```
//Alexander Popov
//St. Norbert College
//CS460
//March 3, 2011
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

using System.Threading;
using System.IO.Ports;

namespace iRobot
{
    public delegate void ReadDelegate(string read);
    public partial class Form1 : Form
    {

        public Form1()
        {
            InitializeComponent();

        }
        #region Globals
        DriveHistory DriveLog = new DriveHistory();
        const int ForwardCommandID = 1;
        const int BackwardsCommandID = 2;
        const int RightCommandID = 3;
        const int LeftCommandID = 4;

        //this variable tells wait loops in turn threads to end so the thread can be↙
      terminated, must be set as volatile so all the threads can see it
        public volatile bool TerminateThread = false;
        //set up globl Lock Variable
        System.Threading.Mutex Mutex = new System.Threading.Mutex(false, "mutex");
        #endregion
        #region Methods
        /// <summary>
        /// Using the stack, turns the robot around and drives him "back" traversing↙
      through the stack and executing the commands backwards
        /// </summary>
        private void BackTrack()
        {
            int[] LastCommand = new int[7];
            byte[] readBufferByte = new byte[5];
            int[] reverseLog = new int[4000];
            byte[] Stop = new byte[] { 137, 0, 0, 0, 0 };
            int angle = 0;
            int distance = 0;
            int LastCommandDistance = 0;
            int LastCommandAngle = 0;
            int LastCommandSpeed = 0;
            int size = DriveLog.GetSize();
            int z = 0;
            int j = 0;

            //Copy the current stack in reverse order
            while (size >= j)
            {
                LastCommand = DriveLog.PopLastCommandLine();
                Array.Reverse(LastCommand);
                LastCommand.CopyTo(reverseLog, j);
                j = j + 7;

            }

            //if the TerminateThread command is not issued, turn the robot around 180 degrees
            if (!TerminateThread)
            {
                LeftTurn();
                LastCommand = DriveLog.PopLastCommandLine();
            }
            Thread.Sleep(20);
            if (!TerminateThread)
            {
                LeftTurn();
                LastCommand = DriveLog.PopLastCommandLine();
            }
            //Traverse the reversed stack
            while (z < size)
            {
                if (TerminateThread)
                    break;
                //Grab the command line from the reversed stack
                for (int i = 0; i < 7; i++)
                    LastCommand[i] = reverseLog[z + i];

                //Convert high/low bytes to integers
                LastCommandDistance = ConvertHighLowToInt(Convert.ToByte(LastCommand[1]), Convert.ToByte(LastCommand[2]));
                LastCommandAngle = ConvertHighLowToInt(Convert.ToByte(LastCommand[3]), Convert.ToByte(LastCommand[4]));
                LastCommandSpeed = Convert.ToInt32(LastCommand[6]) * 50;



                distance = 0;
                angle = 0;

                //Check for CommandID and execute appropriate command
                switch (LastCommand[0])
                {
                    case 1:
                        if (LastCommandDistance == 0)
                            break;
                        if (Mutex.WaitOne() == false)
                            break;
                        if (LastCommandDistance - 13 > 0)
                            LastCommandDistance = LastCommandDistance - 13;

                        DriveForward(LastCommandSpeed);

                        //If the robot turned while moving, perform the turn on the move
                        if (LastCommandAngle > 0)
                            RightMoving(LastCommandSpeed, LastCommandAngle);
                        else if (LastCommandAngle < 0)
                            LeftMoving(LastCommandSpeed, LastCommandAngle);

                        Mutex.ReleaseMutex();
                        //Stop the robot after it drives 'distance'
                        while (distance < LastCommandDistance)
                        {
                            if (DriveLog.IsMoving() == false)
                            {
                                //DriveLog.SubmitSensorDataBuffer();
                                UpdateStackGUI();
                                break;
                            }
                            if (Mutex.WaitOne() == false)
                                break;
                            if (TerminateThread)
                            {
                                UpdateStackGUI();
                                Mutex.ReleaseMutex();
                                break;
                            }

                            readBufferByte = ReadAndReturnSensors();
                            angle = angle + ConvertHighLowToInt(readBufferByte[2], readBufferByte[3]);
                            distance = distance + ConvertHighLowToInt(readBufferByte[0], readBufferByte[1]);

                            Mutex.ReleaseMutex();

                        }

                        if (Mutex.WaitOne() == false)
                            break;
                        sp.Write(Stop, 0, Stop.Length);
                        ReadAndAddSensorsToLog();
                        DriveLog.SetMoving(false);
                        UpdateStackGUI();
                        Mutex.ReleaseMutex();
                        break;
                    case 2:
                        if (LastCommandDistance == 0)
                            break;

                        DriveBackward(LastCommandSpeed);

                        //if turned while moving, perform turn on the move
                        if (LastCommandAngle > 0)
                            RightMoving(LastCommandSpeed, LastCommandAngle);
                        else if (LastCommandAngle < 0)
                            LeftMoving(LastCommandSpeed, LastCommandAngle);

                        //drive 'distance' backwards
                        do
                        {
                            Thread.Sleep(15);
                            if (DriveLog.IsMoving() == false)
                            {
                                UpdateStackGUI();
                                break;
                            }
                            if (Mutex.WaitOne() == false)
                                break;
                            readBufferByte = ReadAndReturnSensors();
                            angle = angle + ConvertHighLowToInt(readBufferByte[2], readBufferByte[3]);
                            distance = distance + ConvertHighLowToInt(readBufferByte[0], readBufferByte[1]);
                        } while (distance > LastCommandDistance);
                        sp.Write(Stop, 0, Stop.Length);
                        ReadAndAddSensorsToLog();
                        DriveLog.SetMoving(false);
                        UpdateStackGUI();
                        Mutex.ReleaseMutex();
                        break;
                    //Turn Right, mirror the turns since you are going backwards
                    case 4:
                        backgroundWorkerRightTurn.RunWorkerAsync(LastCommandAngle * -1);
                        while (backgroundWorkerRightTurn.IsBusy)
                        { }
                        backgroundWorkerRightTurn.CancelAsync();
                        break;
                    //Left Turn, mirror the turns since you are going backwards
                    case 3:
                        backgroundWorkerLeftTurn.RunWorkerAsync(LastCommandAngle * -1);
                        while (backgroundWorkerLeftTurn.IsBusy)
                        { }
                        backgroundWorkerLeftTurn.CancelAsync();
                        break;
                    default:
                        break;
                }
                //Move on to the next command which is seven away
                z = z + 7;
            }
            if (TerminateThread)
            {
                TerminateThread = false;
                return;
            }
            //Turn robot around after finishing the return
            LeftTurn();
            //Remove the turn around command from the current stack
            LastCommand = DriveLog.PopLastCommandLine();
            Thread.Sleep(20);
            LeftTurn();
            LastCommand = DriveLog.PopLastCommandLine();
        }
        /// <summary>
        /// Converts high and low byte to 32 signed int
        /// </summary>
        /// <param name="high"></param>
        /// <param name="low"></param>
        /// <returns></returns>
        public int ConvertHighLowToInt(byte high, byte low)
        {
            return (Int16)(((short)high * (short)256) + (short)low);
        }
        /// <summary>
