

```
1 //Alexander Popov
2 //St. Norbert College
3 //CS460
4 //March 3, 2011
5 using System;
6 using System.Collections.Generic;
7 using System.ComponentModel;
8 using System.Data;
9 using System.Drawing;
10 using System.Linq;
11 using System.Text;
12 using System.Windows.Forms;
13
14 using System.Threading;
15 using System.IO.Ports;
16
17 namespace iRobot
18 {
19     public delegate void ReadDelegate(string read);
20     public partial class Form1 : Form
21     {
22
23         public Form1()
24         {
25             InitializeComponent();
26
27         }
28         #region Globals
29         DriveHistory DriveLog = new DriveHistory();
30         const int ForwardCommandID = 1;
31         const int BackwardsCommandID = 2;
32         const int RightCommandID = 3;
33         const int LeftCommandID = 4;
34
35         //this variable tells wait loops in turn threads to end so the thread can be
36         terminated, must be set as volatile so all the threads can see it
37         public volatile bool TerminateThread = false;
38         //set up globl Lock Variable
39         System.Threading.Mutex Mutex = new System.Threading.Mutex(false, "mutex");
40         #endregion
41         #region Methods
42         /// <summary>
43         /// Using the stack, turns the robot around and drives him "back" traversing
44         through the stack and executing the commands backwards
45         /// </summary>
46         private void BackTrack()
47         {
48             int[] LastCommand = new int[7];
49             byte[] readBufferByte = new byte[5];
50             int[] reverseLog = new int[4000];
51             byte[] Stop = new byte[] { 137, 0, 0, 0, 0 };
52             int angle = 0;
53             int distance = 0;
54             int LastCommandDistance = 0;
55             int LastCommandAngle = 0;
56             int LastCommandSpeed = 0;
57             int size = DriveLog.GetSize();
58             int z = 0;
59             int j = 0;
60
61             //Copy the current stack in reverse order
62             while (size >= j)
63             {
64                 LastCommand = DriveLog.PopLastCommandLine();
65                 Array.Reverse(LastCommand);
66                 LastCommand.CopyTo(reverseLog, j);
```

```
65         j = j + 7;
66     }
67 }
68
69     //if the TerminateThread command is not issued, turn the robot around 180 degrees
70     if (!TerminateThread)
71     {
72         LeftTurn();
73         LastCommand = DriveLog.PopLastCommandLine();
74     }
75     Thread.Sleep(20);
76     if (!TerminateThread)
77     {
78         LeftTurn();
79         LastCommand = DriveLog.PopLastCommandLine();
80     }
81     //Traverse the reversed stack
82     while (z < size)
83     {
84         if (TerminateThread)
85             break;
86         //Grab the command line from the reversed stack
87         for (int i = 0; i < 7; i++)
88             LastCommand[i] = reverseLog[z + i];
89
90         //Convert high/low bytes to integers
91         LastCommandDistance = ConvertHighLowToInt(Convert.ToByte(LastCommand[1]), Convert.ToByte(LastCommand[2]));
92         LastCommandAngle = ConvertHighLowToInt(Convert.ToByte(LastCommand[3]), Convert.ToByte(LastCommand[4]));
93         LastCommandSpeed = Convert.ToInt32(LastCommand[6]) * 50;
94
95
96
97         distance = 0;
98         angle = 0;
99
100        //Check for CommandID and execute appropriate command
101        switch (LastCommand[0])
102        {
103            case 1:
104                if (LastCommandDistance == 0)
105                    break;
106                if (Mutex.WaitOne() == false)
107                    break;
108                if (LastCommandDistance - 13 > 0)
109                    LastCommandDistance = LastCommandDistance - 13;
110
111                DriveForward(LastCommandSpeed);
112
113                //If the robot turned while moving, perform the turn on the
114                move
115                if (LastCommandAngle > 0)
116                    RightMoving(LastCommandSpeed, LastCommandAngle);
117                else if (LastCommandAngle < 0)
118                    LeftMoving(LastCommandSpeed, LastCommandAngle);
119
120                Mutex.ReleaseMutex();
121                //Stop the robot after it drives 'distance'
122                while (distance < LastCommandDistance)
123                {
124                    if (DriveLog.IsMoving() == false)
125                    {
126                        //DriveLog.SubmitSensorDataBuffer();
127                        UpdateStackGUI();
```

```
127         break;
128     }
129     if (Mutex.WaitOne() == false)
130         break;
131     if (TerminateThread)
132     {
133         UpdateStackGUI();
134         Mutex.ReleaseMutex();
135         break;
136     }
137
138     readBufferByte = ReadAndReturnSensors();
139     angle = angle + ConvertHighLowToInt(readBufferByte[2],
readBufferByte[3]);
140     distance = distance + ConvertHighLowToInt(readBufferByte
[0], readBufferByte[1]);
141
142     Mutex.ReleaseMutex();
143
144     }
145
146     if (Mutex.WaitOne() == false)
147         break;
148     sp.Write(Stop, 0, Stop.Length);
149     ReadAndAddSensorsToLog();
150     DriveLog.SetMoving(false);
151     UpdateStackGUI();
152     Mutex.ReleaseMutex();
153     break;
154     case 2:
155         if (LastCommandDistance == 0)
156             break;
157
158         DriveBackward(LastCommandSpeed);
159
160         //if turned while moving, perform turn on the move
161         if (LastCommandAngle > 0)
162             RightMoving(LastCommandSpeed, LastCommandAngle);
163         else if (LastCommandAngle < 0)
164             LeftMoving(LastCommandSpeed, LastCommandAngle);
165
166         //drive 'distance' backwards
167         do
168         {
169             Thread.Sleep(15);
170             if (DriveLog.IsMoving() == false)
171             {
172                 UpdateStackGUI();
173                 break;
174             }
175             if (Mutex.WaitOne() == false)
176                 break;
177             readBufferByte = ReadAndReturnSensors();
178             angle = angle + ConvertHighLowToInt(readBufferByte[2],
readBufferByte[3]);
179             distance = distance + ConvertHighLowToInt(readBufferByte
[0], readBufferByte[1]);
180         } while (distance > LastCommandDistance);
181         sp.Write(Stop, 0, Stop.Length);
182         ReadAndAddSensorsToLog();
183         DriveLog.SetMoving(false);
184         UpdateStackGUI();
185         Mutex.ReleaseMutex();
186         break;
187         //Turn Right, mirror the turns since you are going backwards
188     case 4:
```

```
189         backgroundWorkerRightTurn.RunWorkerAsync (LastCommandAngle * -1);
190     while (backgroundWorkerRightTurn.IsBusy)
191     {
192         backgroundWorkerRightTurn.CancelAsync();
193         break;
194     } //Left Turn, mirror the turns since you are going backwards
195     case 3:
196         backgroundWorkerLeftTurn.RunWorkerAsync (LastCommandAngle * -1);
197     while (backgroundWorkerLeftTurn.IsBusy)
198     {
199         backgroundWorkerLeftTurn.CancelAsync();
200         break;
201     default:
202         break;
203     }
204     //Move on to the next command which is seven away
205     z = z + 7;
206 }
207 if (TerminateThread)
208 {
209     TerminateThread = false;
210     return;
211 }
212 //Turn robot around after finishing the return
213 LeftTurn();
214 //Remove the turn around command from the current stack
215 LastCommand = DriveLog.PopLastCommandLine();
216 Thread.Sleep(20);
217 LeftTurn();
218 LastCommand = DriveLog.PopLastCommandLine();
219 }
220 /// <summary>
221 /// Converts high and low byte to 32 signed int
222 /// </summary>
223 /// <param name="high"></param>
224 /// <param name="low"></param>
225 /// <returns></returns>
226 public int ConvertHighLowToInt(byte high, byte low)
227 {
228     return (Int16)(((short)high * (short)256) + (short)low);
229 }
230 /// <summary>
231 /// Converts an integer to high and low byte, returns byte[] with 0= high
232 and 1 = low
233 /// </summary>
