Autonomous Sentry Robot

Brian Dodge, Nicholas Musco and Trevor Roman

Dept. of Electrical Engineering and Computer Science, University of Central Florida, Orlando, Florida, 32816-2450

***Abstract* — The objective of this project is to create a land based, autonomous surveillance robot that implements mapping and localization. The Autonomous Sentry Robot is a multifaceted security system for use in enclosed buildings. The ASR will be equipped with a variety of sensors and a Microsoft Kinect, allowing it to autonomously navigate and dynamically map its place in space. The ASR will then patrol the space and alert the user in the event of detected motion. The group chose this project because we have a mutual interest in robotic systems and we wanted to develop a cost effective solution for a robotic sentry.**

***Index Terms* — Computer vision, intelligent robots, motion detection, power distribution, robot control, simultaneous localization and mapping.**

I. Introduction

Robots are becoming more prevalent in the world. They are no longer just in movies. They are being used for industrial purposes, and research projects. Robots can be found being used by the military and police departments as drones for aerial surveillance or robots for Explosive Ordnance Disposal. Robots are now found in the home as toys, such as the MIP or Sphero, and cleaning assistants like the iRobot Roomba vacuum cleaner. They can be found in hospitals, being used for surgery. Robots are being developed in labs that can map and localize themselves. Most of the robots mentioned are teleoperated by humans, or perform simple, repetitive tasks. There is an area which seems to not be well covered in consumer robotics. The area is land based, autonomous surveillance robots that implement mapping and localization. The goal of this project is to create a robot that does this.

The Autonomous Sentry Robot is a multifaceted security system for use in enclosed buildings. As previously stated, the ASR will be equipped with a variety of sensors and a Microsoft Kinect, allowing it to autonomously navigate and dynamically map its place in space. It will take the sensor data, and either use an onboard processor or a local computer to use the data for mapping and localization. From this map, it will plot an efficient path to patrol within this space, and its camera and vision capabilities will detect changes in the environment or motion. If changes are detected, the ASR's owner will be alerted through a mobile app, and they will be able to access its camera feed, as well as take control of its movement.

The robot is meant to be fully functional even when operating autonomously. When no users are around, it will need to keep track of its power level and return to the charging station when necessary. The robot must also be able to account for objects that are not picked up by the camera. Sonars and tactile sensors will be employed to ensure the robot doesn't drive over an unseen object. Above all, the ASR must be low power, low maintenance, low latency, easy to operate, and able to map, navigate, and detect reliably.

II. Hardware Design

The ASR is made up of both mechanical and electrical hardware systems. These systems must work together in order for the ASR be successful. That being said, it is important to design the system components to be independent so that if a part needs to be modified or replaced it can be taken care of without affecting the entire system. This section contains the decisions made for the mechanical and electrical hardware of the robot as well as the reasons for those decisions.

A. Mechanical System

The mechanical system of the ASR contains the sections related to the chassis, the drive system, wheels, and the motors. The system has been designed to be as simple as possible.

1. Chassis

The chassis is the main hub for the entire robot. It must be able to accommodate every subsystem. Our team decided to go with the VEX medium chassis kit for the ASR. The kit is extremely well designed and something we have worked with in the past. The kit is 12.592" x 12.92" and can be cut to be smaller if necessary. The dimensions of the medium kit are well within the form factor chosen for the ASR but it is also not too small. The chassis is also relatively inexpensive. As system compatibility is important for the ASR, the factors listed and discussed below also played an important role in choosing the medium VEX chassis.

2. Drive System

Our project requires the ASR to be able to maneuver around many obstacles. For this purpose a highly mobile drive system would generally be preferred. However, as our software would already be challenging, we decided not to add to the difficulty by selecting a complicating drive system. This narrowed our choice down to only one option. A tank drive system was chosen for the ASR due to its simplicity and effectiveness in achieving our overall goal.

