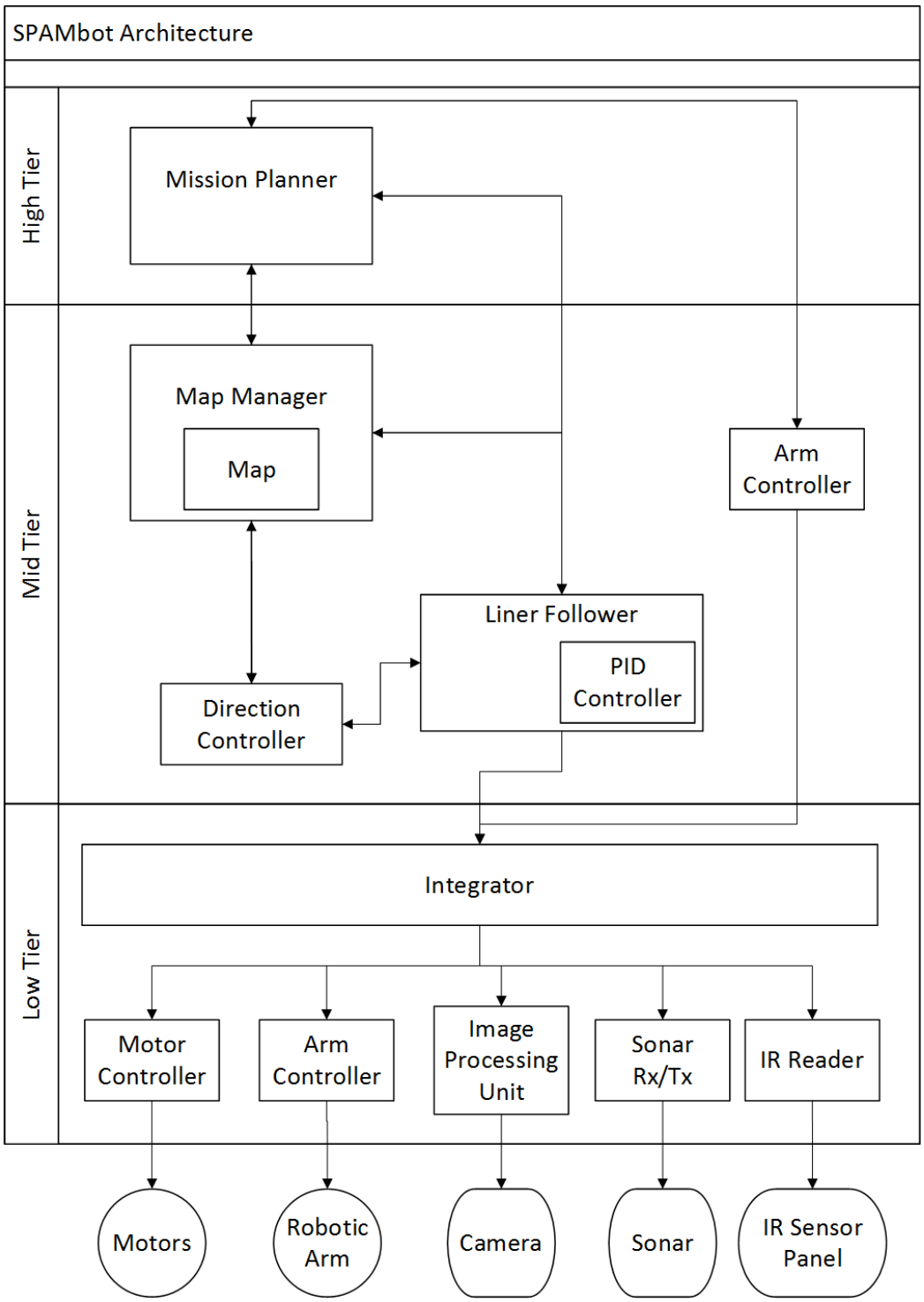
SPAMbot - An Arduino Robot Framework

Introduction

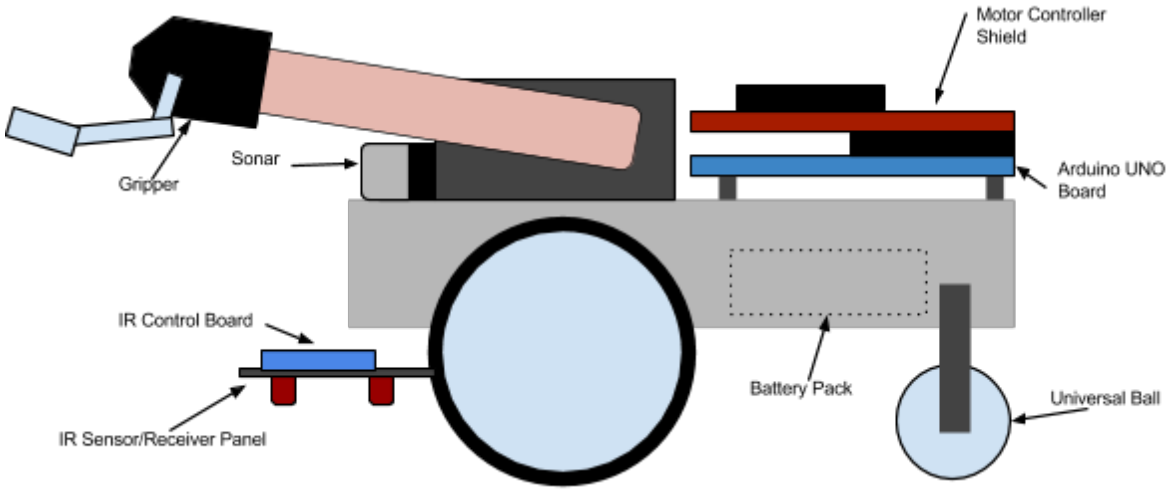
SPAMbot is a robotics framework which was undertaken to achieve higher level functions such as transporting payloads, on top of middle and low level tiers made to create maps, avoid obstacles, follow lines etc. The name *SPAMbot* is derived from the first letters of the developers' names; **S**habir, **P**raneeth, **A**nushanga and **M**ilan.

SPAMbot consist of several interconnected modules and layers to perform different levels of tasks.

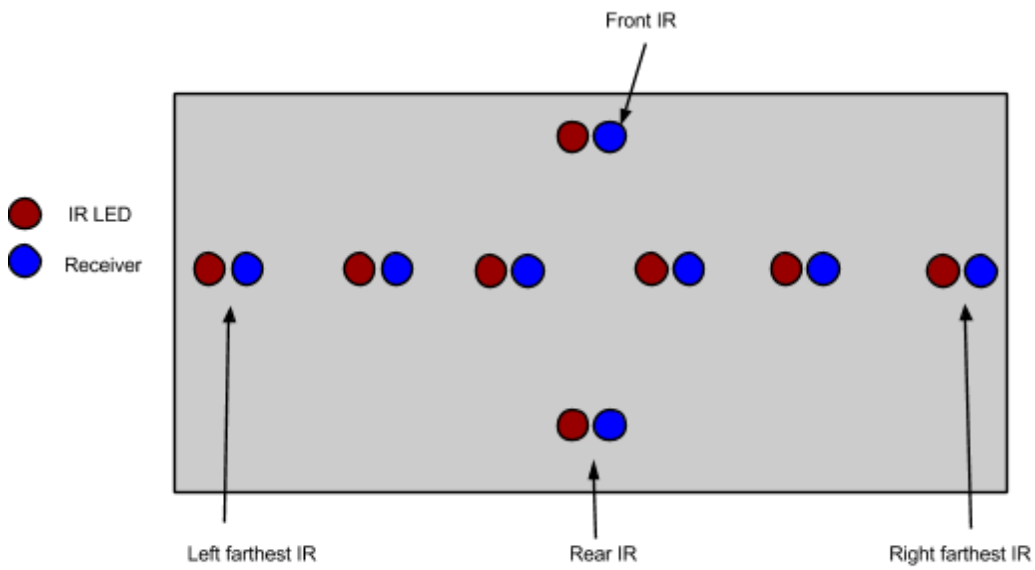


Hardware Structure

Side View



Sensor Panel Structure



Lower Tier

Motor Controller

The motor controller is in control of all the turning that the robot does. Turning a robot can be done in two ways: make it turn until a condition is met or turn it for a predefined amount of time. Both approaches are used in robot design but the latter poses some disadvantages in a practical scenario such if there was a bump in the surface or if the wheels slipped the robot would not have any idea of what is going on, it will turn until the given time has run out. The other approach mitigates such annoyances because it continuously monitors the environment. Therefore, we chose to implement the first mentioned approach. All functions starting with "*turn_*" in ***SPAMbot.ino*** are utility functions of this layer.