234 /// <param name="integerValue"></param>
235 /// <returns></returns>
236 public byte[] ConvertToHighLow(int integerValue)
237 {
238     byte[] result = new byte[2];
239     byte high;
240     byte low;
241     Int16 original = Convert.ToInt16(integerValue);
242     high = Convert.ToByte((original >> 8) & 0xff);
243     low = Convert.ToByte(original & 0xff);
244     result[0] = high;
245     result[1] = low;
246     return result;
247 }
248 /// <summary>
249 /// Used in the BumpSensor thread to stop the robot if its moving forward or
250 turning and its sensor is bumped
251 private void BumpSensorCheckToStop()
```

```
251     {
252         do
253         {
254             try
255             {
256                 //If moving forward
257                 if (DriveLog.GetLastCommandID() != 2 && DriveLog.IsMoving())
258                 {
259                     //Ask for Mutex, lock out serial port
260                     if (Mutex.WaitOne(2000) == false)
261                         return;
262                     sp.DiscardOutBuffer();
263                     sp.DiscardInBuffer();
264                     //Ask for the status of the bumper
265                     byte[] Buff = new byte[] { 142, 7 };
266                     sp.Write(Buff, 0, Buff.Length);
267                     int timeout = 0;
268                     //Read the status of the bumper
269                     while (sp.BytesToRead < 1)
270                     {
271                         Thread.Sleep(15);
272                         timeout++;
273                         if (timeout == 67)
274                             break;
275                     }
276                     byte[] readBuffer = new byte[] { 0 };
277                     sp.Read(readBuffer, 0, 1);
278                     byte[] stopBuffer = new byte[] { 137, 0, 0, 0, 0 };
279                     //If the bumper is pressed
280                     if (Convert.ToInt16(readBuffer[0]) > 0 && Convert.ToInt16
281 (readBuffer[0]) < 4)
282                     {
283                         //if we were turning
284                         if (backgroundWorkerLeftTurn.IsBusy ||
backgroundWorkerRightTurn.IsBusy)
285                         {
286                             TerminateThread = true;
287                             while (backgroundWorkerLeftTurn.IsBusy &&
backgroundWorkerRightTurn.IsBusy)
288                             {
289                                 if (timeout == 10)
290                                     break;
291                                 else
292                                 {
293                                     Thread.Sleep(100);
294                                     timeout++;
295                                 }
296                             }
297                             //TerminateThread = false;
298                             //If the threads closed correctly exit safely
299                             if (timeout >= 10)
300                             {
301                                 //if the thread did not close correctly, make
302 sure the mutex is released for other threads
303                                 backgroundWorkerRightTurn.CancelAsync();
304                                 backgroundWorkerLeftTurn.CancelAsync();
305
306                                 //run stop procedure
307                                 sp.Write(stopBuffer, 0, stopBuffer.Length);
308                                 DriveLog.SetMoving(false);
309                                 ReadAndAddSensorsToLog();
310
311                                 //Dirty Way To Release Abandoned Mutex...wait
for abandoned mutex error then release mutex
```

```

312         try
313         {
314             Mutex.WaitOne(1);
315         }
316         catch (AbandonedMutexException) { };
317         //Mutex.ReleaseMutex();
318     }
319
320     }
321     //If backtracking
322     if (backgroundWorkerBackTrack.IsBusy)
323     {
324         TerminateThread = true;
325         Mutex.ReleaseMutex();
326         //Thread.Sleep(100);
327         while (backgroundWorkerBackTrack.IsBusy) { Thread.
Sleep(50); }
328         if (Mutex.WaitOne(2000) == false)
329             return;
330         //If we finished backtracking, stop
331         sp.Write(stopBuffer, 0, stopBuffer.Length);
332         DriveLog.SetMoving(false);
333         ReadAndAddSensorsToLog();
334         TerminateThread = false;
335     }
336     else
337     {
338         sp.Write(stopBuffer, 0, stopBuffer.Length);
339         DriveLog.SetMoving(false);
340         ReadAndAddSensorsToLog();
341     }
342     }
343     Mutex.ReleaseMutex();
344 }
345 }
346 catch (Exception ex)
347 {
348     Mutex.ReleaseMutex();
349     //MessageBox.Show(ex.Message.ToString());
350 }
351 Thread.Sleep(50);
352 } while (true);
353 }
354 /// <summary>
355 /// Starts the BumpSensor thread which stops the robot if the bump sensor is
bumped
356 /// </summary>
357 /// <returns></returns>
358 public bool Start_CheckToStopThread()
359 {
360     try
361     {
362         Thread StopCheckThread = new Thread(new ThreadStart
(BumpSensorCheckToStop));
363         StopCheckThread.IsBackground = true;
364         if (StopCheckThread.IsAlive == true)
365             return false;
366         else
367         {
368             if (sp.IsOpen)
369             {
370                 StopCheckThread.Start();
371                 return true;
372             }
373             else
374             {

```

```
375         sp.Open();
376         StopCheckThread.Start();
377         return true;
378     }
379 }
380 }
381 catch (Exception ex)
382 {
383     MessageBox.Show(ex.Message.ToString());
384     return false;
385 }
386 }
387 private void DriveBackward(int speed = -1)
388 {
389     try
390     {
391         //Ask for Mutex, lock out serial port
392         if (Mutex.WaitOne() == false)
393             return;
394         if (!sp.IsOpen)
395             sp.Open();
396         //Clear movement distance history on the robot
397         ReadAndAddSensorsToLog();
398         DriveLog.InsertCommand(BackwardsCommandID);
399         short velocity = 0;
400         //Get speed from trackBar if needed
401         if (speed != -1)
402             velocity = (short)speed;
403         else
404             velocity = SelectedVelocity();
405         velocity = (short)(velocity * -1);
406         byte lowByte = (byte)(velocity & 0xff);
407         byte highByte = (byte)((velocity >> 8) & 0xff);
408         byte[] Buff = new byte[] { 137, highByte, lowByte, 128, 0 };
409         sp.DiscardOutBuffer();
410         //Start Moving
411         sp.Write(Buff, 0, Buff.Length);
412         DriveLog.SetMoving(true);
413         Mutex.ReleaseMutex();
414     }
415     catch (Exception ex)
416     {
417         Mutex.ReleaseMutex();
418         //MessageBox.Show(ex.Message.ToString());
419     }
420 }
421 private void DriveForward(int speed = -1)
422 {
423     try
424     {
425         //Ask for Mutex(lock out everyone else)
426         if (Mutex.WaitOne() == false)
427             return;
428         if (!sp.IsOpen)
429             sp.Open();
430         //Clear movement distance history on the robot
431         ReadAndAddSensorsToLog();
432         DriveLog.InsertCommand(ForwardCommandID);
433         short velocity = 0;
434         //Get speed if param is not set
435         if (speed != -1)
436             velocity = (short)speed;
437         else
438             velocity = SelectedVelocity();
439         byte lowByte = (byte)(velocity & 0xff);
440         byte highByte = (byte)((velocity >> 8) & 0xff);
```

```
441         byte[] Buff = new byte[] { 137, highByte, lowByte, 128, 0 };
442         sp.DiscardOutBuffer();
443         //Start Moving
444         sp.Write(Buff, 0, Buff.Length);
445         DriveLog.SetMoving(true);
446         Mutex.ReleaseMutex();
447     }
448     catch (Exception ex) { Mutex.ReleaseMutex(); }
449
450     //sp.Close();
451 }
452 /// <summary>
453 /// Returns 0 to 10 value of the trackBar, creates a delegate to the GUI
454 thread if needed
455 /// </summary>
456 /// <returns></returns>