It is a very simple design to implement as it is essentially a rectangle with four wheels. Each wheel in our system is individually powered to allow the ASR to carry a larger load. In a tank drive system, the left side of the robot and the right side of the robot each act as one. This cuts down on computation needed to figure out which motion would be best for a holonomic system to navigate a room. The VEX chassis is a perfect fit for this drive system. The chassis comes with 4 rails and two bumpers for mounting wheels and motors. Those components are for the four main drive wheels.

3. Wheels

The chassis design was factor in our wheel choice. The tank style drivetrain really works effectively with traction wheels. The traction wheels will allow the robot to turn and strafe with more friction than any of the other wheels we looked into. This means that there will be less slippage to worry about allowing our encoders to give us better data. There are many types of traction wheels available. However, we decided to look at choices that VEX robotics had to offer as they would be directly compatible with our chassis.

All of the VEX wheels are designed to support the VEX chassis and required electronics. As the ASR is not meant to carry a large load, there is no need to compare load specifications for the following wheels. The first option we looked into are 2.75 inch traction wheels. The second option we looked into are 4 inch traction wheels. The ASR may need to be able to move over thresholds to effectively map a space. If the ASR cannot enter a room because it cannot make it over a threshold, it loses much of its functionality. The 2.75 inch traction wheels are sufficient to move over standard thresholds. Therefore the ASR should be able to enter any room with ease. The 2.75 inch wheels were also more cost effective which helps us with our goal of creating a low cost, mapping robot.

These wheels were chosen to work with the chassis. The chassis uses 0.182” standard VEX holes for mounting and the wheels use 0.125” square bars for shafts. The shafts of the wheels will properly fit though the chassis holes for mounting. Bearings and a shaft collars from VEX robotics will be used to hold the wheels in place.

4. Motors

The motors chosen need to be able to support the weight of the robot and all of its components. The robot should weigh no more than 15 lbs. This means that the motors need to have a stall torque greater than 0.85 N-M in order to run properly. The torque value is based on the weight of the robot and the radius of the wheels. The motors should also not have a large current draw in order to maximize battery life. After conducting our motor research, we have chosen to use a DC motor. Having chosen the VEX robotics chassis, we decided that looking at VEX motors would be a good start for system compatibility. The VEX motors we researched are shown in Table 1below.

Motor	**RPM**	**Needs Controller**	**Stall Current (A)**	**Stall Torque (N-M)**	**Price ($)**
393	100	Yes	4.8	1.67	14.99
3 wire	100	No	Not Listed	Not Listed	Not Listed
269	100	Yes	2.6	0.972	12.99

Table 1: Motor Comparison

The 3 wire motors are motors that we already have. The third wire is for PWM signals and therefore it doesn’t need a motor controller. However, there is not a lot of data available on them and we only have three. This eliminates them from being used on the ASR. The 2 wire 269 motors are less expensive than the 2 wire 393 motors. Therefore we have chosen to go with the VEX two wire 393 motors. The motor is a DC motor meaning it runs using DC voltage. That makes it ideal for our system as we are using a battery. DC motors are very easy to control which is a necessity for the ASR. The motors are shown in figure 1 below.

Figure 1: VEX 2 Wire motor 393 [1]

With these motors having two wires it is necessary to get a motor controller for them. As our microcontroller is able to generate PWM signals we have chosen to get the VEX motor controller 29. The motor controller is specifically designed to work with the VEX two wire 393 motors. The motor and motor controller combo is priced at $24.98 on the VEX website making it a great low cost option for our project.

B. Electrical System

The electrical system of the ASR contains the sections related to the power system, the microcontroller, and the sensors. The system has been designed to be as simple as possible.

1. Battery

The battery is an important aspect of the ASR. It needs to have a high capacity and it should be designed for deep cycling. The battery must be able to discharge enough current to run the motors and electronics on the robot. It also shouldn't be affected by the memory effect. After careful research we chose to go with a NiMH battery because they don't require any special care and are safer than Lithium batteries [2]. They also have a higher capacity than NiCd batteries. After that decision was made, two batteries were under consideration. The batteries' specifications can be seen in table 2 below.