        /// Converts an integer to high and low byte, returns byte[] with 0= high and 1 = low
        /// </summary>
        /// <param name="integerValue"></param>
        /// <returns></returns>
        public byte[] ConvertToHighLow(int integerValue)
        {
            byte[] result = new byte[2];
            byte high;
            byte low;
            Int16 original = Convert.ToInt16(integerValue);
            high = Convert.ToByte((original >> 8) & 0xff);
            low = Convert.ToByte(original & 0xff);
            result[0] = high;
            result[1] = low;
            return result;
        }
        /// <summary>
        /// Used in the BumpSensor thread to stop the robot if its moving forward or turning and its sensor is bumped
        /// </summary>
        private void BumpSensorCheckToStop()
        {
            do
            {
                try
                {
                    //If moving forward
                    if (DriveLog.GetLastCommandID() != 2 && DriveLog.IsMoving())
                    {
                        //Ask for Mutex, lock out serial port
                        if (Mutex.WaitOne(2000) == false)
                            return;
                        sp.DiscardOutBuffer();
                        sp.DiscardInBuffer();
                        //Ask for the status of the bumper
                        byte[] Buff = new byte[] { 142, 7 };
                        sp.Write(Buff, 0, Buff.Length);
                        int timeout = 0;
                        //Read the status of the bumper
                        while (sp.BytesToRead < 1)
                        {
                            Thread.Sleep(15);
                            timeout++;
                            if (timeout == 67)
                                break;
                        }
                        byte[] readBuffer = new byte[] { 0 };
                        sp.Read(readBuffer, 0, 1);
                        byte[] stopBuffer = new byte[] { 137, 0, 0, 0, 0 };
                        //If the bumper is pressed
                        if (Convert.ToInt16(readBuffer[0]) > 0 && Convert.ToInt16(readBuffer[0]) < 4)
                        {
                            //if we were turning
                            if (backgroundWorkerLeftTurn.IsBusy || backgroundWorkerRightTurn.IsBusy)
                            {
                                TerminateThread = true;
                                while (backgroundWorkerLeftTurn.IsBusy && backgroundWorkerRightTurn.IsBusy)
                                {
                                    if (timeout == 10)
                                        break;
                                    else
                                    {
                                        Thread.Sleep(100);
                                        timeout++;
                                    }
                                }

                                //TerminateThread = false;
                                //If the threads closed correctly exit safely
                                if (timeout >= 10)
                                {
                                    //if the thread did not close correctly, make sure the mutex is released for other threads

                                    backgroundWorkerRightTurn.CancelAsync();
                                    backgroundWorkerLeftTurn.CancelAsync();

                                    //run stop procedure
                                    sp.Write(stopBuffer, 0, stopBuffer.Length);
                                    DriveLog.SetMoving(false);
                                    ReadAndAddSensorsToLog();

                                    //Dirty Way To Release Abandoned Mutex...wait for abandoned mutex error then release mutex
                                    try
                                    {
                                        Mutex.WaitOne(1);
                                    }
                                    catch (AbandonedMutexException) { };
                                    //Mutex.ReleaseMutex();
                                }

                            }
                            //If backtracking
                            if (backgroundWorkerBackTrack.IsBusy)
                            {
                                TerminateThread = true;
                                Mutex.ReleaseMutex();
                                //Thread.Sleep(100);
                                while (backgroundWorkerBackTrack.IsBusy) { Thread.  ↙
    Sleep(50); }
                                if (Mutex.WaitOne(2000) == false)
                                    return;
                                //If we finished backtracking, stop
                                sp.Write(stopBuffer, 0, stopBuffer.Length);
                                DriveLog.SetMoving(false);
                                ReadAndAddSensorsToLog();
                                TerminateThread = false;
                            }
                            else
                            {
                                sp.Write(stopBuffer, 0, stopBuffer.Length);
                                DriveLog.SetMoving(false);
                                ReadAndAddSensorsToLog();
                            }
                        }
                        Mutex.ReleaseMutex();
                    }
                }
                catch (Exception ex)
                {
                    Mutex.ReleaseMutex();
                    //MessageBox.Show(ex.Message.ToString());
                }
                Thread.Sleep(50);
            } while (true);
        }
        /// <summary>
        /// Starts the BumpSensor thread which stops the robot if the bump sensor is↙
     bumped
        /// </summary>
        /// <returns></returns>
        public bool Start_CheckToStopThread()
        {
            try
            {
                Thread StopCheckThread = new Thread(new ThreadStart                 ↙
    (BumpSensorCheckToStop));
                StopCheckThread.IsBackground = true;
                if (StopCheckThread.IsAlive == true)
                    return false;
                else
                {
                    if (sp.IsOpen)
                    {
                        StopCheckThread.Start();
                        return true;
                    }
                    else
                    {
                        sp.Open();
                        StopCheckThread.Start();
                        return true;
                    }
                }
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message.ToString());
                return false;
            }
        }
        private void DriveBackward(int speed = -1)
        {
            try
            {
                //Ask for Mutex, lock out serial port
                if (Mutex.WaitOne() == false)
                    return;
                if (!sp.IsOpen)
                    sp.Open();
                //Clear movement distance history on the robot
                ReadAndAddSensorsToLog();
                DriveLog.InsertCommand(BackwardsCommandID);
                short velocity = 0;
                //Get speed from trackBar if needed
                if (speed != -1)
                    velocity = (short)speed;
                else
                    velocity = SelectedVelocity();
                velocity = (short)(velocity * -1);
                byte lowByte = (byte)(velocity & 0xff);
                byte highByte = (byte)((velocity >> 8) & 0xff);
                byte[] Buff = new byte[] { 137, highByte, lowByte, 128, 0 };
                sp.DiscardOutBuffer();
                //Start Moving
                sp.Write(Buff, 0, Buff.Length);
                DriveLog.SetMoving(true);
                Mutex.ReleaseMutex();
            }
            catch (Exception ex)
            {
                Mutex.ReleaseMutex();
                //MessageBox.Show(ex.Message.ToString());
            }
        }
        private void DriveForward(int speed = -1)
        {
            try
            {
                //Ask for Mutex(lock out everyone else)
                if (Mutex.WaitOne() == false)
                    return;
                if (!sp.IsOpen)
                    sp.Open();
                //Clear movement distance history on the robot
                ReadAndAddSensorsToLog();
                DriveLog.InsertCommand(ForwardCommandID);
                short velocity = 0;
                //Get speed if param is not set
                if (speed != -1)
                    velocity = (short)speed;
                else
                    velocity = SelectedVelocity();
                byte lowByte = (byte)(velocity & 0xff);
                byte highByte = (byte)((velocity >> 8) & 0xff);
                byte[] Buff = new byte[] { 137, highByte, lowByte, 128, 0 };
