http://www.joyglobal.com/clientCSS/images/logo_JoyGlobal.png

**Implementation of an Agile Educational Robot**



**May 22, 2015**

Logan Beaver

Justin Campbell

Tyler Paddock

Ronald Shipman

***Advisor***: Dr. Luis A. Rodriguez

Contents

[Executive Summary 3](#_Toc420018202)

[Hardware Construction 3](#_Toc420018203)

[Pneumatic System 6](#_Toc420018204)

[Electrical System 6](#_Toc420018205)

[Control System 10](#_Toc420018206)

[Results and Future Work 14](#_Toc420018207)

[References 15](#_Toc420018208)

[Appendix I – XboxInterface.dll 16](#_Toc420018209)

[Appendix II – Java Application – Human Machine Interface 19](#_Toc420018210)

# Executive Summary

The Milwaukee School of Engineering (MSOE) participates in Science Technology Engineering and Mathematics (STEM) outreach events for prospective students. The school will benefit greatly from having a sophisticated robotic control system to build excitement about STEM as well as sparking interest in fluid power, automation, and the controls fields. An agile pneumatic robot is not only a complicated control system that can be used to get young people excited about STEM, but it will also increase the prestige of MSOE knowing that a group of seniors attending the school were able to design and build the system from the ground up. In addition it also provides an exciting opportunity for future groups to iterate on the design and integrate new and exciting features.

Implementation of the agile educational robot began. From the design quarter constraints were determined. The most important constraints are listed below:

* A maximum weight of 35 kg for portability
* Maximum size of 0.75 m x 0.75 m x 1.0 m box for portability
* Custom debug panel creation to facilitate troubleshooting
* MATLAB and Simulink model support to allow mechanical engineering students to update control algorithms without knowledge of C/C++
* Electronic fuses and shielding to protect the robot and operator during use and maintenance
* Mechanical protection to reduce the risk of pinching and self-collision damage to the robot
* An easy to access emergency stop to quickly depower the robot
* A pressure relief valve to reduce the risk of overloading and damaging pneumatic components

The design work done on this project was a continuation of the work done by Kevin Lee during the Research Experience for Undergraduates (REU) at MSOE. His work involved deriving a dynamic model for a simplified quadruped robot. Now with our team’s design work finished the implementation of this walking robot was conducted. All necessary components were ordered, tested, and assembled. Pneumatic components were mostly provided by the vendor Numatics. Numatics valves and cylinders are used in the implementation. From the previous design quarter a dog like robot was offered as the final design. The chassis of this dog-like robot was constructed out of T-slotted aluminum framing. The legs of the robot constructed out of solid aluminum. Electrical components were fabricated onto prototyping boards and tested. Software was written in Java, Windows C code, and MATLAB Simulink. The user interface was implemented in java, the controller driver was interfaced with c code dll, and the microcontroller code was written in Simulink block diagram logic.

# Hardware Construction

The chassis of the robot is a construction of 6105-T5 aluminum extrusion and 6061 aluminum plate. The extrusions were cut to length, while various pieces had angles milled. These pieces were then connected with plates and fasteners for the t-slotted aluminum. One change in the design of the chassis was the cross beam added in the middle of the chassis, shown in Figure 1. This change was to ensure added strength against twisting. In addition, more pieces of extruded aluminum were added to the bottom of the chassis, to allow the bottom plate of the chassis to be mounted. Electrical and pneumatic components were then mounted to this bottom plate.

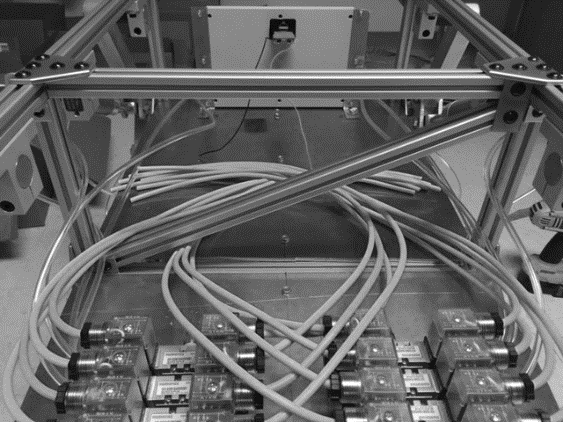


Figure 1: Cross beam of robot's chassis

In this phase of the project a hip design was chosen, consisting of a tube holder and a steel shaft lathed down to the appropriate diameter. The tube holder, shown in Figure 2, being manufactured for extrusions with larger slotting, required the chassis to be further machined to accommodate the tube holders. Much like the plates used to connect the individual pieces of extrusion, the tube holders were then slid into the slots of the chassis.

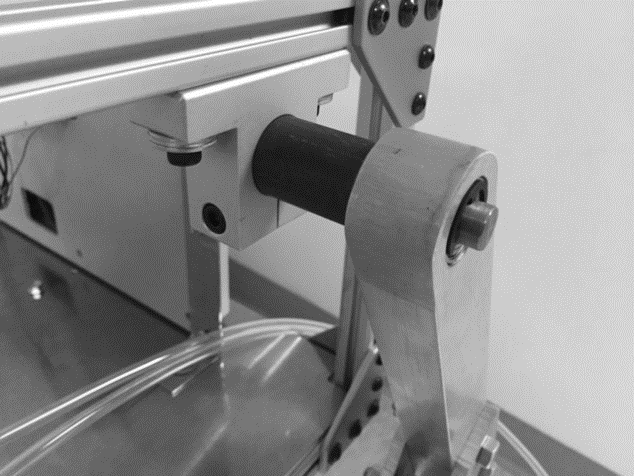


Figure 2: Hip joint attached to robot chassis

The legs of the robot are constructions of 6061 aluminum bar. The aluminum bar was cut to appropriate lengths, and much like the extrusions for the chassis, had required angles milled. The leg segments were rounded and also had notches cut in to produce the knee joint of the robot. Plates, displayed in Figure 3, were then cut to the appropriate size to produce points for the air cylinders to attach to the thigh and shank segments of the legs, as well as the chassis.