Battery Brand	Voltage (V)	Capacity (mAh)	Price ($)
Tenergy	7.2	3800	23.99
Tenergy	7.2	2000	9.99
Tenergy	7.2	5000	32.99

Table 2: Battery Comparison

For the ASR we chose the Tenergy 7.2V 5000mAh NiMH battery. We chose this battery because it has a higher capacity than most other 7.2V batteries. The battery is able to deliver 40A of current which is well above what the ASR can draw. The battery is designed to not be affected by the memory effect. Therefore it can be charged at any stage instead of only when it has been completely discharged. The battery is 7.2V making it perfect for running the motors we have chosen. The battery is relatively inexpensive and costs $32.99. The battery is pictured in figure 6.2.1-1 below.

Figure 2: Tenergy 5000mAh NiMh Battery [3]

2. Charger

Our group has chosen to go with the Tenergy Universal Smart Charger with the charge rate of 2A. This will allow a slightly faster charge time to get the ASR back out into the room for its patrol. The battery charger is able to detect the battery voltage to ensure a proper charge and is equipped with a temperature sensor to ensure that the battery doesn't overheat. The charger is shown in figure 3 below.

3. Power Distribution

The ASR's batteries will essentially have three loads, one load from the Microsoft Kinect at 12 V, one from the motors at 7.2 V, and one load from the microcontroller, microcomputer, and sensors at 5 V. Since the motors run on the same voltage as the batteries, they will not require a separate power distribution board. However, we will need a power distribution for both 12 V and 5 V.

The ATmega328P, tactile sensors, and ultrasonic sensors all run at 5 V. The maximum current draw from them is 1.10 A. Since we have experience from a previous laboratory with 5 V voltage regulation, we decided to use what we know. We decided to use the LM2576, a 5V switching voltage regulator. We will be

using TI reference design [3], since it does exactly what we need, as shown below in the Figure 3.

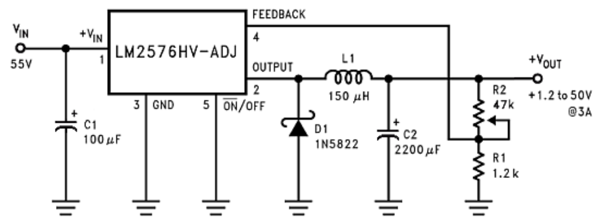

Figure 3: Microcontroller and Sensor Power Supply

For the Microsoft Kinect Sensor, we will require a 12 V voltage regulator. We designed one in TI Webench Power Architect as seen in Figure 4. The voltage regulator uses a LM2587.

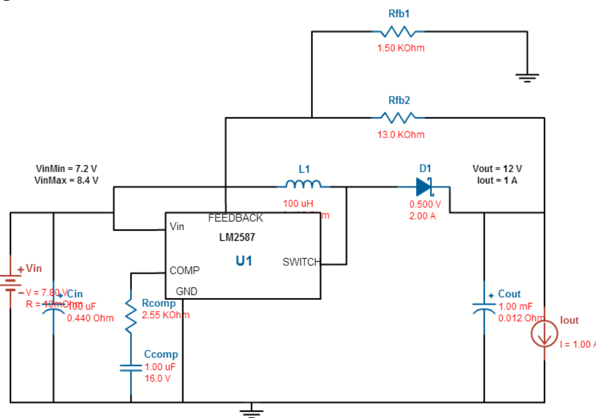

Fig. 4: 12 V Power Supply

4. *Microcontroller*

From our research, we saw that the ATMega328P has a faster clock frequency, but less Program Memory, less RAM, less I/O pins, and less USARTs/SPIs than the ATmega2560. Since we do not plan to do anything complex with the microcontroller and need for it to react to obstacles quickly, we decided on using the ATmega328P. We will be using it with an Arduino bootloader to simplify the programming required, thus saving us some time. All the microcontroller needs to do is take movement commands and send the commands to the motor controllers, and take sensor data from the tactile sensor, ultrasonic sensors, and motor encoders and then send movement commands to the motors, if needed. The faster clock frequency actions for the robot.

