             }
33             less_than_45(navcon, side, sensor_system->incidence);
34             navcon->colour = White;
35             navcon->outside_sensor = false;
36         }
37     }
38     break;
39
40     case Reverse:
41         // until MARV has reversed for 6cm, keep reversing....
42         if (motor_system->distance < 60) {
43             return;
44         }
45
46         navcon->prev = Reverse;
47         navcon->state = Stop;
48
49         break;
50     case RotateLeft:
51         // if the rotation is still in progress, then keep rotating
52         if (motor_system->rotation < navcon->AOI_correction) {
53             return;
54         }
55
56         navcon->colour = White;
57         navcon->state = Forward;
58
59         break;
60     case RotateRight:
61         // if the rotation is still in progress, then keep rotating
62         if (motor_system->rotation < navcon->AOI_correction) {
63             return;
64         }
65
66         navcon->colour = White;
67         navcon->state = Forward;
68
69         break;
70     case Stop:
71         if (navcon->prev == Forward) {
72             navcon->state = Reverse;
73             return;
74         }
75
76         navcon->state = navcon->next;
77
78         break;
79     case MazeDone:
80         navcon->AOI_correction = 360;
81         navcon->state = RotateRight;
82
83         break;

```

2. Testing

Testing was initially done using a custom serial terminal. The SNC was put into maze state, and data from the SS and MDPS was emulated to test the NAVCON's components. After verifying that all components of the NAVCON and state transition algorithm worked independently, the HUB, the prescribed testing software, was used to test requirements and specifications. A QTP was run, and once all bugs were addressed, then next QTP was tackled. The previous QTP was rerun after successful completion of the first, ensuring that no further bugs were created due to code changes/additions.

2.5.3 Constraints and trade-offs

2.5.3.1 Design Constraints

The design constraints that were present include

- frequency of sensor measurements received from the SS and MDPS
- accuracy of sensor measurements received from the SS and MDPS
- amount of memory available on the Arduino nano
- amount of program memory available on the Arduino nano
- the digital communications protocol prescribed
- the number of I/O pins available on the Arduino nano
- physical size of the subsystem
- programming language used to code the Arduino nano
- commercially available component values
- voltage of the available supply rails
- time in which the project had to be completed
- physical size of the LCD required

Most design constraints did not have a noticeable impact on the final implementation. Constraints of note include the number of I/O pins available, physical size of the subsystem, time in which the project had to be completed, physical size of the LCD, and the accuracy of sensor measurements.

2.5.3.2 Trade-offs

The constraint which had the most impact on the final implementation was the accuracy of the sensor measurements received by the SS and MDPS. The SS's designer opted to use static lighting for the colour sensor. This meant that the sensor used a single voltage to determine the colour under the sensor. The consequence of this was that when a sensor is over white, it outputs a high voltage, and as it moves over a black line, for example, its output voltage slowly decreases as it moves over the line, until finally outputting the colour black. It was decided an algorithm would be developed to accommodate this, sacrificing the speed with which the MARV can navigate the maze. The SNC keeps track of all colours received until white is detected again, using the lowest voltage colour for navigation. This means that the MARV has to move slowly over a line before the NAVCON can determine which navigation action should be performed. However, the main objective of the MARV is to navigate a maze. Therefore, ignoring this and simply using the colour sensed first would mean the MARV would have little chance of actually completing the correct navigation action, resulting in a lost robot...

The time given to complete the project had an impact on the microcontroller used in the implementation. The complexity of using an LCD required the use of an Arduino compatible MCU, since the Arduino IDE has excellent support for controlling the output of an LCD. Another consideration was the PIC18F45K22, because it was familiar. However, Microchip does not provide libraries for LCD control, and therefore a set of functions to complete the task would have had to be developed. This task alone would have taken the entire duration of the project, considering the fact that the I2C protocol and LCD OpCodes were unfamiliar.

The physical size of the subsystem, LCD and number of I/O pins available can be grouped together. All three had an effect on the microcontroller chosen, and method used for control of the LCD. The physical size of the subsystem meant that an Arduino nano, or similar sized microcontroller was required. The Arduino nano was chosen. It has 22 I/O pins. Controlling an LCD in parallel mode requires about 10 pins, so the I2C mode was selected for the implementation. It uses 4 pins, however since it uses a serial format, there is more delay in updating the screen.

2.6 SNC QUALIFICATION TESTS RESULTS

Table 1: A summary of all the SNC Qualification Test Procedure (QTP)'s expected and measured results.