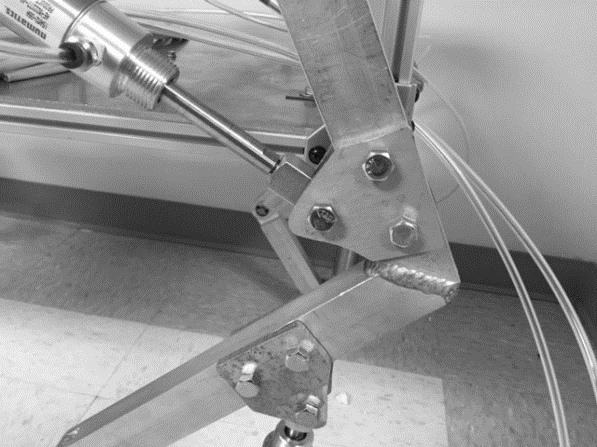


Figure 3: Attachment plate between air cylinder and leg segment

These plates were then bolted to the legs. In order to increase the robot’s range of motion, its legs were flipped 180 degrees about the vertical axis, shown in Figure 4.

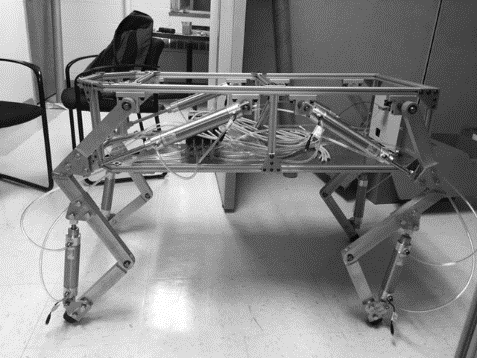
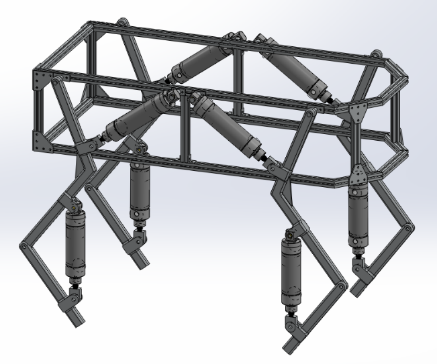


Figure 4: Comparison of robot’s old leg orientation (left) and new orientation (right).

The feet of the robot consist of a rubber coating. The final result, shown in Figure 5, was of multiple dipped layers.

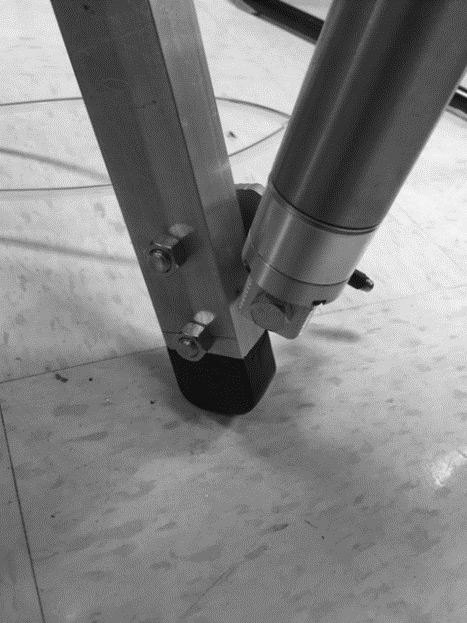


Figure 5: Finished foot of robot

# Pneumatic System

The final pneumatic system of the robot consists of 8 dual-acting air cylinders with position feedback sensors, two manifolds housing 8 dual-solenoid 4 port/3 position directional control valves, a 10 gallon air compressor, supplying 6.3 cubic feet of airflow per minute (CFM) at 40 psi and 5.7 CFM at 90 psi, and necessary connectors and tubing to interface the components. In addition to these components, a soft start/dump valve was added to the system to ensure safe pressurization and depressurization of the system during use and after use respectively. During this phase of the project, the bore diameter of the air cylinders was decreased from 2 inches to 1.5 inches. This decision was made after it was determined that a smaller bore diameter would be adequate in generating the torques required to drive the legs of the robot.

# Electrical System

The electronics of the robot are broken up into two major subsystems, the motherboard and the debug panel. The motherboard was designed to contain the auxiliary electronics and signal conditioning components needed for the robot. The debug panel contains all necessary electronics to display battery levels and other statuses of the robot.

To power the electronics two different sets of batteries are used. A 9 volt battery is used to power the microcontroller. The valves are powered by multiple 12 volt batteries connected in series. The valves run on 24 volt power and the signal condition circuits use 12 volt power.

The debug panel subsystem contains a physical panel with light emitting diodes (LED) and connections for banana plug cables. The LEDs are used to show battery levels and the status of the robot. The banana plug connectors are used to interface to Milwaukee School of Engineering’s test equipment in the labs. Banana plugs are used because they are standard on test equipment. A USB slot is also included on the debug panel to assist in programming the microcontroller without removing it in the robot. The following figure shows a brief layout of the debug panel components.

