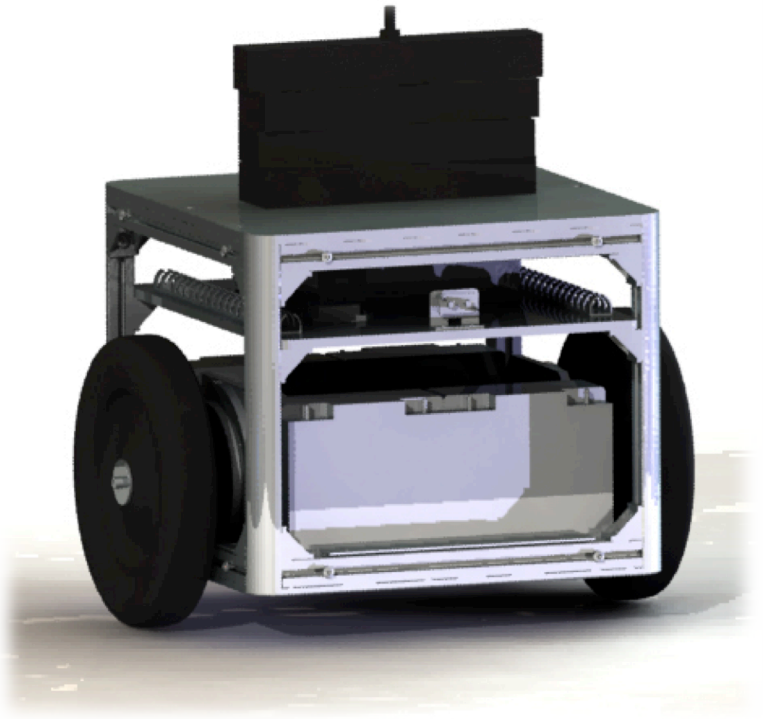
January 2012

Two-Wheel Self Balancing Robot

David H. Linke
Worcester Polytechnic Institute



Two-Wheel Self Balancing Robot

A Major Qualifying Project submitted to the faculty of
Worcester Polytechnic Institute
in partial fulfillment of the requirements for the
Degree of Bachelor of Science
in
Electrical and Computer Engineering

By David Linke

Advised by Professors Yiming Rong & Xinming Huang
December 2011

Table of Contents

Table of Figures

Abstract

This project set out to use an existing two-wheeled self balancing robot and add additional functionality to it for a general item distribution application. It was necessary to repair and stabilize the robot prior to adding functionality. The new functionality included making the robot avoid obstacles using a sensor as well as adding light and sound indicators.

Acknowledgments

I would like to thank my Chinese teammates Ding Yu, Yang Qian, and Zhong Jian for all their help in both making this project a success and making my experience in China amazing. While they all had classes and jobs or interviews to attend, they made themselves available for assistance whenever I needed it even at the expense of spending time with their friends or families. I cannot thank them enough for their efforts and friendship.

I would also like to thank Marzhana Mozhanova for all of her contributions in the PQP and first couple weeks of the MQP. It's a shame that visa issues prevented her from working on this MQP but her influence and support were most appreciated.

I would also like to thank all of the graduate students in the CIMS & Robotics Research Center on the Yanchang campus and in the Service Robot Research Lab on the Baoshan campus. Specifically, I would like to thank Wang Pengchong for his help throughout the entire project. He went above and beyond what was required of him as a graduate student in assisting us with any problems we came across as well as manufacturing the sensor mount for us. I would like to thank Wang Shuai for initial sensor testing help as well as Jiang Liangyin for assisting us in getting the SHUBot robot operational.

I would like to express my deepest gratitude to Shanghai University for hosting me and taking part in WPI's unique global projects program. Additionally I would like to thank WPI for working so hard to allow students to complete project work abroad safely, comfortably, and effectively.

Finally, I would like to thank all of the advisors that offered their time and guidance to this project. Thank you to Professor Yingzhong Tian of Shanghai University for providing us with direction, resources, and knowledge in all aspects of the project. Thank you to Professor Yiming Rong for ensuring the project would be successful and ensuring

our welfare. And finally, thank you to Professor Xinming Huang for his guidance and moral support provided remotely from WPI.

Introduction

Due to recent innovations and developments in engineering it is now possible to create robots and vehicles that can balance on two parallel wheels. While it is simple for vehicles such as bicycles and motorcycles to balance on two wheels while in motion, it is difficult for vehicles to balance where the wheels are aligned parallel to one another. The same principles apply for balancing two-wheeled vehicles whether the wheels are on the same axis or parallel, they just need to be taken into consideration and applied differently. Since two-wheeled vehicles are naturally unstable, external forces must be applied to keep them balanced. While the potential of two-wheeled vehicles where the wheels are on the same axis has already been realized, it has yet to be fully explored for vehicles where the wheels are parallel to one another. There are many possibilities for these types of vehicles and robots. So far they only exist for human transportation and proof of concept purposes.

Two-wheeled robots have several advantages over their four-wheeled counterparts that would make them more suitable for a variety of applications. They can turn with no radius for better operation in confined spaces. They can be simpler, smaller, and lighter structurally and because of this, consume less power. There are many potential applications these robots would perform better in that have yet to be explored. Since there are so many possible uses and not many current two-wheeled robots in use there is a large prospective market to be explored.

This project sought to make use of available resources at Shanghai University to add functionality to an existing two-wheeled robot for a specific application. Previous research has been conducted at Shanghai University on the principles of two-wheeled self balancing robots. Due to this research two robots were available for us to use for this project, although both were not fully functional. We decided to focus on a general item distribution application for adding additional features to a robot using the resources available to us. These functions included avoiding obstacles and indicating the robots status. Additionally,

some team members were to focus on researching existing control algorithms and demonstrating the robots assembly visually.

This report will cover the background research conducted on two-wheel self-balancing robots as well as the purpose, objectives, design details, and potential future work of this project.

Background

History

Two-wheeled self-balancing robots have been of great interest to engineers and researchers recently. Many have been developed independently of each other using the same basic principles. Some of more widely known two-wheeled self-balancing robots include Joe le pendule, LegWay, Nbot, Equibot and the Segway personal transporter.

Joe le pendule (Grasser, D'Arrigo, Colombi, & Rufer) was designed in 2002 by Felix Grasser with his group from the Industrial Electronics Laboratory at the Swiss Federal Institute of Technology (EPFL) in Lausanne. The robot uses an accelerometer and a gyroscope to sense the tilt angular rate of the body, and two incremental encoders to measure the position and velocity of the robot. It also has two decoupled control systems - one that balances the robot and controls its forward/backward motion and one that allows it to spin around its vertical axis for turning. It functions using a radio control and can be seen in Figure 1.

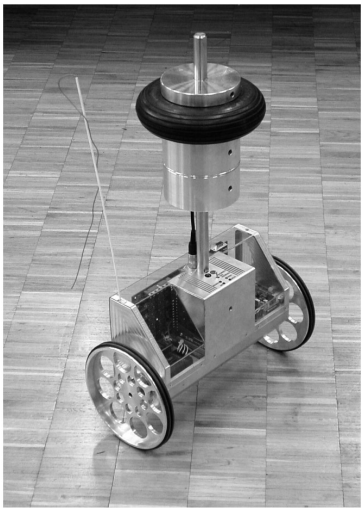


Figure 1: Joe le pendule Robot

LegWay (Hassenplug, 2002) was built by Steve Hassenplug using Lego bricks and the Lego MindStorms platform in 2002. While other hardware was necessary for it to function due to a lack of resolution in the MindStorms sensors, it proved that a complex two-wheeled self-balancing robot was possible to be built using primarily “toy” components. It functions using two electro-optical proximity detectors to balance and detect lines. LegWay can be seen in Figure 2.

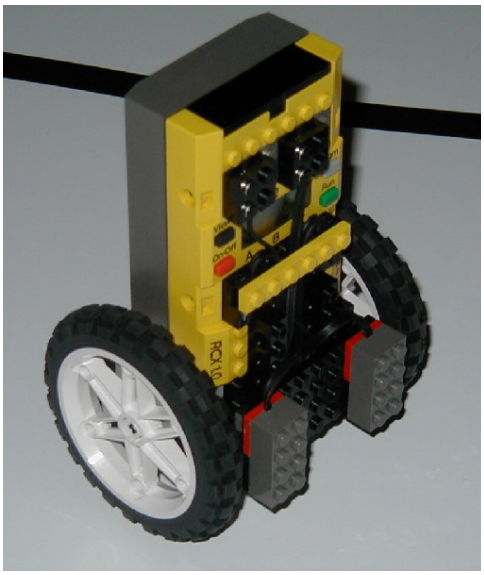


Figure 2: LegWay Robot

The Nbot (Anderson, 2010) was designed by David P. Anderson, the director of the Geophysical Imaging Laboratory at Southern Methodist University in Dallas. This robot features an HC11 microcontroller, ADXL202 accelerometer, rate gyroscope, tilt sensor and optical encoders on the motors to function. The gyroscope and accelerometer are combined with complementary filters to act as an inertial reference sensor. It was also featured as NASA's Cool Robot of the Week on May 19, 2003. It can be seen in Figure 3.

Figure 3: Nbot Robot

Another robot similar to the Legway is the Equibot (Piponi, 2006) by Dan Piponi, introduced in 2006. This robot is unique in that it uses a Sharp GP2D120 infrared range sensor to measure the distance to the ground. From this distance an ATMega32 microcontroller calculates the tilt angle of the robot, which is then used as the basis for balancing it. This robot can be seen in Figure 4.

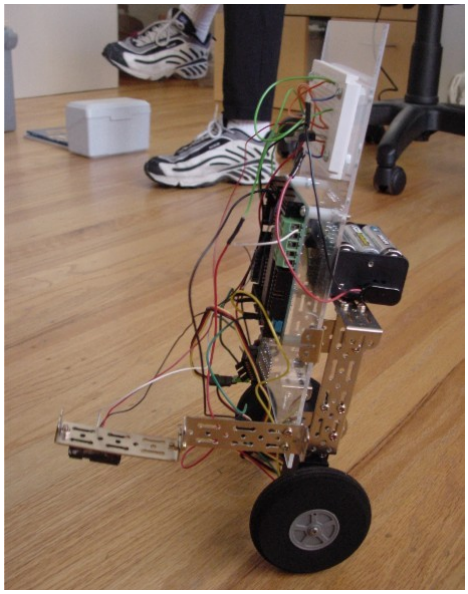


Figure 4: Equibot Robot

Perhaps the most well known of two-wheel self balancing robots is the Segway personal transporter (Kamen, 2011) created by Dean Kamen in 2001. Unlike most of these robots, this one is designed for human transportation with ease of use in mind. It contains two tilt sensors and five gyroscopes - three for measuring the forward and backward tilt angles and the corresponding rate of change, and two for redundancy and more reliable results. This robot can be seen in Figure 5.

Figure 5: Segway Personal Transporter

After researching the current two-wheeled self balancing robots we were shown the possible robots we would be working with. Upon seeing them and discussing the many possibilities for this project, a purpose was formulated.

Purpose

The purpose for this project is to repair and repurpose an older two-wheeled self balancing robot for a general item distribution application.

Technology

The key technologies that allow these robots to balance on two wheels are gyroscopes, accelerometers, microcontrollers, motor drivers, and motor encoders. Gyroscopes measure the rotation around axes while accelerometers measure the acceleration for all three axes. Using information from these sensors as well as information from the motor encoders, a microcontroller performs calculations in real-time. These calculations are then used to drive the motors with the appropriate torque to keep the robot balanced.

At Shanghai University two robots using these technologies were at our disposal; GBOT1001, a company produced robot for research purposes, and SHUBot, a robot created by Shanghai University graduate students. While we initially intended to use the GBOT1001 for this project, it became clear after working with it for a while that it would not function as necessary and the switch to SHUBot was made. Because both were worked with for a significant period of time the technologies behind both will be discussed.

GBOT1001

GBOT1001 is a two-wheeled self balancing robot produced by Googol Technology Ltd based in Shenzhen and Hong Kong. This robot has been produced for educational and research purposes. This robot was designed to be modular with an open architecture so its functionality can be customized. It functions using Simulink in Matlab 6.5 running on top of the Windows 98 operating system. It weighs 20kg and its size is 26cm x 45cm x 73cm. It uses an inverted pendulum model with a freely moveable base. The wheels are

independently driven to allow for turning and greater stability. The maximum speed is 1.6m/s and the maximum climbing angle is 20°. This robot can be seen in Figure 6. A block diagram of how this robot functions can be seen in Figure 7. This demonstrates how the hardware of the robot interacts to allow it to successfully self balance. This particular robot has been taken apart and rebuilt a number of times using different components by Shanghai University students for research purposes which has resulted in it not functioning as it should.