457 private int GetTrackBarSpeedValue()
458 {
459     int Velocity = 0;
460     if (this.trackBarSpeed.InvokeRequired)
461     {
462         Invoke(new MethodInvoker(
463             delegate
464             {
465                 Velocity = Convert.ToInt16(trackBarSpeed.Value);
466             }));
467     }
468     else
469         Velocity = Convert.ToInt16(trackBarSpeed.Value);
470
471     return Velocity;
472 }
473 /// <summary>
474 /// Gets the turn on the move sensetivity value, creates a delegeate to the
475 GUI thread if needed
476 /// </summary>
477 /// <returns></returns>
478 private int GetTrackBarDegree ()
479 {
480     int degrees = 0;
481     if (this.trackBarDegree.InvokeRequired)
482     {
483         Invoke(new MethodInvoker(
484             delegate
485             {
486                 degrees = Convert.ToInt16(trackBarDegree.Value);
487             }));
488     }
489     else
490         degrees = Convert.ToInt16(trackBarDegree.Value);
491
492     return degrees;
493 }
494 /// <summary>
495 /// returns an array of size 5 with sensor data: distancehigh,distancelow,
496 anglehigh, anglelow, bumpsensor
497 /// </summary>
498 /// <returns></returns>
499 private byte[] ReadAndReturnSensors()
500 {
501     try
502     {
503         sp.DiscardOutBuffer();
504         sp.DiscardInBuffer();
505         //Ask for sensors, distances and angles
506         byte[] writeBuffer = new byte[] { 149, 3, 19, 20, 7 };
```



```
504         sp.Write(writeBuffer, 0, writeBuffer.Length);
505         int timeout = 0;
506         //Read sensors, distances and angles as requested above
507
508         //wait for 5 bytes to come into the serial port
509         while (sp.BytesToRead < 5 && timeout < 10)
510         {
511             Thread.Sleep(15);
512             timeout++;
513         }
514         byte[] readBuffer = new byte[5];
515         if (timeout == 10)
516         {
517             for (int i = 0; i < 5; i++)
518                 readBuffer[i] = 0;
519             return readBuffer;
520         }
521         //Read 5 bytes from the serial port
522         sp.Read(readBuffer, 0, 5);
523         //textBox2.Text = System.Text.ASCIIEncoding.GetEncoding(0).GetString
524         (readBuffer);
525         sp.DiscardInBuffer();
526         return readBuffer;
527     }
528     catch (Exception ex)
529     {
530         return null;
531     }
532     /// <summary>
533     /// This function reads the sensor data and adds it to the log if the top of
534     the stack is a CommandID (indicating the robot was performing action)
535     /// </summary>
536     private void ReadAndAddSensorsToLog()
537     {
538         try
539         {
540             sp.DiscardOutBuffer();
541             sp.DiscardInBuffer();
542             //Ask for sensors, distances and angles
543             byte[] writeBuffer = new byte[] { 149, 3, 19, 20, 7 };
544             sp.Write(writeBuffer, 0, writeBuffer.Length);
545
546             //If the top of the stack is a CommandID, add corresponding sensor
547             data to the stack
548             if (DriveLog.IsTopACommandID())
549             {
550                 int timeout = 0;
551                 //Wait for 5 bytes to come into the serial port
552                 while (sp.BytesToRead < 5 && timeout < 10)
553                 {
554                     Thread.Sleep(15);
555                     timeout++;
556                 }
557                 //Read sensors, distances and angles as requested above
558                 byte[] readBuffer = new byte[5];
559                 if (timeout == 10)
560                 {
561                     for (int i = 0; i < 5; i++)
562                         readBuffer[i] = 0;
563                 }
564                 else
565                     sp.Read(readBuffer, 0, 5);
566                 //if not in the GUI thread, create a delegate to grab speed from
567                 the main thread
568                 if (trackBarSpeed.InvokeRequired)
```

```
566         {
567             Invoke(new MethodInvoker(
568                 delegate
569                 {
570                     DriveLog.InsertSensorData(Convert.ToInt32(readBuffer[0])
571 , Convert.ToInt32(readBuffer[1]), Convert.ToInt32(readBuffer[2]), Convert.
572 ToInt32(readBuffer[3]), Convert.ToInt32(readBuffer[4]), trackBarSpeed.Value);
573                 }
574             ));
575         }
576     else
577     {
578         DriveLog.InsertSensorData(Convert.ToInt32(readBuffer[0]),
579 Convert.ToInt32(readBuffer[1]), Convert.ToInt32(readBuffer[2]), Convert.ToInt32
580 (readBuffer[3]), Convert.ToInt32(readBuffer[4]), trackBarSpeed.Value);
581     }
582     //textBox2.Text = System.Text.ASCIIEncoding.GetEncoding(0).GetString
583 (readBuffer);
584     sp.DiscardInBuffer();
585     UpdateStackGUI();
586 }
587 catch (Exception ex)
588 {
589     //MessageBox.Show(ex.Message.ToString());
590 }
591 }
592 /// <summary>
593 /// Rotate orientation picture x amount of degrees
594 /// </summary>
595 /// <param name="degrees"></param>
596 private void RotatePicture(int degrees)
597 {
598     if (this.pictureRoomba.InvokeRequired)
599     {
600         Invoke(new MethodInvoker(delegate { RotatePicture(degrees); }));
601     }
602     else
603     {
604         //make a graphics object from the image
605         Image returnImage = pictureRoomba.Image;
606         Graphics g = Graphics.FromImage(returnImage);
607         //move rotation point to center of image
608         g.TranslateTransform((float)returnImage.Width / 2, (float)
609 returnImage.Height / 2);
610         //rotate
611         g.RotateTransform(degrees);
612         //move image back
613         g.TranslateTransform(-(float)returnImage.Width / 2, -(float)
614 returnImage.Height / 2);
615         //ensure the image preserve high quality
616         g.InterpolationMode = System.Drawing.Drawing2D.InterpolationMode.
617 HighQualityBicubic;
618         //draw image
619         g.DrawImage(returnImage, new Point(0, 0));
620         g.Dispose();
621         //refresh picture box
622         pictureRoomba.Image = returnImage;
623         pictureRoomba.Refresh();
624     }
625 }
626 }
627 /// <summary>
```

```
624     /// Starts the 'turn right' or 'turn left' threads if they are not already started
625     /// </summary>
626     /// <param name="DirectionID">Integer ID representing which direction to turn: 3-Right Turn, 4-Left Turn</param>
627     /// <param name="degrees">Degrees to turn</param>
628     void StartTurningThread(int DirectionID, int degrees = 0)
629     {
630         try
631         {
632             switch (DirectionID)
633             {
634                 case 3:
635                     if (backgroundWorkerRightTurn.IsBusy)
636                     {
637                         MessageBox.Show("The robot is still turning, please wait for the operation to complete");
638                         break;
639                     }
640                     if (degrees != 0)
641                     {
642                         backgroundWorkerRightTurn.RunWorkerAsync(degrees);
643                     }
644                     else
645                         backgroundWorkerRightTurn.RunWorkerAsync(-90);
646                     break;
647                 case 4:
648                     if (backgroundWorkerLeftTurn.IsBusy)
649                     {
650                         MessageBox.Show("The robot is still turning, please wait for the operation to complete");
651                         break;
652                     }
653                     if (degrees != 0)
654                         backgroundWorkerLeftTurn.RunWorkerAsync(degrees);
655                     else
656                         backgroundWorkerLeftTurn.RunWorkerAsync(90);
657                     break;
658             }
659         }
660     }
661     catch (Exception ex)
662     { MessageBox.Show(ex.Message); }
663 }
664
665     /// <summary>
666     /// Return the selected velocity of the trackbar * 50, representing the range of 0 to max speed the robot can go.