A challenge faced in the implementation of this code was, when the motors suddenly start to work the voltages of the IR sensors drop and this makes them register false information. A small time delay was added to prevent from our algorithms reading the false information and prematurely exiting the function without turning as needed.

Robotic Arm Controller

The Arm Controller of the lower tier is implemented to accommodate the need for lifting and moving payloads spread across the navigation map. We focus on the explanation of the hardware setup of the Robotic Arm Controller in this section. We further elaborate on the software implementation in the section dedicated for the Arm Controller in the *"Mid Tier"* of the architecture.

The arm and gripper of SPAMbot is built using the units of the ***AL5A Robotic Arm Kit*** produced by ***Lynxmotion Inc***. The default kit comprises of 5 servos. The sample explained in the kit is a fully independently operable Arm with 5 degrees of freedom. The shoulder, elbow, wrist and gripper movements were managed by 4 servos and the 5th contributed to produce complete rotations of the arm-unit. This Arm is controlled using a *"BotBoarduino"* circuit board. The *BotBoarduino* is an Arduino-Duemilanove compatible microcontroller made specifically for the Lynxmotion robots. It retains the normal Arduino shield connections as well. However, for the purpose of implementing the Arm Controller of SPAMbot, we have only made use of some hardware units of this kit and directly plugged them to our main Arduino-ATMega shield.

Our initial attempt to build a fully functional gripper control using 4 servos was unsuccessful. This was primarily due to the weight that the robot had to carry and the power-load required by 4 concurrently operating servos. This setup required an additional power source for the robot and also had unstable movements when the Arm controller was in operation-trying to pick/drop a payload. Hence the hardware design was reduced to only use 2 servos connected directly to our primary arduino. This setup reduced the servo-power consumption to a manageable level. Furthermore, the robot was more stable in carrying out the Arm Controller movements.

An "*HS-755HB"* large-scale servo is directly mounted to the body of the robot. A "U" bend connected to this servo extends the rotational movement in a direction parallel to the robot. The large servo is fixed such that: 90 degrees anti-clockwise rotation makes the arm fully extended forwards *(in the direction of the robot's movement)* and 90 degrees clockwise rotation aligns the arm bent backwards on the body of the robot. Moreover, an *"HS-645MG"* standard-size servo is fixed to the end of the "U-bend". A gripper kit directly mounted to this standard servo helps to achieve the gripper motion of the arm. However, the default gripper unit available in the Lynxmotion kit has a spread of only 3cm. Hence, this unit alone cannot be used to grip payloads of larger width. Given that the payloads of the given mission are 10cm wide, we had to devise a mechanism to increase the gripper spread. To solve this problem we built two similar-looking "L" shaped extenders with the units from "Lego-Mindstorms" kit. Each one of them were an approximate 5cm long. These units were directly mounted to the two ends of the existing gripper kit. Hence, we could achieve a combined spread of 13cm. When the gripper kit was fully closed the spread between the extended units reduced to an approximate 10cm. Light-rubber labels were attached to the extended arm units in order to increase friction when gripping an object.

With this setup for the Arm-Controller, the large-servo *(mounted to the body of the robot which rotates the arm in a direction parallel to the itself)* brings the gripper in place for the payload to be picked/dropped or carried. This setup does not allow picking-up of payloads found anywhere in the map. SPAMbot can only carry objects that are found right in-front of it - in its path of navigation. We

also have the limitation of anticipating the gripper-ends to correctly fall on either sides of the object without colliding with it *(when extending the arm).* If the robot is aligned at an angle to its navigation path, then we have the risk of the gripper-extensions colliding with the payload whilst the arm opens up.

Image Processing Unit

A method for recognizing the box number was approach from image processing to make it tolerable for errors such as the box being picked up slightly rotated manner, the changing lighting conditions etc. Atemel CPU in the Arduino Uno that we were using for the robot is not powerful enough to carry out major tasks and the image processing, therefore we used Raspberry Pi with OpenCV. A video input from a camera was processed frame by frame for the existence of a template that matched the static image of black and white which was to be found in any box. After this is found the adjacent color code was extracted by extending the area. By converting to grayscale and machine the intensity levels the code of the box could be recovered. Challenges faced in implementing this were, the data transportation should be done on i2c because the serial link was used for the sonar and the template that was used to match had an effective region of use beyond and closer than that region it failed to recognize the code properly.

Sonar

We used a typical sonar module in order to detect obstacles in front of the robot. We have attached a simple sonar module to A0 and A1 as Transmitters (Trigger) and Receivers (Echo) respectively. The key idea is to transmit a sample of ultrasonic wave, sense it back, and calculate the *half of the time* difference between Tx and Rx, assuming the speed of sound is 344.424 m/s.

