KINECT/RC: CONTROLLING A REMOTE CONTROL CAR WITH THE KINECT

A Project Report

Presented To

Eastern Washington University

Cheney, WA

In Partial Fulfillment of the Requirements

for the Degree

Masters of Science

By

Devin Howard

Spring 2012

Table of Contents

1 Project Goals

The objective of this project is to create a program (herein after called Kinect/RC) that allows a user to easily control a remote control car using a natural user interface (NUI), specifically, the Microsoft Kinect. To accomplish this project, various goals need to be met. Video needs to be provided from a camera mounted on the remote control car to allow the user to view the direction in which the car is headed. This is a more natural and easier way to control the car than watching the car directly. Additionally, the program needs to provide telemetry data such as direction and velocity of the car to confirm to the user that their intended hand motions are being recognized the way they wish (Hands in a position that expresses turning left should show that a left turn is being sent to the remote control car). The program should also display the skeletal data that is determined by the Kinect so the user is able to verify they are being recognized correctly. Thus, for example, the user can correct their positioning should part of their body be cut off from the view of the Kinect. The Kinect has an adjustable elevation of ±27° [1] so the program needs to provide the user the ability to adjust this elevation. This adjustment will allow the placement of the Kinect at any height within its tolerance of 2 ft. to 6 ft. [1] above the floor. The project also must be written using design patterns and other object-oriented principles to develop a modular and easily maintainable program.

2 Specifications

2.1 Hardware

To accomplish the goals set forth for this project, certain requirements need to be met. The remote control car needs to have a transmitter that can connect to a computer or be modified to allow it to connect to a computer. The remote control car needs to be modified to allow the mounting of a wireless camera that streams video to the program. The camera needs to work for an appropriate

amount of time (30 minutes minimum) off battery power for a typical usage scenario. It also needs to be light enough not to hinder the performance of the remote control car. The streaming video needs to have low enough latency to allow a user to react to obstacles in the path of the remote control car. The quality and resolution of the streaming video needs to be high enough to allow the user to recognize objects in the video easily. Lastly, a Microsoft Kinect is required as the motion-sensing interface to the control program.

2.2 Software

To develop the program, the official Microsoft Kinect SDK was used. The recommended language from Microsoft, C#, was used [1]. The software was written using design patterns where appropriate as well as using object-oriented principles such as the SOLID principles (Single responsibility, Open/close, Liskov substitution, Interface segregation, and Dependency inversion) [2]. The program was designed to be modular to allow other developers to provide their own plugins for other hardware. This flexibility will allow developers to use the program for their own custom adapted remote control hardware without having to rewrite the main program.

3 Overall Architecture

3.1 Hardware

The following is a description of the hardware components of the Kinect/RC project. Figure 1 - Block Diagram shows how the components are connected to each other. You may also look at the associated figures of each component for additional information on their use in the program.

Kinect – Senses the user's body joints in three-dimensional space. (Figure 15, Figure 16)

Computer – Kinect/RC runs on the computer and processes the input from the Kinect to tell the mbed microcontroller how to control the remote control car.

Wi-Fi Access Point – Facilitates communication between the wireless camera mounted on the remote control car and the computer for displaying video to the user. (Figure 13)

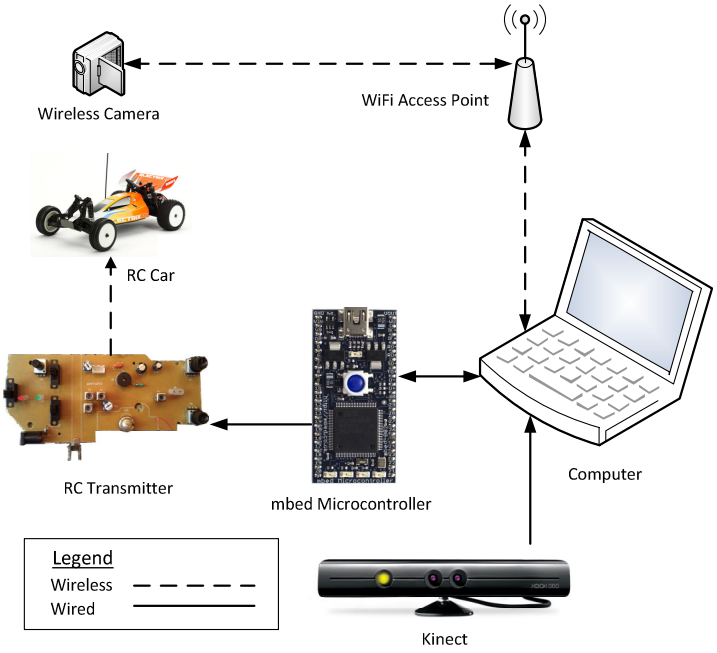

Figure 1 - Block Diagram

Remote Control Car – The remote control car that is controlled by the user. (Figure 14)