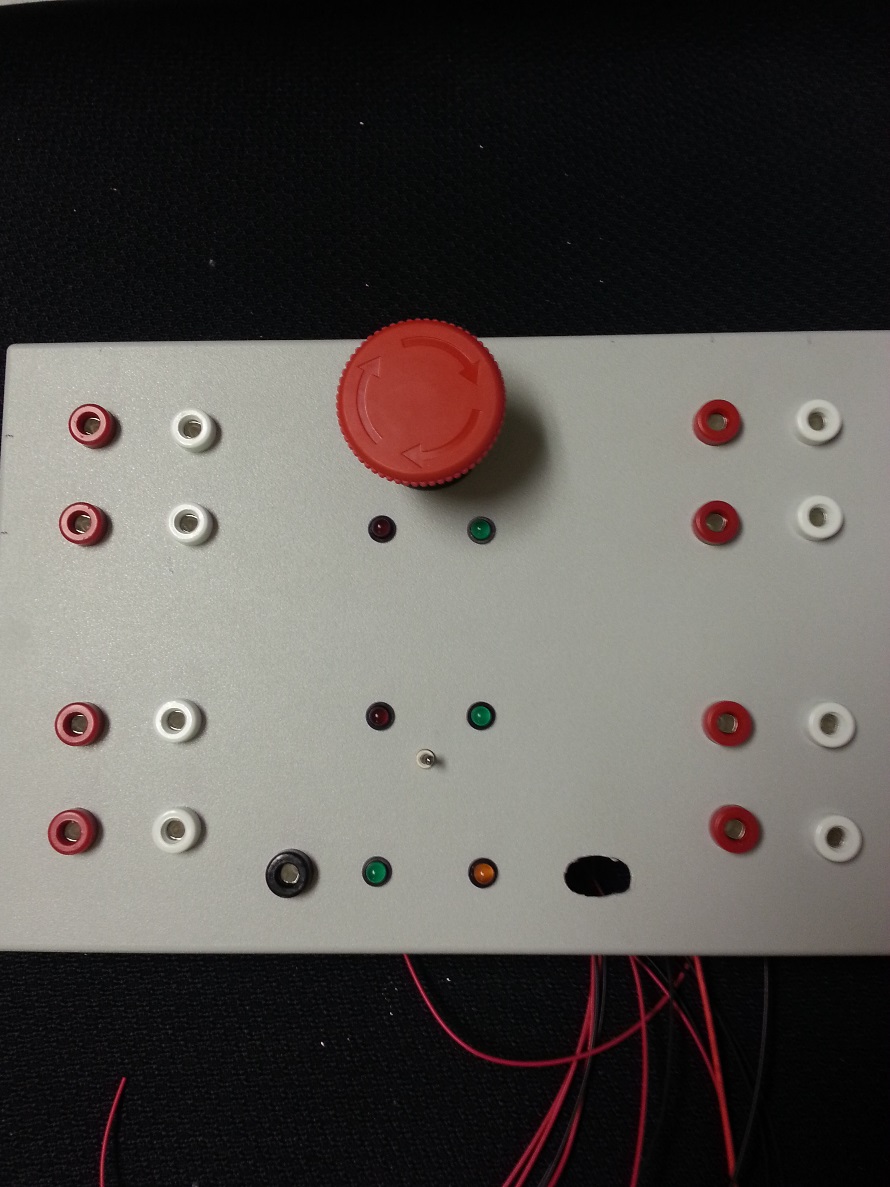


Figure 6: Debug Panel Component Layout

A nine volt battery level indicator was created using a 6.2 zener diode and two transistors. The circuit drives two different color LEDs to indicate if the voltage needs to be replaced. If the green LED is on then the battery is at a good level. Once the green LED goes out the operator should replace the nine volt battery. If the red LED goes out the microcontroller operation is not guaranteed to be successful based on the remaining batter voltage. At this condition the nine volt battery needs to be replaced.

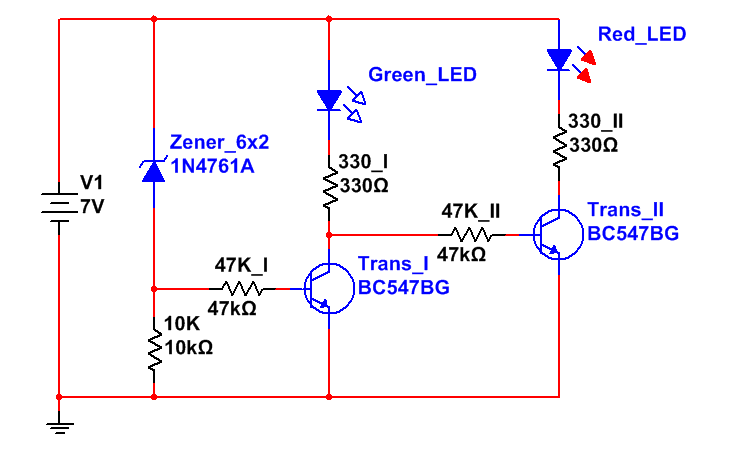


Figure 7: 9 Volt Battery Level Indicator

The battery level indicator is needed because of the discharge curve of a nine volt battery. The battery maintains close to its specified level and then drops quickly. The indicator allows the operators to change the battery at an appropriate time. The nine volt battery discharge curves are shown below in figure 8.

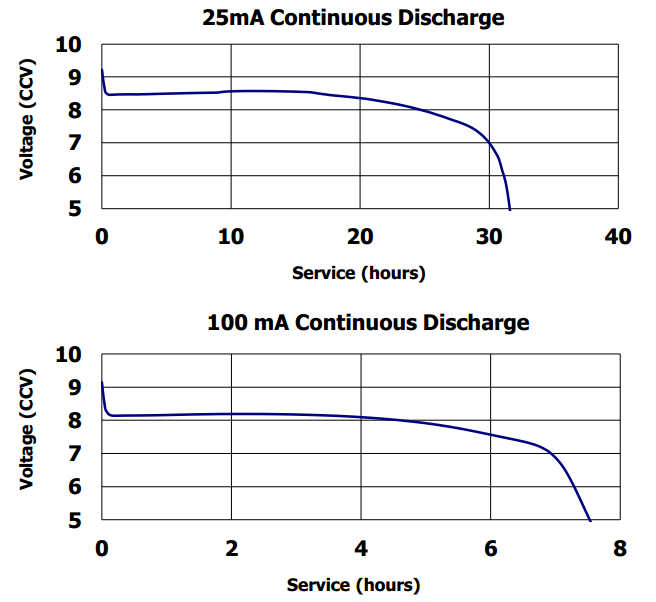


Figure 8: Energizer 9 volt battery discharge curve [1]

From the previous quarter a low pass filter was designed to convert the pulse-width modulated outputs of the Arduino microcontroller into analog signals. This quarter the filters were implemented on prototyping boards. Each board has two third order low pass filters and doubling amplifiers. Each board has seven connection points. The connections include two inputs and two outputs a positive rail and negative rail for the amplifiers and a ground connection. The layout of the prototype board is shown in figure 9.