667     /// If not in the GUI thread, creates a delegate to GUI and returns the value
668     /// </summary>
669     /// <returns></returns>
670     private short SelectedVelocity()
671     {
672         Int16 Velocity = 0;
673         if (this.trackBarSpeed.InvokeRequired)
674         {
675             Invoke(new MethodInvoker(
676                 delegate
677                 {
678                     Velocity = Convert.ToInt16(50 * trackBarSpeed.Value);
679                 }));
680         }
681         else
682             Velocity = Convert.ToInt16(50 * trackBarSpeed.Value);
683     }
```

```
684         return Velocity;
685     }
686     /// <summary>
687     /// Used for development, creates a delegate to the GUI to print output in
the textBox2
688     /// </summary>
689     /// <param name="text"></param>
690     private void SetText(string text)
691     {
692         // InvokeRequired required compares the thread ID of the
693         // calling thread to the thread ID of the creating thread.
694         // If these threads are different, it returns true.
695         if (this.textBox2.InvokeRequired)
696         {
697             ReadDelegate d = new ReadDelegate(SetText);
698             this.Invoke(d, new object[] { text });
699         }
700         else
701         {
702             this.textBox2.Text = text;
703         }
704     }
705     /// <summary>
706     /// Turn right, default 90 degrees, else insert optional degree to turn
707     /// </summary>
708     /// <param name="degrees"></param>
709     private void RightTurn(int degrees = -90) // = -90)
710     {
711         try
712         {
713             if (Mutex.WaitOne() == false)
714                 return;
715             if (TerminateThread)
716                 return;
717             if (!sp.IsOpen)
718                 sp.Open();
719             //clear turn angle
720             ReadAndAddSensorsToLog();
721             //RIGHT TURN
722
723             //Kill the thread if needed
724             if (degrees > 0 || TerminateThread)
725                 return;
726
727             //RIGHT TURN
728             DriveLog.InsertCommand(RightCommandID);
729             DriveLog.SetMoving(true);
730
731             //Start moving one wheel while keeping the other still
732             byte[] Right = new byte[] { 145, 255, 166, 0, 100 };
733             sp.Write(Right, 0, Right.Length);
734             sp.DiscardOutBuffer();
735             int angle = 0;
736             int distance = 0;
737             byte[] readBufferByte = new byte[5];
738             //Turn while turned less than -90 degrees
739             do
740             {
741                 //If the robot is not moving submit buffer and exit
742                 if (DriveLog.IsMoving() == false)
743                 {
744                     DriveLog.SubmitSensorDataBuffer();
745                     UpdateStackGUI();
746                     return;
747                 }
748                 //Read how far we went since last asked, and add it to buffer
```

```
749         readByteBuffer = ReadAndReturnSensors();
750         angle = angle + ConvertHighLowToInt(readByteBuffer[2],
readByteBuffer[3]);
751         distance = distance + ConvertHighLowToInt(readByteBuffer[0],
readByteBuffer[1]);
752         DriveLog.InsertSensorDataIntoBuffer(distance, angle,
readByteBuffer[4], GetTrackBarSpeedValue());
753     } while ((angle > degrees || angle == 0) && !TerminateThread);
754     //Reset thread termination
755     TerminateThread = false;
756     //Stop the robot and submit the buffer
757     byte[] Stop = new byte[] { 137, 0, 0, 0, 0 };
758     sp.Write(Stop, 0, Stop.Length);
759     DriveLog.SetMoving(false);
760     DriveLog.SubmitSensorDataBuffer();
761     UpdateStackGUI();
762     RotatePicture(angle * -1);
763     Mutex.ReleaseMutex();
764
765 }
766 catch (Exception ex)
767 {
768     byte[] Stop = new byte[] { 137, 0, 0, 0, 0 };
769     sp.Write(Stop, 0, Stop.Length);
770     DriveLog.SetMoving(false);
771     Mutex.ReleaseMutex();
772     MessageBox.Show(ex.Message.ToString());
773 }
774
775
776 }
777 /// <summary>
778 /// Turn left, default 90 degrees, else insert optional degree to turn
779 /// </summary>
780 /// <param name="degrees"></param>
781 private void LeftTurn(int degrees = 90) // = 90
782 {
783     try
784     {
785         if (Mutex.WaitOne() == false)
786             return;
787         if (!sp.IsOpen)
788             sp.Open();
789         if (TerminateThread)
790             return;
791         //clear turn angle
792         ReadAndAddSensorsToLog();
793         //RIGHT TURN
794
795         //Kill the thread if needed
796         if (degrees < 0 || TerminateThread)
797             return;
798
799         DriveLog.SetMoving(true);
800         DriveLog.InsertCommand(LeftCommandID);
801         //Start moving one wheel while the other stays still
802         byte[] Buff = new byte[] { 145, 0, 100, 255, 156 };
803         sp.Write(Buff, 0, Buff.Length);
804         sp.DiscardInBuffer();
805         int angle = 0;
806         int distance = 0;
807         byte[] readByteBuffer = new byte[5];
808         int[] readBufferInt = new int[5];
809         //Keep turning while the angle turned is less than 90
810         do
811         {
```

```

812         //Thread.Sleep(30);
813
814         //If for some reason the robot is no longer moving, submit and
exit
815         if (DriveLog.IsMoving() == false)
816         {
817             DriveLog.SubmitSensorDataBuffer();
818             UpdateStackGUI();
819             return;
820         }
821         //Read and add latest angle turned to the whole angle turned and
add to buffer
822         readByteBuffer = ReadAndReturnSensors();
823         angle = angle + ConvertHighLowToInt(readByteBuffer[2],
readByteBuffer[3]);
824         distance = distance + ConvertHighLowToInt(readByteBuffer[0],
readByteBuffer[1]);
825         DriveLog.InsertSensorDataIntoBuffer(distance, angle,
readByteBuffer[4], GetTrackBarSpeedValue());
826     } while (angle < degrees && !TerminateThread);
827     //Reset thread termination
828     TerminateThread = false;
829     //Stop the robot
830     byte[] Stop = new byte[] { 137, 0, 0, 0, 0 };
831     sp.Write(Stop, 0, Stop.Length);
832     DriveLog.SetMoving(false);
833     DriveLog.SubmitSensorDataBuffer();
834     UpdateStackGUI();
835     RotatePicture(angle * -1);
836     Mutex.ReleaseMutex();
837 }
838 catch (Exception ex)
839 {
840     byte[] Stop = new byte[] { 137, 0, 0, 0, 0 };
841     sp.Write(Stop, 0, Stop.Length);