All the supplementary methods for the sonar are implemented in ***sonar.ino*** and are called as needed while following the lines. Whenever it detects an obstacle within the provided range (in this case, 10 cm), it hands over the process to the Arm controller to grab the cubical obstacle. However, since these readings from the sonar may be a bit noisy and may lead to false positive detections, the robot was programmed to stop when it detects an obstacle and slowly move backwards while keep scanning to confirm that there is an actual obstacle.

IR Reader

For the purpose of following lines in the map, we have used IR sensor array which consisted 8 IR LED along with receivers. Specifically, couple of 4-way tracking module have mounted in order to capture straight lines, crosses and t-joints. It worked with the voltage of 5V and The detection distance can be adjusted from 1 mm to 60 cm. However, we have observed that the closer distance performance more stable, white reflection furthest distance.

After mounting the sensors with correct order which is described in the previous section (Hardware Structure) we have used the output of these sensors as digital inputs for the Arduino. The reason that we use this type of sensor formation because we wanted to detect current location of the robot very precious. Therefor we put front IR and Rear IR to detect crosses and t-joints and farthest left and right IRs to increase the accuracy of the turnings. Other remain sensors were used for typical line following task. Since the control board of the sensors gives digital outputs as 1s and 0s, it was easier to handle the readings and figure out the correct locations. However there were instances where we have encountered some problems because of unstable power supply and surface reflections which resulted continuous tuning over and over.