Figure 6: GBOT1001 Robot

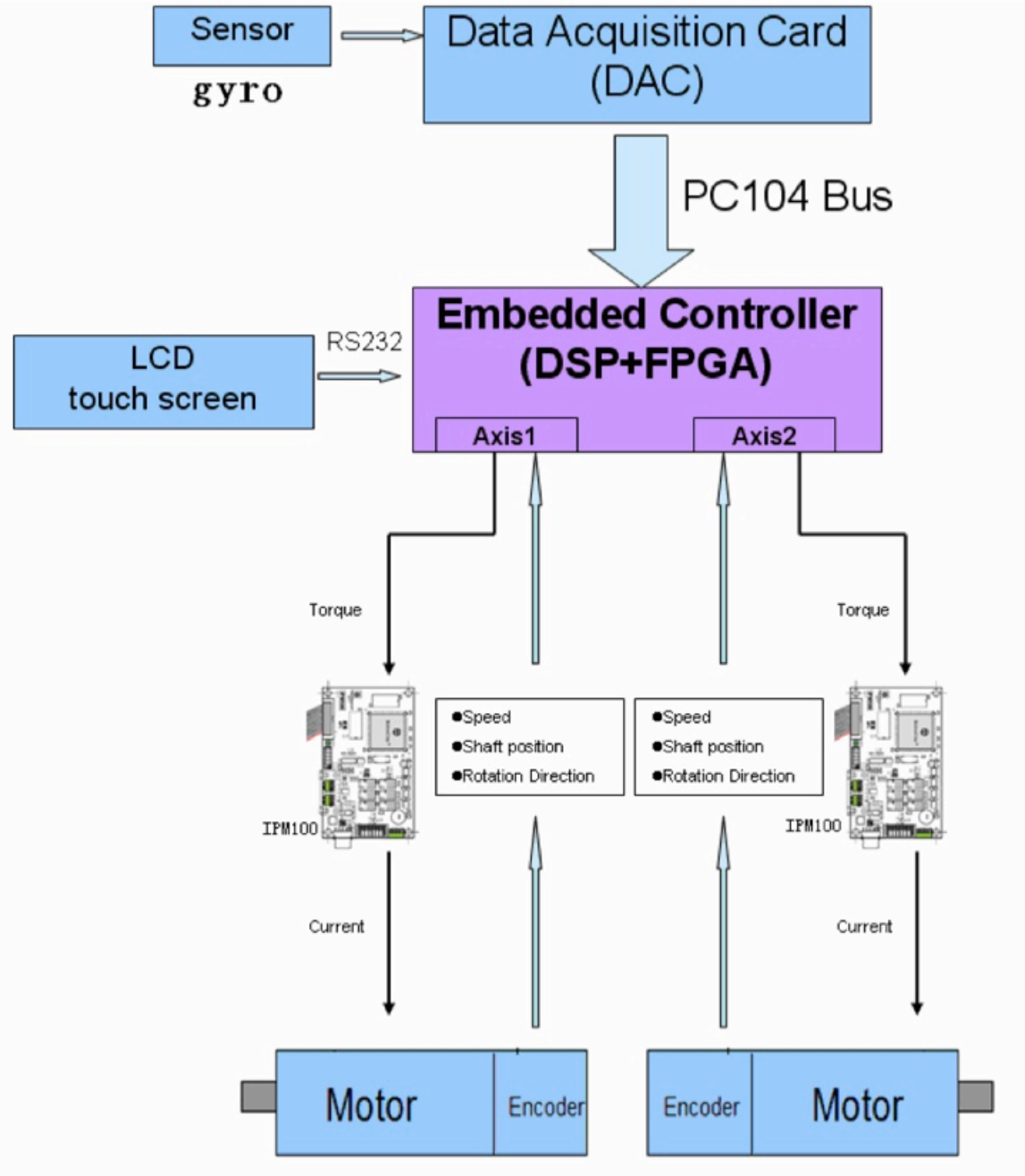


Figure 7: Block Diagram of GBOT1001 Robot

SHUBot

SHUBot is a two-wheeled self balancing robot created by Shanghai University graduate students. It can be controlled with a PC interface as well as an RF remote. It uses a National Instruments cRIO-9014 controller as well as a PC running LabVIEW 2010 to function. This controller features a chassis that allows up to 8 National Instruments modules to be used. Three modules were used for the operation of SHUBot; the NI9205 analog input module for interpreting the RF remote control signals, the NI9263 analog output module to control the motor drivers, and the NI9411 digital input module for reading motor encoder data. Additionally we had an NI9401 digital input/output module to be used for adding additional functionality. This robot used a combined gyroscope/accelerometer, the MicroStrain 3DM-GX1, to sense information regarding the robots orientation and movements. The robot also uses Accelnet ACJ-055-18 motor drivers for precise control over the motors movements. A picture of the SHUBot robot can be seen in Figure 8. A block diagram of the SHUBot we created can be seen in Figure 9.

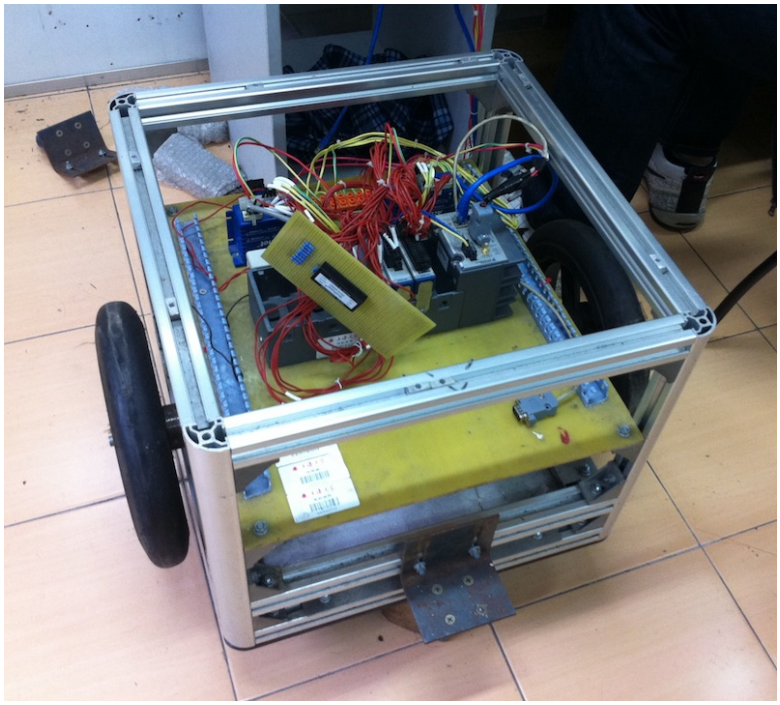


Figure 8: SHUBot Robot

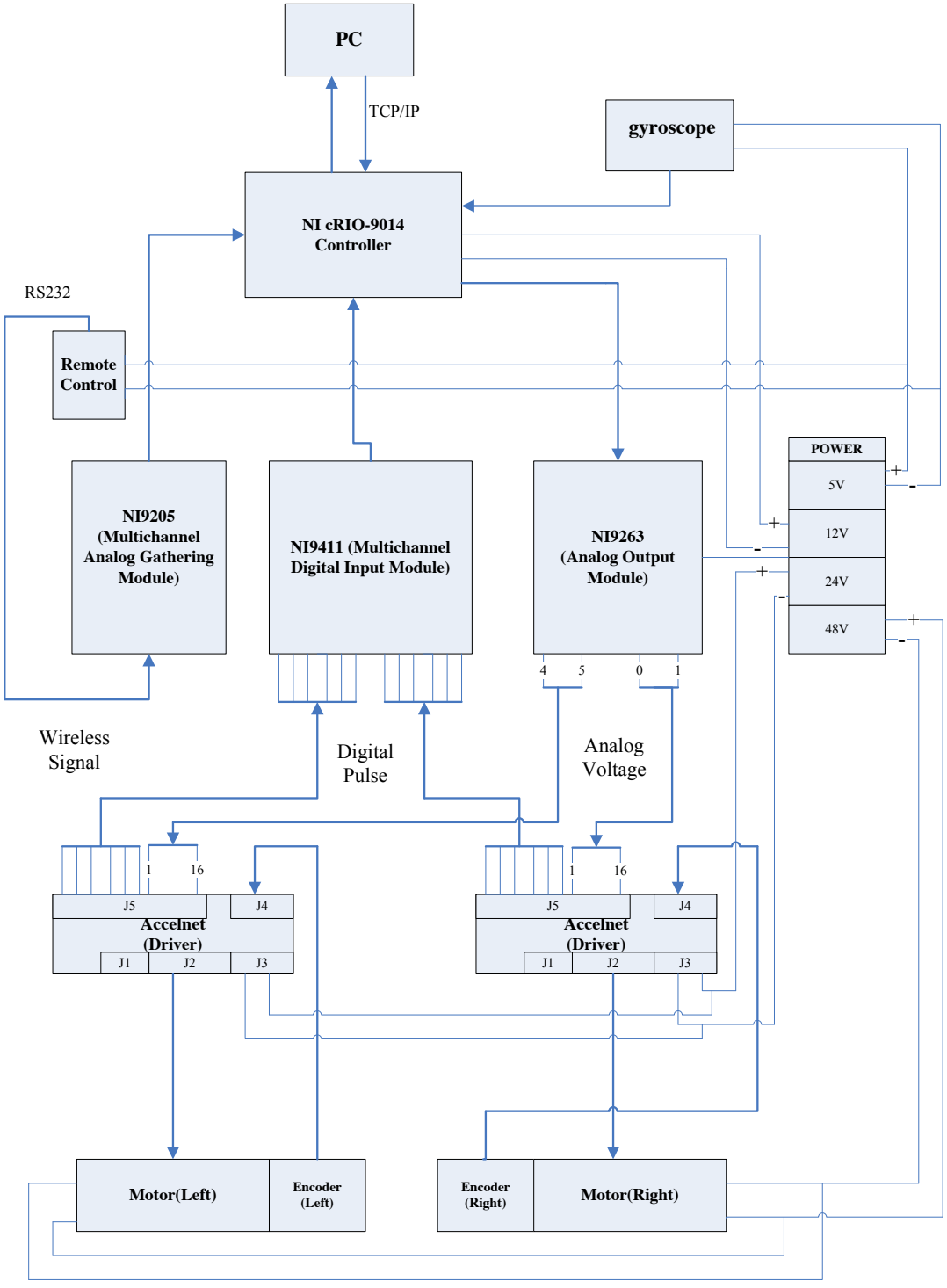


Figure 9: Block Diagram of SHUBot Robot

Objectives

The objectives for this project evolved over the first few weeks of working on this project as it became clear what we would able to accomplish with the resources available. The final objectives we settled on can be seen below. Some of the objectives will not be covered in detail in this report as I did not work on them at all and have no English literature on them.

1. To research existing control algorithms.
    - The purpose of this objective is to research the Pole Assignment and Linear Quadratic Regulator control algorithms and perform simulations with them to find the most optimal algorithm and parameters for balancing the robot. This objective was assigned to Yang Qian and will not be covered in this report.
2. To stabilize the robot.
    - The purpose of this objective is to ensure the robot can remain stable for a significant period of time. We defined this to mean remaining stationary for greater than 20 seconds.
3. To ensure proper movement of the robot.
    - The purpose of this objective is to ensure the robot can move forwards, backwards, and perform a zero radius turn.
4. To allow the robot to detect and avoid obstacles.
    - The purpose of this objective is to have the robot be able to both detect when an obstacle is in its way and avoid them.
5. To indicate the robots status with light and sound.
    - The purpose of this objective is to demonstrate the status of the robot with both light and sound indicators. The status is defined as initialization of the robot, information regarding obstacles, and termination of the robot.

6. To demonstrate the robots assembly visually.
    - The purpose of this objective is to demonstrate the assembly of the robot visually through both static and animated three-dimensional CAD modeling. This objective was assigned to Zhong Jian and will not be covered in this report.

Design

In this section all of the design work will be covered. Ding Yu and myself did this work, where he was in charge of hardware and I software. However, there was significant overlap in our responsibilities and we worked closely together on both due to the necessity of hardware and software functioning together.

Initial Setup

GBOT1001

We started off working with the GBOT1001 robot. This robot required no hardware modifications to initially get it up and running. Since it had an integrated controller/computer running a Windows 98 install from the factory it also did not need any software modifications. However once we started working with it we discovered a number of problems that at first were minor. The biggest problem was the battery would only last a few minutes so it had to remain tethered to AC power at all times. Another problem was the touchscreen was non-functional, mandating that a keyboard and mouse had to be used to operate it. Additionally the remote control could only move it forwards; the backwards functionality was broken. Finally it would not remain stable, which will be described in the next section, stabilization.

SHUBot

After working with the GBOT1001 we switched to the SHUBot (the reasons for this will be explained in the stabilization section). Since the SHUBot requires a computer to run and we did not have the original computer, we had to do extensive work to get it up and running. We did not initially know the version of LabVIEW that would work so we ended up installing 3 different versions before finding one that worked, the 2010 version. We had the LabVIEW project files previously used but did not have many of the resource libraries they required to run. We spent a couple days tracking down all the necessary files and

integrating them into the project. Once all the resource libraries were integrated we were able to successfully connect to the SHUBot and run the program. However the SHUBot could not balance because it was missing its gyroscope. It happened to use the same gyroscope that the GBOT1001 used, the MicroStrain 3DM-GX1, so it was removed from the GBOT1001. We could not connect the gyroscope to the SHUBot initially as it used a proprietary tiny connector. After searching for this connector at some electronics shops around Shanghai, we gave up on this connector and found a new one we could replace it with. Once the male and female connectors were attached to the gyroscope and SHUBot respectively, the SHUBot could see the gyroscope data and run properly. From there we moved onto stabilizing the SHUBot.

Stabilization

GBOT1001

Stabilizing the GBOT1001 proved to be a difficult task. It initially appeared like a simple task since it is based on graphical blocks in Matlab with few adjustable parameters. However once we started changing the parameters trying to stabilize it, it became clear it was not as simple as it seemed. The main operating graphical user interface can be seen in Figure 10. This interface contains the blocks that each have individual adjustable parameters. There are four different adjustable blocks; LQR parameters for balancing, PD controller parameters for turning, a noise filter parameter, and gyroscope offset parameters. Each of these parameters was adjusted to try to stabilize the GBOT1001.

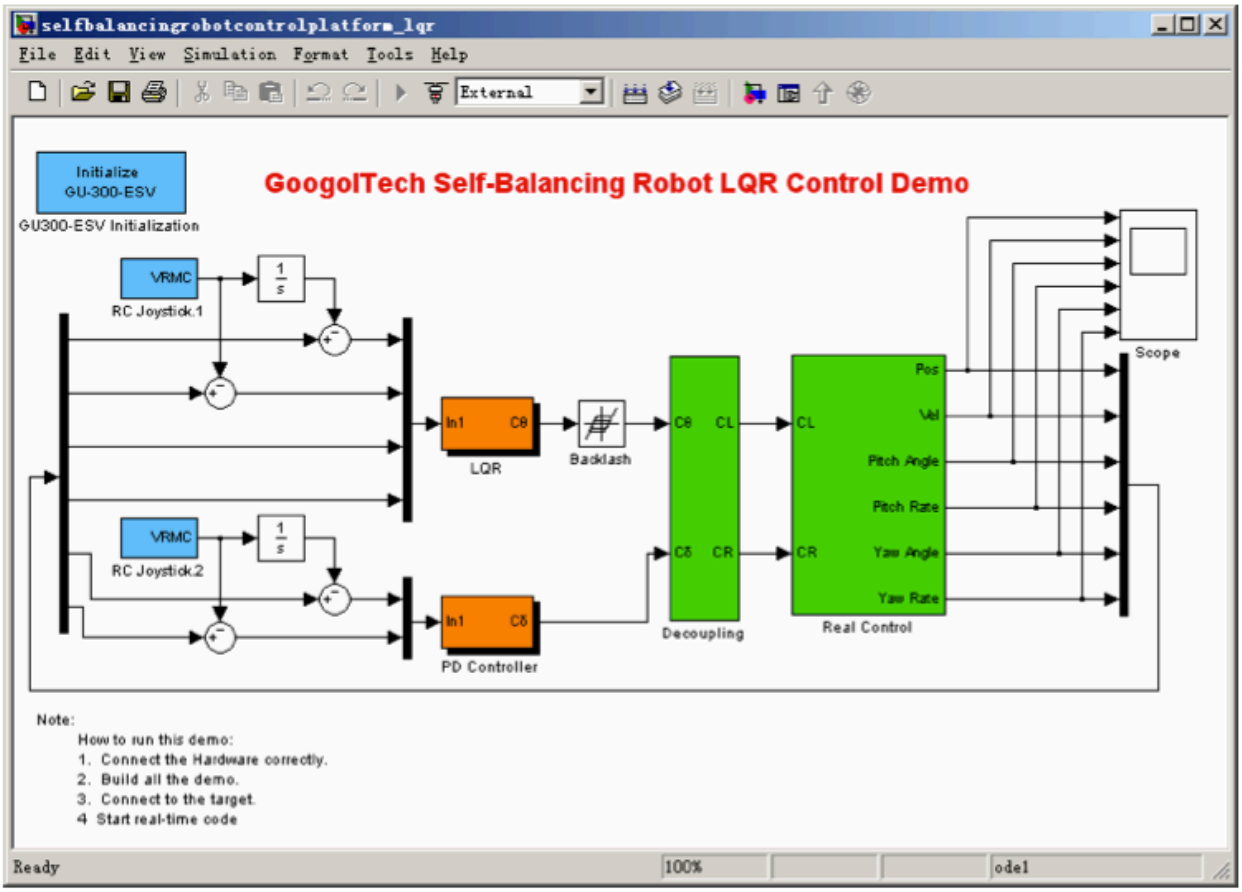


Figure 10: Matlab Graphical User Interface for GBOT1001

The first set of parameters adjusted were the LQR parameters for balancing. These are Kx, Kx', Ka, and Ka'. We could not find any documentation on what these parameters meant so we used the default values as a guide and experimented from there. We learned through experimenting that the larger Kx is the faster the response will be, that Kx' must be positive or the GBOT1001 cannot be controlled, that Ka must be negative or the GBOT1001 will balance in the wrong direction, and that the smaller Ka' is the faster the response will be. The below table contains the sets of values we found that worked best for these parameters.