Remote Control Transmitter – Sends the direction and velocity to the remote control car as dictated by the dual potentiometers. (Figure 17)

Mbed Microcontroller – Adjusts the MCP4261 digital dual potentiometer that in turn controls the voltage the remote control transmitter senses and thus changes the direction or velocity of the remote control car appropriately. (Figure 17)

Wireless Camera – Mounted on the remote control car, the camera sends video to the computer for the user to view. (Figure 19, Figure 20)

3.2 Software

The following is a description of the software components of the Kinect/RC project. Figure 2 shows how the components are connected to each other.

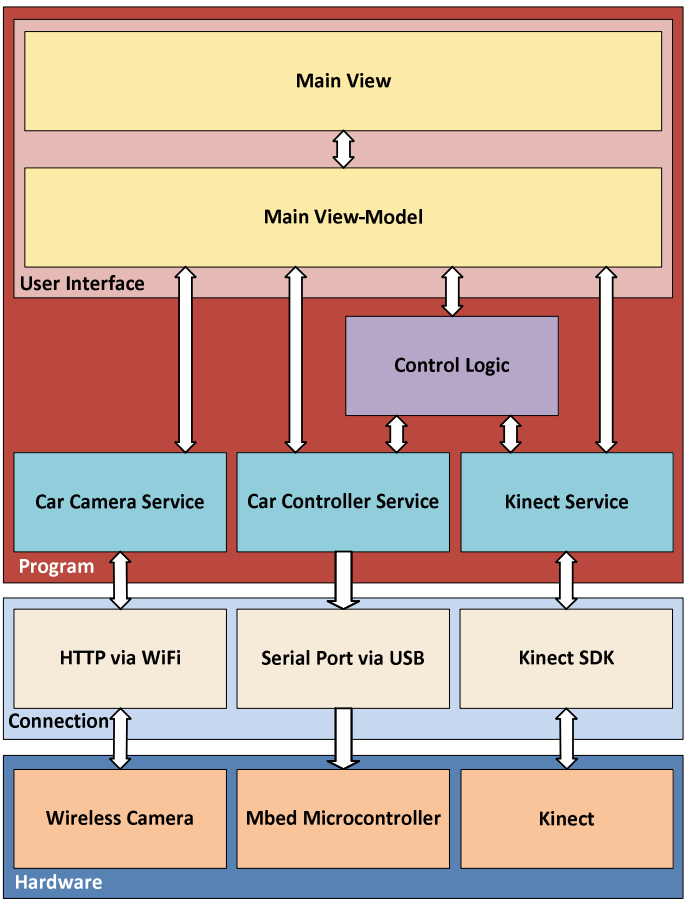

Figure 2 - Architectural Diagram

Main View – An XML file that declares the user interface controls that the user sees and interacts with when the program is running. The various user controls include the skeletal viewer, instrumentation, and video from the wireless camera on the remote control car. See section 6.4.1 Custom User Controls for information on the user controls used in the program and section 11.2 User Interface for examples.

Main View-Model – Defines the data that the Main View displays to the user. It also facilitates communication to underlying services when a user interacts with the main view. This includes adjusting the elevation angle of the Kinect and the connection information for the wireless camera. See section 6.3.1 Model View View-Model for additional information.

Control Logic – The business logic that translates the skeletal points data of the user returned by the Kinect to the direction and velocity values needed to control the remote control car. This data is then sent to the appropriate service to control the car as intended. See section 6.4.6 ControlLogic Class for specific information on this component.

Car Camera Service – A singleton service [2] that is responsible for connecting to the wireless camera on the remote control car and to return the images received from the camera to the view-model for displaying to the user. See section 6.4.4 CarCameraService Class for further information.

Car Controller Service – A singleton service [2] that is responsible for sending the direction and velocity to the mbed controller as is dictated by the control logic. See section 6.4.7 CarControllerService for further information.