5. *Sensors*

To be successful, our robot will require long range, medium range, and short range sensors. For the short range sensors, we choose the VEX bumper sensor. It will complement the HC-SR04 ultrasonic distance sensor. It'll work well for a medium range sensor with a range of 2 cm to 4 m. The bumper sensor can handle anything that is missed. We will have two bumper sensors in the front of the robot, along with one ultrasonic sensor. These sensors should be able to handle close to medium range object detection. If they are triggered, the microcontroller will react and move the robot away from the object. We will also have the same configurations on the back of the robot. This will cover the cases when the robot backs up and it'll ensure that it does not run into anything while backing up.

III. Software Design

The ASR is a complex system of interconnected sub-subsystems. The overall system has distinct inputs and outputs, and so should the individual subsystems. With this approach in mind, our design attempts to be as modular as possible. This way modifications can be made to one system without too much impact on other systems. This approach allows us to utilize ROS as a general purpose framework. First, a high level view of the overall system architecture will be presented. Following this, each system will be looked at in more depth. ROS requires the use of many different nodes and packages for things like creating coordinate transforms between sensor frames, viewing data, and performing navigation procedures using a map. Since these are features of ROS itself and not modules we will be programming, they will not be discussed.

A. *High Level Software System Architecture*

The overall system is contained within and being executed on a laptop running Ubuntu Linux. The subsystems are, manual navigation, autonomous navigation, mapping and localization, motion detection, and the state manager. The inputs to the system are the map, streams of data from various sensors, and input from the user. The outputs of the system are the current generated map which is fed back in as an input, alerts which are pushed to the user's Gmail account, and locomotion data to the motors.

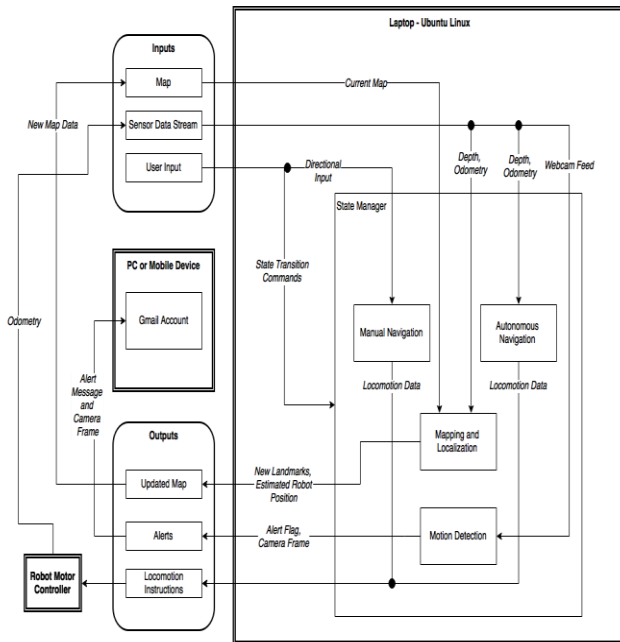

Figure 5: High Level Software Architecture

The above diagram does not demonstrate the order of execution in the system, but the relationship of inputs and outputs to each subsystem, and the overall system itself. Arrows flowing in are inputs, arrows flowing out are outputs. The type of I/O data is indicated on each line. Some of these subsystems are running concurrently, so dedicated threads are necessary, luckily ROS safely handles this. For instance, if the user decides to map autonomously, both the autonomous navigation and mapping/localization subsystems will be executing. The map will be being updated while the robot is planning its path, and sending locomotion instructions to the robot's wheel controller. The black dots provide no functionality, but instead indicate connected branches for better clarity.

1. State Manager

The state manager is a singleton class which sets the given state of the robot based on input from the user to the terminal. Abstract states like "Autonomous Mapping Mode" selected in the state manager are not actually representative of a single state in the manager, but rather two states operating simultaneously. This approach helps us eliminate redundancy and keep code more modular for ease of modification. The State Manager class is simple in terms of methods and variables. The *SetState*() method takes in a boolean array which flags states that are to be set active and inactive.