Parameter	**Values [1]**	**Values [2]**	**Values [3]**	**Values [4]**
Kx	-1.2	.5	.5	.2
Kx'	-3.2762	.1	.2	.1
Ka	-25.559	-30	-30	-30.4
Ka'	-4.54	-3	-2	-2

The next set of parameters we adjusted were the PD controller parameters for turning. These are Kb and Kb'. These parameters adjust the signals before the decoupling block. Since turning seemed to work fairly well we did not experiment much with these. We found Kb to work well around -.5 and Kb' to work well at -.1, it's default value. The below table contains the two sets of values we found to work best for these parameters.

Parameter	**Values [1]**	**Values [2]**
Kb	-.54	-.5
Kb'	-.1	-.1

The next parameter we adjusted was the noise filter parameter. This is a cutoff parameter for a backlash filter to remove noise from the waveforms between the decoupling and real control blocks. Through testing we found that setting it to .1 and .5 resulted in no obvious difference in performance. However these had extraordinarily long compiling times. The default value was .3 so we used that or .25 although the difference in

performance was negligible. The below table contains these values that we found to work best.

Parameter	Value [1]	Value [2]
Cutoff	.25	.3

The final parameters we adjusted were the gyroscope offset parameters. These are AR and ARO, the angle offset and angle rate offset respectively. We found that they should both be around 500. If they were too large or too small stability would be greatly decreased. The below table contains the sets of values we found to work best.

Parameter	Values [1]	Values [2]	Values [3]	Values [4]
AO	510	505	507	506
ARO	505	505	505	504

Overall we found the K and A parameters to be most important in stabilizing the GBOT1001. However even after adjusting all these parameters for a week we were unable to meet our objective of having it remain stable for greater than 20 seconds. We tried contacting the company that produced it for help and they gave us a fresh copy of the program files. Once we installed them the GBOT1001 would still not balance. The company suggested it was a hardware problem that they could not help us with. Additionally as we were testing different parameters the GBOT1001 was deteriorating. We suspected there was information loss between components and there were increasingly frequent shutdowns that delayed work by at least 30 minutes every time it rebooted. Since it had so many issues and we wasted a significant amount of time on just one objective we decided it would be best to switch to an alternative robot. This robot, SHUBot, was our only other option. The SHUBot had a simpler hardware platform to debug things more easily and ran LabVIEW, another graphical programming application. However the SHUBot had some disadvantages to the GBOT1001 such as having no battery and requiring a computer. We felt it was the best option to move forwards with this MQP so we made the switch.

SHUBot

Once the SHUBot was configured properly we were able to start stabilizing it. However, due to the SHUBot's extremely low center of gravity and large wheels it could remain stable without even being on. In order to show that it could balance well ~20kg of weights were added to the top to actually make it less naturally stable. Once these weights were added we adjusted a number of parameters to try and stabilize it.

The main graphical user interface for adjusting control parameters on the SHUBot can be seen in Figure 11. This interface was originally in Chinese but with the help of my partners I translated some of it into English.

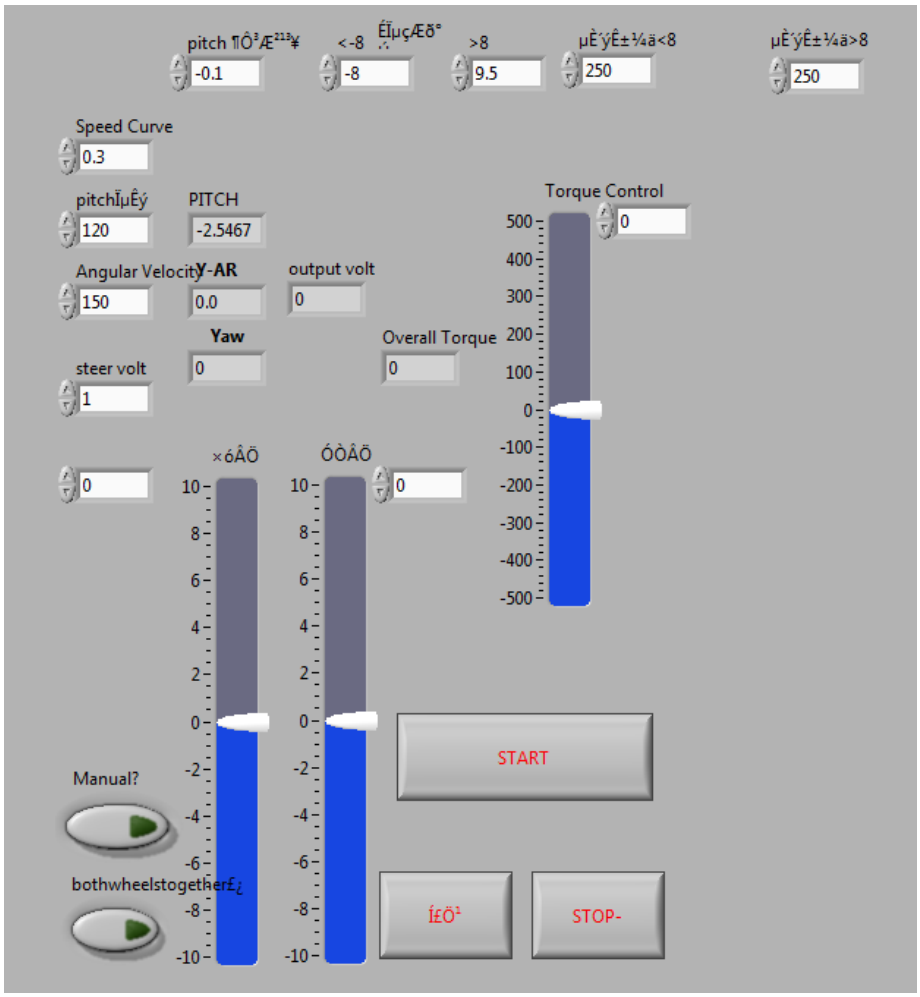


Figure 11: LabVIEW Graphical User Interface for SHUBot Control

This interface features five different adjustable parameters relating to stabilization. Three of these were based off theoretical modeling done by the graduate students that created the SHUBot. However, they did not take into account the additional weights on top of the SHUBot so adjustments had to be made. The first of these parameters is speed curve. Its theoretical value was .2 but we adjusted it to .3 to increase the response speed a bit. The second parameter is the pitch function. Its theoretical value was around 100 but we found it to be best at 120. The third parameter is angular velocity. Its theoretical value was also around 100 but we found it to be best at 150.

There are two additional parameters that were very important. The startup voltage when the SHUBot turns on is the first. This can range from 0V to 10V and is extremely important to getting the SHUBot initially stable. We experimented with values between 7V and 10V and found 9.5V to be the best. The second is waiting time, or how long the startup voltage should be applied. If it is too long the SHUBot will overshoot and if it's too short the SHUBot will undershoot trying to stabilize. We found a value of 250ms to be the best. With these values we were able to meet our objective of having the SHUBot remain stable for greater than 20 seconds. However as we added additional functionality to the SHUBot it had less cycles to devote to calculating values necessary for balancing and performance suffered a bit. While adding additional features we had to be very mindful of this, always ensuring that it could remain stable for greater than 20 seconds. Once stability was verified we moved onto movement.

Movement

One of our objectives was to ensure proper movement of the SHUBot. Once stabilization was achieved we focused on trying to get the SHUBot to move well. We found that the SHUBot could turn right and left well using the RF remote control. However it had no forwards and backwards movement capabilities. While looking into how to add these capabilities we discovered from discussions with Professor Tian that the robot suffered from mechanical design flaws. When the graduate students originally designed it they selected components such that the gear ratio was too great for the motors to control the wheels quickly. He said they attempted to get it to move properly but failed. Despite this news we still felt we should work on getting it to move forwards and backwards since it was one of our objectives.

We started out trying to get it to move by manually applying voltages to the wheels as it was balancing. We were somewhat successful in that we could make it move a short distance; however after this short distance it would ramp up in speed and lose balance. We tried adjusting the voltages based on timing so it would increase initially then decrease once it started moving. This worked slightly better but had the same issues. We came to the conclusion that it would be difficult to program due to the non-linear nature of controlling this movement and that hardware issues would have prevented it from ever working properly. Since we could not get it to move without eventually losing stability we decided it would be best to move on to meeting the other objectives.

Sensor Integration

One of our objectives was to make the SHUBot avoid obstacles. In order to meet this objective we needed to implement a sensor that could measure the distance of obstacles from the SHUBot. We had two different types of sensors to work with; infrared and ultrasonic. The infrared sensor, a Sharp GP2D12 optoelectronic device, was tested first as it was simpler to operate. It can be seen in Figure 12 and can measure distances from 10cm to 80cm away. We wired it to an oscilloscope and power supply and observed the resulting waveforms as we moved objects back and forth in front of it. It was fairly inaccurate, only showing a proper voltage when a highly reflective object was placed directly in front of it at short distances. Because of this we decided to test the ultrasonic sensor.

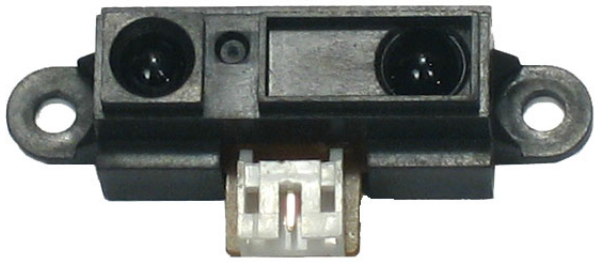


Figure 12: Sharp GP2D12 Optoelectronic Sensor

The ultrasonic sensor, an SRF05 ultrasonic ranger, is more accurate but also more complicated to operate than the infrared sensor. It can be seen in Figure 13 and can accurately measure distances between 2mm and 3m. It functions by sending it a trigger pulse longer than 10µs and measuring a response pulse that can range from 100µs to 25ms. It has two modes of operation depending on how it is wired; it can either use a single pin for the trigger pulse and response pulse or separate pins for each. Since none of the modules we had to work with would allow for a single port to be both an input and output, we opted to use separate pins for the trigger and response pulses.

Figure 13: SRF05 Ultrasonic Sensor

To initially test this sensor it was connected to a power supply, a square wave generator on the trigger pulse pin, and an oscilloscope on the response pulse pin. We were able to see the voltage waveforms fluctuate appropriately as objects were moved back and fourth in front of it. It was much more accurate than the infrared sensor and had a wider range of values it could detect so we opted to move forwards with this sensor.

In order to operate this sensor more accurately and integrate it into the SHUBot we decided to start developing the LabVIEW code for it as well as wiring it to a module. We decided to use the NI9401 Digital Input/Output module since it was the only module that we could both send a trigger pulse and measure the response pulse with. Wiring the sensor to this module was quite simple; the schematic can be seen in Figure 14. The sensor is connected to +5V power and ground, the COM port of the NI9401 module is connected to ground, the sensor output is connected to the DIO0 port and the sensor input is connected to the DIO4 port. The reason for using DIO0 and DIO4 is that DIO0-DIO3 and DIO4-DIO7 are grouped together as either inputs or outputs and cannot be selected as an input or output individually.