Kinect Service – A singleton service [2] that is responsible for managing the connection with the Kinect SDK and sending the data to the control logic for translation into direction and velocity settings. See section 6.4.6 ControlLogic Class for further information.

4 User Experience

The Kinect/RC program is easy to operate for any user. See Figure 21 - User Interface for a screenshot of the user interface. All the services are off when the program is opened. In the top right of the user interface, the user types in the URL for the wireless camera that is attached to the remote control car along with a username and password if needed. The user then clicks the Connect button below the camera information. Once clicked, the Connect button changes to a Disconnect button. The Connect button starts the CarControllerService, KinectService, and CarCameraService. The user interface then displays a skeleton representation of the user in the top left, color video from the Kinect in the bottom left and the video from the wireless camera in the center. The instruments on the bottom right are then enabled. The user then has access to a slider and button to set the elevation angle of the Kinect so it can better see the user. Below the elevation controls is the direction and velocity of the remote control car represented as percentages. There are also two bars that visually represent the direction and velocity.

The user starts controlling the remote control car by raising their hands out in front of them at half their arm's length as if they are holding on to a steering wheel. If their hands are in this position and horizontal to each other, a countdown timer in the bottom right of the user interface will start counting to zero from three seconds. During this time, the user must hold their hands steady. If their hands move to their waste or are not horizontal then the countdown stops and resets. Once the countdown timer hits zero, the background of the user interface turns from white to green to indicate the user now has control of the remote control car.

To turn the remote control car, the user rotates their hands like a steering wheel. The scale for steering is based on a 90° range from center. The car itself has a 35° range from center. The increased range gives the user a greater sense of control of the remote control car. The amount of rotation the user turns their hands will proportionally turn the wheels. If the user turns their hands slightly to the right, the car's wheels will turn slightly to the right. If the user turns their hands to a full 90° to the left, the car's wheels will turn to the farthest position the car is capable of (35°) to the left. Conversion between the user's steering range and the car's turning range is explained in section 6.4.6.4 CalculateDirection() method.

To make the car move, the user moves both their hands away from themselves to accelerate the car forward, and towards themselves to accelerate the car in reverse. The distance they move their hands proportionally sets the velocity of the car. If they move their hands slightly forward from their initial position then the car moves slowly. If they extend their hands fully, then the car will move at its maximum set speed. The maximum speed of the car can be changed in the settings file of the program. Calculation of the velocity is explained in section 6.4.6.5 CalculateVelocity() method.

When the user wishes to stop controlling the car, all they do is drop both their hands to their sides. The user interface background will change back to white to indicate that the user is no longer in control of the car. The user then may either close the program or click the Disconnect button.

5 Hardware Design

5.1 RC Car

The remote control car being used is an Electrix RC Boost model [3]. It is a hobby quality car chosen for its replaceable parts and affordable price over a toy quality car. A larger battery pack was purchased to extend the testing time between charges. The car was also modified to mount the Airlink 101 wireless camera. A hole was drilled in the plastic chassis cover to hold a ¼-inch nut and screw. It was then reinforced with a brass metal strip that mounts to the car itself (See Figure 20 - Remote Control Car Mount with Airlink 101 1620W Camera). The car was further modified to hold a smart phone instead of the Airlink 101 camera by removing the spoiler mount. Then a 2 in. x 2.5 in. plastic plate was mounted vertically to where the spoiler was. The smart phone is then attached to the plastic plate via rubber bands (See Figure 19 - Remote Control Car Mount with Smart Phone).

5.2 Car Controller

The remote control transmitter uses two 5KΩ potentiometers as voltage dividers. The transmitter senses the difference in voltage between the two ends of each potentiometer then sends the appropriate direction or velocity setting to the remote control car. The middle position of each potentiometer is considered the neutral position for the car, wheels straight and stationary. The ends of each potentiometer indicate maximum turning to the left/right and velocity in the forward/reverse directions as appropriate.

The potentiometers were replaced with an MCP4261 digital dual potentiometer. The MCP4261 has 256 different steps of resistance and is controlled over a Serial Peripheral Interface (SPI) [4]. An mbed microcontroller is used to control the MCP4261 based on the input received from the computer. The mbed microcontroller is a prototyping microcontroller that supports a wide range of hardware protocols. It is programmed using the mbed SDK using C/C++. The mbed is connected to the computer via USB and communicates with Kinect/RC via a virtual serial port [5]. Commands are sent over the serial port connection to the mbed, which in turn adjusts the digital potentiometers. The serial port connection runs at 115200 bit/s.