The current state can be retrieved with *GetState*() which retrieves the global variable *flags*[ ], containing whatever states were last set. The global variable *mapComplete* is set when the mapping state has completed and is used by the state manager to decide which options are available to the user. The state manager needs no knowledge of sensors or any other input as these are inputs to the classes of the subsystems. It needs no output except for other classes to be able to retrieve the current state for lower level decision making. Flags can be set by subclasses when certain procedures have been completed.

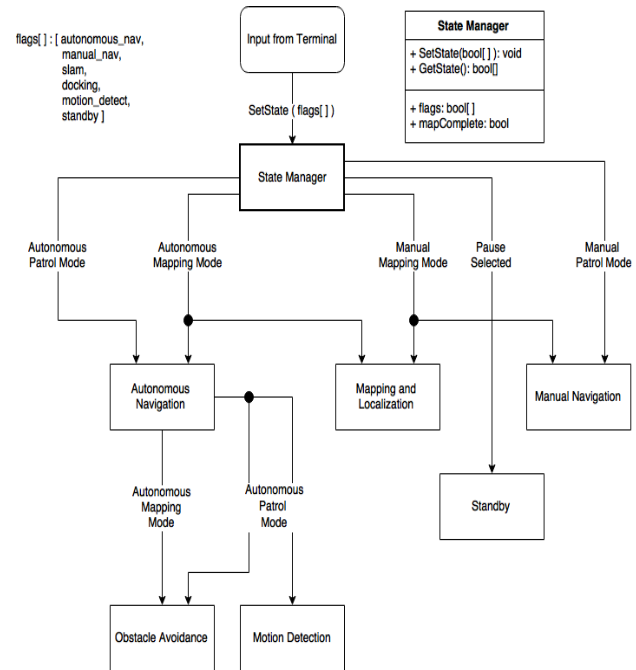

Figure 6: State Manager Architecture

1. Autonomous Exploration

During autonomous exploration, the autonomous_nav and slam flags are marked true, therefore, the Autonomous Navigation and SLAM subsystems are both active and executing. This is the state that occurs when the user selects Autonomous Mapping in the state manager. The autonomous navigation states essentially function like a finite state machines. Autonomous exploration starts by immediately moving forward while checking on sensor data, if no obstacles are detected it will briefly switch to the locomotion state to transmit motion data, then return to the wander state and repeat.

If any obstacle avoidance warnings are triggered, then it will immediately trigger the stop state, switch to locomotion and transmit data, then return and switch to the avoid obstacle state. The avoid obstacle state contains logic for determining and calculating a new heading. Once the heading is calculated, it switches to the locomotion state and transmits data to reflect the new heading.

Following this, it returns to the wander state and repeats the whole process. The wander state also checks if the standby flag has been triggered. If it has, the robot is told to stop and then exit this state and wait for instructions.

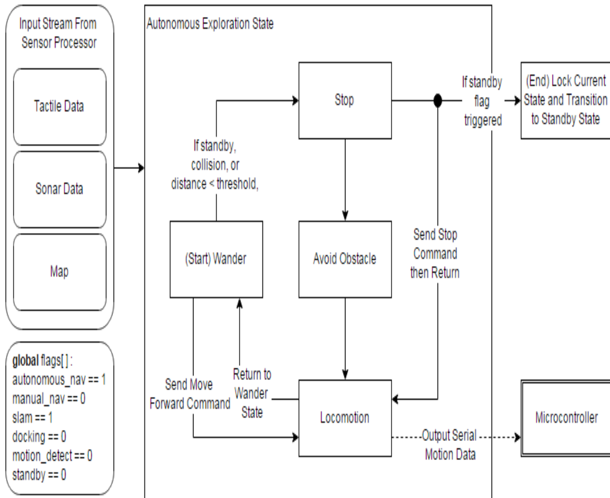

Figure 7: Autonomous Exploration State Architecture

2. *Autonomous Patrol*

During autonomous patrol, the autonomous_nav and motion_detect flags are marked true, therefore, the Autonomous Navigation and Patrol subsystems are both active and executing. This is the state that occurs when the user selects Autonomous Patrol in the state manager. First, the set of patrol nodes are read in from the user so that the nearest node can be determined. Nodes are essentially just coordinates on the map, so calculation is fairly simple. Once a goal node is determined, the navigation state is triggered.