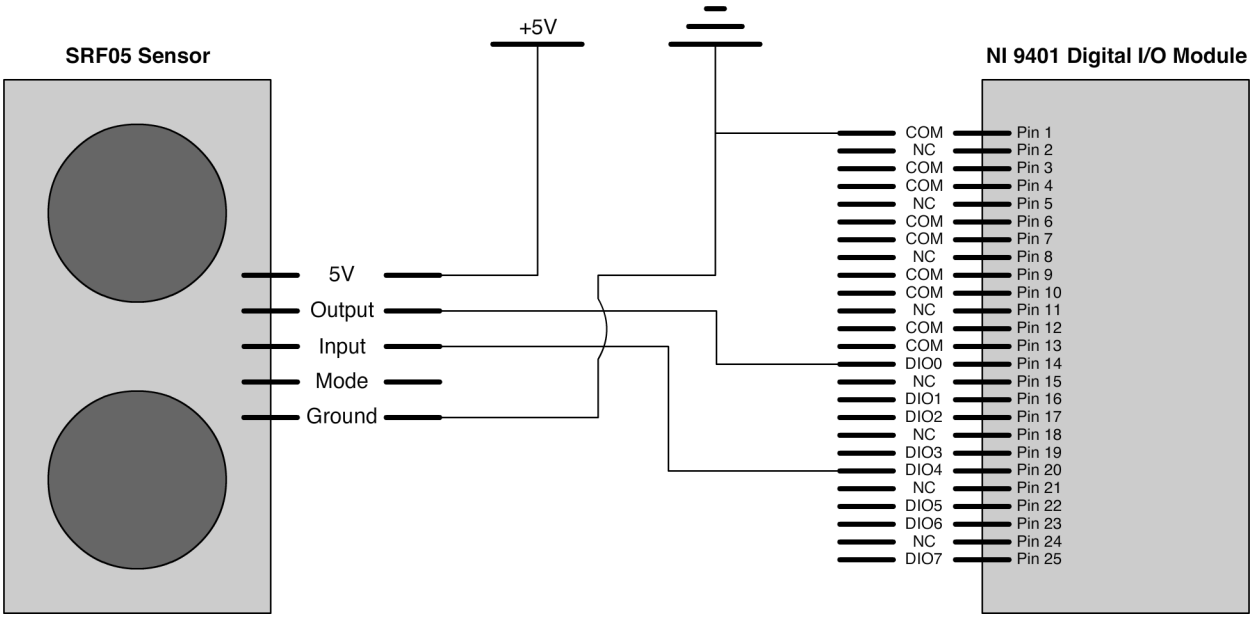


Figure 14: SRF05 Sensor to NI9401 Module Schematic

Once we successfully wired the sensor to the module we wanted to test it. Development on the sensor started before we had SHUBot up and running so we used a National Instruments cDAQ-9178 USB chassis with the NI9401 module for testing. A simple test program was written following the sensors operation protocols. In order to find the distance in centimeters the response pulse width in µs is divided by 58. This number was given in the technical documentation for the sensor and was calculated based on the speed of sound in µs. The block diagram for this program can be seen in Figure 15 and an enlarged version can be seen in the appendix. The graphical user interface for this program can be seen in Figure 16. A picture of the test setup can be seen in Figure 17.

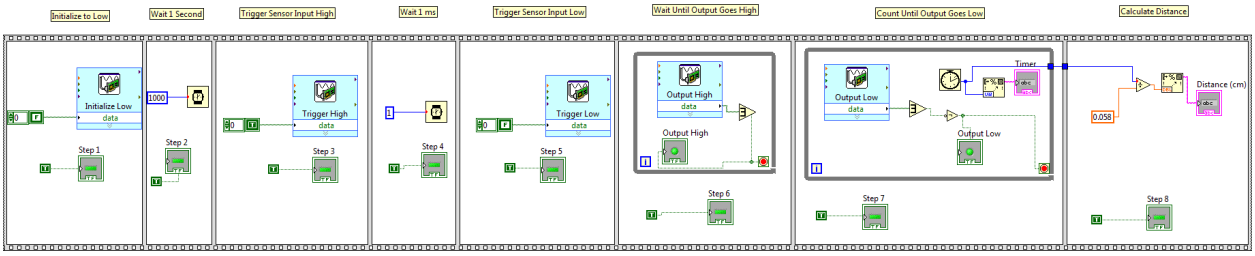


Figure 15: LabVIEW Block Diagram for SRF05 Sensor Testing

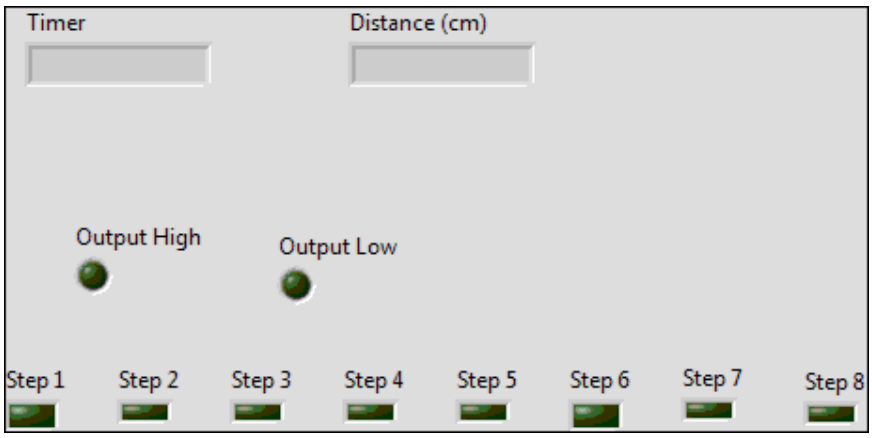


Figure 16: LabVIEW GUI for SRF05 Sensor Testing

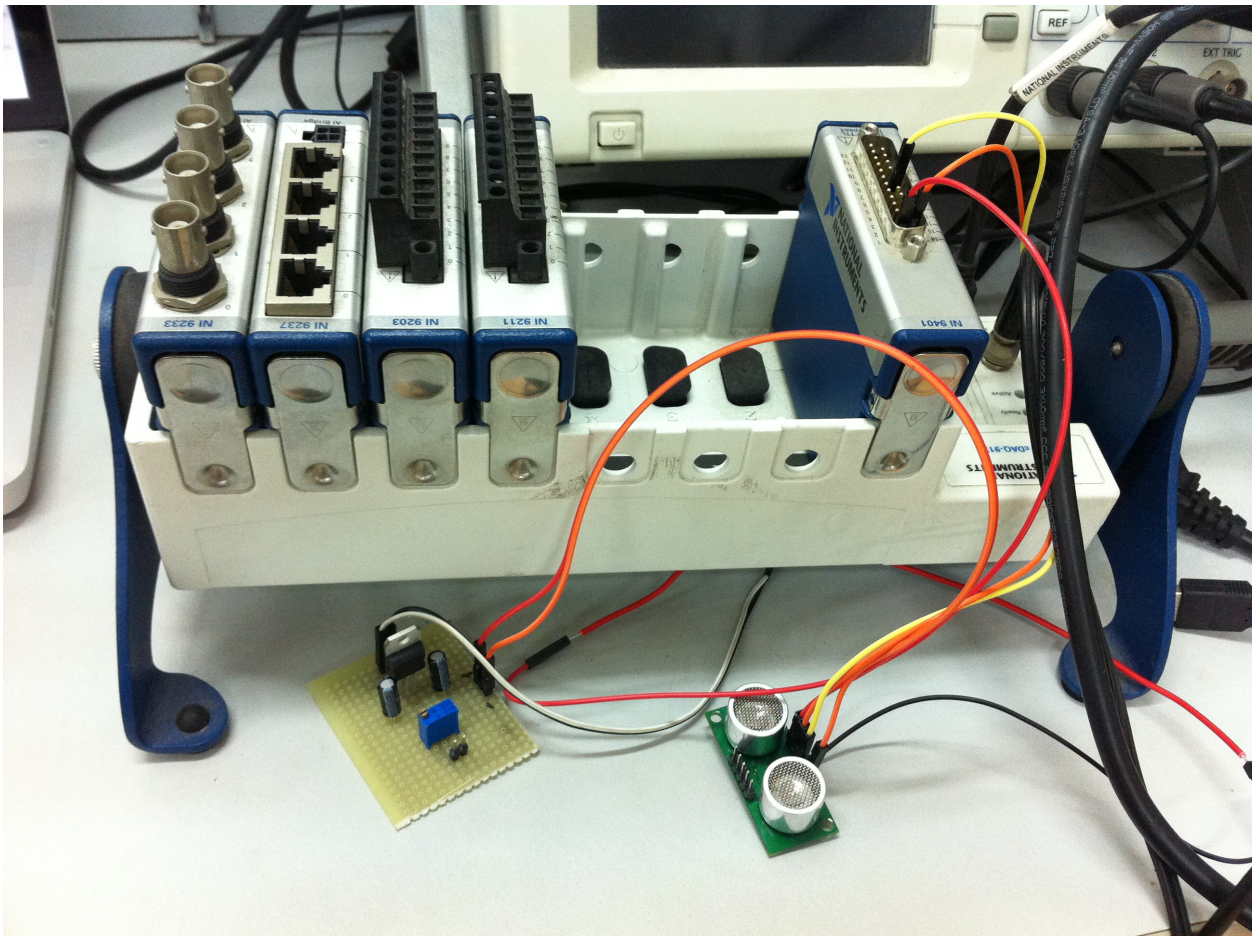


Figure 17: SRF05 Sensor Test Setup

The graphical user interface shows a timer readout, the distance in cm, the status of the trigger pulse with an output high and output low LED indicator, and LED indicators for each step in the block diagram to track where issues arose. After testing this program and trying to make it work it just wouldn't function properly. We discovered that using the cDAQ-9178 USB chassis was problematic as the maximum clock speed it could run at is 1MHz. The maximum resolution at this clock speed is 1ms and the response pulse is in µs so it just wasn't accurate enough to function properly. Once we managed to get the SHUBot up and running we then continued testing the sensor on the SHUBot, which uses the cRIO-9014 controller with much higher clock speeds.

Programming the sensor to work with the SHUBot took some time. The first approach we took was to use µs timers to measure the response pulse width but this required running the program in FPGA mode. However, when we attempted to run the program in FPGA mode errors would come up or compiling would take hours and never end. We tried installing different drivers and using different versions of LabVIEW but we were unable to get FPGA mode working after days of troubleshooting. We then looked into alternative approaches to measure the response pulse width and found that the module had a number of specialty modes. The input port on the module was configured to be a pulse width measurement counter on the high pulse. Using this mode we were able to successfully measure the response pulse width of the sensor, however the distance the sensor read was a bit "jumpy" and fluctuated. In order to resolve this a more sophisticated program was written that averaged multiple distance samples for a more accurate reading. The block diagram for this program can be seen in Figure 18 and an enlarged version can be seen in the appendix. The graphical user interface for this program can be seen in Figure 19.

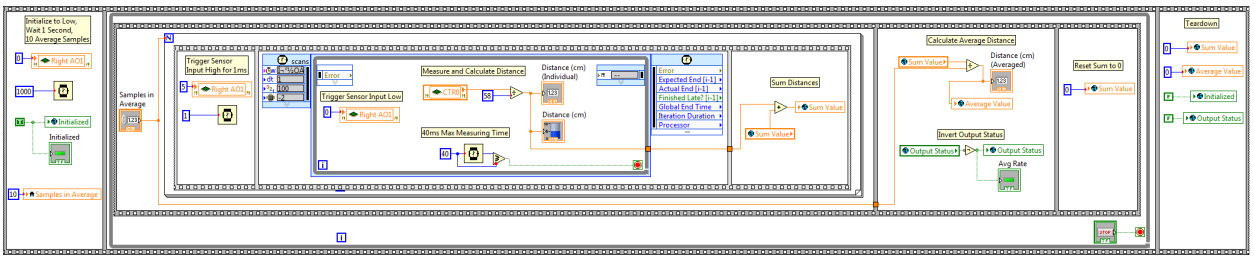


Figure 18: LabVIEW Block Diagram for SRF05 Sensor

Figure 19: LabVIEW GUI for SRF05 Sensor

This program runs in it's own while loop outside of the main SHUBot code. It starts by initializing the trigger output to low, waiting 1s, and setting the number of average samples to 10. Then the trigger output is set to high for 1ms before entering the scanning loop. Inside the scanning loop the trigger output is set to low, the loop is timed for 40ms to account for the longest distance the sensor can measure, and the distance is measured and calculated in cm as well as being graphed in the GUI. This distance is then summed in a

shared variable and the loop repeated as many times as the average samples is set to. Once it reaches that many iterations the average distance is calculated by dividing the summed distance by the average samples. A shared variable called output status is then inverted. This variables purpose is to visually show how long it takes for the average distance to be calculated as well as to be used by the LED indicators which will be described later in this report. After this step the sum is reset to 0 and the whole process repeated until the stop button is pressed. Once the stop button is pressed there is a teardown which resets all the variables to 0. This program allowed us to obtain distance readings properly and we were able to verify the distance readings were accurate with a measuring tape.

This program required minor changes after the addition of the sound controller and LED indicators. We needed additional outputs that the other modules could not provide so we decided to use another module, the NI9411 digital input module, to measure the response pulse width. This module was configured with the same specialty counter as the NI9401 module. Additionally we decided to use the NI9263 analog output module to send the trigger pulse. In software the only changes we had to make were changing the ports to the new modules. However the hardware required all new wiring to these new modules. The COM ports for both modules were tied to ground, the sensor input was connected to the A01 port of the NI9263, and the sensor output was connected to the DI0a port of the NI9411.The final sensor schematic can be seen in Figure 20.

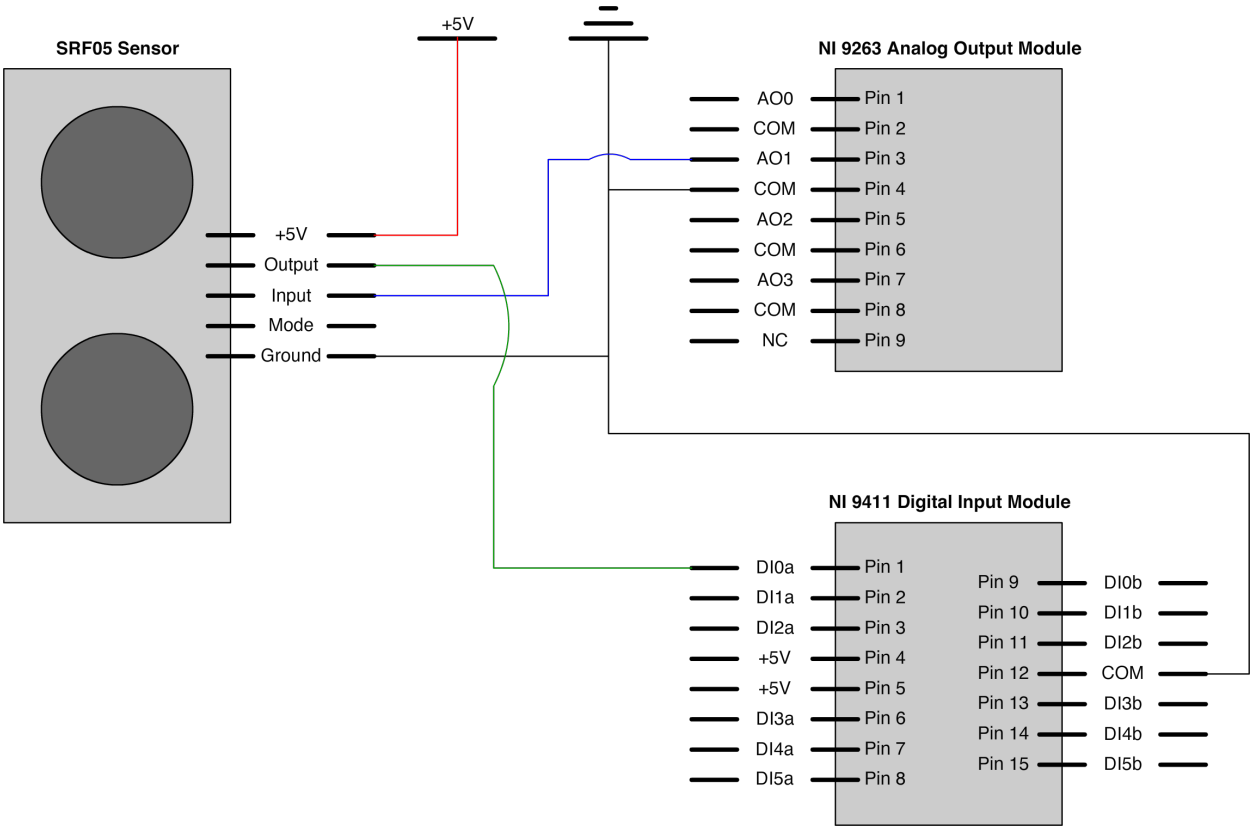


Figure 20: SRF05 Sensor Schematic

Once the sensor was functioning properly we needed to mount it to the SHUBot. To do this we designed a custom mount and modeled it in CAD using SolidWorks 2007. The technical drawing of the mount can be seen in Figure 21. Wang Pengchong, a graduate student helping us, manufactured the mount using traditional machinery out of aluminum while we assisted with measurements. The sensor was then secured to this mount using risers and the mount was affixed to the SHUBot. The completed sensor mounted on the SHUBot can be seen in Figure 22. After completing work on the sensor we moved onto integrating the LED indicators.