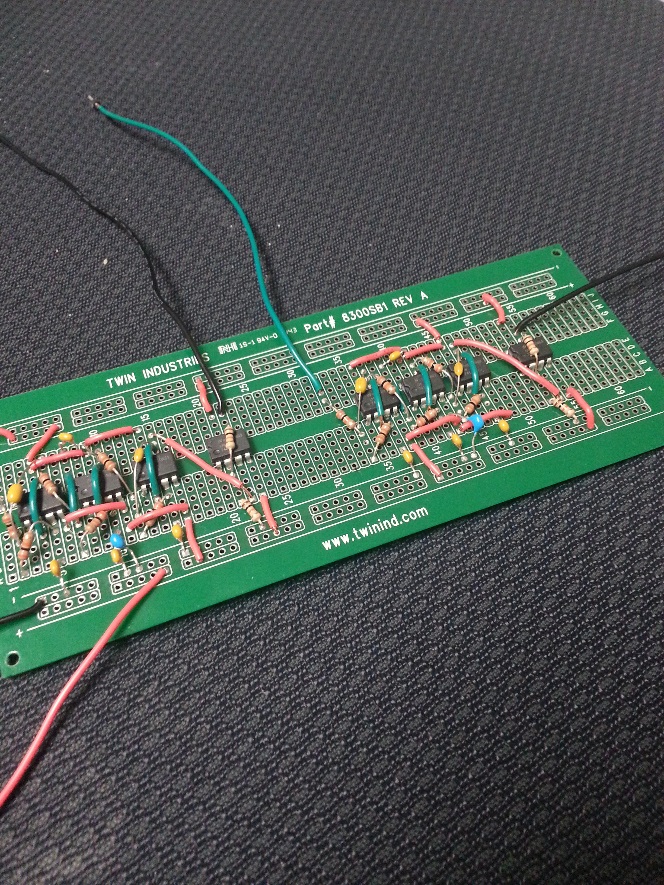


Figure 9: Prototyping Board with LPF and Amplifier

A few changes were made from the initial design. The capacitor values of the filter were changed to be standard capacitor values. The circuit diagram of a single filter and amplifier is shown in figure 10 and 11.

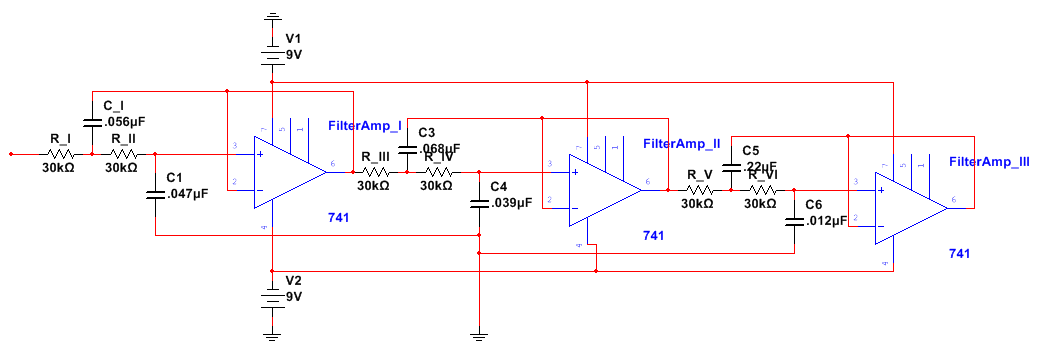


Figure 10: Low Pass Filter Layout

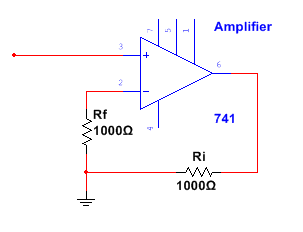


Figure 11: Amplifier Layout

The low pass filter converts the microcontroller PWM into an analog signal. The max value of this analog signal is 5 volts. The scaling amplifier afterwards increase the filter output by a gain of two.

In since the batteries are not truly 12 volts on each battery the voltage could be higher than needed. A voltage regulator circuit is used to limit the voltage to 24 volts. The circuit is taken directly from the voltage regulator datasheet. Figure 12 is the image taken from the datasheet.

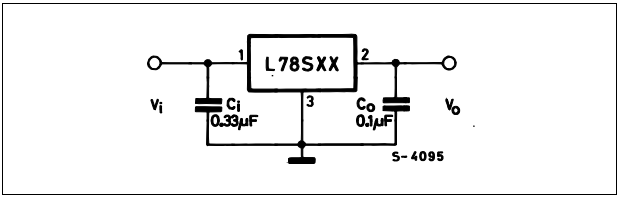


Figure 12: Voltage Regulator Circuit From Datasheet [2]

# Control System

The robot control system consists of two parts, a Simulink control architecture running on an Arduino Mega 2650 and a GUI developed in Java which runs on 64 bit windows. The Java application has two dependent libraries, Java Native Access (JNA) and RXTX. JNA is used to call native machine code to read from an xbox controller. The native code is the XboxInterface.dll which was a custom C code project completed by our team. The C code calls windows xbox driver code to read from the controller.

RXTX is a java library used to communicate with the serial ports on the PC. When the *connect* button is pressed in the GUI RXTX searches for available serial ports and connects to the first available to find the XBee radio transmitter/receiver. The java application calls the XboxInterface.dll to get a current controller conditions and identify a user request. The current user request is sent out the Xbee device to the robot’s receiver.

The Simulink code is cross compiled using the Arduino software package. Due to hardware limitations the control code was simplified to controlling the valve command with user input. The following block diagram shows the code run by the Arduino at runtime:

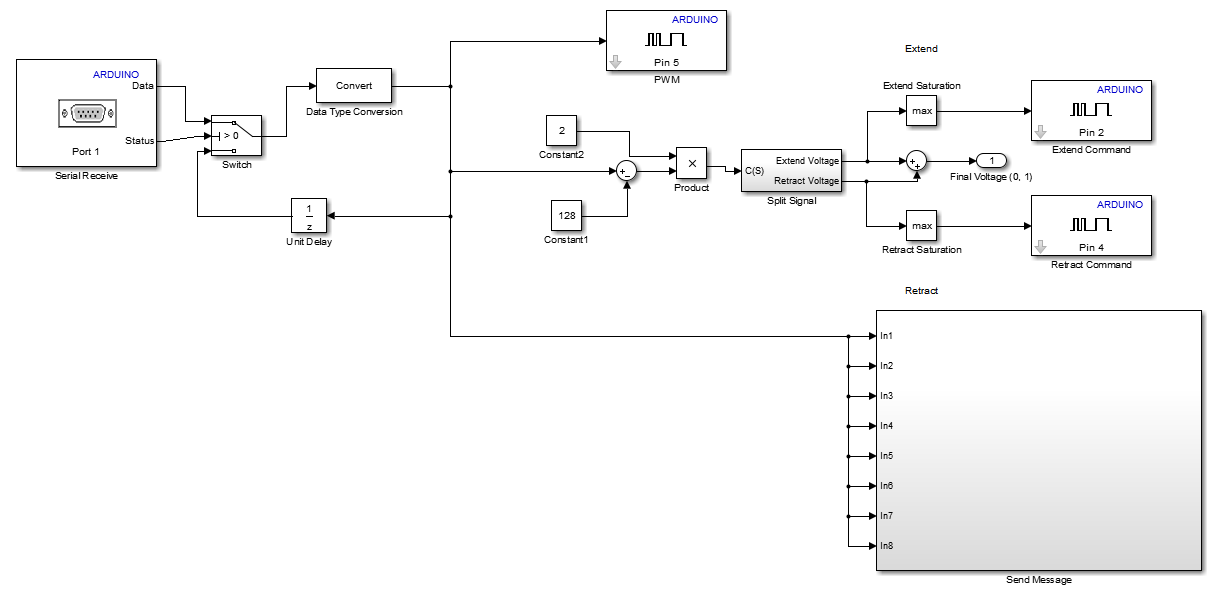


Figure 12: Code executed at runtime on the arduino. The blocks with the word Arduino in the upper right are from the arduino support toolbox. The send message block consists of a mux and a serial write on Port 1.

The most important part of this block diagram is the Split Signal block. Because each valve has two solenoids two analog signals are required to effectively control it. The split signal subsystem takes in a command and splits it into a positive extend and positive retract signal. At least one of these commands is always zero. The Split Signal subsystem can be seen below:

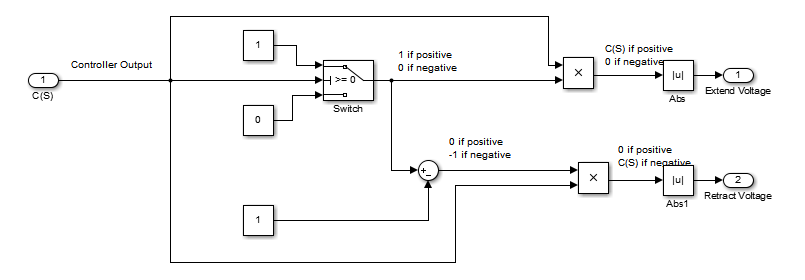


Figure 13: The Split Signal sub diagram. The absolute value blocks were added during system debugging and may be removed.

In addition more code was developed to characterize the frequency response of the valve and cylinder. It uses a sinusoidal signal generator and outputs the current sine value while outputting an extended or retract signal. The frequency response test code can be seen below:

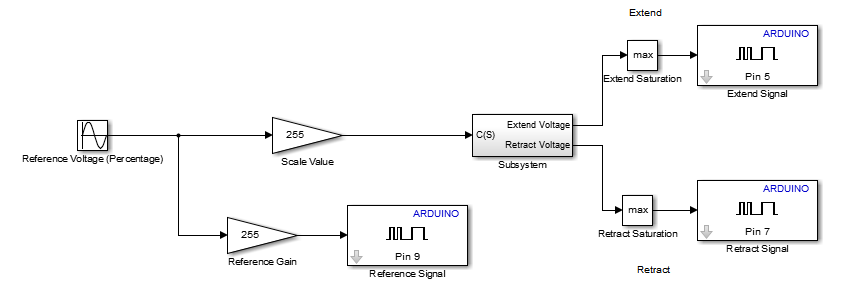


Figure 14: Cylinder frequency response test. The included subsystem is the signal split block.

In addition a much more sophisticated control system was developed based on position feedback of the cylinders and a state machine with user input. The high level diagram for the control system is given below:

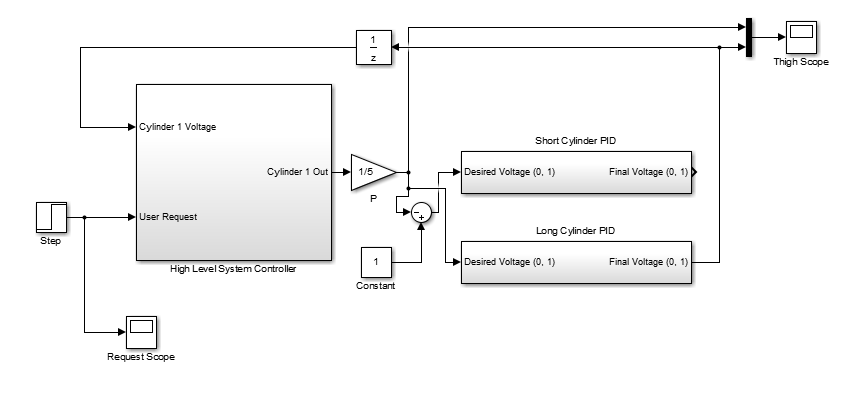


Figure 15: High level diagram for the sophisticated control system. The high level system controller uses the user request, simulated by the step, and the feedback to calculate the next foot position.

The cylinder PID’s read the feedback voltage and calculate an error based on desired position. They then use P control to adjust their output signal. Inside the high level system controller the cylinder feedback is used along with a user signal to determine the next foot position. The Unit Delay (1/Z) block is required on the feedback due to the discrete implementation of the software.

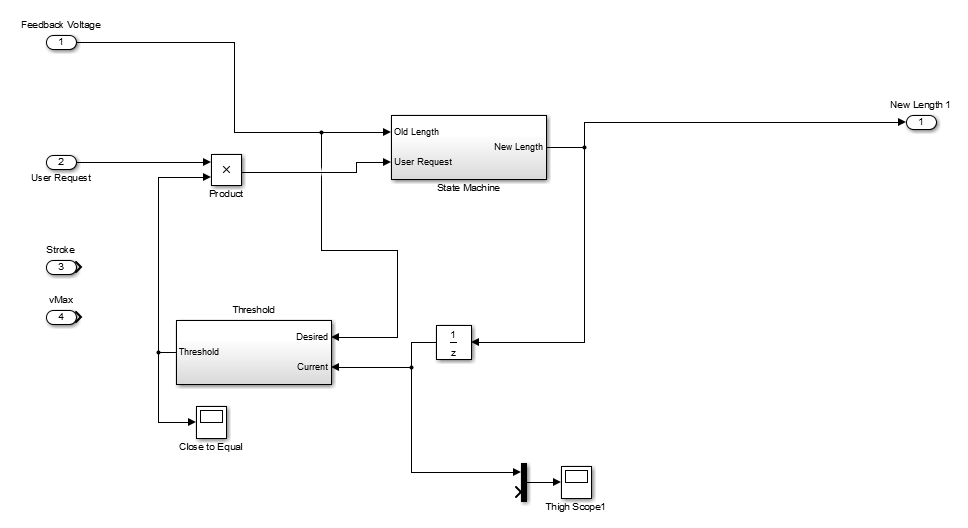


Figure 16: The high level system controller. The feedback voltage is compared to the previous value and adjusts the user request. If the value is out of the threshold the user reques is set to zero.