In the navigation state, obstacle avoidance sensors are checked. If no warnings are triggered, the locomotion state is triggered and motion data is transmitted; moving the robot a unit of distance towards the goal, then returning to the navigation state. If an obstacle is detected or the standby flag is triggered, this state functions the same as in the autonomous exploration state. If the robot arrives at the node, it stops itself completely and transitions itself to the detect motion state. Motion detection is described later in the design section. Once motion detection is complete, it returns to the starting state and determines a new node to travel to, repeating the whole process.

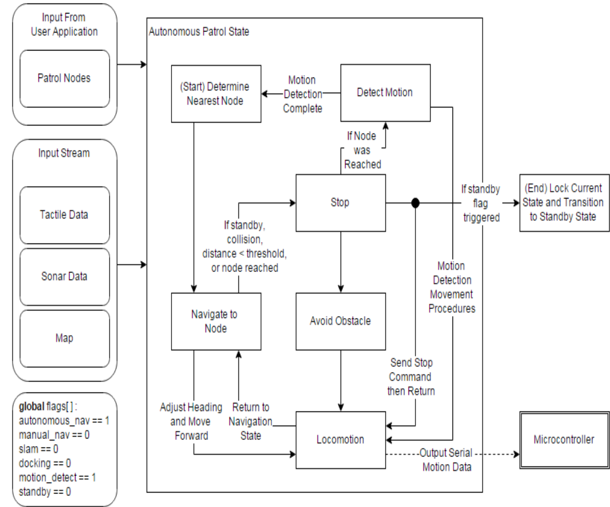

Figure 8: Autonomous Patrol State Architecture

3. *Manual Navigation*

During manual exploration, the manual_nav and slam flags are marked true, therefore, the Manual Navigation and SLAM subsystems are both active and executing. This is the state that occurs when the user selects Manual Mapping in the state manager. This state is very simple. The starting state listens for user input from the state manager. The input is in the form of directions which the user wants the robot to move. If a command is received, the locomotion state is triggered, motion data is transmitted, and then it returns to the listener. If the standby flag is triggered, this state is exited and the robot waits for instruction.

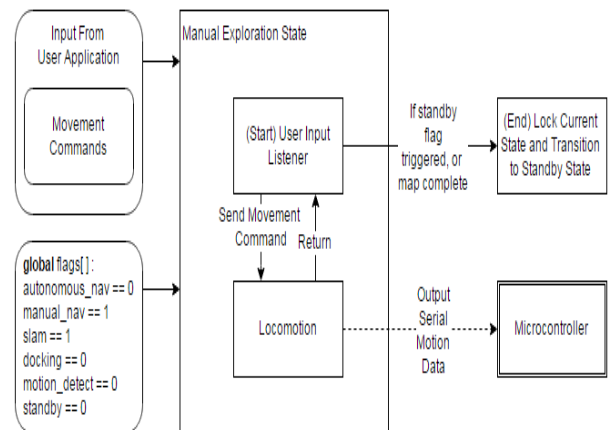

Figure 9: Manual Exploration State Architecture

4. *Manual Patrol*

During manual exploration, only the manual_nav flag is marked true, therefore, the Manual Navigation subsystem is active and executing. This is the state that occurs when the user selects Manual Patrol in the state manager. This state state functions exactly the same as the Manual Exploration state, except that the user has the additional option to capture a frame from the webcam if they wish to take a picture.