842     DriveLog.SetMoving(false);
843     Mutex.ReleaseMutex();
844     MessageBox.Show(ex.Message.ToString() + " " + ex.TargetSite.Name.
ToString());
845 }
846 }
847     /// <summary>
848     /// Turn right on the move, increases speed of one wheel by 50 and turns x
degrees depending on sensetivity track bar
849     /// Optionally insert speed at which to do the turn and how many degrees to
turn
850     /// </summary>
851     /// <param name="speed"></param>
852     /// <param name="degrees">-10000 says I should grab the value from track bar
, else designate one here</param>
853     private void RightMoving(int speed = -1, int degrees = -10000)
854     {
855         //Read velocity, set left wheel to that velocity. Change the velocity of
the right wheel to +50 or -50 depending on direction
856         try
857         {
858             if (DriveLog.IsMoving() == false)
859                 return;
860             if (Mutex.WaitOne() == false)
861                 return;
862             if (TerminateThread)
863                 return;
864             if (!sp.IsOpen)
865                 sp.Open();
866
867             int direction = DriveLog.GetLastCommandID();

```

```

868         byte speedLowByteLeft = 0;
869         byte speedHighByteLeft = 0;
870         int directionAdjustment = 1;
871         short velocity = 0;
872
873         //set speed bytes depending on direction for the left wheel
874         switch (direction)
875         {
876             case 1:
877                 //get velocity from param or from the trackBar
878
879                 if (speed != -1)
880                     velocity = (short) speed;
881                 else
882                     velocity = SelectedVelocity();
883                 speedLowByteLeft = (byte) (velocity & 0xff);
884                 speedHighByteLeft = (byte) ((velocity >> 8) & 0xff);
885                 directionAdjustment = 1;
886                 break;
887             case 2:
888                 //get velocity from param or from the trackBar
889                 if (speed != -1)
890                     velocity = (short) (speed * -1);
891                 else
892                     velocity = (short) (SelectedVelocity() * -1);
893                 speedLowByteLeft = (byte) (velocity & 0xff);
894                 speedHighByteLeft = (byte) ((velocity >> 8) & 0xff);
895                 directionAdjustment = -1;
896                 break;
897         }
898         //Set velocity of the Right wheel to be +50 or -50 so the robot will
899         turn on the move
900         byte speedHighByteRight = 0;
901         byte speedLowByteRight = 0;
902         if (velocity > 0 && velocity + 50 < 32768)
903         {
904             velocity = (short) (velocity + 50);
905         }
906         else if (velocity < 0 && velocity - 50 > -32768)
907         {
908             velocity = (short) (velocity - 50);
909         }
910         speedHighByteRight = ConvertToHighLow(velocity)[0];
911         speedLowByteRight = ConvertToHighLow(velocity)[1];
912         //Send a small script: ask for Sensor Data and set new movement so
913         one wheel starts moving 'faster' by 50
914         //The script is needed so the time between the last data returned
915         and new movement is as small as possible so as to keep
916         //the distances/angles as accurate as possible
917         byte[] Buff = new byte[] { 152, 10, 149, 3, 19, 20, 7, 145,
918         speedHighByteRight, speedLowByteRight, speedHighByteLeft, speedLowByteLeft };
919         sp.Write(Buff, 0, Buff.Length);
920         byte[] Start = new byte[] { 153 };
921         sp.Write(Start, 0, Start.Length);
922         //Read sensors/distance that we asked for in the above script, add
923         it to buffer, submit it.
924         int timeout = 0;
925         while (sp.BytesToRead < 5 && timeout < 10)
926         {
927             Thread.Sleep(15);
928             timeout++;
929         }
930         byte[] readBuffer = new byte[5];
931         sp.Read(readBuffer, 0, 5);
932         int angle = ConvertHighLowToInt(readBuffer[2], readBuffer[3]);

```

```

928         int distance = ConvertHighLowToInt(readBuffer[0], readBuffer[1]);
929         //Add to buffer and then submit the buffer into stack(add the buffer
to the log itself).
930         DriveLog.InsertSensorDataIntoBuffer(distance, angle, readBuffer[4],
GetTrackBarSpeedValue());
931         DriveLog.SubmitSensorDataBuffer();
932         //END SUBMITTING OF PREVIOUS MOVEMENT
933
934         //Start Right TURN READ/WAIT LOOP
935         DriveLog.InsertCommand(direction);
936         if(DriveLog.IsMoving() == false)
937             DriveLog.SetMoving(true);
938
939         Mutex.ReleaseMutex();
940         angle = 0;
941         distance = 0;
942         int turnAngle = 0;
943
944         if (angle != -10000)
945             turnAngle = angle;
946         else
947             turnAngle = GetTrackBarDegree();
948
949         byte[] readBufferByte = new byte[5];
950         do
951         {
952             if (TerminateThread)
953                 break;
954             Thread.Sleep(60);
955             if (DriveLog.IsMoving() == false)
956             {
957                 DriveLog.SubmitSensorDataBuffer();
958                 UpdateStackGUI();
959                 return;
960             }
961             if (Mutex.WaitOne() == false)
962                 return;
963             readBufferByte = ReadAndReturnSensors();
964             angle = angle + ConvertHighLowToInt(readBufferByte[2],
readBufferByte[3]);
965             distance = distance + ConvertHighLowToInt(readBufferByte[0],
readBufferByte[1]);
966             //textBox2.Text = textBox2.Text + ":" + angle;
967
968             DriveLog.InsertSensorDataIntoBuffer(distance, angle,
readBufferByte[4], GetTrackBarSpeedValue());
969             Mutex.ReleaseMutex();
970             } while ((angle > -1 * turnAngle * directionAdjustment) || angle ==
0);
971         //FINISH TURNING, START MOVING FORWARD AGAIN
972         if (Mutex.WaitOne() == false)
973             return;
974         //GET ORIGINAL/NEW SPEED(RESET SPEED OF BOTH WHEELS TO EQUAL TO EACH
OTHER)
975         if (speed != -1)
976             velocity = (short)speed;
977         else
978             velocity = (short)(SelectedVelocity() * directionAdjustment);
979
980         byte velocityHight = ConvertToHighLow(velocity)[0];
981         byte velocityLow = ConvertToHighLow(velocity)[1];
982
983         DriveLog.SubmitSensorDataBuffer();
984         UpdateStackGUI();
985         //START MOVING STRAIGHT
986         byte[] Move = new byte[] { 137, velocityHight, velocityLow, 128, 0 }

```



```

;
987         sp.Write(Move, 0, Move.Length);
988         Thread.Sleep(200);
989         ReadAndAddSensorsToLog();
990         DriveLog.InsertCommand(direction);
991         RotatePicture(angle* -1);
992         Mutex.ReleaseMutex();
993     }
994 }
995 catch (Exception ex)
996 {
997     byte[] Stop = new byte[] { 137, 0, 0, 0, 0 };
998     sp.Write(Stop, 0, Stop.Length);
999     DriveLog.SetMoving(false);
1000    Mutex.ReleaseMutex();
1001    MessageBox.Show(ex.Message.ToString());
1002 }
1003
1004 }
1005 }
1006     /// <summary>
1007     /// Turn right on the move, increases speed of one wheel by 50 and turns x
1008     degrees depending on sensetivity track bar
1009     /// Optionally insert speed at which to do the turn and how many degrees to
1010     turn
1011     /// </summary>
1012     /// <param name="speed"></param>
1013     /// <param name="degrees">-10000 says I should grab the value from track bar
1014     , else designate one here</param>
1015     private void LeftMoving(int speed = -1, int degrees = -10000)
1016     {
1017         //MIRRORS Function "RightMoving", REFER TO RightMoving FOR EXTENSIVE
1018         DOCUMENTATION
1019         try
1020         {
1021             if (DriveLog.IsMoving() == false)
1022                 return;
1023             if (Mutex.WaitOne() == false)
1024                 return;
1025             if (!sp.IsOpen)
1026                 sp.Open();
1027             if (TerminateThread)
1028                 return;
1029             int direction = DriveLog.GetLastCommandID();
1030             byte speedLowByteLeft = 0;
1031             byte speedHighByteLeft = 0;
1032             byte speedHighByteRight = 0;
1033             byte speedLowByteRight = 0;
1034             int directionAdjustment = 1;
1035             short velocity = 0;
1036             switch (direction)
1037             {
1038                 case 1:
1039                     if (speed != -1)
1040                         velocity = (short) speed;
1041                     else
1042                         velocity = SelectedVelocity();
1043                     speedLowByteRight = (byte) (velocity & 0xff);
1044                     speedHighByteRight = (byte) ((velocity >> 8) & 0xff);
1045                     directionAdjustment = 1;
1046                     break;
1047                 case 2:
1048                     if (speed != -1)
1049                         velocity = (short) speed;
1050                     else
1051                         velocity = (short) (SelectedVelocity() * -1);

```

```
1048         speedLowByteRight = (byte)(velocity & 0xff);
1049         speedHighByteRight = (byte)((velocity >> 8) & 0xff);
1050         directionAdjustment = -1;
1051         break;
1052     }
1053     //Set the velocity of LEFT WHEEL to go faster than the right
1054
1055     if (velocity > 0 && velocity + 50 < 32768)
1056     {
1057         velocity = (short)(velocity + 50);
1058     }
1059     else if (velocity < 0 && velocity - 50 > -32768)
1060     {
1061         velocity = (short)(velocity - 50);
1062     }
1063     speedHighByteLeft = ConvertToHighLow(velocity)[0];
1064     speedLowByteLeft = ConvertToHighLow(velocity)[0];
1065     byte[] Buff = new byte[] { 152, 10, 149, 3, 19, 20, 7, 145,
speedHighByteRight, speedLowByteRight, speedHighByteLeft, speedLowByteLeft };
1066     sp.Write(Buff, 0, Buff.Length);
1067     byte[] Start = new byte[] { 153 };
1068     sp.Write(Start, 0, Start.Length);
1069
1070     //Read sensors/distance that we asked for in the above script, add
it to buffer, submit it.