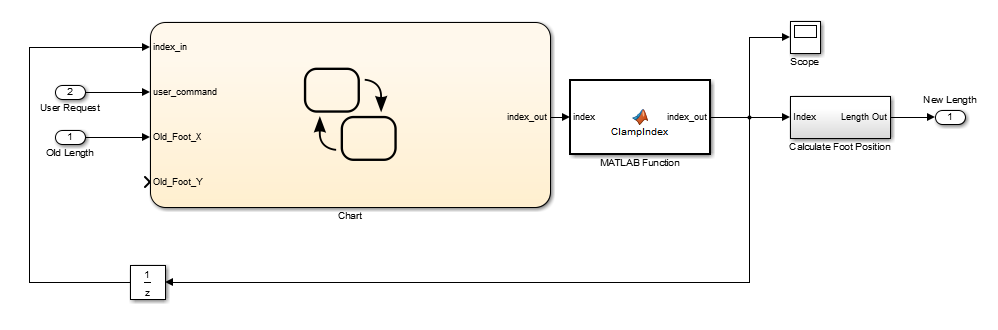


Figure 17: The state machine subsystem. The chart is a Stateflow diagram which contains several states which are transitioned based on the user request. The ClampIndex simply ensures the index never goes out of bounds.

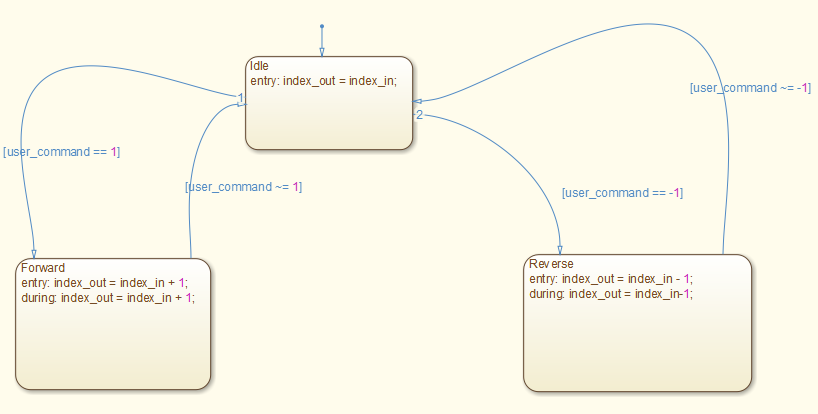


Figure 18: The stateflow diagram. The default state is Idle which holds the leg still, the forward state increments the index, and the reverse state decrements the index.

The complex control structure was validated using the Bimba air cylinders with position feedback. To implement wireless capability portions of the serial in and out systems from Figure 12 could be used.

# Results and Future Work

The robot was constructed in subsystems. The electrical and control subsystems were tested throughout implementation and development. Upon completion of the first leg of the robot the cylinders were attached and driven using available servo valves. At this time the system was pressurized and connected into the filter and amplifier prototyping boards. The user interface was used to drive the leg forward and backward manually.

The next steps for this project include adding sensors to detect disturbances and tipping during walking motions. Using those sensors correction can be made to the gait of the robot. Additionally, advanced gaits and motions need to be programmed and tested.

One problem that occurred during the project was learning that the pneumatic valves are controlled by multiple solenoids after much of the design was completed assuming one solenoid. As a result the number of PWM outputs on the microcontroller need to be doubled to control the robot. Another result of this problem is that each leg needs twice as much signal conditioning circuitry.

# References

|  |  |
| --- | --- |
| [1] | Energizer, "www.energizer.com," [Online]. Available: www.energizer.com. [Accessed 26 2 2015]. |
| [2] | ST Microelectronics, "http://www.farnell.com," February 2003. [Online]. Available: http://www.farnell.com/datasheets/76145.pdf. [Accessed 20 May 2015]. |

# Appendix I – XboxInterface.dll

// dllmain.cpp : Defines the entry point for the DLL application.

// Author: Tyler Paddock

// Date: 4/01/2015

// Purpose: Interface with the XboxInput function on a windows machine

// Returns Button states from an xbox controller

#include "stdafx.h"

#include "XInput.h"

#include "Windows.h"

#include "math.h"

#include "XboxInterface.h"

BOOL APIENTRY DllMain( HMODULE hModule,

DWORD ul\_reason\_for\_call,

LPVOID lpReserved

)

{

return TRUE;

}

//Function Declarations

extern "C" \_\_declspec(dllexport) int getLeftJoyStick(int);

extern "C" \_\_declspec(dllexport) int getRightJoyStick(int);

extern "C" \_\_declspec(dllexport) int getButtonStates(int);

extern "C" \_\_declspec(dllexport) short getTriggerStates(int);

//The code implementation

#include "XboxInterface.cpp"

\_\_declspec(dllexport) int getLeftJoyStick(int controllerNumber){

DWORD dwResult;

XINPUT\_STATE state;

ZeroMemory(&state, sizeof(XINPUT\_STATE));

dwResult = XInputGetState(controllerNumber, &state);

//Declare retrun value

int retVal = 0;

if (dwResult == ERROR\_SUCCESS){

//Controller

retVal = (state.Gamepad.sThumbLX << SHORT\_SHIFT) | state.Gamepad.sThumbLY;

}

return retVal;

}

\_\_declspec(dllexport) int getRightJoyStick(int controllerNumber){

DWORD dwResult;

XINPUT\_STATE state;

ZeroMemory(&state, sizeof(XINPUT\_STATE));

dwResult = XInputGetState(controllerNumber, &state);

//Declare retrun value

int retVal = 0;

if (dwResult == ERROR\_SUCCESS){

//Controller

retVal = (state.Gamepad.sThumbRX << SHORT\_SHIFT) | state.Gamepad.sThumbRY;

}

return retVal;

}

\_\_declspec(dllexport) int getButtonStates(int controllerNumber){

DWORD dwResult;

XINPUT\_STATE state;