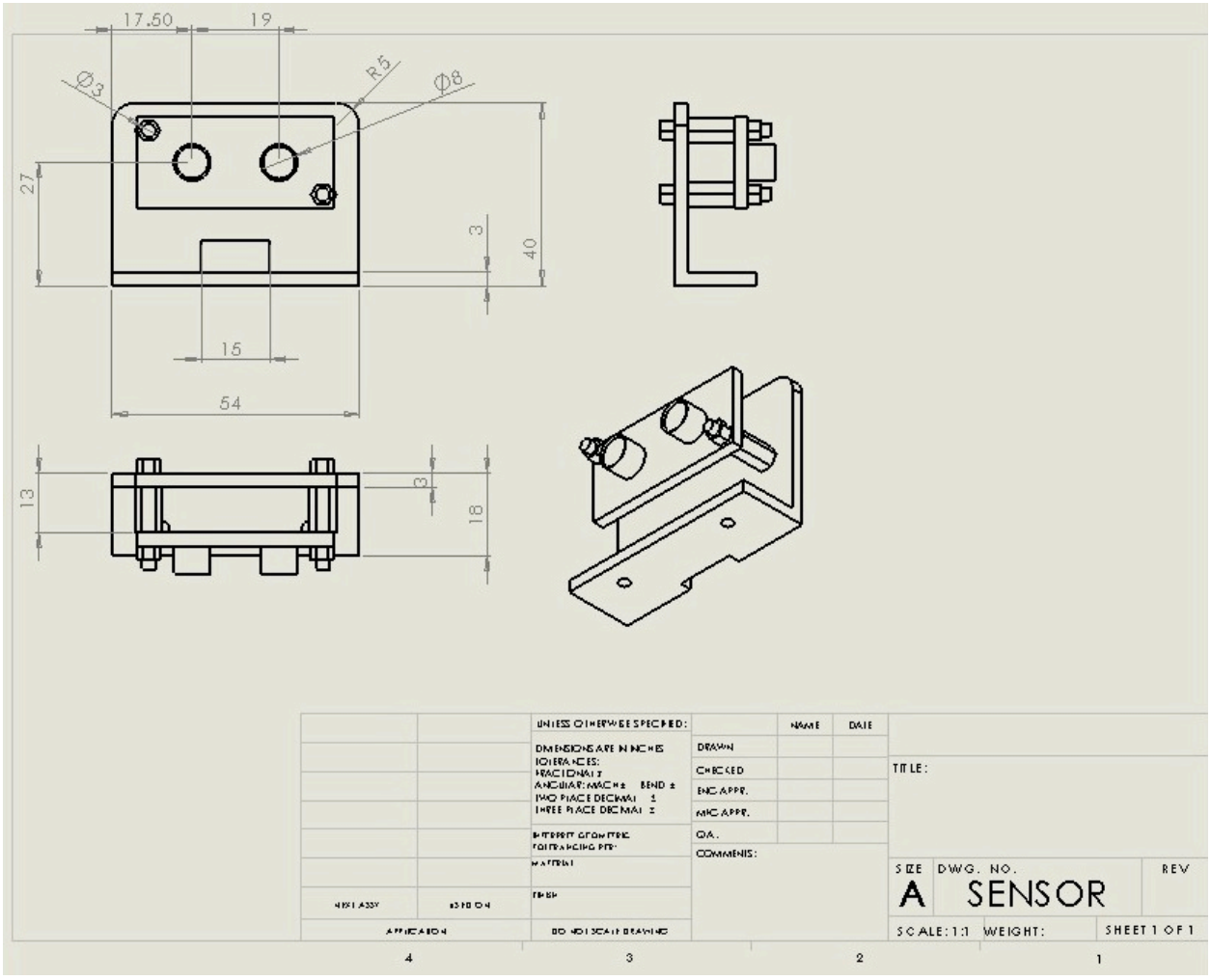


Figure 21: SRF05 Sensor Mount Technical Drawing

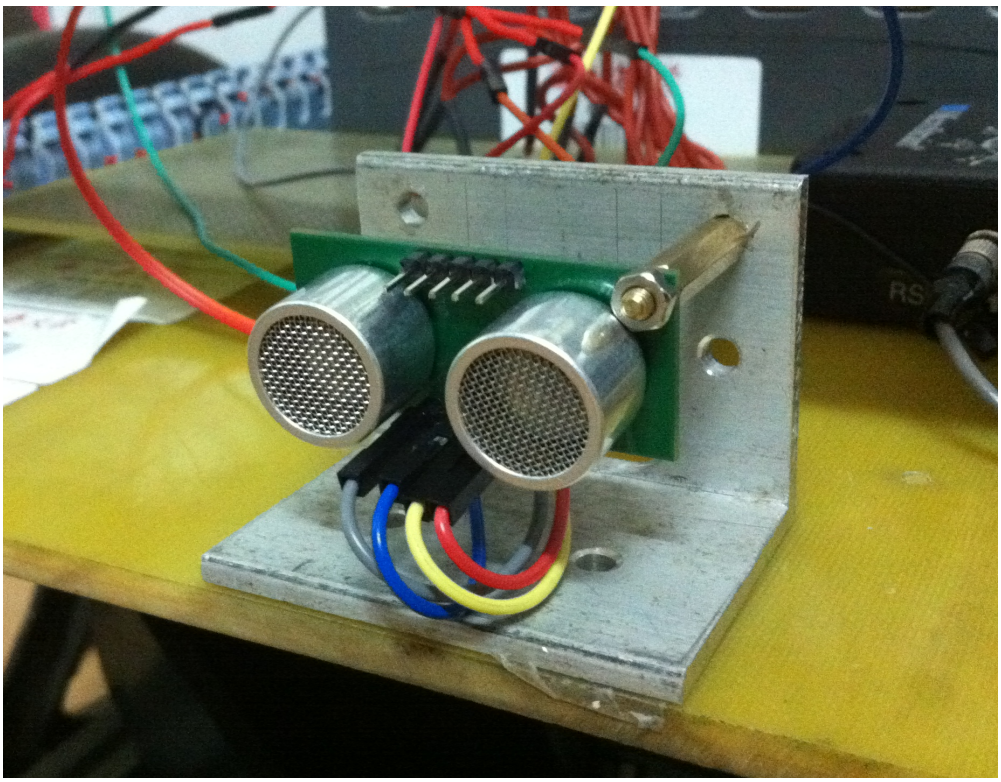


Figure 22: SRF05 Sensor Mounted on SHUBot

LED Integration

One of our objectives was to indicate the SHUBot's status using light and sound. In order to meet this objective we needed to integrate LED's into the SHUBot. We decided to use two tri-color LED modules; one in strip form to be wrapped around the SHUBot and one in "headlight" form to be used to simulate eyes. These tri-color LED modules have 3 inputs to control whether red, blue, green, or any combination of these diodes are illuminated. There are seven possible combinations of colors that can produce red, yellow, green, greenish-blue, blue, purple, and white light. These LED modules use a single 12V supply and a ground connection for each color that when connected allow them to illuminate.

To control the LED's we decided to use the NI9401 digital input/output module. Unlike the sensor, the LED's could not be wired directly to the module as it uses 12V power instead of 5V. We decided to use relays to control the LED's and found some OMRON G6B-1114P subminiature PCB relays in our lab to use. We soldered these relays to a stripboard and connected the LED modules and NI9401 module to them as seen in the schematic in Figure 23. A test program was written in LabVIEW to control each LED color individually using simple boolean switches on the control ports. The block diagram for this program can be seen in Figure 24. The graphical user interface can be seen in Figure 25.

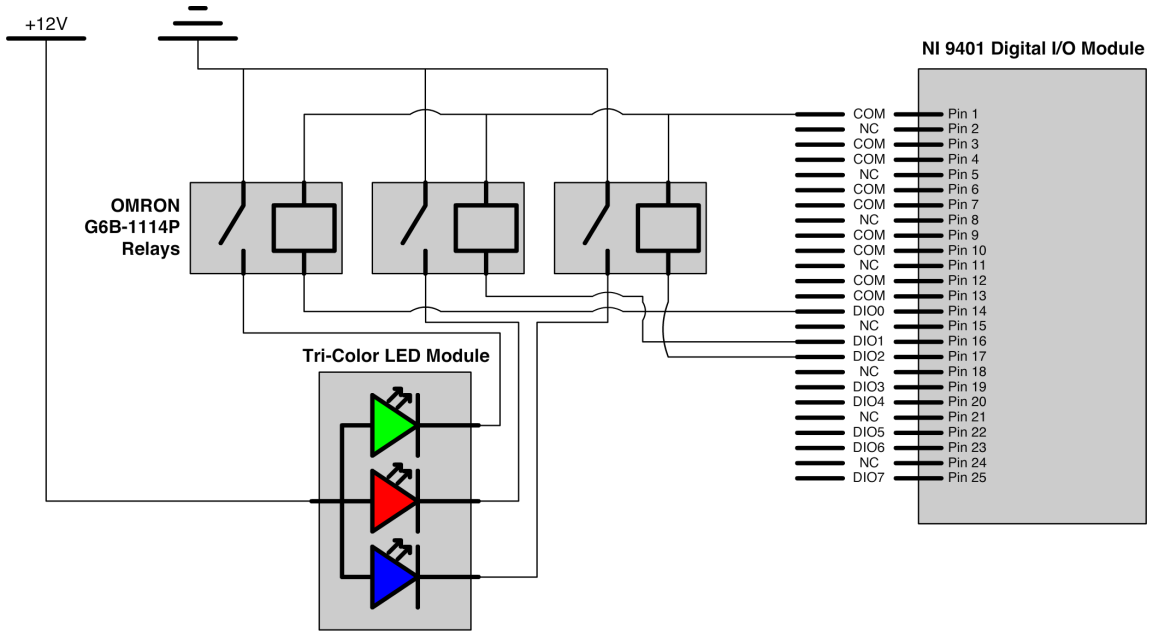


Figure 23: Tri-Color LED Module Schematic

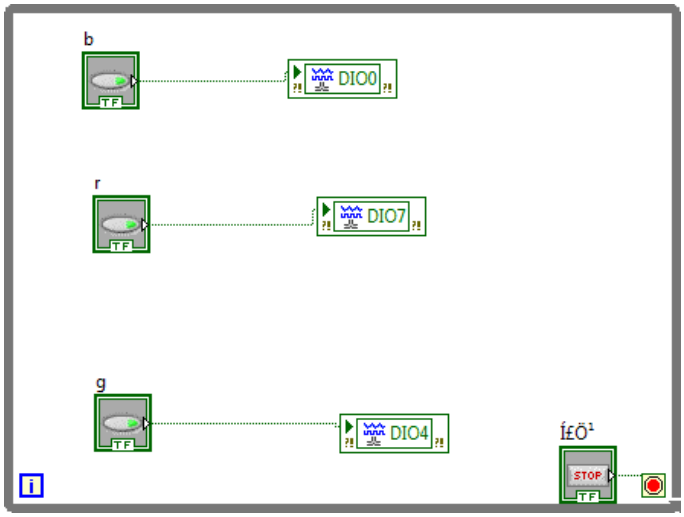


Figure 24: LabVIEW Block Diagram for Tri-Color LED Module Testing

Figure 25: LabVIEW GUI for Tri-Color LED Module Testing

Upon testing the LED's with this hardware configuration we encountered an issue where the LED's would only illuminate some of the time. Additionally only two colors could be illuminated at once with them all turning off when attempting to illuminate all three. After some time troubleshooting trying to figure out the reasons for this I realized that the NI9401 module was not providing enough current to drive the relays properly. Our initial response was to find smaller relays that needed less current but these were difficult to find. We bought the smallest relays we could find at an electronics market, Songle SRD-SLC's, but they still required more current than could be provided, about 2mA with a 4.3V draw.

In order to increase the current to drive the relays we created a simple driver circuit using an SS9013 NPN transistor. This transistor is connected to the NI9401 module at the base with a 1kΩ resistor being used to set the current driving the transistor to 5mA. The transistor emitter is connected to ground and the collector is connected to the 5V supply. A diode is connected across the relay coil to prevent damage to the transistor due to a back electromagnetic field pulse caused by the relays inductance when it is switched off. The schematic for this updated circuit can be seen in Figure 26.