1071     int timeout = 0;
1072     while (sp.BytesToRead < 5 && timeout < 10)
1073     {
1074         Thread.Sleep(15);
1075         timeout++;
1076     }
1077     byte[] readBuffer = new byte[5];
1078     sp.Read(readBuffer, 0, 5);
1079     int angle = ConvertHighLowToInt(readBuffer[2], readBuffer[3]);
1080     int distance = ConvertHighLowToInt(readBuffer[0], readBuffer[1]);
1081     DriveLog.InsertSensorDataIntoBuffer(distance, angle, readBuffer[4],
GetTrackBarSpeedValue());
1082     DriveLog.SubmitSensorDataBuffer();
1083     //END SUBMITTING OF PREVIOUS MOVEMENT
1084
1085     //Start Right TURN READ
1086     DriveLog.InsertCommand(direction);
1087     if (DriveLog.IsMoving() == false)
1088         DriveLog.SetMoving(true);
1089
1090     Mutex.ReleaseMutex();
1091     angle = 0;
1092     distance = 0;
1093
1094     int turnAngle = 0;
1095
1096     if (angle != -10000)
1097         turnAngle = angle;
1098     else
1099         turnAngle = GetTrackBarDegree();
1100
1101     byte[] readBufferByte = new byte[5];
1102     do
1103     {
1104         if (TerminateThread)
1105             break;
1106         Thread.Sleep(60);
1107         if (DriveLog.IsMoving() == false)
1108         {
1109             DriveLog.SubmitSensorDataBuffer();
1110             UpdateStackGUI();
```

```

1111         return;
1112     }
1113     if (Mutex.WaitOne() == false)
1114         return;
1115     readBufferByte = ReadAndReturnSensors();
1116     angle = angle + ConvertHighLowToInt(readBufferByte[2],
readBufferByte[3]);
1117     distance = distance + ConvertHighLowToInt(readBufferByte[0],
readBufferByte[1]);
1118
1119     DriveLog.InsertSensorDataIntoBuffer(distance, angle,
readBufferByte[4], GetTrackBarSpeedValue());
1120     Mutex.ReleaseMutex();
1121     } while ((angle < turnAngle * directionAdjustment) || angle == 0);
1122     //FINISH TURNING, START MOVING STRAIGHT
1123     if (Mutex.WaitOne() == false)
1124         return;
1125
1126     if (speed != -1)
1127         velocity = (short) speed;
1128     else
1129         velocity = (short) (SelectedVelocity() * directionAdjustment);
1130
1131     byte velocityHight = ConvertToHighLow(velocity)[0];
1132     byte velocityLow = ConvertToHighLow(velocity)[1];
1133
1134     DriveLog.SubmitSensorDataBuffer();
1135     UpdateStackGUI();
1136     byte[] Move = new byte[] { 137, velocityHight, velocityLow, 128, 0 };
;
1137     sp.Write(Move, 0, Move.Length);
1138     Thread.Sleep(200);
1139     ReadAndAddSensorsToLog();
1140     DriveLog.InsertCommand(direction);
1141     RotatePicture(angle* -1);
1142     Mutex.ReleaseMutex();
1143
1144     }
1145     catch (Exception ex)
1146     {
1147         byte[] Stop = new byte[] { 137, 0, 0, 0, 0 };
1148         sp.Write(Stop, 0, Stop.Length);
1149         DriveLog.SetMoving(false);
1150         Mutex.ReleaseMutex();
1151         MessageBox.Show(ex.Message.ToString());
1152     }
1153     }
1154     /// <summary>
1155     the stack
1156     /// </summary>
1157     private void UpdateStackGUI()
1158     {
1159         if (this.listBoxStack.InvokeRequired)
1160         {
1161             //listBoxStack.Invoke(UpdateStackGUI());
1162             Invoke(new MethodInvoker(
1163                 delegate
1164                 {
1165                     int[] stack = DriveLog.GetStack();
1166                     listBoxStack.Items.Clear();
1167                     for (int i = 0; i < DriveLog.GetSize(); i++)
1168                         listBoxStack.Items.Add(stack[i].ToString());
1169                     if (listBoxStack.Items.Count > 0)
1170                         listBoxStack.SelectedIndex = listBoxStack.Items.Count - 1;
1171                 }

```

```
1172         });
1173     }
1174     else
1175     {
1176         int[] stack = DriveLog.GetStack();
1177         listBoxStack.Items.Clear();
1178         for (int i = 0; i < DriveLog.GetSize(); i++)
1179             listBoxStack.Items.Add(stack[i].ToString());
1180         if (listBoxStack.Items.Count > 0)
1181             listBoxStack.SelectedIndex = listBoxStack.Items.Count - 1;
1182     }
1183 }
1184 #endregion
1185 #region Events
1186 private void Form1_Load(object sender, EventArgs e)
1187 {
1188     SetPortPopUp subForm = new SetPortPopUp(this);
1189     subForm.Show();
1190     //Start_CheckToStopThread();
1191     //Use The Below function to capture arrow-key presses on the main form
1192     //this.KeyPreview = true;
1193 }
1194 private void buttonSendCommand_Click(object sender, EventArgs e)
1195 {
1196     try
1197     {
1198         if (!sp.IsOpen)
1199             sp.Open();
1200         // Write a message into the port.
1201
1202         //byte[] Buff = new byte[] {128,131, 152, 17, 137 ,1 ,44 ,128 ,0
1203         ,156 ,1 ,144 ,137 ,1 ,44 ,0 ,1 ,157 ,0 ,90, 153};
1204         //byte[] Buff = new byte[] {137, 0, 100, 128, 0};
1205         string[] input = textBox1.Text.Split(',');
1206
1207         byte[] Buff = new byte[input.Length];
1208
1209         for (int i = 0; i < input.Length; i++)
1210         {
1211             Buff[i] = Convert.ToByte(input[i]);
1212         }
1213         sp.Write(Buff, 0, Buff.Length);
1214
1215         //sp.Close();
1216     }
1217     catch (Exception ex)
1218     {
1219         MessageBox.Show(ex.Message.ToString());
1220     }
1221 }
1222 private void buttonBackTrack_Click(object sender, EventArgs e)
1223 {
1224     try
1225     {
1226         backgroundWorkerBackTrack.RunWorkerAsync();
1227     }
1228     catch (Exception ex)
1229     {
1230         MessageBox.Show(ex.Message.ToString());
1231     }
1232 }
1233 private void button_forward_Click(object sender, EventArgs e)
1234 {
1235     DriveForward();
1236 }
1237 private void button_stop_Click(object sender, EventArgs e)
```

```
1237     {
1238         try
1239         {
1240             int timeout = 0;
1241             //If we are in the middle of the turn, begin thread termination.
1242             if(backgroundWorkerLeftTurn.IsBusy || backgroundWorkerRightTurn.