ZeroMemory(&state, sizeof(XINPUT\_STATE));

dwResult = XInputGetState(controllerNumber, &state);

//Declare retrun value

int retVal = 0;

if (dwResult == ERROR\_SUCCESS){

//Controller Connected

retVal = state.Gamepad.wButtons;

}

return retVal;

}

//short = byte(leftTrigger) + byte(rightTrigger)

\_\_declspec(dllexport) short getTriggerStates(int controllerNumber){

DWORD dwResult;

XINPUT\_STATE state;

ZeroMemory(&state, sizeof(XINPUT\_STATE));

dwResult = XInputGetState(controllerNumber, &state);

//Declare retrun value

short retVal = 0;

if (dwResult == ERROR\_SUCCESS){

//Controller Connected

retVal = ((state.Gamepad.bLeftTrigger) << BYTE\_SHIFT) | state.Gamepad.bRightTrigger;

}

return retVal;

}

#ifndef XboxInterface\_H

#define XboxInterace\_H

#include "stdafx.h"

#include "XInput.h"

#include "Windows.h"

#include "math.h"

#define SHORT\_SHIFT 16

#define BYTE\_SHIFT 8

//DWORD readControllerState(XINPUT\_STATE\*, int);

#endif

# Appendix II – Java Application – Human Machine Interface

**package** arc.msoe.hmi;

**import** java.awt.BorderLayout;

**import** java.awt.FlowLayout;

**import** java.awt.GridLayout;

**import** java.awt.event.ActionEvent;

**import** java.awt.event.ActionListener;

**import** java.awt.image.BufferedImage;

**import** java.io.File;

**import** java.io.IOException;

**import** javax.imageio.ImageIO;

**import** javax.swing.ImageIcon;

**import** javax.swing.JButton;

**import** javax.swing.JFrame;

**import** javax.swing.JLabel;

**import** javax.swing.JPanel;

**import** arc.msoe.hmi.comms.InputThread;

/\*\*

\* Creates the main GUI and serves as the bridge between user input and the Xbee controller

\* Provides updates on the current state of the robot

\*

\* 3/15/2015

\* **@author** beaverl

\*

\*/

**public** **class** MainGUI **extends** JFrame{

//Frame Constants

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**int** height = 600;

**int** width = 800;

String heading = "A.R.C. Robot HMI";

//helper classes

RTComms comms; //IO with the Xbee unit

InputThread input; //parses user input

Thread inputThread;

//boolean values

**boolean** validController; //is the controller connected?

**boolean** commsConnected; //have the comms connected?

**boolean** faultInComs; //have the comms faulted?

//GUI elements

//images

ImageIcon ARCLogo; //ARC Logo

JLabel ARCLogoLabel;

JLabel title; //program title

JLabel subtitle; //program subtitle

//buttons

JButton exit; //button used to close the program

JButton connect; //starts the comms

//comms indicators

ImageIcon commsOn; //comms active

ImageIcon commsFault; //comms fault

ImageIcon commsConnecting; //comms connecting

ImageIcon commsOff; //comms inactive

JLabel commsOnLabel;

JLabel commsOffLabel;

JLabel commsFaultLabel;

JLabel commsConnectLabel;

//cylinder indicators

JLabel Leg1T;

JLabel Leg1S;

JLabel Leg2T;

JLabel Leg2S;

JLabel Leg3T;

JLabel Leg3S;

JLabel Leg4T;

JLabel Leg4S;

ImageIcon robot; //robot image

JLabel robotLabel;

//controller indicators

ImageIcon controllerOn; //controller active

ImageIcon controllerFault; //controller fault

ImageIcon controllerOff; //controller inactive

JLabel controlOnLabel;

JLabel controlOffLabel;

JLabel controlFaultLabel;

//misc JLabels

JLabel controllerRef; //current controller command

JLabel commsText;

JLabel inputText;

**public** MainGUI() {

**this**.setSize(width, height); //set the size of the window

**this**.setDefaultCloseOperation(***DO\_NOTHING\_ON\_CLOSE***); //close the window when exit is pressed

**this**.setTitle(heading); //window title

**this**.setLocationRelativeTo(**null**); //center window

setupComponents();

createGUI(); //create the GUI

**this**.setVisible(**true**); //make the window visible

}

/\*\*

\* loads images and creates initial GUI elements

\*

\*/

**private** **void** setupComponents() {

input = **new** InputThread();

inputThread = **new** Thread(input);

inputThread.start();

System.***out***.println(System.*getProperty*("sun.arch.data.model"));

comms = **new** RTComms();

//declare images

BufferedImage LEDOnImage = **null**;

BufferedImage LEDOffImage = **null**;

BufferedImage LEDErrorImage = **null**;

BufferedImage LEDIdleImage = **null**;

BufferedImage ARCLogoImage = **null**;

BufferedImage DogeBotImage = **null**;

//load images with IO

**try** {

LEDOnImage = ImageIO.*read*(**new** File("res/ActiveImage.png"));

LEDOffImage = ImageIO.*read*(**new** File("res/OffImage.png"));

LEDErrorImage = ImageIO.*read*(**new** File("res/ErrorImage.png"));

LEDIdleImage = ImageIO.*read*(**new** File("res/IdleImage.png"));

ARCLogoImage = ImageIO.*read*(**new** File("res/ARCLogo.png"));

DogeBotImage = ImageIO.*read*(**new** File("res/DogebotImage.png"));

} **catch**(IOException e) {

System.***err***.println("COULD NOT LOAD ALL IMAGE ICONS");

e.printStackTrace();

System.*exit*(-1);

}

//initialize ImageIcons

ARCLogo = **new** ImageIcon(ARCLogoImage); //ARC Logo

robot = **new** ImageIcon(DogeBotImage); //robot image

commsOn = **new** ImageIcon(LEDOnImage); //comms active

commsFault = **new** ImageIcon(LEDErrorImage); //comms fault

commsOff = **new** ImageIcon(LEDOffImage); //comms inactive

commsConnecting = **new** ImageIcon(LEDIdleImage);

controllerOn = **new** ImageIcon(LEDOnImage); //controller active

controllerFault = **new** ImageIcon(LEDErrorImage); //controller fault

controllerOff = **new** ImageIcon(LEDOffImage); //controller inactive

//initialize JLabel Icons

ARCLogoLabel = **new** JLabel(ARCLogo);

robotLabel = **new** JLabel(robot);

commsOnLabel = **new** JLabel(commsOn);

commsOffLabel = **new** JLabel(commsOff);

commsFaultLabel = **new** JLabel(commsFault);

commsConnectLabel = **new** JLabel(commsConnecting);

controlOnLabel = **new** JLabel(commsOn);

controlOffLabel = **new** JLabel(commsOff);

controlFaultLabel = **new** JLabel(commsFault);