                sp.DiscardOutBuffer();
                //Start Moving
                sp.Write(Buff, 0, Buff.Length);
                DriveLog.SetMoving(true);
                Mutex.ReleaseMutex();
            }
            catch (Exception ex) { Mutex.ReleaseMutex(); }

            //sp.Close();
        }
        /// <summary>
        /// Returns 0 to 10 value of the trackBar, creates a delegate to the GUI thread if needed
        /// </summary>
        /// <returns></returns>
        private int GetTrackBarSpeedValue()
        {
            int Velocity = 0;
            if (this.trackBarSpeed.InvokeRequired)
            {
                Invoke(new MethodInvoker(
              delegate
              {
                  Velocity = Convert.ToInt16(trackBarSpeed.Value);
              }));
            }
            else
                Velocity = Convert.ToInt16(trackBarSpeed.Value);

            return Velocity;
        }
        /// <summary>
        /// Gets the turn on the move sensetivity value, creates a delegeate to the GUI thread if needed
        /// </summary>
        /// <returns></returns>
        private int GetTrackBarDegree()
        {
            int degrees = 0;
            if (this.trackBarDegree.InvokeRequired)
            {
                Invoke(new MethodInvoker(
              delegate
              {
                  degrees = Convert.ToInt16(trackBarDegree.Value);
              }));
            }
            else
                degrees = Convert.ToInt16(trackBarDegree.Value);

            return degrees;
        }
        /// <summary>
        /// returns an array of size 5 with sensor data: distancehigh,distancelow, anglehigh, anglelow, bumpsensor
        /// </summary>
        /// <returns></returns>
        private byte[] ReadAndReturnSensors()
        {
            try
            {
                sp.DiscardOutBuffer();
                sp.DiscardInBuffer();
                //Ask for sensors, distances and angles
                byte[] writeBuffer = new byte[] { 149, 3, 19, 20, 7 };
                sp.Write(writeBuffer, 0, writeBuffer.Length);
                int timeout = 0;
                //Read sensors, distances and angles as requested above

                //wait for 5 bytes to come into the serial port
                while (sp.BytesToRead < 5 && timeout < 10)
                {
                    Thread.Sleep(15);
                    timeout++;
                }
                byte[] readBuffer = new byte[5];
                if (timeout == 10)
                {
                    for (int i = 0; i < 5; i++)
                        readBuffer[i] = 0;
                    return readBuffer;
                }
                //Read 5 bytes from the serial port
                sp.Read(readBuffer, 0, 5);
                //textBox2.Text = System.Text.ASCIIEncoding.GetEncoding(0).GetString↙
    (readBuffer);
                sp.DiscardInBuffer();
                return readBuffer;
            }
            catch (Exception ex)
            {
                return null;
            }
        }
        /// <summary>
        /// This function reads the sensor data and adds it to the log if the top of↙
     the stack is a CommandID (indicating the robot was performing action)
        /// </summary>
        private void ReadAndAddSensorsToLog()
        {
            try
            {
                sp.DiscardOutBuffer();
                sp.DiscardInBuffer();
                //Ask for sensors, distances and angles
                byte[] writeBuffer = new byte[] { 149, 3, 19, 20, 7 };
                sp.Write(writeBuffer, 0, writeBuffer.Length);

                //If the top of the stack is a CommandID, add corresponding sensor  ↙
    data to the stack
                if (DriveLog.IsTopACommandID())
                {
                    int timeout = 0;
                    //Wait for 5 bytes to come into the serial port
                    while (sp.BytesToRead < 5 && timeout < 10)
                    {
                        Thread.Sleep(15);
                        timeout++;
                    }
                    //Read sensors, distances and angles as requested above
                    byte[] readBuffer = new byte[5];
                    if (timeout == 10)
                    {
                        for (int i = 0; i < 5; i++)
                            readBuffer[i] = 0;
                    }
                    else
                        sp.Read(readBuffer, 0, 5);
                    //if not in the GUI thread, create a delegate to grab speed from↙
     the main thread
                    if (trackBarSpeed.InvokeRequired)
                    {
                        Invoke(new MethodInvoker(
                        delegate
                        {
                            DriveLog.InsertSensorData(Convert.ToInt32(readBuffer[0]), Convert.ToInt32(readBuffer[1]), Convert.ToInt32(readBuffer[2]), Convert.ToInt32(readBuffer[3]), Convert.ToInt32(readBuffer[4]), trackBarSpeed.Value);
                        }
                        ));
                    }
                    else
                        DriveLog.InsertSensorData(Convert.ToInt32(readBuffer[0]), Convert.ToInt32(readBuffer[1]), Convert.ToInt32(readBuffer[2]), Convert.ToInt32(readBuffer[3]), Convert.ToInt32(readBuffer[4]), trackBarSpeed.Value);
                }
                //textBox2.Text = System.Text.ASCIIEncoding.GetEncoding(0).GetString(readBuffer);
                sp.DiscardInBuffer();

                UpdateStackGUI();
            }
            catch (Exception ex)
            {
                //MessageBox.Show(ex.Message.ToString());
            }
        }
        /// <summary>
        /// Rotate orientation picture x amount of degrees
        /// </summary>
        /// <param name="degrees"></param>
        private void RotatePicture(int degrees)
        {
            if (this.pictureRoomba.InvokeRequired)
            {
                Invoke(new MethodInvoker(delegate { RotatePicture(degrees); }));
            }
            else
            {
                //make a graphics object from the image
                Image returnImage = pictureRoomba.Image;
                Graphics g = Graphics.FromImage(returnImage);
                //move rotation point to center of image
                g.TranslateTransform((float)returnImage.Width / 2, (float)returnImage.Height / 2);
                //rotate
                g.RotateTransform(degrees);
                //move image back
                g.TranslateTransform(-(float)returnImage.Width / 2, -(float)returnImage.Height / 2);

                //ensure the image preserve high quality
                g.InterpolationMode = System.Drawing.Drawing2D.InterpolationMode.HighQualityBicubic;

                //draw image
                g.DrawImage(returnImage, new Point(0, 0));

                g.Dispose();

                //refresh picture box
                pictureRoomba.Image = returnImage;
                pictureRoomba.Refresh();
            }

        }
        /// <summary>
        /// Starts the 'turn right' or 'turn left' threads if they are not already started
        /// </summary>
        /// <param name="DirectionID">Integer ID representing which direction to turn: 3-Right Turn, 4-Left Turn</param>
        /// <param name="degrees">Degrees to turn</param>
        void StartTurningThread(int DirectionID, int degrees = 0)
        {
            try
            {
                switch (DirectionID)
                {
                    case 3:
                        if (backgroundWorkerRightTurn.IsBusy)
                        {
                            MessageBox.Show("The robot is still turnining, plese wait for the operation to complete");
                            break;
                        }
                        if (degrees != 0)
                        {
                            backgroundWorkerRightTurn.RunWorkerAsync(degrees);
                        }
                        else
                            backgroundWorkerRightTurn.RunWorkerAsync(-90);
                        break;
                    case 4:
                        if (backgroundWorkerLeftTurn.IsBusy)
                        {
                            MessageBox.Show("The robot is still turnining, plese wait for the operation to complete");
                            break;
                        }
                        if (degrees != 0)
                            backgroundWorkerLeftTurn.RunWorkerAsync(degrees);
                        else
                            backgroundWorkerLeftTurn.RunWorkerAsync(90);
                        break;

                }
            }
            catch (Exception ex)
            { MessageBox.Show(ex.Message); }

        }
        /// <summary>
        /// Return the selected velocity of the trackbar * 50, representing the range of 0 to max speed the robot can go.