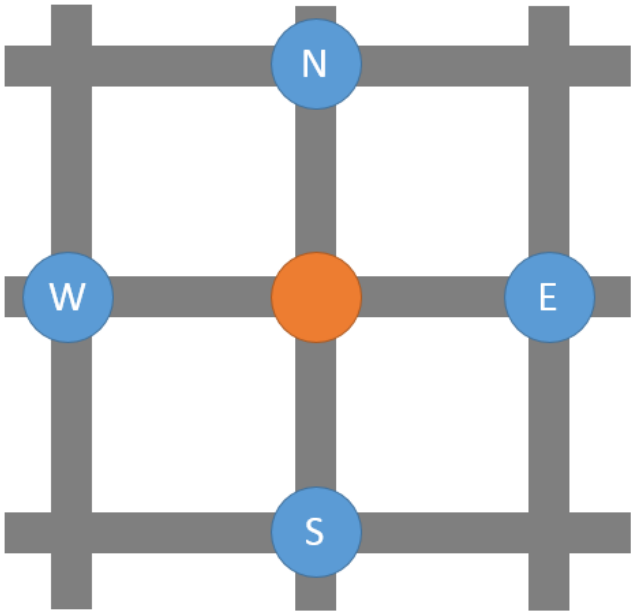
Mid Tier

Direction Controller

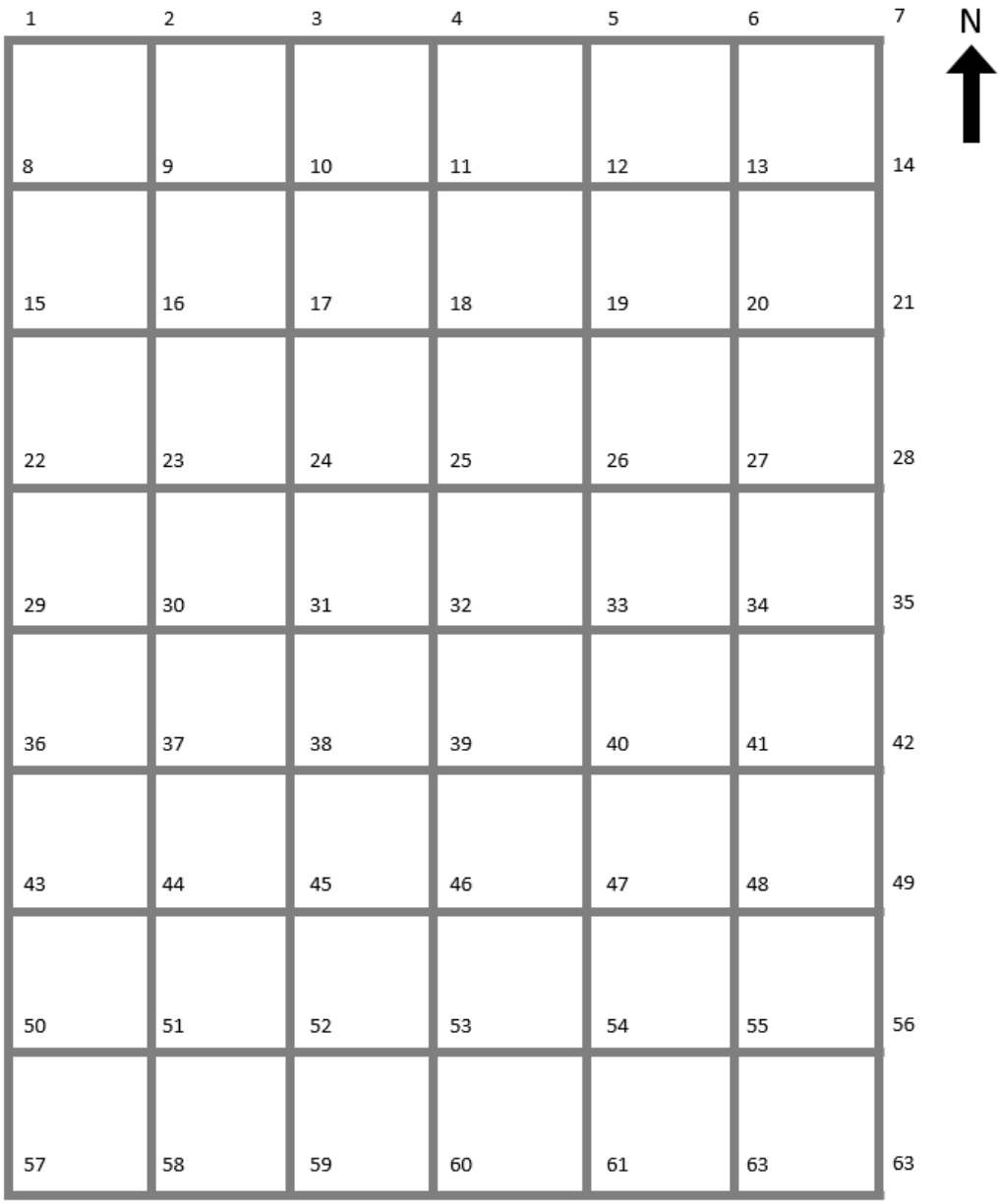
This module keeps track of the heading (*cur_heading*) of the robot and allows the programmer to turn the robot by simply using the direction it needs to be turned. Initially the robot is assumed to be facing North and we can pass the directions that it needs to turn using an array (*orders*) which contains a list of predefined directions. We use four main directions; S_LEFT, S_RIGHT, S_FORW and S_TURN180. Whenever we ask the robot to turn either of these directions, it calculates the resultant direction that it needs to turn with respective to the map's frame (N,S,E,W). These directions are sequentially fed into the motor controller where it checks for the next turn on each junction.

Map Manager

This module creates a higher level grid representing the lines and junctions of the target environment. Since the robot easily can identify a junction (cross, T or L), it always go for the next junction to calculate the next direction. These junctions are numbered from left-to-right, top-to-bottom manner; starting from 1. If we consider these junctions as nodes of a graph, each of them have four neighbors; North, South, East and West.



In this case, we have used 7 x 9 grid as the environment and following diagram shows the spread of numbered junctions.



This needs to be initialized at the very beginning of the program and for the easiness of coding, this is implemented as a linear array. This segment uses another array with the size of NUM_NODES (63 in this case). They are to keep track of the availability of a particular node, which is updated either the as predefined or from sonar readings. It is essential to maintain such data, in order to find the shortest path between two given nodes, in the Mission Planner stage.

Line Follower

Following a line seems easier to implement but has complications in the real world. As we found out, simple models which turn based on direct input of the IR sensors fail to correct their errors fast enough. PID controllers have solved this problem before and we thought of implementing it, the model that we used had penalties for the errors. The output of the PID was multiplied by a constant and handed over to the motors such that stronger errors will make the motors spin more in different directions. And while in the line following loop other IR sensors that are there to checking crossing points are polled for input if they indicate that there is a cross present the mission planner is polled for the direction to be taken, after that is received the turning is done by the motor controller.