//initialize JLabels

title = **new** JLabel("A.R.C. HMI"); //program title

subtitle = **new** JLabel("Developed by Logan Beaver"); //program subtitle

Leg1T = **new** JLabel("Leg 1 Thigh Extension: 0 inches");

Leg1S = **new** JLabel("Leg 1 Shank Extension: 0 inches");

Leg2T = **new** JLabel("Leg 2 Thigh Extension: 0 inches");

Leg2S = **new** JLabel("Leg 2 Shank Extension: 0 inches");

Leg3T = **new** JLabel("Leg 3 Thigh Extension: 0 inches");

Leg3S = **new** JLabel("Leg 3 Shank Extension: 0 inches");

Leg4T = **new** JLabel("Leg 4 Thigh Extension: 0 inches");

Leg4S = **new** JLabel("Leg 4 Shank Extension: 0 inches");

controllerRef = **new** JLabel("Current Controller Reference: NONE"); //current controller command

commsText = **new** JLabel("Comms status: off");

inputText = **new** JLabel("Input Status: off");

//initialize Buttons

connect = **new** JButton("Connect"); //starts the comms

exit = **new** JButton("Exit"); //button used to close the program

connect.addActionListener(**new** ConnectListener());

exit.addActionListener(**new** EscapeListener());

}

/\*\*

\* organizes all elements for the main GUI

\*/

**private** **void** createGUI() {

**this**.setLayout(**new** BorderLayout());

//creating JPanel holders

JPanel titlePanel = **new** JPanel(); //contains title and logos

JPanel bodyPanel = **new** JPanel(); //contains buttons and status indicators

titlePanel.setLayout(**new** BorderLayout());

bodyPanel.setLayout(**new** BorderLayout());

//place the GUI elements in the Layouts in the JPanels in the JFrame

titlePanel.add(title);

titlePanel.add(subtitle, BorderLayout.***SOUTH***);

titlePanel.add(ARCLogoLabel, BorderLayout.***EAST***);

add(titlePanel, BorderLayout.***NORTH***);

JPanel buttonPanel = **new** JPanel();

buttonPanel.setLayout(**new** GridLayout(1,2));

buttonPanel.add(connect);

buttonPanel.add(exit);

bodyPanel.add(buttonPanel, BorderLayout.***NORTH***);

**this**.add(bodyPanel);

JPanel statusPanel = **new** JPanel();

statusPanel.setLayout(**new** FlowLayout());

statusPanel.add(commsText);

statusPanel.add(commsOffLabel);

statusPanel.add(inputText);

statusPanel.add(controlOffLabel);

bodyPanel.add(statusPanel);

// this.add(Leg1T);

// this.add(Leg1S);

// this.add(Leg2T);

// this.add(Leg2S);

// this.add(Leg3T);

// this.add(Leg3S);

// this.add(Leg4T);

// this.add(Leg4S);

// this.add(robotLabel);

}

**public** **static** **void** main(String[] args) {

**new** MainGUI(); //create the GUI on startup

}

**private** **void** dllTest() {

comms.dllTest();

System.***out***.println("testing DLL");

}

**class** EscapeListener **implements** ActionListener{

@Override

**public** **void** actionPerformed(ActionEvent event) {

//gracefully shut down

input.close(); //stops controller input thread

// **TODO** disconnect comms

//close the program

System.*exit*(0);

}

}

**class** ConnectListener **implements** ActionListener {

@Override

**public** **void** actionPerformed(ActionEvent event) {

//send a connection request to RT Comms

//wait for timeout

//if the connection is accepted, the comms are not faulting

dllTest();

System.***out***.println("connecting...");

}

}

}

**package** arc.msoe.hmi.comms;

/\*\*

\* A thread that reads from the Xbee wireless communicator and updates RTComms

\*

\* 3/12/2015

\* **@author** beaverl

\*

\*/

**public** **class** ReadThread **extends** Thread {

}

**package** arc.msoe.hmi.comms;

/\*\*

\* Takes input from the user controller and provides the requested action to MainGUI

\*

\* 3/15/2015

\* **@author** beaverl

\*

\*/

**public** **class** InputThread **extends** Thread {

**boolean** connected;

**volatile** **boolean** running;

**public** InputThread() {

//check for a controller

connected = **true**; //controller found

running = **true**;

}

**public** **void** run() {

**while** (running) { //while the thread is running

**if** (connected) { //if a controller is connected

System.***out***.println("reading input...");

System.***out***.println("collecting data...");

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

} **else** { //look for controller

}

}

}

**public** **void** close() {

running = **false**;

}

}

**package** arc.msoe.hmi;

**import** com.sun.jna.Library;

**import** com.sun.jna.Native;

**import** arc.msoe.hmi.comms.\*;

/\*\*

\* This class is called by the MainGUI to read/write data to the robot

\* Uses ReadThread and WriteThread to communicate with the Xbee wireless communicator

\*

\* 3/15/2015

\* **@author** beaverl

\* **@author** paddockt

\*

\*/

**public** **class** RTComms {

ReadThread read;

WriteThread write;

DLL lib;

**public** RTComms() {

**try** {

String path = System.*getProperty*("user.dir") + "\\res\\";

System.***out***.println(path);

lib = (DLL) Native.*loadLibrary*(path + "DLLTest.dll",

DLL.**class**);

} **catch** (Error e) {

e.printStackTrace();

System.*exit*(-1);

}

}

**public** **void** dllTest() {

**double** sa = 0;

**double** vol = 0;

System.***out***.println("The radius should be two: " + lib.GetSphereSAandVol(1, sa, vol));

System.***out***.println("Surface area = " + sa);

System.***out***.println("Volume = " + vol);

}

//Test interface using JNA to load a .dll file

**public** **interface** DLL **extends** Library {

// FREQUENCY is expressed in hertz and ranges from 37 to 32767

// DURATION is expressed in milliseconds

**public** **int** GetSphereSAandVol(**double** radius, **double** sa, **double** vol);

}

}