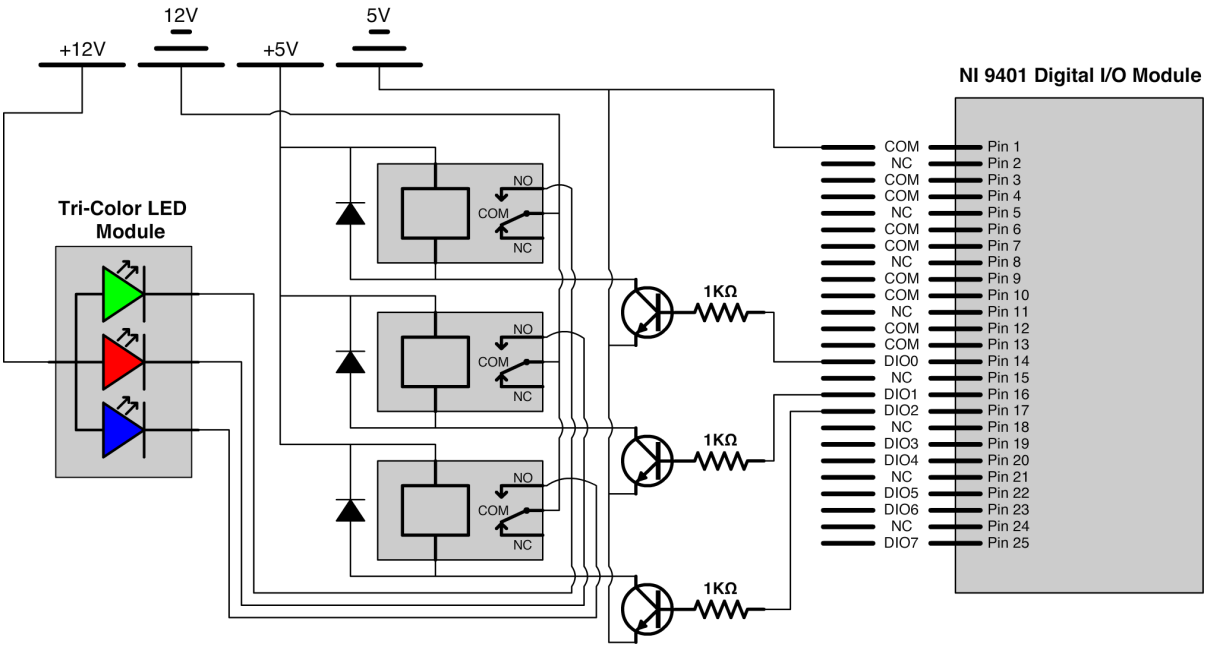


Figure 26: Tri-Color LED Module Updated Schematic

This hardware configuration worked flawlessly and we moved on to integrating it into the main SHUBot program. We decided the LED's should have both a manual and automatic mode of operation. The manual mode was implemented using a color slider with integers representing each possible color. The automatic mode was implemented using the average distance measurements from the sensor. If an object was greater than 80cm away the LED's would illuminate green. If an object was between 40cm and 80cm away the LED's would illuminate yellow. If an object was less than 40cm away the LED's would illuminate red. These settings provide good visual feedback as to how close the SHUBot is to an obstacle. Additionally the LED's were programmed to strobe at the rate the average distance is being calculated which could be turned on or off with a button. The LED's were initialized to off in the first frame of the program and tore down to off in the last frame, while the main LED program resided in the main SHUBot operational loop. The initialization block diagram can be seen in Figure 27. The main block diagram can be seen in Figure 28. The teardown block diagram can be seen in Figure 29. The graphical user interface for the tri-color LED module operation can be seen in Figure 30. This GUI includes the color slider for manual color selection, buttons to turn on manual mode or strobe functionality, and digital LED indicators to show which colors are illuminated in the software. After completing work on the LED indicators we moved onto the sound integration.

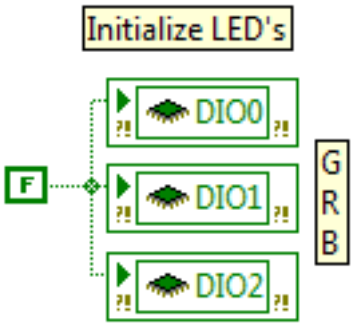


Figure 27: LabVIEW Block Diagram for Tri-Color LED Module Initialization

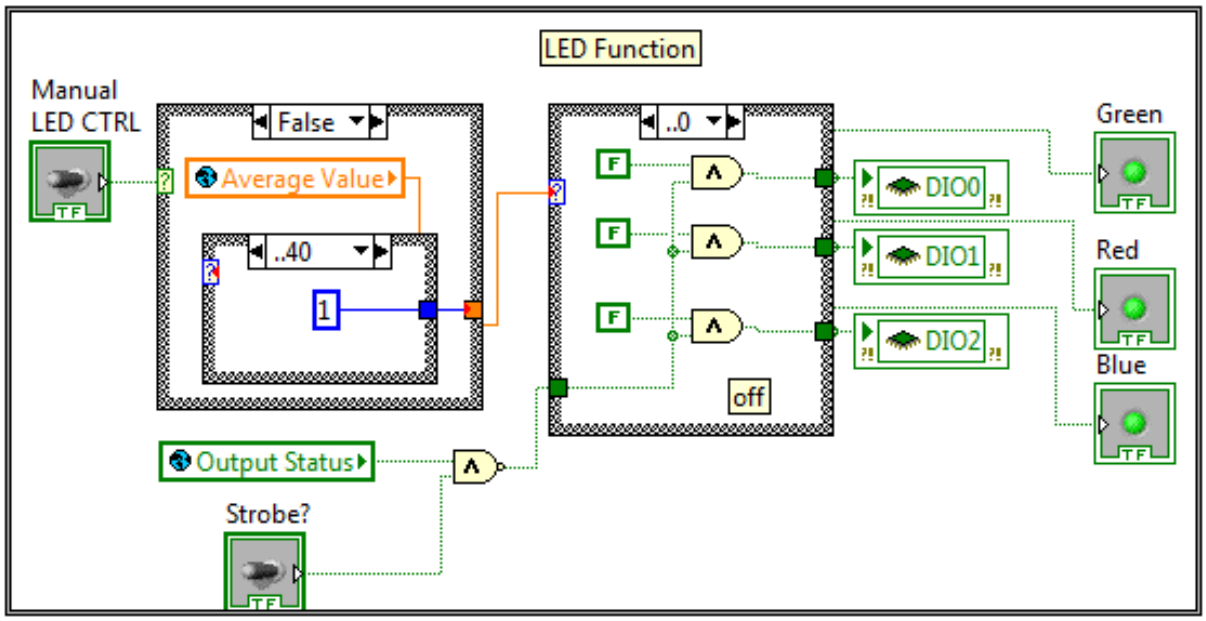


Figure 28: LabVIEW Block Diagram for Tri-Color LED Module Operation

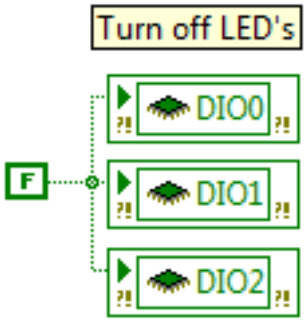


Figure 29: LabVIEW Block Diagram for Tri-Color LED Module Teardown

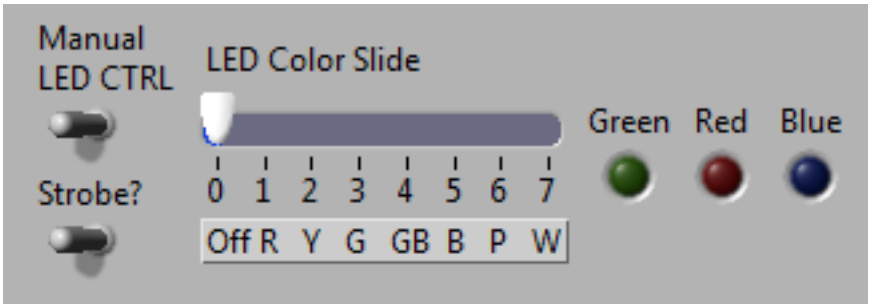


Figure 30: LabVIEW GUI for Tri-Color LED Module Operation

Sound Integration

One of our objectives was to indicate the SHUBot's status using light and sound. In order to meet this objective we needed to integrate a sound controller and speakers into the SHUBot as well as generate the audio files that the SHUBot will "speak." We first decided look for a sound controller that we would be able to most easily integrate and operate in the SHUBot. Our requirements were to use a minimal number of ports and have all processing done by the sound controller. After searching for a while we came across a Chinese controller that would meet our requirements, the Tao-line TXSDMP3220 MP3 controller. This controller can be seen in Figure 31.

Figure 31: TXDSDMP3220 Sound Controller

This controller plays audio files in a variety of formats off an SD card. It has 9 different modes of operation including serial port control, infrared remote control, timer based control, binary decoding control, direct selection control, and automatic playback mode. The two that were most suitable for our application were binary decoding control and direct selection control. Binary decoding allows for 2^n+1 number of audio files to be played where n is the number of ports and must be less than or equal to 8. Direct selection

allows for n number of audio files to be played back where n is the number of ports and must be less than or equal to 8. We opted to use direct selection control as we did not have too many things we wanted SHUBot to say and it was simpler to program for, not requiring a data latch signal like the binary decoding control. It operates by sending a low pulse greater than 10ms to a port to play that ports corresponding audio file.

While we waited for this controller to arrive in the mail we created the audio files it would play. We decided to have a welcome sound when the SHUBot turns on, a goodbye sound when the SHUBot turns off, an alert when an obstacle is detected, and an alert when the SHUBot is actively avoiding an obstacle. The scripts and file names for these sounds can be seen in the table below.

001.mp3	Hello. My name is SHUBot. I am a two-wheeled self balancing robot created by Shanghai University and Worcester Polytechnic Institute students.
002.mp3	Farewell.
003.mp3	Obstacle Ahead.
004.mp3	Avoiding Obstacle.

These audio files were created using three programs on Mac OSX 10.7; the command terminal, XLD audio converter, and Audacity audio editor. The files were initially generated in the terminal using the command "say.' For example, the Farewell file was created by inputting "say -o 002.aif Farewell. extra." The –o modifier says to write the audio to a file 002.aif while the "extra" is text spoken so that the Farewell does not get cut off. This was a problem initially when testing this command, as it would say "Farewell" without the extra spoken text. Once this file was generated it was edited with Audacity to convert the "extra" to silence and saved to a lossless WAV format to prevent audio degradation due to repeated compression. This WAV file was then converted to its final form using XLD audio converter, a 256kbps CBR MP3 file.

The SD card needed to be configured properly for the controller to be able to read the files. It was formatted using the FAT16 file system then a folder called "Config" was

created at the root level. Inside this folder a standard text file called “Config.txt” was created containing the operating mode number. Since we chose the direct selection mode the number 6 was used. The audio files were then placed in a folder called “Music” at the root level and titled 001.mp3, 002.mp3, 003.mp3, and 004.mp3. These correspond to ports 0, 1, 2, and 3 respectively. Once the SD card was properly configured with the audio files and the controller received we began testing it.

The first controller we received underwent some initial testing and in the course of this testing was damaged. A graduate student searching for a suitable power supply to test it accidentally plugged in a laptop power supply with a much greater voltage than it could handle, essentially frying it. A second identical controller was ordered and was found to be mostly functional, only having one issue. Whenever it was turned on it had to be manually reset before it would function. In order to solve this issue we decided to take advantage of an MCU reset pin on the board, which resets the module if a low pulse greater than 10ms triggers it. Once we established that this board was functional we worked on integrating it into the SHUBot.

To communicate with this controller we decided to use the remaining five output ports on the NI9401 digital input/output module. Since this controller was active low (all ports must be high unless being triggered), and the module does not provide much driving current, we opted to use relays to control the controller. The same relays and relay driver circuits that we used with the LED’s were used except the power supplied was 5V instead of 12V. The schematic for the sound controller circuit can be seen in Figure 32. The NI901 outputs were connected to the GPIO0-GPIO3 audio control ports and the MCU reset port. This allowed us to play back the four audio files created as well as reset the controller from the software. Some inexpensive USB powered speakers were connected to the sound controllers standard 3.5mm audio output. A female USB jack was soldered to the stripboard and connected to the 5V supply to power these speakers. These speakers were then mounted on the top of the SHUBot using strong double-sided tape.

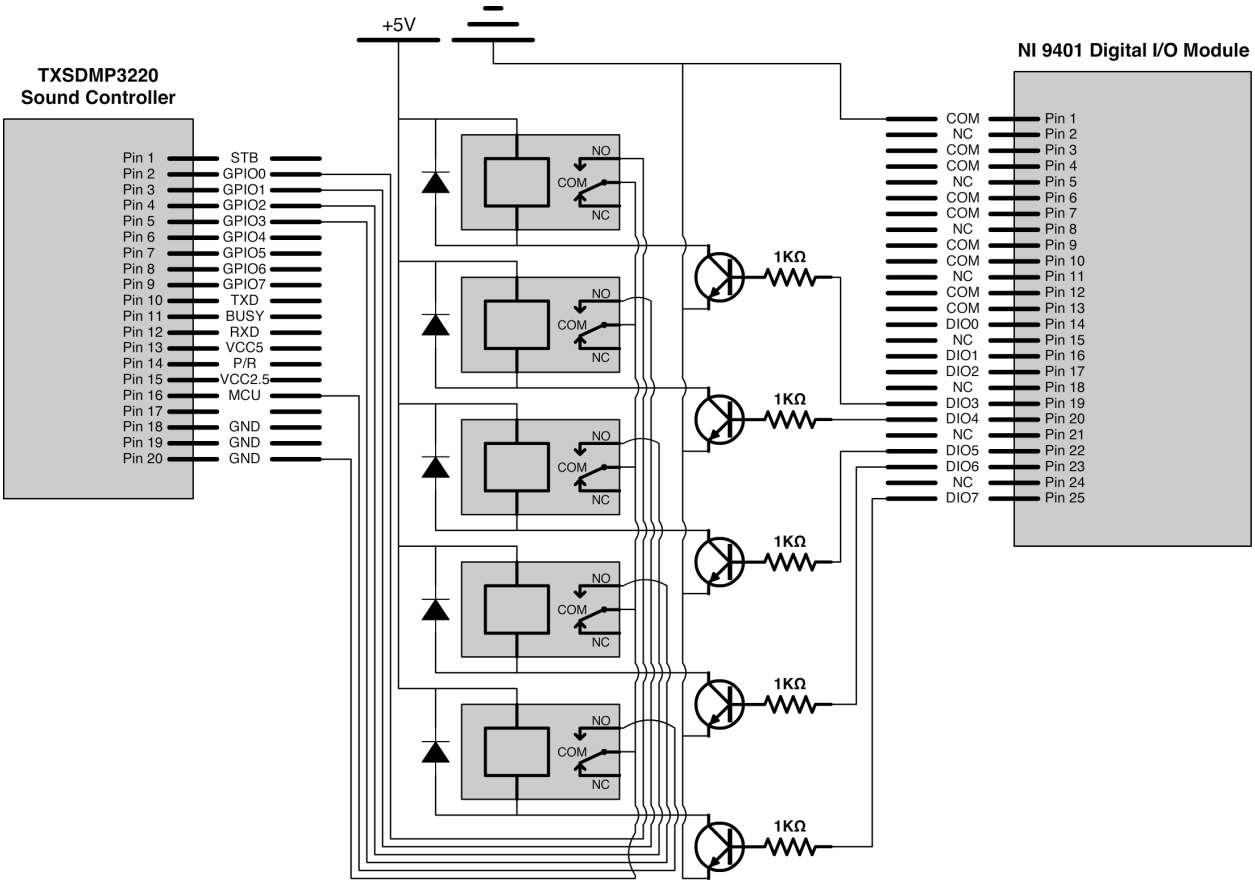


Figure 32: TXDSDMP3220 Sound Controller Schematic

Once the hardware was working and the controller configured we moved onto integrating the sound functionality into the main SHUBot code. The program starts out by initializing all the sound ports and a sound boolean variable to false. It also initializes a sound introduction variable to true. These variables are then utilized in the main SHUBot while loop to control which sounds are played. A simple case structure is used to determine if sounds should play or not based on the sound boolean variable. If it is false, another case structure is entered based on the sound introduction variable. If this variable is true a 15ms pulse is sent to trigger the introductory audio file and then the sound introduction variable set to false so it does not play again. If the sound boolean variable is true, the case structure contains both the sound introduction code as well as code for both the obstacle avoiding sounds and the sound controller reset functionality. The obstacle avoiding reset

functionality is quite simple, with a boolean button being placed in the graphical user interface directly controlling the port attached to the MCU reset on the controller. The obstacle avoiding sound functionality is a bit complex in that it runs off a counter to control how often phrases are said. If obstacle avoidance is turned on and the counter reaches 30 cycles, “obstacle ahead” is played if there is an obstacle detected between 40cm and 80cm away. If an obstacle is detected less than 40cm away, “obstacle avoided” is played. If no obstacle closer than 80cm is detected nothing is said. Every time this case structure is entered sound count is reset back to 0. Once the SHUBot is shut down, a teardown structure is entered that sends a pulse to play the phrase “farewell.”