IsBusy)
1243             {
1244                 TerminateThread = true;
1245                 while (backgroundWorkerLeftTurn.IsBusy &&
backgroundWorkerRightTurn.IsBusy)
1246                 {
1247                     if (timeout == 10)
1248                         break;
1249                     else
1250                     {
1251                         Thread.Sleep(100);
1252                         timeout++;
1253                     }
1254                 }
1255
1256                 //TerminateThread = false;
1257                 //If the threads closed correctly exit safely
1258                 if (timeout < 10)
1259                 {
1260                     return;
1261                 }
1262                 else
1263                 {
1264                     //if the thread did not close correctly, make sure the mutex
is released for other threads
1265
1266                     backgroundWorkerRightTurn.CancelAsync();
1267                     backgroundWorkerLeftTurn.CancelAsync();
1268                     //Dirty Way To Release Abandoned Mutex...wait for abandoned
mutex error then release mutex
1269                     try
1270                     {
1271                         Mutex.WaitOne(1);
1272                     }
1273                     catch (AbandonedMutexException) { };
1274                     //Mutex.ReleaseMutex();
1275                 }
1276             }
1277
1278             //stop BackTracking
1279             if (backgroundWorkerBackTrack.IsBusy)
1280             {
1281                 timeout = 0;
1282                 TerminateThread = true;
1283                 while(backgroundWorkerBackTrack.IsBusy)
1284                 {
1285                     if (timeout == 10)
1286                         break;
1287                     else
1288                     {
1289                         Thread.Sleep(100);
1290                         timeout++;
1291                     }
1292                 }
1293
1294                 //TerminateThread = false;
1295                 //If the threads closed correctly exit safely
1296                 if (timeout < 10)
1297                 {
1298                     return;

```

```
1299         }
1300         else
1301         {
1302             //if the thread did not close correctly, make sure the mutex
is released for other threads
1303
1304             backgroundWorkerBackTrack.CancelAsync();
1305             //Dirty Way To Release Abandoned Mutex...wait for abandoned
mutex error then release mutex
1306             try
1307             {
1308                 Mutex.WaitOne(1);
1309             }
1310             catch (AbandonedMutexException) { };
1311             //Mutex.ReleaseMutex();
1312         }
1313         //TerminateThread = true;
1314         //Thread.Sleep(1000);
1315         //while (backgroundWorkerBackTrack.IsBusy) { Thread.Sleep(50); }
1316         //TerminateThread = false;
1317         //return;
1318     }
1319     if (!sp.IsOpen)
1320         sp.Open();
1321     byte[] Buff = new byte[] { 137, 0, 0, 0, 0 };
1322     sp.DiscardOutBuffer();
1323     sp.Write(Buff, 0, Buff.Length);
1324     DriveLog.SetMoving(false);
1325     ReadAndAddSensorsToLog();
1326     TerminateThread = false;
1327     if (timeout >= 10)
1328     {
1329         //MessageBox.Show("Thread did not terminate correctly, log data
may be inaccurate.");
1330     }
1331     }
1332     catch (Exception ex)
1333     {
1334         MessageBox.Show(ex.Message);
1335     }
1336     //CollectDistance();
1337     //byte[] Buff2 = new byte[] { 142, 19 };
1338     //sp.Write(Buff2, 0, Buff2.Length);
1339     //sp.Close();
1340 }
1341 private void button_backwards_Click(object sender, EventArgs e)
1342 {
1343     DriveBackward();
1344 }
1345 private void button_left_Click(object sender, EventArgs e)
1346 {
1347     StartTurningThread(LeftCommandID);
1348 }
1349 private void button_right_Click(object sender, EventArgs e)
1350 {
1351     StartTurningThread(RightCommandID);
1352 }
1353 private void buttonClear_Click(object sender, EventArgs e)
1354 {
1355     DriveLog.ClearLog();
1356     UpdateStackGUI();
1357     pictureRoomba.Image = iRobot.Properties.Resources.roomba;
1358     pictureRoomba.Refresh();
1359 }
1360 private void buttonAngleRight_Click(object sender, EventArgs e)
1361 {
```

```
1362         RightMoving();
1363     }
1364     private void buttonLeftAngle_Click(object sender, EventArgs e)
1365     {
1366         LeftMoving();
1367     }
1368     private void trackBarSpeed_ValueChanged(object sender, EventArgs e)
1369     {
1370         if (!DriveLog.IsMoving())
1371             return;
1372         else
1373         {
1374             if (DriveLog.GetLastCommandID() == 1)
1375             {
1376                 ReadAndAddSensorsToLog();
1377                 DriveForward(GetTrackBarSpeedValue() * 50);
1378             }
1379             else if (DriveLog.GetLastCommandID() == 2)
1380             {
1381                 ReadAndAddSensorsToLog();
1382                 DriveBackward(GetTrackBarSpeedValue() * 50);
1383             }
1384         }
1385     }
1386 }
1387 //This function is used to capture keyboard input, was never implemented,
1388 only tested
1389 protected override bool ProcessKeyPreview(ref System.Windows.Forms.Message
m)
1390 {
1391     //textBox2.Text = m.WParam.ToString();
1392     //switch (m.WParam.ToInt32())
1393     //{
1394     //    case 13:
1395     //        textBox2.Text = "enter";
1396     //        break;
1397     //    case 32:
1398     //        textBox2.Text = "space";
1399     //        break;
1400     //    case 37: // <--- left arrow.
1401     //        textBox2.Text="you pressed the left arrow!\n";
1402     //        // do stuff for Left Arrow here.
1403     //        break;
1404     //    case 38: // <--- up arrow.
1405     //        textBox2.Text="you pressed the up arrow!\n";
1406     //        // do stuff for Up Arrow here.
1407     //        break;
1408     //    case 39: // <--- right arrow.
1409     //        textBox2.Text="you pressed the right arrow!\n";
1410     //        // do stuff for Right Arrow here.
1411     //        break;
1412     //    case 40: // <--- down arrow.
1413     //        textBox2.Text="you pressed the down arrow!\n";
1414     //        // do stuff for Down Arrow here.
1415     //        break;
1416     //}
1417     return false;
1418 }
1419 private void backgroundWorkerRightTurn_DoWork(object sender, DoWorkEventArgs
e)
1420 {
1421     RightTurn((int)e.Argument);
1422 }
1423 private void backgroundWorkerLeftTurn_DoWork(object sender, DoWorkEventArgs
e)
```

```
1424     {
1425         LeftTurn((int)e.Argument);
1426     }
1427     private void backgroundWorkerBackTrack_DoWork_1(object sender,
DoWorkEventArgs e)
1428     {
1429         BackTrack();
1430     }
1431     private void safeModeToolStripMenuItem_Click(object sender, EventArgs e)
1432     {
1433         try
1434         {
1435             if (!sp.IsOpen)
1436                 sp.Open();
1437             byte[] Buff = new byte[] { 128, 131 };
1438             sp.Write(Buff, 0, Buff.Length);
1439             Start_CheckToStopThread();
1440         }
1441         catch (Exception ex)
1442         {
1443             MessageBox.Show(ex.ToString());
1444         }
1445     }
1446     private void closePortToolStripMenuItem_Click(object sender, EventArgs e)
1447     {
1448         if (sp.IsOpen)
1449             sp.Close();
1450     }
1451     private void toolStripSplitButton1_ButtonClick(object sender, EventArgs e)
1452     {
1453         toolStripSplitButton1.ShowDropDown();
1454     }
1455     private void terminateBacktrackToolStripMenuItem_Click(object sender,
EventArgs e)
1456     {
1457         if (backgroundWorkerBackTrack.IsBusy)
1458             backgroundWorkerBackTrack.CancelAsync();
1459         try
1460         {
1461             Mutex.WaitOne(1);
1462         }
1463         catch (AbandonedMutexException) { };
1464         button_stop.PerformClick();
1465     }