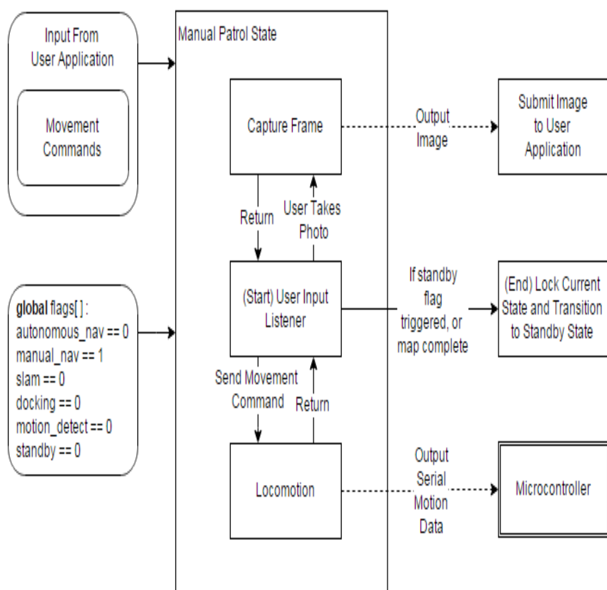

Figure 10: Manual Patrol State Architecture

5. *SLAM*

We take a black box view of SLAM because we're not writing any of the code to actually implement it. Instead we utilize the *hector_slam* stack that is part of ROS. With *hector_slam* we have a verified SLAM algorithm with easily accessible inputs and outputs. It's only input is the formatted depth data from our Kinect, as well as the current map. It's output is the updated map, which some other states have access to. The depth data is formatted into a faked laser scan with the *depth_image_to_laserscan* node that is part of ROS.

6. *Motion Detection*

Motion detection is a subsystem of autonomous navigation. Our design uses differential image comparisons to detect changes in a video feed on a per-frame basis. Motion detection becomes active during the autonomous patrol state, when the motion detection sub-state is triggered. This state looks for motion for a set amount of time, then rotates 90 degrees, after it has rotated 360 degrees, motion detection is complete. This state starts by opening the video feed and grabbing three frames from the webcam. Following this, it converts the three images to grayscale, then calculates the differential of the first two. It then thresholds and blurs those images, thresholds them again, and looks for contours in the image. If any are found then motion was detected. Otherwise no motion is present. OpenCV functions are utilized for some of the image processing procedures.

If a certain number of white pixels are discovered, then motion has been detected, so submit an alert with this image to the user's Gmail account. If motion wasn't detected, transition to the wait state. During the wait state it checks how long motion detection at this angle has been running. If the time limit hasn't been reached, then continue the loop of motion detection, reading in another frame and repeating the process. If the time limit of this detection has been reached, then instruct the robot to rotate 90 degrees and repeat the process like above. If the robot has rotated 360 degrees, then motion detection is complete, exit this state and return to the autonomous patrol state. If the standby flag is triggered, the robot exits this state and waits for instructions.