The initialization block diagram can be seen in Figure 33. The main block diagram can be seen in Figure 34. The teardown block diagram can be seen in Figure 35. The graphical user interface for the sound controller operation can be seen in Figure 36. This GUI contains a button to turn the sound on or off as well as to reset the sound controller module. The sound controller was mounted on the stripboard the relay circuits were soldered onto and can be seen in Figure 37. After completing work on the sound integration we moved onto making the SHUBot avoid obstacles.

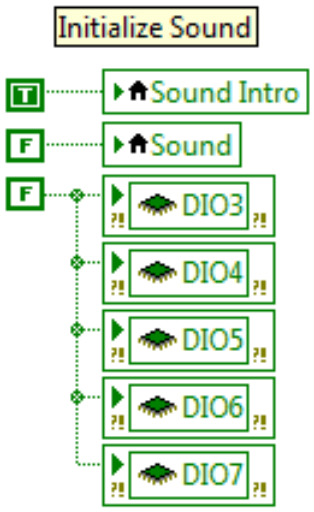


Figure 33: LabVIEW Block Diagram for TXDSDMP3220 Sound Controller Initialization

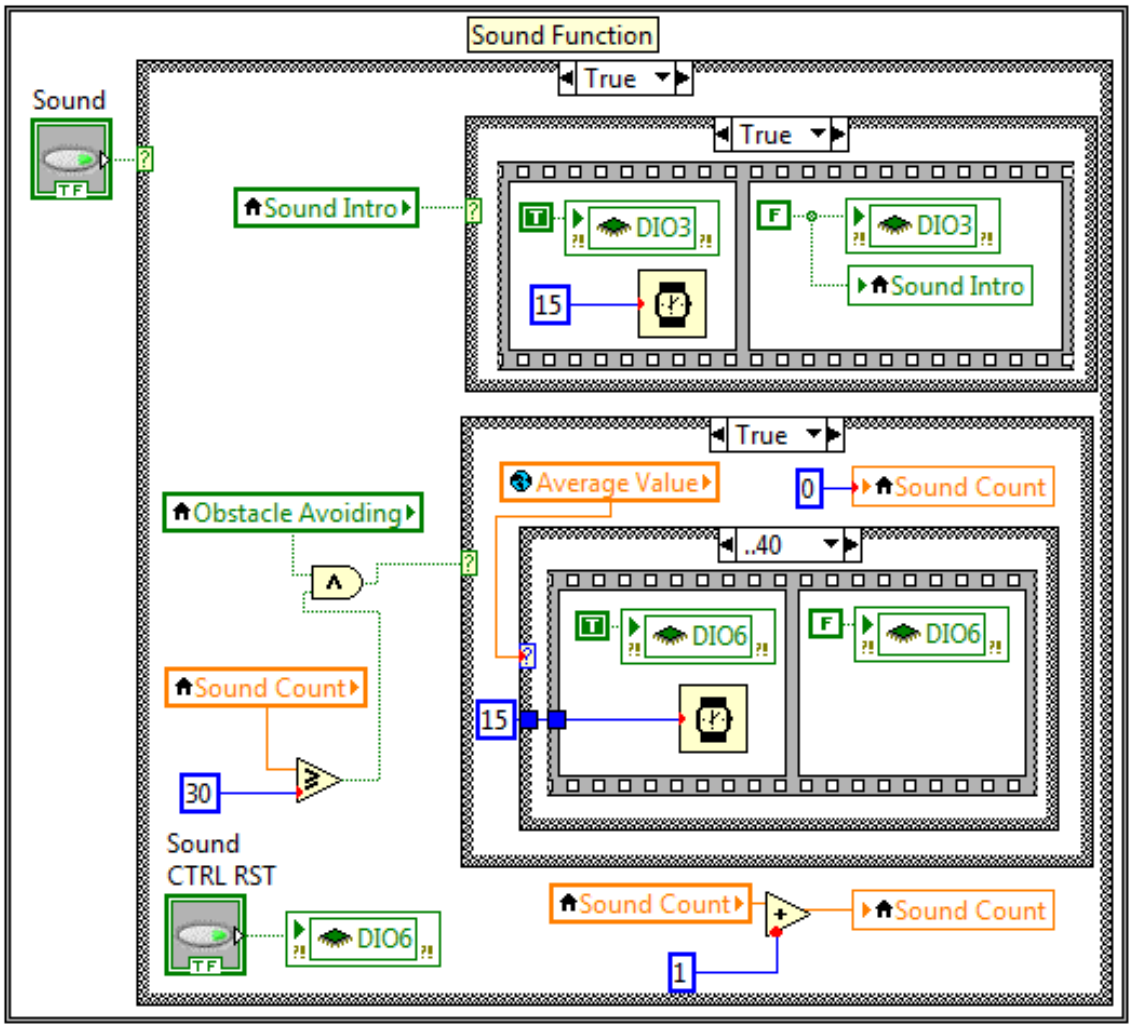


Figure 34: LabVIEW Block Diagram for TXDSDMP3220 Sound Controller Operation

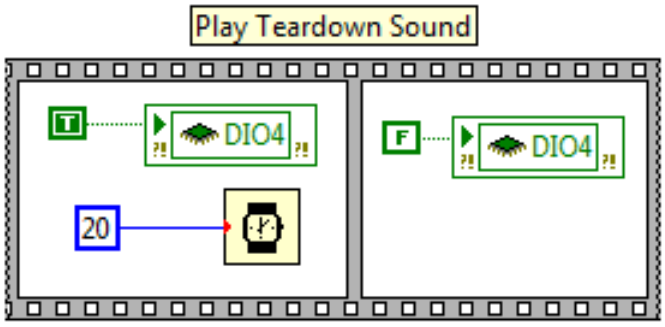


Figure 35: LabVIEW Block Diagram for TXDSDMP3220 Sound Controller Teardown

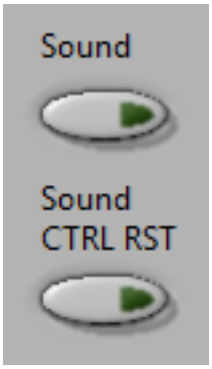


Figure 36: LabVIEW GUI for TXDSDMP3220 Sound Controller Operation

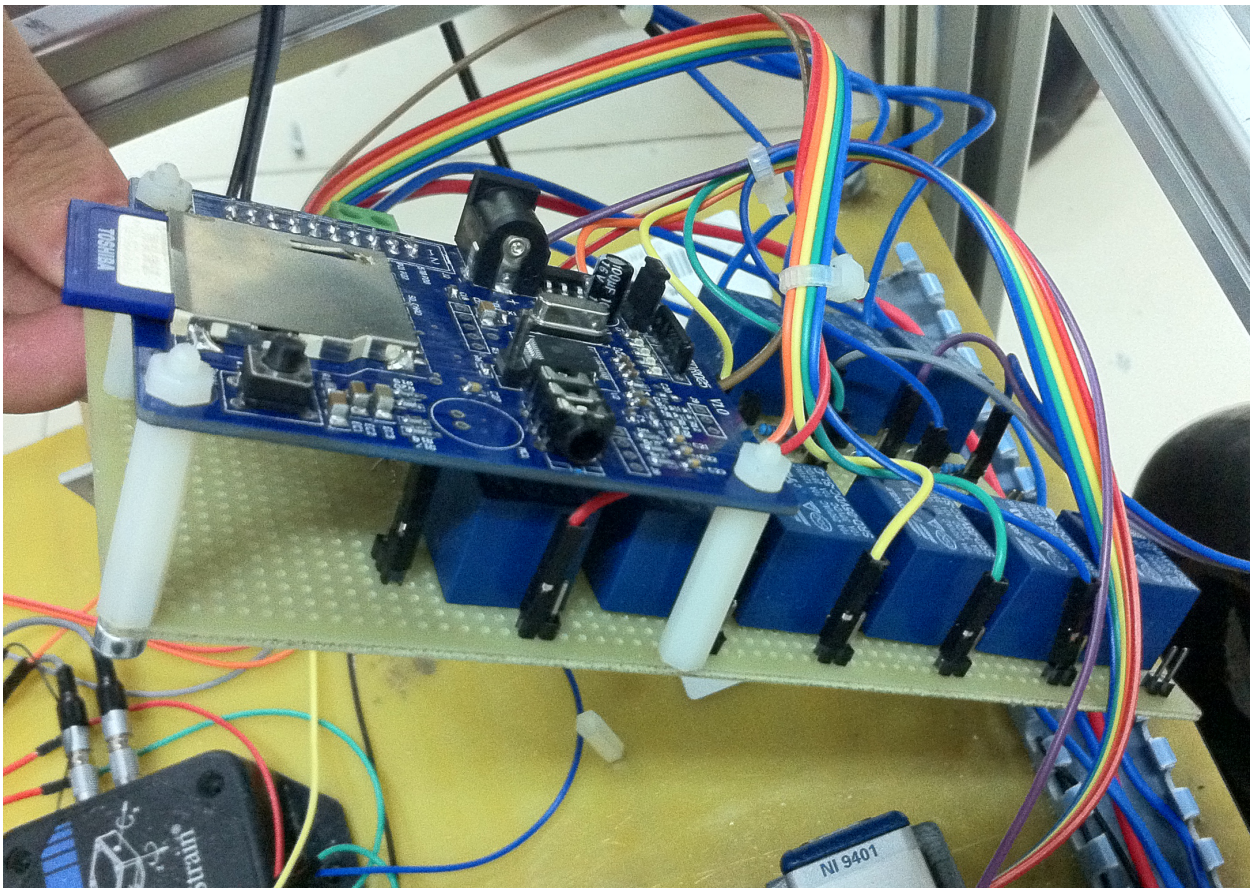


Figure 37: Completed Relay and Sound Controller Circuit

Obstacle Avoidance

One of our objectives was to make the SHUBot avoid obstacles. In order to meet this objective we implemented a sensor to measure the distance of obstacles from the SHUBot. Once this information was available we were able to go ahead and make the SHUBot obstacle avoiding. One of the main issues with SHUBot is its inability to move forwards and backwards. Since we could not get this working and still wanted to meet our obstacle avoidance objective we decided the best thing we could do was have the SHUBot turn away from any obstacles it encounters. This was the easiest of all functionality we had to implement as it required no hardware additions and was simple to program in software. It is programmed to turn right if the obstacle avoiding button is turned on and the average distance of an obstacle is less than 40cm away. It turns right by imitating a remote control button press to turn right, writing true to the same variable. The block diagram for the obstacle avoiding operation can be seen in Figure 38. The graphical user interface for the obstacle avoiding operation can be seen in Figure 39. Once the SHUBot could avoid obstacles we moved onto integrating everything into a single program and GUI.

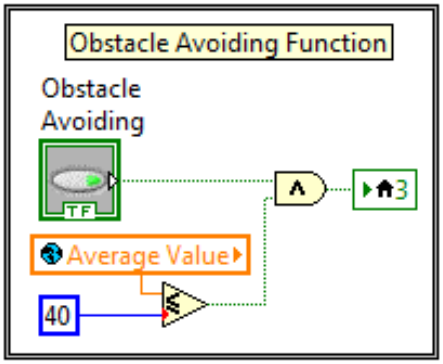


Figure 38: LabVIEW Block Diagram for Obstacle Avoiding Operation

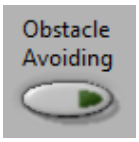


Figure 39: LabVIEW GUI for Obstacle Avoiding Operation

Complete Functionality

Once all the functionality we developed was completed we needed to integrate it all into cleanly into the main SHUBot program and graphical user interface. All of the GUI elements from the functionality we developed were assembled into a single block inside the main GUI. This block can be seen in Figure 40.