        /// If not in the GUI thread, creates a delegate to GUI and returns the value
        /// </summary>
        /// <returns></returns>
        private short SelectedVelocity()
        {
            Int16 Velocity = 0;
            if (this.trackBarSpeed.InvokeRequired)
            {
                Invoke(new MethodInvoker(
              delegate
              {
                  Velocity = Convert.ToInt16(50 * trackBarSpeed.Value);
              }));
            }
            else
                Velocity = Convert.ToInt16(50 * trackBarSpeed.Value);

            return Velocity;
        }
        /// <summary>
        /// Used for development, creates a delegate to the GUI to print output in the textBox2
        /// </summary>
        /// <param name="text"></param>
        private void SetText(string text)
        {
            // InvokeRequired required compares the thread ID of the
            // calling thread to the thread ID of the creating thread.
            // If these threads are different, it returns true.
            if (this.textBox2.InvokeRequired)
            {
                ReadDelegate d = new ReadDelegate(SetText);
                this.Invoke(d, new object[] { text });
            }
            else
            {
                this.textBox2.Text = text;
            }
        }
        /// <summary>
        /// Turn right, default 90 degrees, else insert optional degree to turn
        /// </summary>
        /// <param name="degrees"></param>
        private void RightTurn(int degrees = -90)// = -90)
        {
            try
            {
                if (Mutex.WaitOne() == false)
                    return;
                if (TerminateThread)
                    return;
                if (!sp.IsOpen)
                    sp.Open();
                //clear turn angle
                ReadAndAddSensorsToLog();
                //RIGHT TURN

                //Kill the thread if needed
                if (degrees > 0 || TerminateThread)
                    return;

                //RIGHT TURN
                DriveLog.InsertCommand(RightCommandID);
                DriveLog.SetMoving(true);

                //Start moving one wheel while keeping the other still
                byte[] Right = new byte[] { 145, 255, 166, 0, 100 };
                sp.Write(Right, 0, Right.Length);
                sp.DiscardOutBuffer();
                int angle = 0;
                int distance = 0;
                byte[] readBufferByte = new byte[5];
                //Turn while turned less than -90 degrees
                do
                {
                    //If the robot is not moving submit buffer and exit
                    if (DriveLog.IsMoving() == false)
                    {
                        DriveLog.SubmitSensorDataBuffer();
                        UpdateStackGUI();
                        return;
                    }
                    //Read how far we went since last asked, and add it to buffer
```

```
                        readBufferByte = ReadAndReturnSensors();
                        angle = angle + ConvertHighLowToInt(readBufferByte[2], ↙
    readBufferByte[3]);
                        distance = distance + ConvertHighLowToInt(readBufferByte[0], ↙
    readBufferByte[1]);
                        DriveLog.InsertSensorDataIntoBuffer(distance, angle, ↙
    readBufferByte[4], GetTrackBarSpeedValue());
                    } while ((angle > degrees || angle == 0) && !TerminateThread);
                    //Reset thread termination
                    TerminateThread = false;
                    //Stop the robot and submit the buffer
                    byte[] Stop = new byte[] { 137, 0, 0, 0, 0 };
                    sp.Write(Stop, 0, Stop.Length);
                    DriveLog.SetMoving(false);
                    DriveLog.SubmitSensorDataBuffer();
                    UpdateStackGUI();
                    RotatePicture(angle * -1);
                    Mutex.ReleaseMutex();

                }
                catch (Exception ex)
                {
                    byte[] Stop = new byte[] { 137, 0, 0, 0, 0 };
                    sp.Write(Stop, 0, Stop.Length);
                    DriveLog.SetMoving(false);
                    Mutex.ReleaseMutex();
                    MessageBox.Show(ex.Message.ToString());
                }


            }
            /// <summary>
            /// Turn left, default 90 degrees, else insert optional degree to turn
            /// </summary>
            /// <param name="degrees"></param>
            private void LeftTurn(int degrees = 90) //= 90)
            {
                try
                {
                    if (Mutex.WaitOne() == false)
                        return;
                    if (!sp.IsOpen)
                        sp.Open();
                    if (TerminateThread)
                        return;
                    //clear turn angle
                    ReadAndAddSensorsToLog();
                    //RIGHT TURN

                    //Kill the thread if needed
                    if (degrees < 0 || TerminateThread)
                        return;

                    DriveLog.SetMoving(true);
                    DriveLog.InsertCommand(LeftCommandID);
                    //Start moving one wheel while the other stays still
                    byte[] Buff = new byte[] { 145, 0, 100, 255, 156 };
                    sp.Write(Buff, 0, Buff.Length);
                    sp.DiscardInBuffer();
                    int angle = 0;
                    int distance = 0;
                    byte[] readBufferByte = new byte[5];
                    int[] readBufferInt = new int[5];
                    //Keep turning while the angle turned is less than 90
                    do
                    {
```

```
                    //Thread.Sleep(30);

                    //If for some reason the robot is no longer moving, submit and exit
                    if (DriveLog.IsMoving() == false)
                    {
                        DriveLog.SubmitSensorDataBuffer();
                        UpdateStackGUI();
                        return;
                    }
                    //Read and add latest angle turned to the whole angle turned and add to buffer
                    readBufferByte = ReadAndReturnSensors();
                    angle = angle + ConvertHighLowToInt(readBufferByte[2], readBufferByte[3]);
                    distance = distance + ConvertHighLowToInt(readBufferByte[0], readBufferByte[1]);
                    DriveLog.InsertSensorDataIntoBuffer(distance, angle, readBufferByte[4], GetTrackBarSpeedValue());
                } while (angle < degrees && !TerminateThread);
                //Reset thread termination
                TerminateThread = false;
                //Stop the robot
                byte[] Stop = new byte[] { 137, 0, 0, 0, 0 };
                sp.Write(Stop, 0, Stop.Length);
                DriveLog.SetMoving(false);
                DriveLog.SubmitSensorDataBuffer();
                UpdateStackGUI();
                RotatePicture(angle * -1);
                Mutex.ReleaseMutex();
            }
            catch (Exception ex)
            {
                byte[] Stop = new byte[] { 137, 0, 0, 0, 0 };
                sp.Write(Stop, 0, Stop.Length);
                DriveLog.SetMoving(false);
                Mutex.ReleaseMutex();
                MessageBox.Show(ex.Message.ToString() + " " + ex.TargetSite.Name.ToString());
            }
        }
        /// <summary>
        /// Turn right on the move, increases speed of one wheel by 50 and turns x degrees depending on sensetivity track bar
        /// Optionally insert speed at which to do the turn and how many degrees to turn
        /// </summary>
        /// <param name="speed"></param>
        /// <param name="degrees">-10000 says I should grab the value from track bar, else designate one here</param>
        private void RightMoving(int speed = -1, int degrees = -10000)
        {
            //Read velocity, set left wheel to that velocity. Change the velocity of the right wheel to +50 or -50 depending on direction
            try
            {
                if (DriveLog.IsMoving() == false)
                    return;
                if (Mutex.WaitOne() == false)
                    return;
                if (TerminateThread)
                    return;
                if (!sp.IsOpen)
                    sp.Open();

                int direction = DriveLog.GetLastCommandID();
                byte speedLowByteLeft = 0;
                byte speedHighByteLeft = 0;
                int directionAdjustment = 1;
                short velocity = 0;

                //set speed bytes depending on direction for the left wheel
                switch (direction)
                {
                    case 1:
                        //get velocity from param or from the trackBar

                        if (speed != -1)
                            velocity = (short)speed;
                        else
                            velocity = SelectedVelocity();
                         speedLowByteLeft = (byte)(velocity & 0xff);
                         speedHighByteLeft = (byte)((velocity >> 8) & 0xff);
                         directionAdjustment = 1;
                        break;
                    case 2:
                        //get velocity from param or from the trackBar
                        if (speed != -1)
                            velocity = (short)(speed * -1);
                        else
                            velocity = (short)(SelectedVelocity() * -1);
                         speedLowByteLeft = (byte)(velocity & 0xff);
                         speedHighByteLeft = (byte)((velocity >> 8) & 0xff);
                         directionAdjustment = -1;
                         break;
                }
                //Set velocity of the Right wheel to be +50 or -50 so the robot will turn on the move
                byte speedHighByteRight = 0;
                byte speedLowByteRight = 0;
                if (velocity > 0 && velocity + 50 < 32768)
                {
                    velocity = (short)(velocity + 50);
                }
                else if (velocity < 0 && velocity - 50 > -32768)
                {
                    velocity = (short)(velocity - 50);
                }
                speedHighByteRight =  ConvertToHighLow(velocity)[0];
                speedLowByteRight = ConvertToHighLow(velocity)[0];
                //Send a small script: ask for Sensor Data and set new movement so one wheel starts moving 'faster' by 50
                //The script is needed so the time between the last data returned and new movement is as small as possible so as to keep
                //the distances/angles as accurate as possible
                byte[] Buff = new byte[] {152,10, 149, 3, 19, 20, 7, 145, speedHighByteRight, speedLowByteRight, speedHighByteLeft, speedLowByteLeft };
                sp.Write(Buff, 0, Buff.Length);
                byte[] Start = new byte[] { 153 };
                sp.Write(Start, 0, Start.Length);
                //Read sensors/distance that we asked for in the above script, add it to buffer, submit it.
                int timeout = 0;
                while (sp.BytesToRead < 5 && timeout < 10)
                {
                    Thread.Sleep(15);
                    timeout++;
                }
                byte[] readBuffer = new byte[5];
                sp.Read(readBuffer, 0, 5);

                int angle = ConvertHighLowToInt(readBuffer[2], readBuffer[3]);
                int distance = ConvertHighLowToInt(readBuffer[0], readBuffer[1]);
                //Add to buffer and then submit the buffer into stack(add the buffer to the log itself).
                DriveLog.InsertSensorDataIntoBuffer(distance, angle, readBuffer[4], GetTrackBarSpeedValue());
                DriveLog.SubmitSensorDataBuffer();
                //END SUBMITTING OF PREVIOUS MOVEMENT

                //Start Right TURN READ/WAIT LOOP
                DriveLog.InsertCommand(direction);
                if(DriveLog.IsMoving() == false)
                    DriveLog.SetMoving(true);

                Mutex.ReleaseMutex();
                angle = 0;
                distance = 0;
                int turnAngle = 0;

                if (angle != -10000)
                    turnAngle = angle;
                else
                    turnAngle = GetTrackBarDegree();

                byte[] readBufferByte = new byte[5];
                do
                {
                    if (TerminateThread)
                        break;
                    Thread.Sleep(60);
                    if (DriveLog.IsMoving() == false)
                    {
                        DriveLog.SubmitSensorDataBuffer();
                        UpdateStackGUI();
                        return;
                    }
                    if (Mutex.WaitOne() == false)
                        return;
                    readBufferByte = ReadAndReturnSensors();
                    angle = angle + ConvertHighLowToInt(readBufferByte[2], readBufferByte[3]);
                    distance = distance + ConvertHighLowToInt(readBufferByte[0], readBufferByte[1]);
                    //textBox2.Text = textBox2.Text + ":" + angle;

                    DriveLog.InsertSensorDataIntoBuffer(distance, angle, readBufferByte[4], GetTrackBarSpeedValue());
                    Mutex.ReleaseMutex();
                } while ((angle > -1 * turnAngle * directionAdjustment) || angle == 0);
                //FINISH TURNING, START MOVING FORWARD AGAIN
                if (Mutex.WaitOne() == false)
                    return;
                //GET ORIGINAL/NEW SPEED(RESET SPEED OF BOTH WHEELS TO EQUAL TO EACH OTHER)
                if (speed != -1)
                    velocity = (short)speed;
                else
                    velocity = (short)(SelectedVelocity() * directionAdjustment);

                byte velocityHight = ConvertToHighLow(velocity)[0];
                byte velocityLow = ConvertToHighLow(velocity)[1];

                DriveLog.SubmitSensorDataBuffer();
                UpdateStackGUI();
                //START MOVING STRAIGHT
                byte[] Move = new byte[] { 137, velocityHight, velocityLow, 128, 0 }
```

```
;
            sp.Write(Move, 0, Move.Length);
            Thread.Sleep(200);
            ReadAndAddSensorsToLog();
            DriveLog.InsertCommand(direction);
            RotatePicture(angle* -1);
            Mutex.ReleaseMutex();

        }
        catch (Exception ex)
        {
            byte[] Stop = new byte[] { 137, 0, 0, 0, 0 };
            sp.Write(Stop, 0, Stop.Length);
            DriveLog.SetMoving(false);
            Mutex.ReleaseMutex();
            MessageBox.Show(ex.Message.ToString());
        }


    }
    /// <summary>
    /// Turn right on the move, increases speed of one wheel by 50 and turns x ↙
degrees depending on sensetivity track bar
    /// Optionally insert speed at which to do the turn and how many degrees to ↙
turn
    /// </summary>
    /// <param name="speed"></param>
    /// <param name="degrees">-10000 says I should grab the value from track bar↙
, else designate one here</param>
    private void LeftMoving(int speed = -1, int degrees = -10000)
    {
        //MIRRORS Function "RightMoving", REFER TO RightMoving FOR EXTENSIVE    ↙
DOCUMENTATION
        try
        {
            if (DriveLog.IsMoving() == false)
                return;
            if (Mutex.WaitOne() == false)
                return;
            if (!sp.IsOpen)
                sp.Open();
            if (TerminateThread)
                return;
            int direction = DriveLog.GetLastCommandID();
            byte speedLowByteLeft = 0;
            byte speedHighByteLeft = 0;
            byte speedHighByteRight = 0;
            byte speedLowByteRight = 0;
            int directionAdjustment = 1;
            short velocity = 0;
            switch (direction)
            {
                case 1:
                    if (speed != -1)
                        velocity = (short)speed;
                    else
                        velocity = SelectedVelocity();
                    speedLowByteRight = (byte)(velocity & 0xff);
                    speedHighByteRight = (byte)((velocity >> 8) & 0xff);
                    directionAdjustment = 1;
                    break;
                case 2:
                    if (speed != -1)
                        velocity = (short)speed;
                    else
                        velocity = (short)(SelectedVelocity() * -1);
                    speedLowByteRight = (byte)(velocity & 0xff);
                    speedHighByteRight = (byte)((velocity >> 8) & 0xff);
                    directionAdjustment = -1;
                    break;
            }
            //Set the velocity of LEFT WHEEL to go faster than the right

            if (velocity > 0 && velocity + 50 < 32768)
            {
                velocity = (short)(velocity + 50);
            }
            else if (velocity < 0 && velocity - 50 > -32768)
            {
                velocity = (short)(velocity - 50);
            }
            speedHighByteLeft = ConvertToHighLow(velocity)[0];
            speedLowByteLeft = ConvertToHighLow(velocity)[0];
            byte[] Buff = new byte[] { 152, 10, 149, 3, 19, 20, 7, 145,    ↙
speedHighByteRight, speedLowByteRight, speedHighByteLeft, speedLowByteLeft };
            sp.Write(Buff, 0, Buff.Length);
            byte[] Start = new byte[] { 153 };
            sp.Write(Start, 0, Start.Length);

            //Read sensors/distance that we asked for in the above script, add  ↙
it to buffer, submit it.