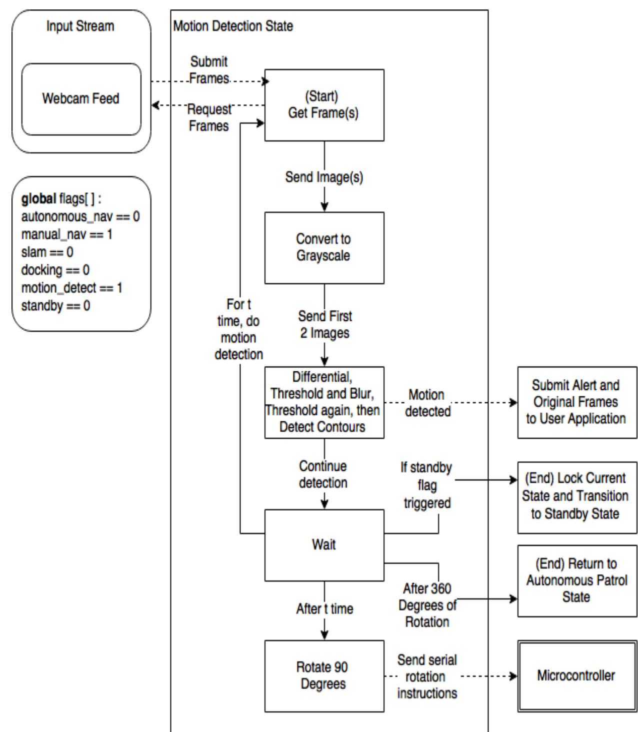

Figure 11: Motion Detection State Architecture

IV. Conclusion

A project of this magnitude ushers in an entirely new perspective on working as a team. This project required that all members be able to contribute effectively in order to create a good project. From the software perspective, our overall design and framework was implemented effectively. However ROS has proven to be very tricky. ROS helped immensely with concurrency, hardware drivers, visualization, communication, and the many open-source nodes and stacks made implementing some features very simple. However, ROS has an extremely steep learning curve and can be incredibly cryptic. Many of the available packages aren't documented well or it isn't clear how to use them. The majority of development time was spent trying to understand how ROS works, debug ROS errors, and crawling through ROS Answers for solutions to problems. The navigation stack has been particularly difficult to understand as we are still having issues with autonomous navigation. The cryptic nature of ROS also made it hard to explain the software systems to other team members and made code collaboration almost impossible. As a result, some of our software systems fell short of what we wanted, but we will continue to work on them until our deadline. As of the writing of this document, the systems fully operational are the state manager, manual navigation, SLAM, motion detection, and alert reporting. Autonomous navigation is still a work in progress. From a hardware perspective, the biggest issue has been working with the Kinect. Initially we intended to run the Kinect on 5V which would have required internal modification to surface mount components. In the end we decided due to time that it would be best to add the 12V regulator to our design.

Acknowledgement

The authors wish to acknowledge the assistance and support of Dr. Samuel Richie.

References

[1] Vexrobotics.com, 'Motion - Robot Accessories - Products - VEX EDR - VEX Robotics', 2015. [Online]. Available: http://www.vexrobotics.com/vex/products/accessories/motion?ref=home. [Accessed: 23- Jul- 2015].

[4] Batteryuniversity.com, 'Nickel-based Batteries Information – Battery University', 2015. [Online]. Available: http://batteryuniversity.com/learn/article/nickel_based_batteries. [Accessed: 23- Jul- 2015].

[3] Tenergy.com, 'Smart Universal Charger for NiMH / NiCd Battery pack 6V - 12V', 2015. [Online]. Available: http://www.tenergy.com/01025. [Accessed: 23- Jul- 2015].

[4] Texas Instruments, "LM2576/LM2576HV Series SIMPLE SWITCHER® 3A Step-Down Voltage Regulator ," LM2756/LM2576HV datasheet, June 1999 [Revised April 2013]

Biography

BRIAN DODGE IS CURRENTLY A SENIOR AT THE UNIVERSITY OF CENTRAL FLORIDA, AND WILL BE RECEIVING HIS BACHELOR'S OF SCIENCE IN ELECTRICAL ENGINEERING IN AUGUST OF 2015. BRIAN HOPES TO PURSUE A CAREER IN ELECTRICAL ENGINEERING AND/OR ROBOTICS, HE IS A MEMBER OF TAU BETA PI AND ETA KAPPA NU.

NICHOLAS MUSCO IS A SENIOR ENGINEERING STUDENT. NICHOLAS HOPES TO PURSUE A CAREER IN ROBOTICS AND ELECTRONICS HARDWARE FOR EITHER THEME PARKS OR SPACE EXPLORATION. HE JOINED LUNAR KNIGHTS AT UCF AS THE POWER MANAGER FOR THE 2014-2015 NASA RMC AND IS INVOLVED WITH THE LIMBITLESS SOLUTIONS LEG TEAM.

TREVOR ROMAN IS CURRENTLY A SENIOR AT THE UNIVERSITY OF CENTRAL FLORIDA, AND WILL BE RECEIVING HIS BACHELOR'S OF SCIENCE IN COMPUTER ENGINEERING IN AUGUST OF 2015. TREVOR WILL PURSUE A CAREER IN SOFTWARE ENGINEERING AND HOPES TO FOCUS ON ROBOTICS, COMPUTER VISION, AI, OR OTHER INTERESTING SOFTWARE.