SPECIFICATION	EXPECTED TEST RESULT	TEST RESULTS
Power Supply		
The subsystem may not draw more than 1A	With the LCD drawing 35mA, and the other circuits drawing 10mA from simulation \approx 45mA	40mA
state transition and critical diagnostic display		
adhere to the SCS	all data packets sent comply with the SCS	all data packets sent comply with the SCS
transition states correctly upon detecting a touch	state transitions occur when a touch is detected, according to figure 11	state transitions occur when a touch is detected, according to figure 11
Await end-of-calibration (EoC) signals from the SS and MDPS within the calibration state	maze state cannot be entered before the SS and MDPS are done with calibration	maze state cannot be entered before the SS and MDPS are done with calibration
Display all critical diagnostics	All critical diagnostics are displayed and updated when required	All critical diagnostics are displayed and updated when required
The MARV returns to the Idle state when the end-of-maze command is received from the SS	The MARV transitions to the Idle state when the end-of-maze command is received from the SS	The MARV transitions to the Idle state when the end-of-maze command is received from the SS
Clap and Snap detection		

Detect a clap of hands and snap of fingers with a minimum volume of 70 dB	A clap/snap is detected with a minimum volume of 70 dB	A clap/snap is detected with a minimum volume of 70 dB
Green encounter		
The MARV crosses a green/red line for $\theta_i \leq 5^\circ$	When θ_i received by the SS is less than 5, and the line encountered is green/red the NAVCON remains in the Forward state	the NAVCON remains in the Forward state for a green/red line at $\theta_i \leq 5^\circ$
The MARV attempts to correct θ_i when $5^\circ < \theta_i < 45^\circ$	The NAVCON sends the appropriate output to the MDPS to rotate the MARV toward the line with an angle of rotation = θ_i	The NAVCON sends the appropriate output to the MDPS such that the MARV rotates toward the line with an angle of rotation = θ_i
The MARV makes a small correction of 5° toward the line when $\theta_i \geq 45^\circ$	The NAVCON sends the appropriate output to the MDPS to rotate the MARV toward the line with an angle of rotation = 5°	The NAVCON sends the appropriate output to the MDPS such that the MARV rotates toward the line with an angle of rotation = 5°
Blue encounter		
The MARV rotates 90° when a blue/black line is detected with $\theta_i < 45^\circ$	The NAVCON sends the appropriate output to the MDPS to rotate the MARV clockwise with an angle of rotation = $90^\circ \pm \theta_i$	The NAVCON sends the appropriate output to the MDPS to rotate the MARV clockwise with an angle of rotation = $90^\circ \pm \theta_i$
The MARV steers away from a blue/black line with $\theta_i \geq 45^\circ$	The NAVCON sends the appropriate output to the MDPS to rotate the MARV away from the line with an angle of rotation = 5°	The NAVCON sends the appropriate output to the MDPS to rotate the MARV away from the line with an angle of rotation = 5°
The MARV rotates 180° when a blue/black line is detected with $\theta_i < 45^\circ$ for the second time in a row	The NAVCON sends the appropriate output to the MDPS to rotate the MARV clockwise with an angle of rotation = $180^\circ \pm \theta_i$	The NAVCON sends the appropriate output to the MDPS to rotate the MARV clockwise with an angle of rotation = $180^\circ \pm \theta_i$

2.7 SNC CONCLUSIONS AND RECOMMENDATIONS

Through the use of continuous iteration and re-evaluation of requirements, as well as frequent testing, a successful SNC that met all requirements was developed. The ability of the SNC to remember the past colours of lines encountered allowed it to address all cases almost perfectly. The use of clear requirements and accurate simulation allowed for the development of a touch and clap/snap sensors that were accurate and reliable. This, in combination with a robust state transition algorithm and stringent communication reception allowed the MARV to reliably change states in the intended fashion.

Although not technically deviating from the specifications, the systems shortfall was exposed when the MARV encountered two lines at the same time (a corner). This situation occurred because the MARV had a tendency to veer to one side. When this happened, the NAVCON responded to the line which

was seen by the inner sensor first. This would occasionally cause the MARV to become confused, and execute several 180° turns in a row, before eventually fixing its trajectory.

This same problem with the motor system meant that the MARV sometimes encountered a line that it was supposed to be travelling parallel to. In this case, the NAVCON would tell the MDPS to execute a 90° turn, when not required.

These two shortfalls motivate some of the recommendations for the subsystem. One way to fix them be to implement a centering algorithm, in which the MARV identifies that it has encountered a corner, and corrects itself, rather than treating it in the same way as encountering one line.

Another recommendation would be to increase the MARV's capability in terms of the type of mazes it is able to navigate. As it is now, the MARV would not be able to navigate mazes in which there are dead ends. The graph-based algorithm discussed in section 2.3.2.2, would allow the MARV to navigate any maze. In conclusion, the SNC met all specifications, and was able to navigate a maze in the majority of cases.

A Abbreviations

QTP Qualification Test Procedure	20
SNC State and Navigation Control	1