            int timeout = 0;
            while (sp.BytesToRead < 5 && timeout < 10)
            {
                Thread.Sleep(15);
                timeout++;
            }
            byte[] readBuffer = new byte[5];
            sp.Read(readBuffer, 0, 5);
            int angle = ConvertHighLowToInt(readBuffer[2], readBuffer[3]);
            int distance = ConvertHighLowToInt(readBuffer[0], readBuffer[1]);
            DriveLog.InsertSensorDataIntoBuffer(distance, angle, readBuffer[4], ↙
GetTrackBarSpeedValue());
            DriveLog.SubmitSensorDataBuffer();
            //END SUBMITTING OF PREVIOUS MOVEMENT

            //Start Right TURN READ
            DriveLog.InsertCommand(direction);
            if (DriveLog.IsMoving() == false)
                DriveLog.SetMoving(true);

            Mutex.ReleaseMutex();
            angle = 0;
            distance = 0;

            int turnAngle = 0;

            if (angle != -10000)
                turnAngle = angle;
            else
                turnAngle = GetTrackBarDegree();

            byte[] readBufferByte = new byte[5];
            do
            {
                if (TerminateThread)
                    break;
                Thread.Sleep(60);
                if (DriveLog.IsMoving() == false)
                {
                    DriveLog.SubmitSensorDataBuffer();
                    UpdateStackGUI();
                    return;
                }
                if (Mutex.WaitOne() == false)
                    return;
                readBufferByte = ReadAndReturnSensors();
                angle = angle + ConvertHighLowToInt(readBufferByte[2], readBufferByte[3]);
                distance = distance + ConvertHighLowToInt(readBufferByte[0], readBufferByte[1]);

                DriveLog.InsertSensorDataIntoBuffer(distance, angle, readBufferByte[4], GetTrackBarSpeedValue());
                Mutex.ReleaseMutex();
            } while ((angle < turnAngle * directionAdjustment) || angle == 0);
            //FINISH TURNING, START MOVING STRAIGHT
            if (Mutex.WaitOne() == false)
                return;

            if (speed != -1)
                velocity = (short)speed;
            else
                velocity = (short)(SelectedVelocity() * directionAdjustment);

            byte velocityHight = ConvertToHighLow(velocity)[0];
            byte velocityLow = ConvertToHighLow(velocity)[1];

            DriveLog.SubmitSensorDataBuffer();
            UpdateStackGUI();
            byte[] Move = new byte[] { 137, velocityHight, velocityLow, 128, 0 };
            sp.Write(Move, 0, Move.Length);
            Thread.Sleep(200);
            ReadAndAddSensorsToLog();
            DriveLog.InsertCommand(direction);
            RotatePicture(angle* -1);
            Mutex.ReleaseMutex();

        }
        catch (Exception ex)
        {
            byte[] Stop = new byte[] { 137, 0, 0, 0, 0 };
            sp.Write(Stop, 0, Stop.Length);
            DriveLog.SetMoving(false);
            Mutex.ReleaseMutex();
            MessageBox.Show(ex.Message.ToString());
        }
    }
    /// <summary>
    /// Clears the stack listbox and re-adds all the values from the stack to the stack
    /// </summary>
    private void UpdateStackGUI()
    {
        if (this.listBoxStack.InvokeRequired)
        {
            //listBoxStack.Invoke(UpdateStackGUI());
            Invoke(new MethodInvoker(
           delegate
           {
               int[] stack = DriveLog.GetStack();
               listBoxStack.Items.Clear();
               for (int i = 0; i < DriveLog.GetSize(); i++)
                   listBoxStack.Items.Add(stack[i].ToString());
               if (listBoxStack.Items.Count > 0)
                   listBoxStack.SelectedIndex = listBoxStack.Items.Count - 1;
           }
```

```
                ));
            }
            else
            {
                int[] stack = DriveLog.GetStack();
                listBoxStack.Items.Clear();
                for (int i = 0; i < DriveLog.GetSize(); i++)
                    listBoxStack.Items.Add(stack[i].ToString());
                if (listBoxStack.Items.Count > 0)
                    listBoxStack.SelectedIndex = listBoxStack.Items.Count - 1;
            }
        }
        #endregion
        #region Events
        private void Form1_Load(object sender, EventArgs e)
        {
            SetPortPopUp subForm = new SetPortPopUp(this);
            subForm.Show();
            //Start_CheckToStopThread();
            //Use The Below function to capture arrow-key presses on the main form
            //this.KeyPreview = true;
        }
        private void buttonSendCommand_Click(object sender, EventArgs e)
        {
                try
                {
                    if (!sp.IsOpen)
                        sp.Open();
                    // Write a message into the port.