Arm Controller

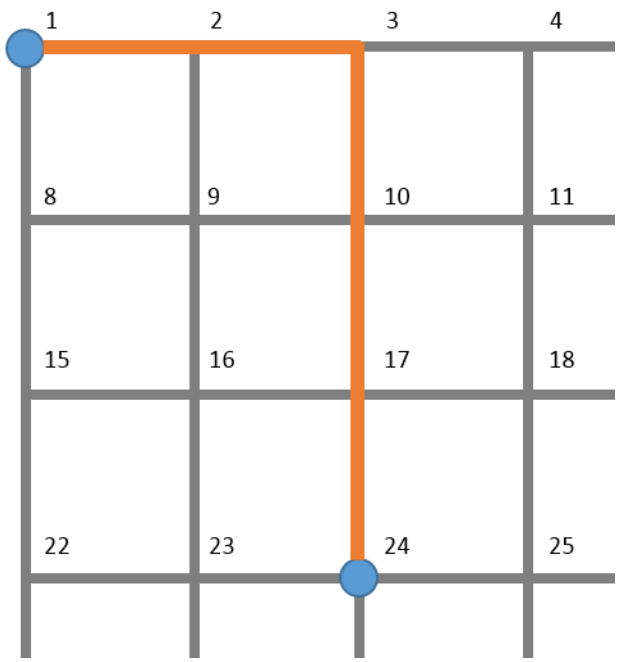
We describe the software implementation of the Arm-Controller here. The hardware setup is explained in the earlier section in the *"Lower Tier"*. The two servos *(larger/shoulder and standard/gripper)* are directly wired to the primary arduino on which all other components also operate. The gripper functionality is written to lift and drop payloads from the ground and from on top another payload. Hence, the appropriate angles at which the *shoulder-servo* needs to be aligned was deduced by trial and error method. Thus, these angles for gripping and placing payloads from the ground/on-top-of-another-payload are predefined. We also predefine an angle at which the shoulder is kept when the robot is moving whilst carrying a payload. Initially the gripper-servo angle is set such that the gripper is fully open.

The software implementation only has two methods to control the Arm movements: ***grabBox()*** and ***dropBox()***. Both methods take in a single argument indicating the positioning of the payload when executing the action - whether on the ground *(indicated as **BOTTOM**)* or on top another payload *(indicated by **TOP**)*. The shoulder-servo rotates the arm by the appropriate predefined angle in order to place the gripper in alignment to pick or drop from/on the TOP or BOTTOM. Upon gripping a payload the shoulder rotates backwards an angle sufficient enough for the payload to remove contact with the ground/another-payload. We also maintain an extra variable ***("whichBox" = TOP or BOTTOM)*** to indicate the position from which the current payload has been lifted. This enables SPAMbot to know the current position of its arm and allows it to correctly deduce the amount of shoulder-rotation required before releasing the payload.

High Tier

Mission Planner

The key idea in this tier is to move the robot one place to another in the grid. However, the actual movement is controlled by the lower tiers, the main task of this phase is to generate the relevant directions which are needed to navigate the robot to the destination. Hence, we apply A Star graph search in order to find the shortest path between the source and the target nodes.



If we assume that we need to travel from 1 to 24, the A Star algorithm will provide the path as [1, 2, 3, 10, 17, 24] when using *findPath(startNode, endNode)*. However, since the mid tier expects a list of directions, at each node, relevant direction is calculated with respect to the current direction of the robot. Then we would obtain the expected movement from *getNextHeading(node1, node2)*, when we provide the two adjacent nodes along with the current heading. This the above example, if the robot was facing EAST, the order list would be [S_FORW, S_FORW, S_RIGHT, S_FORW, S_FORW].

Moreover, in the first traversal, robot doesn't have any idea if a node in the middle is occupied by an obstacle or not. Hence a separate array (*available*) is maintained, where we can control the shortest path we obtain. For instance, if we enable the nodes column by column manner, the robot cannot move towards node 2 at the first movement. Hence, we program the robot first to move to 22. Then we enable the 2nd column, which allows the robot to move along that particular column. Then after the robot comes to node 2, we enable the 3rd column. Likewise, this allows the robot to scan the whole map and mark the nodes with obstacles.

Furthermore, when the robot is traveling from a node to another, it keeps scanning the path using the sonar sensor in order to detect any obstacles. It is by default enabled in the PID mode to avoid sudden crashes with obstacles. If he detect an obstacle, it alters the original plan and triggers the sub routing of grabbing the obstacle and moving it to a predefined node in the map.

For more information, please contact the authors.

Shabir : shabir.tck@gmail.com
Praneeth : gnomez.grave@gmail.com
Madura (Anushanga) : Madura.x86@gmail.com
Milan : milanharindu.ucsc@gmail.com