1466     private void startBumpSensorThreadToolStripMenuItem_Click(object sender,
EventArgs e)
1467     {
1468         Start_CheckToStopThread();
1469     }
1470     private void setPortToolStripMenuItem_Click(object sender, EventArgs e)
1471     {
1472         SetPortPopUp subForm = new SetPortPopUp(this);
1473         subForm.Show();
1474     }
1475     /* private void sp_DataReceived(object sender, SerialDataReceivedEventArgs e)
1476     {
1477         string readdata = sp.ReadExisting();
1478         ASCIIEncoding encoding = new ASCIIEncoding();
1479         byte[] byte_answer = encoding.GetBytes(readdata);
1480         SetText(string.Join(",",byte_answer));
1481     }
1482     //byte[] byte_buffer = new byte[sp.BytesToRead];
1483     //sp.Read(byte_buffer, 0, byte_buffer.Length);
1484     //sp.DiscardInBuffer();
1485     //for (int i = 0; i < sp.BytesToRead; i++) {
```



```
1487         //}
1488         //SetText(string.Join(",", byte_buffer));
1489     }*/
1490     #endregion
1491
1492 }
1493 public class DriveHistory
1494 {
1495     int[] _DriveHistoryStack = new int[5000];
1496     byte[] _Buffer = new byte[6];
1497     private int top = 0;
1498     private int size = 0;
1499     private int lastcommandId = 0;
1500     private int lastcommandOffset = 0;
1501     private bool moving = false;
1502
1503     /// <summary>
1504     /// Inserts a CommandID on top of the stack.(Forward 1, Backwards 2, Right 3
1505     , left 4)
1506     /// </summary>
1507     /// <param name="command"></param>
1508     public void InsertCommand(int command)
1509     {
1510         _DriveHistoryStack[top] = command;
1511         lastcommandId = command;
1512         lastcommandOffset = top;
1513         top++;
1514         size++;
1515     }
1516     /// <summary>
1517     /// Adds sensor data to the stack
1518     /// </summary>
1519     /// <param name="distancehightByte"></param>
1520     /// <param name="distancelowByte"></param>
1521     /// <param name="anglehightByte"></param>
1522     /// <param name="anglelowByte"></param>
1523     /// <param name="bumpSensor"></param>
1524     /// <param name="speed"></param>
1525     public void InsertSensorData(int distancehightByte, int distancelowByte, int
1526     anglehightByte, int anglelowByte, int bumpSensor, int speed)
1527     {
1528         if (top == 4993)
1529             return;
1530         if (!IsTopACommandID())
1531         {
1532             InsertCommand(1);
1533             top++;
1534             size++;
1535         }
1536         _DriveHistoryStack[top] = distancehightByte;
1537         top++;
1538         size++;
1539         _DriveHistoryStack[top] = distancelowByte;
1540         top++;
1541         size++;
1542         _DriveHistoryStack[top] = anglehightByte;
1543         top++;
1544         size++;
1545         _DriveHistoryStack[top] = anglelowByte;
1546         top++;
1547         size++;
1548         _DriveHistoryStack[top] = bumpSensor;
1549         top++;
1550         size++;
1551         _DriveHistoryStack[top] = speed;
1552         top++;
```

```
1551         size++;
1552     }
1553     /// <summary>
1554     /// Adds sensor data to an internal buffer. Caution, this function does not
1555     submit the data to the stack.
1556     /// </summary>
1557     /// <param name="distance"></param>
1558     /// <param name="angle"></param>
1559     /// <param name="bump"></param>
1560     /// <param name="speed"></param>
1561     public void InsertSensorDataIntoBuffer(int distance, int angle, int bump,
1562     int speed)
1563     {
1564         if (distance >= 32767 || distance <= -257 || angle >= 32767 || angle <=
1565         -257)
1566         {
1567             InsertSensorData(_Buffer[0], _Buffer[1], _Buffer[2], _Buffer[3],
1568             _Buffer[4], _Buffer[5]);
1569             for (int i = 0; i < 5; i++)
1570                 _Buffer[i] = 0;
1571             if (IsMoving() == true)
1572                 InsertCommand(GetLastCommandID());
1573         }
1574         byte[] distanceBytes = new byte[2];
1575         byte[] angleBytes = new byte[2];
1576         distanceBytes = ConvertToHighLow(distance);
1577         angleBytes = ConvertToHighLow(angle);
1578         _Buffer[0] = distanceBytes[0];
1579         _Buffer[1] = distanceBytes[1];
1580         _Buffer[2] = angleBytes[0];
1581         _Buffer[3] = angleBytes[1];
1582         _Buffer[4] = (byte)bump;
1583         _Buffer[5] = (byte)speed;
1584     }
1585     /// <summary>
1586     /// Adds the data in the buffer to the log stack
1587     /// </summary>
1588     public void SubmitSensorDataBuffer()
1589     {
1590         if (lastcommandOffset == top - 1)
1591         {
1592             InsertSensorData(_Buffer[0], _Buffer[1], _Buffer[2], _Buffer[3],
1593             _Buffer[4], _Buffer[5]);
1594             for (int i = 0; i < 5; i++)
1595                 _Buffer[i] = 0;
1596             //if (IsMoving() == true)
1597             //    InsertCommand(GetLastCommandID());
1598         }
1599     }
1600     /// <summary>
1601     /// Returns last command line from the top of the stack of size 7: 0-speed 1-
1602     -bump 2-ang.low 3-ang.high 4-dist.low 5-dist.high 6-id
1603     /// </summary>
1604     /// <returns></returns>
1605     public int[] PopLastCommandLine()
1606     {
1607         int[] LastCommand = new int[7];
1608         for (int i = 0; i < 7; i++)
1609             LastCommand[i] = 0;
1610         try
1611         {
1612             if (top < 6)
1613                 return LastCommand;
1614             else
1615             {
1616                 for (int i = 0; i < 7; i++)
```

```
1611         LastCommand[i] = _DriveHistoryStack[top - 1 - i];
1612         top = top - 7;
1613         size = size - 7;
1614         if (top != 0)
1615             lastcommandOffset = top - 1;
1616         else
1617             lastcommandOffset = top;
1618         lastcommandId = _DriveHistoryStack[lastcommandOffset];
1619         return LastCommand;
1620     }
1621 }
1622 catch (Exception ex)
1623 {
1624     MessageBox.Show(ex.Message.ToString()); return LastCommand;
1625 };
1626 }
1627 public void ClearLog()
1628 {
1629     for (int i = 0; i < size; i++)
1630         _DriveHistoryStack[i] = 0;
1631     size = 0;
1632     top = 0;
1633     lastcommandId = 0;
1634     lastcommandOffset = 0;
1635 }
1636 public int GetTop()
1637 {
1638     return top;
1639 }
1640 public int GetLastCommandID()
1641 {
1642     return lastcommandId;
1643 }
1644 public bool IsTopACommandID()
1645 {
1646     if (top - 1 == lastcommandOffset && top != 0)
1647         return true;
1648     else
1649         return false;
1650 }
1651 public bool IsMoving()
1652 {
1653     return moving;
1654 }
1655 public void SetMoving(bool isMoving)
1656 {
1657     moving = isMoving;
1658 }
1659 /// <summary>
1660 /// Returns the whole stack
1661 /// </summary>
1662 /// <returns></returns>
1663 public int[] GetStack()
1664 {
1665     return _DriveHistoryStack;
1666 }
1667 public int GetSize()
1668 {
1669     return size;
1670 }
1671 /// <summary>
1672 /// Converts an integer to high and low byte, returns byte[] with 0= high
and 1 = low
1673 /// </summary>
1674 /// <param name="integerValue"></param>
1675 /// <returns></returns>
```

```
1676     private byte[] ConvertToHighLow(int integerValue)
1677     {
1678         byte[] result = new byte[2];
1679         byte high;
1680         byte low;
1681         Int16 original = Convert.ToInt16(integerValue);
1682         high = Convert.ToByte((original >> 8) & 0xff);
1683         low = Convert.ToByte(original & 0xff);
1684         result[0] = high;
1685         result[1] = low;
1686         return result;
1687     }
1688     /// <summary>
1689     /// Converts high and low byte to 32 signed int
1690     /// </summary>
1691     /// <param name="high"></param>
1692     /// <param name="low"></param>
1693     /// <returns></returns>
1694     private int ConvertHighLowToInt(byte high, byte low)
1695     {
1696         return (Int16)((((short)high * (short)256) + (short)low));
1697     }
1698 }
1699 }
1700
```