                    //byte[] Buff = new byte[] {128,131, 152, 17, 137 ,1 ,44 ,128 ,0↙
    ,156 ,1 ,144 ,137 ,1 ,44 ,0 ,1 ,157 ,0 ,90, 153};
                    //byte[] Buff = new byte[] {137, 0, 100, 128, 0};
                    string[] input = textBox1.Text.Split(',');

                    byte[] Buff = new byte[input.Length];

                    for (int i = 0; i < input.Length; i++)
                    {
                        Buff[i] = Convert.ToByte(input[i]);
                    }
                    sp.Write(Buff, 0,Buff.Length);

                    //sp.Close();
                }
                catch (Exception ex)
                {
                    MessageBox.Show(ex.Message.ToString());
                }
        }
        private void buttonBackTrack_Click(object sender, EventArgs e)
        {
                try
                {
                    backgroundWorkerBackTrack.RunWorkerAsync();
                }
                catch (Exception ex)
                {
                    MessageBox.Show(ex.Message.ToString());
                }
        }
        private void button_forward_Click(object sender, EventArgs e)
        {
            DriveForward();
        }
        private void button_stop_Click(object sender, EventArgs e)
        {
            try
            {
                int timeout = 0;
               //If we are in the middle of the turn, begin thread termination.
                if(backgroundWorkerLeftTurn.IsBusy || backgroundWorkerRightTurn.IsBusy)
                {
                    TerminateThread = true;
                    while (backgroundWorkerLeftTurn.IsBusy && backgroundWorkerRightTurn.IsBusy)
                    {
                        if (timeout == 10)
                            break;
                        else
                        {
                            Thread.Sleep(100);
                            timeout++;
                        }
                    }

                    //TerminateThread = false;
                    //If the threads closed correctly exit safely
                    if (timeout < 10)
                    {
                        return;
                    }
                    else
                    {
                        //if the thread did not close correctly, make sure the mutex is released for other threads

                        backgroundWorkerRightTurn.CancelAsync();
                        backgroundWorkerLeftTurn.CancelAsync();
                        //Dirty Way To Release Abandoned Mutex...wait for abandoned mutex error then release mutex
                        try
                        {
                            Mutex.WaitOne(1);
                        }
                        catch (AbandonedMutexException) { };
                        //Mutex.ReleaseMutex();
                    }

                }
                //stop BackTracking
                if (backgroundWorkerBackTrack.IsBusy)
                {
                    timeout = 0;
                    TerminateThread = true;
                    while(backgroundWorkerBackTrack.IsBusy)
                    {
                    if (timeout == 10)
                            break;
                        else
                        {
                            Thread.Sleep(100);
                            timeout++;
                        }
                    }

                    //TerminateThread = false;
                    //If the threads closed correctly exit safely
                    if (timeout < 10)
                    {
                        return;
                    }
                    else
                    {
                        //if the thread did not close correctly, make sure the mutex is released for other threads

                         backgroundWorkerBackTrack.CancelAsync();
                        //Dirty Way To Release Abandoned Mutex...wait for abandoned mutex error then release mutex
                        try
                        {
                            Mutex.WaitOne(1);
                        }
                        catch (AbandonedMutexException) { };
                        //Mutex.ReleaseMutex();
                    }
                    //TerminateThread = true;
                    //Thread.Sleep(1000);
                    //while (backgroundWorkerBackTrack.IsBusy) { Thread.Sleep(50); }
                    //TerminateThread = false;
                    //return;
                }
                if (!sp.IsOpen)
                    sp.Open();
                byte[] Buff = new byte[] { 137, 0, 0, 0, 0 };
                sp.DiscardOutBuffer();
                sp.Write(Buff, 0, Buff.Length);
                DriveLog.SetMoving(false);
                ReadAndAddSensorsToLog();
                TerminateThread = false;
                if (timeout >= 10)
                {
                    //MessageBox.Show("Thread did not terminate correctly, log data may be inaccurate.");
                }
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
            }
            //CollectDistance();
            //byte[] Buff2 = new byte[] { 142, 19 };
            //sp.Write(Buff2, 0, Buff2.Length);
            //sp.Close();
        }
        private void button_backwards_Click(object sender, EventArgs e)
        {
            DriveBackward();
        }
        private void button_left_Click(object sender, EventArgs e)
        {
            StartTurningThread(LeftCommandID);
        }
        private void button_right_Click(object sender, EventArgs e)
        {
            StartTurningThread(RightCommandID);
        }
        private void buttonClear_Click(object sender, EventArgs e)
        {
            DriveLog.ClearLog();
            UpdateStackGUI();
            pictureRoomba.Image = iRobot.Properties.Resources.roomba;
            pictureRoomba.Refresh();
        }
        private void buttonAngleRight_Click(object sender, EventArgs e)
        {
            RightMoving();
        }
        private void buttonLeftAngle_Click(object sender, EventArgs e)
        {
            LeftMoving();
        }
        private void trackBarSpeed_ValueChanged(object sender, EventArgs e)
        {
            if (!DriveLog.IsMoving())
                return;
            else
            {
                if (DriveLog.GetLastCommandID() == 1)
                {
                    ReadAndAddSensorsToLog();
                    DriveForward(GetTrackBarSpeedValue() * 50);
                }
                else if (DriveLog.GetLastCommandID() == 2)
                {
                    ReadAndAddSensorsToLog();
                    DriveBackward(GetTrackBarSpeedValue() * 50);
                }

            }

        }
        //This function is used to capture keyboard input, was never implemented, 
    only tested
        protected override bool ProcessKeyPreview(ref System.Windows.Forms.Message 
    m)
        {
            //textBox2.Text = m.WParam.ToString();
            //switch (m.WParam.ToInt32())
            //{
            //    case 13:
            //        textBox2.Text = "enter";
            //        break;
            //    case 32:
            //        textBox2.Text = "space";
            //        break;
            //    case 37: // <--- left arrow.
            //        textBox2.Text=("you pressed the left arrow!\n");
            //        // do stuff for Left Arrow here.
            //        break;
            //    case 38: // <--- up arrow.
            //        textBox2.Text=("you pressed the up arrow!\n");
            //        // do stuff for Up Arrow here.
            //        break;
            //    case 39: // <--- right arrow.
            //        textBox2.Text=("you pressed the right arrow!\n");
            //        // do stuff for Right Arrow here.
            //        break;
            //    case 40: // <--- down arrow.
            //        textBox2.Text=("you pressed the down arrow!\n");
            //        // do stuff for Down Arrow here.