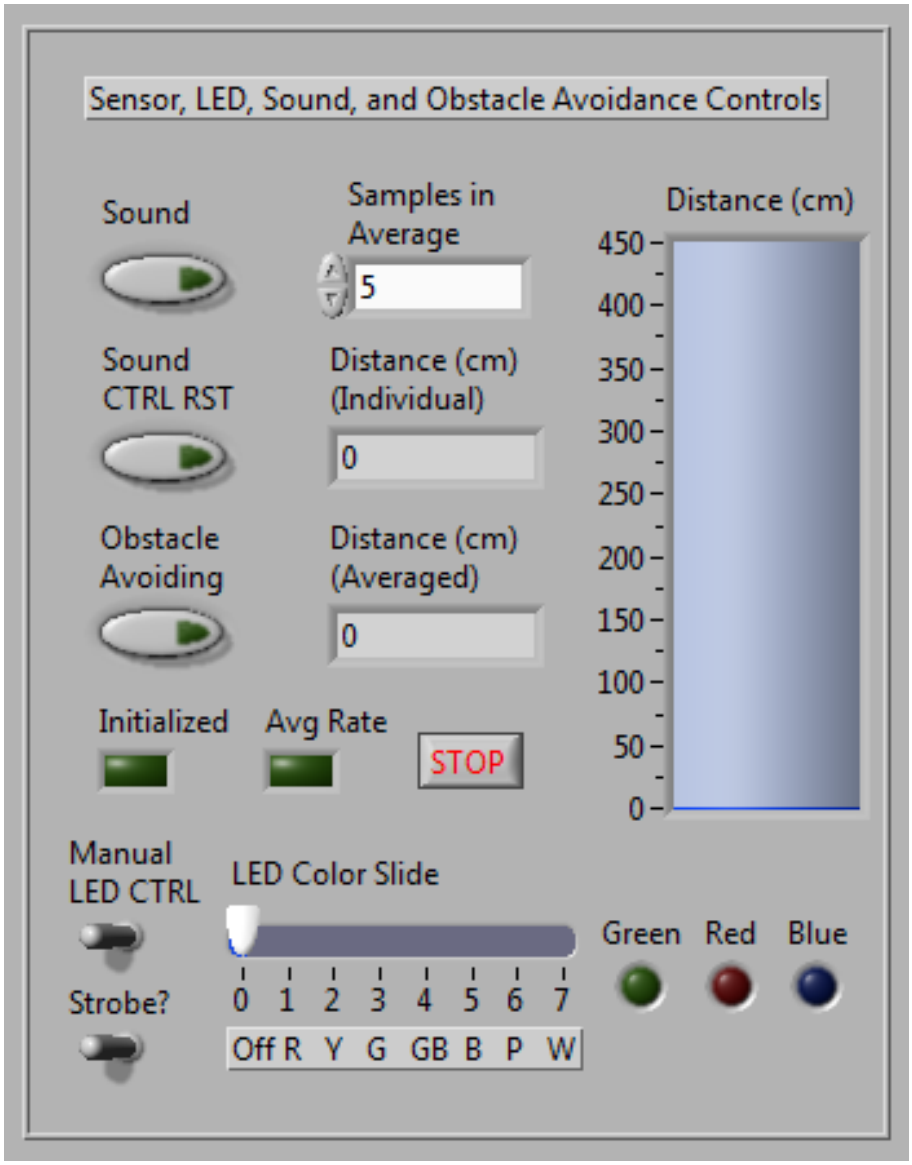


Figure 40: LabVIEW GUI for All Developed Functionality

This GUI was designed to fit into the overall SHUBot GUI easily without obstructing any existing features. It sits in between the movement controls and graphical data and can be seen in Figure 41.

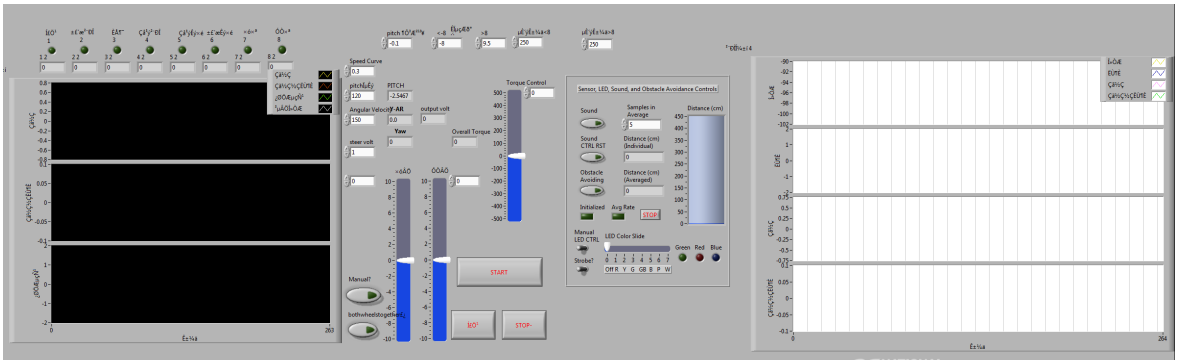


Figure 41: Complete SHUBot LabVIEW GUI

While the GUI was most important to design clearly for usability, the block diagram was also organized to be well defined and modular. The complete block diagram for the entire SHUBot program can be seen in Figure 42. The LED block is labeled 1, the obstacle avoiding block 2, the sound block 3, and the sensor block 4. By laying things out clearly, including the initialization and teardown frames, the program is easier to understand and modify.

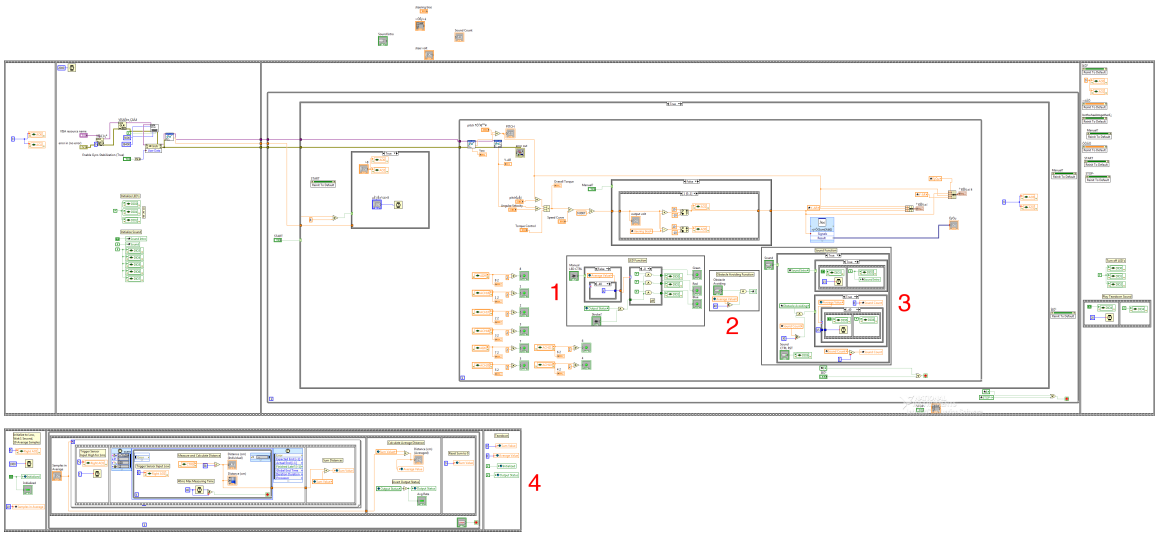


Figure 42: Complete SHUBot LabVIEW Block Diagram

Future Work

There are many future possibilities for this project should others continue working on it. The SHUBot could be redesigned for wireless operation. This would necessitate either changing hardware platforms or redesigning the SHUBot to incorporate a host computer running LabVIEW. It would also necessitate the addition of a battery to provide power without being tied to AC power.

Another area in which SHUBot could be improved is forwards and backwards movement. Due to its current mechanical components it is not capable of moving and balancing at the same time. Redesigning the wheel and motor structures to be able to react more quickly as well as using the motor encoder information would allow this to be possible.

One of the biggest problems we encountered while coding was a loss in performance as we added additional features. The way the code is laid out, commands are not executed based on any priority and are fairly unstructured in there order. It would be of significant benefit to restructure the code for priority based execution to improve performance. This would allow additional features to be added without taking away from the cycles necessary to keep SHUBot balancing properly.

Finally the SHUBot could use some additional remote control functionality. We only used it for turning and emergency stop functions, however it endless possibilities. Should the SHUBot be redesigned for wireless operation this work would be incredibly useful.

Conclusion

Overall this project was a success. A timeline of the tasks we accomplished can be seen in Figure 43. We started out working with the GBOT1001, a robot that had too many issues to function properly. We then switched to the SHUBot robot and were able to complete nearly all of our objectives. Existing control algorithms were researched and simulated to explore the best algorithms for balancing. The SHUBot was adjusted so that it could remain stable for greater than 20 seconds. Proper turning movement of the SHUBot was ensured, although we were unable to make it move forwards and backwards due to mechanical design issues. The SHUBot was able to detect and avoid obstacles based on a sensor we integrated into it. Light and sound indicators were implemented to visually and audibly indicate the SHUBot's status. And finally, the SHUBot's assembly was demonstrated visually through a three dimensional CAD animation. A CAD rendering of the SHUBot can be seen in the appendix. By adding all these additional functions and completing research we were able to contribute to the SHUBot's future usability as a two wheeled self balancing robot. We hope that it remains a focus of future projects so that it can be improved upon even further.

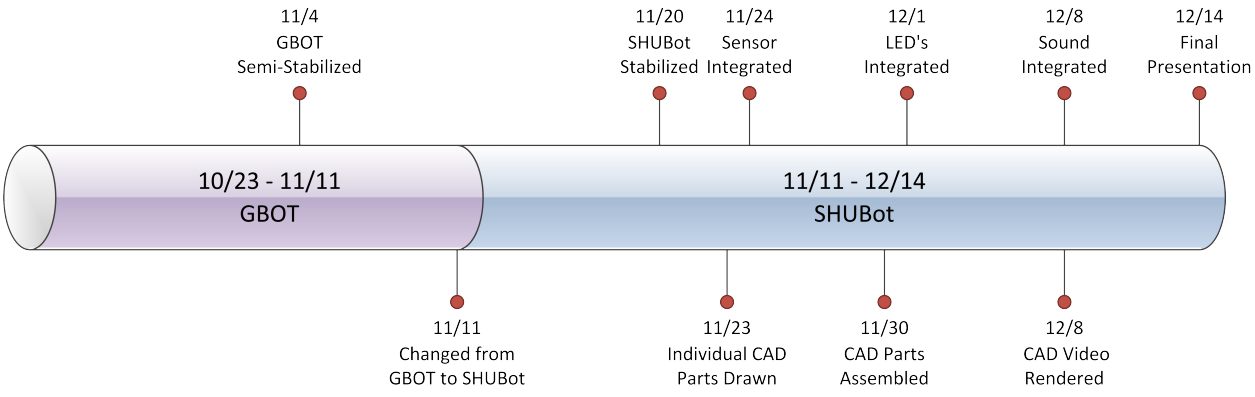


Figure 43: Timeline of Completed Tasks

Bibliography

Anderson, D. P. (2010). *nBot Balancing Robot*. Retrieved 2011, from Southern Methodist University: http://www.geology.smu.edu/~dpa-www/robo/nbot/

Grasser, F., D'Arrigo, A., Colombi, S., & Rufer, A. *JOE: A Mobile, Inverted Pendulum.* Swiss Federal Institute of Technology Lausanne, Laboratory of Industrial Electronics.

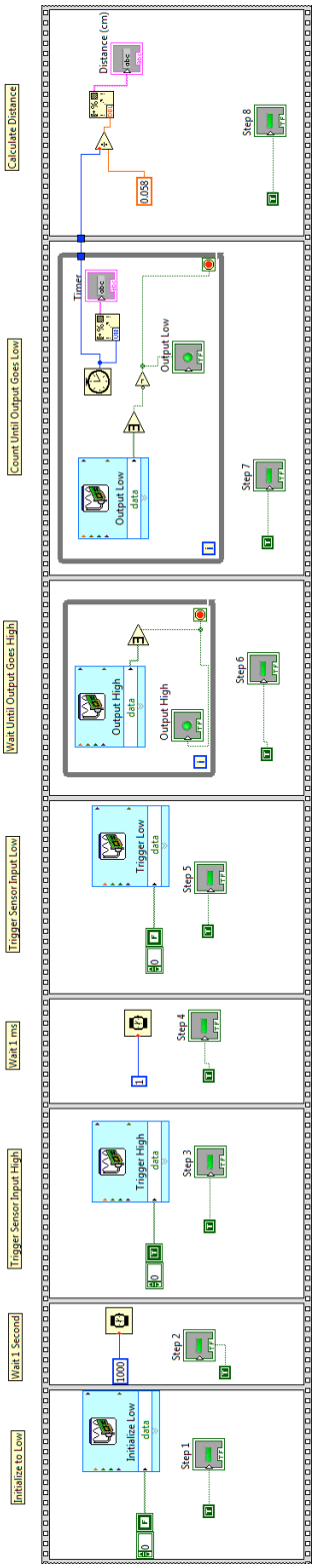
Hassenplug, S. (2002). *Steve's LegWay*. Retrieved 2011, from Team Hassenplug: http://www.teamhassenplug.org/robots/legway/

Kamen, D. (2011). Retrieved 2011, from Segway: http://www.segway.com/

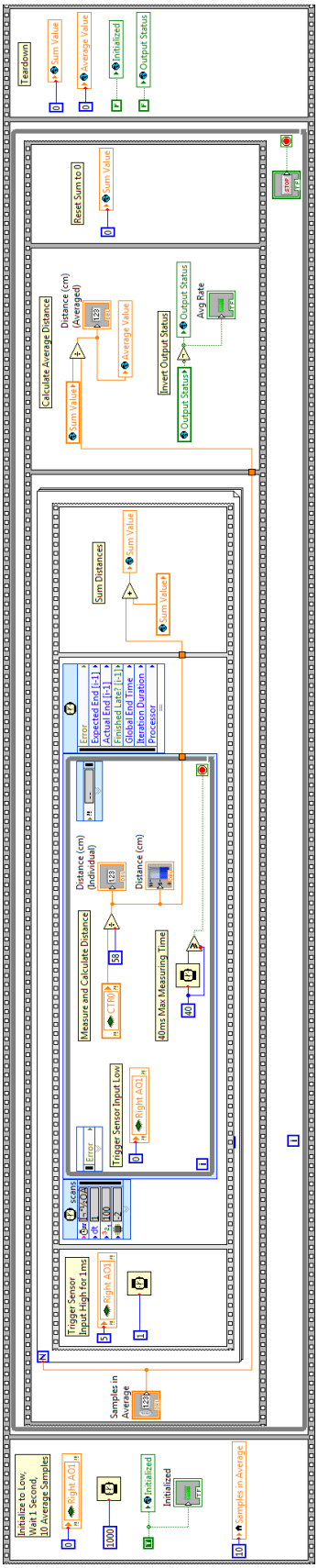
Piponi, D. (2006). *Equibot the Balancing Robot*. Retrieved 2011, from SIGFPE: http://homepage.mac.com/sigfpe/Robotics/equibot.html

Appendix

Appendix 1: LabVIEW Block Diagram for SRF05 Sensor Testing



Appendix 2: LabVIEW Block Diagram for SRF05 Sensor



Appendix 3: SHUBot CAD Rendering