            //        break;
            //}
            return false;
        }
        private void backgroundWorkerRightTurn_DoWork(object sender, DoWorkEventArgs
     e)
        {
            RightTurn((int)e.Argument);
        }
        private void backgroundWorkerLeftTurn_DoWork(object sender, DoWorkEventArgs 
    e)
```

```
        {
            LeftTurn((int)e.Argument);
        }
        private void backgroundWorkerBackTrack_DoWork_1(object sender, 
      DoWorkEventArgs e)
        {
            BackTrack();
        }
        private void safeModeToolStripMenuItem_Click(object sender, EventArgs e)
        {
            try
            {
                if (!sp.IsOpen)
                    sp.Open();
                byte[] Buff = new byte[] { 128, 131 };
                sp.Write(Buff, 0, Buff.Length);
                Start_CheckToStopThread();
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.ToString());
            }
        }
        private void closePortToolStripMenuItem_Click(object sender, EventArgs e)
        {
            if (sp.IsOpen)
                sp.Close();
        }
        private void toolStripSplitButton1_ButtonClick(object sender, EventArgs e)
        {
            toolStripSplitButton1.ShowDropDown();
        }
        private void terminateBacktrackToolStripMenuItem_Click(object sender, 
      EventArgs e)
        {
            if (backgroundWorkerBackTrack.IsBusy)
                backgroundWorkerBackTrack.CancelAsync();
            try
            {
                Mutex.WaitOne(1);
            }
            catch (AbandonedMutexException) { };
            button_stop.PerformClick();

        }
        private void startBumpSensorThreadToolStripMenuItem_Click(object sender, 
      EventArgs e)
        {
            Start_CheckToStopThread();
        }
        private void setPortToolStripMenuItem_Click(object sender, EventArgs e)
        {
            SetPortPopUp subForm = new SetPortPopUp(this);
            subForm.Show();
        }
       /* private void sp_DataReceived(object sender, SerialDataReceivedEventArgs e)
        {
            string readdata = sp.ReadExisting();
            ASCIIEncoding encoding = new ASCIIEncoding();
            byte[] byte_answer = encoding.GetBytes(readdata);
            SetText(string.Join(",",byte_answer));

            //byte[] byte_buffer = new byte[sp.BytesToRead];
            //sp.Read(byte_buffer, 0, byte_buffer.Length);
            //sp.DiscardInBuffer();
            //for (int i = 0; i < sp.BytesToRead; i++) {
```

```
            //}
            //SetText(string.Join(",", byte_buffer));
        }*/
        #endregion

    }
    public class DriveHistory
    {
        int[] _DriveHistoryStack = new int[5000];
        byte[] _Buffer = new byte[6];
        private int top = 0;
        private int size = 0;
        private int lastcommandId = 0;
        private int lastcommandOffset = 0;
        private bool moving = false;

        /// <summary>
        /// Inserts a CommandID on top of the stack.(Forward 1, Backwards 2, Right 3↙
    , left 4)
        /// </summary>
        /// <param name="command"></param>
        public void InsertCommand(int command)
        {
            _DriveHistoryStack[top] = command;
            lastcommandId = command;
            lastcommandOffset = top;
            top++;
            size++;
        }
        /// <summary>
        /// Adds sensor data to the stack
        /// </summary>
        /// <param name="distancehightByte"></param>
        /// <param name="distancelowByte"></param>
        /// <param name="anglehightByte"></param>
        /// <param name="anglelowByte"></param>
        /// <param name="bumpSensor"></param>
        /// <param name="speed"></param>
        public void InsertSensorData(int distancehightByte,int distancelowByte,int  ↙
    anglehightByte, int anglelowByte, int bumpSensor, int speed)
        {
            if (top == 4993)
                return;
            if (!IsTopACommandID())
            {
                InsertCommand(1);
                top++;
                size++;
            }
            _DriveHistoryStack[top] = distancehightByte;
            top++;
            size++;
            _DriveHistoryStack[top] = distancelowByte;
            top++;
            size++;
            _DriveHistoryStack[top] = anglehightByte;
            top++;
            size++;
            _DriveHistoryStack[top] = anglelowByte;
            top++;
            size++;
            _DriveHistoryStack[top] = bumpSensor;
            top++;
            size++;
            _DriveHistoryStack[top] = speed;
            top++;
            size++;
        }
        /// <summary>
        /// Adds sensor data to an internal buffer. Caution, this function does not ↙
    submit the data to the stack.
        /// </summary>
        /// <param name="distance"></param>
        /// <param name="angle"></param>
        /// <param name="bump"></param>
        /// <param name="speed"></param>
        public void InsertSensorDataIntoBuffer(int distance, int angle, int bump,   ↙
    int speed)
        {
            if (distance >= 32767 || distance <= -257 || angle >= 32767 || angle  <=↙
     -257)
            {
                InsertSensorData(_Buffer[0], _Buffer[1], _Buffer[2], _Buffer[3],    ↙
    _Buffer[4], _Buffer[5]);
                for (int i = 0; i < 5; i++)
                    _Buffer[i] = 0;
                if (IsMoving() == true)
                    InsertCommand(GetLastCommandID());
            }
            byte[] distanceBytes = new byte[2];
            byte[] angleBytes = new byte[2];
            distanceBytes = ConvertToHighLow(distance);
            angleBytes = ConvertToHighLow(angle);
            _Buffer[0] = distanceBytes[0];
            _Buffer[1] = distanceBytes[1];
            _Buffer[2] = angleBytes[0];
            _Buffer[3] = angleBytes[1];
            _Buffer[4] = (byte)bump;
            _Buffer[5] = (byte)speed;
        }
        /// <summary>
        /// Adds the data in the buffer to the log stack
        /// </summary>
        public void SubmitSensorDataBuffer()
        {
            if (lastcommandOffset == top - 1)
            {
                InsertSensorData(_Buffer[0], _Buffer[1], _Buffer[2], _Buffer[3],    ↙
    _Buffer[4], _Buffer[5]);
                for (int i = 0; i < 5; i++)
                    _Buffer[i] = 0;
                //if (IsMoving() == true)
                //    InsertCommand(GetLastCommandID());
            }
        }
        /// <summary>
        /// Returns last command line from the top of the stack of size 7: 0-speed 1↙
    -bump 2-ang.low 3-ang.high 4-dist.low 5-dist.high 6-id
        /// </summary>
        /// <returns></returns>
        public int[] PopLastCommandLine()
        {
            int[] LastCommand = new int[7];
            for (int i = 0; i < 7; i++)
                LastCommand[i] = 0;
            try
            {
                if (top < 6)
                    return LastCommand;
                else
                {
                    for (int i = 0; i < 7; i++)
```

```
                        LastCommand[i] = _DriveHistoryStack[top - 1 - i];
                    top = top - 7;
                    size = size - 7;
                    if (top != 0)
                        lastcommandOffset = top - 1;
                    else
                        lastcommandOffset = top;
                    lastcommandId = _DriveHistoryStack[lastcommandOffset];
                    return LastCommand;
                }
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message.ToString()); return LastCommand;
            };
        }
        public void ClearLog()
        {
            for (int i = 0; i < size; i++)
                _DriveHistoryStack[i] = 0;
            size = 0;
            top = 0;
            lastcommandId = 0;
            lastcommandOffset = 0;
        }
        public int GetTop()
        {
            return top;
        }
        public int GetLastCommandID()
        {
            return lastcommandId;
        }
        public bool IsTopACommandID()
        {
            if (top - 1 == lastcommandOffset && top != 0)
                return true;
            else
                return false;
        }
        public bool IsMoving()
        {
            return moving;
        }
        public void SetMoving(bool isMoving)
        {
            moving = isMoving;
        }
        /// <summary>
        /// Returns the whole stack
        /// </summary>
        /// <returns></returns>
        public int[] GetStack()
        {
            return _DriveHistoryStack;
        }
        public int GetSize()
        {
            return size;
        }
        /// <summary>
        /// Converts an integer to high and low byte, returns byte[] with 0= high and 1 = low
        /// </summary>
        /// <param name="integerValue"></param>
        /// <returns></returns>
        private byte[] ConvertToHighLow(int integerValue)
        {
            byte[] result = new byte[2];
            byte high;
            byte low;
            Int16 original = Convert.ToInt16(integerValue);
            high = Convert.ToByte((original >> 8) & 0xff);
            low = Convert.ToByte(original & 0xff);
            result[0] = high;
            result[1] = low;
            return result;
        }
        /// <summary>
        /// Converts high and low byte to 32 signed int
        /// </summary>
        /// <param name="high"></param>
        /// <param name="low"></param>
        /// <returns></returns>
        private int ConvertHighLowToInt(byte high, byte low)
        {
            return (Int16)(((short)high * (short)256) + (short)low);
        }
    }
}

```