The MCP4261 requires two-byte commands for Read and Write and requires a one-byte command for increment and decrement. The first four significant bits state the memory address, which will be read or written to. The next two bits indicate the type of command (read / write / increment / decrement). The last 9 bits state the value, which is to be written. These bits are ignored on a read command. On a successful execution of a command, the ninth bit will be high on the returning two bytes [3]. See section 11.6.1 Transmission Byte Formats for figures of the send and receive bytes.

5.3 Kinect

The Microsoft Kinect is a motion sensing input device released for the Xbox 360. It allows a user to control a program using body motions such as waving a hand. The device has an RGB camera, infrared laser, infrared sensor and microphone array. See section 11.1.1 Kinect for diagrams of the Kinect and its hardware features. The RGB camera is used as a source of color images. The infrared laser is diffracted by a lens with 36,000 specifically placed holes. The laser light hits an object and then is reflected back to the infrared sensor. The Kinect sensor triangulates where the point of laser light reflected from and generates a value for each pixel of the sensor that indicates the distance from the Kinect to the object at that specific point on the sensor [6]. This data is then sent to the computer for processing skeleton joints

for skeleton tracking or for use in a program as raw depth data. The microphone array contains four microphones for improved sound recording quality and source localization [1]. The Kinect/RC program does not use the microphone array.

5.4 Wireless Camera

The wireless camera, in this case, a smart phone camera, is used for video transmission from the remote control car to the user interface of the Kinect/RC program. The smart phone runs a program that transmits JPEG images from the camera over HTTP as an MJPEG stream. The computer then receives this stream and displays the resulting images to the user.

The first wireless camera to be used was an Airlink 101 AIC1620W network camera. It was chosen because it provided an MPEG4 stream over the Real Time Streaming Protocol (RTSP) and was easily convertible to battery power since it ran off 12 volts [7]. This camera was modified to run off 8 AA size batteries. The camera failed to meet expectations for a couple of reasons.

The Windows Presentation Foundation framework provides a built-in user control called MediaElement. This control has built in support for RTSP with no additional coding [8]; however when tested, it was discovered the implementation is incompatible with the Airlink camera.

Research was done to find an alternative to the MediaElement control that would support the RTSP protocol used by the Airlink Camera. There was only one library that was found called VlcDotNet. VlcDotNet is a wrapper around the VLC media player. This library was not tested due to the RTSP support not yet being implemented at the time of research [9]. A decision was made to instead use the MJPEG stream the Airlink camera provides.

The second problem with the camera is that its performance would degrade after 10 minutes. After that, latency in MJPEG frames would increase dramatically to more than 1 second and shortly

after, the camera would stop transmitting images. Constantly replacing the batteries of the camera was not ideal for using the Kinect/RC program.

It was determined that a better solution was needed, one that could provide a similar streaming protocol of the Airlink camera but have a much longer battery life. An Android OS based smart phone was decided as the best choice. The reasoning for this is that a smart phone is built for battery life that lasts multiple hours during calls, its highest battery use. Also, the Android SDK provides native code support so that video processing on the phone could be done natively on the hardware itself and not be impacted by a virtual machine [10].

There is a free app on the Android Marketplace Called IP Webcam that transmits an MJPEG stream over HTTP so there would be no modification needed of Kinect/RC since the code to use the MJPEG stream from the Airlink camera could be reused. IP Webcam supports all resolutions provided by the camera on the phone and includes a setting for the JPEG compression. The IP webcam program's user interface is written in Java while the video processing is written in C [11]. Having the code written in C allows the program to provide an efficient MJPEG stream without the performance penalty of running in the Java Virtual Machine.

The first Android smart phone acquired for the project was an LG Vortex. The Vortex camera is capable of a resolution from 176x144 to 640x480 and the CPU is a Qualcomm MSM7627 running at 600 MHz. The smart phone has a maximum frame rate of 30 fps [12]. The various resolutions and JPEG compression settings were tested to find the ideal balance of picture quality and performance (See section 7.2.1 Previous Wireless Cameras Performance). It was determined that a resolution of 320x240 and a JPEG compression quality at 50 worked best. Unfortunately, the video stream would start to freeze randomly every couple of seconds regardless of the resolution or JPEG compression settings. This situation is not ideal since the user needs to see where the remote control car is travelling at all times.

A second Android phone was acquired with a faster processor. The HTC Desire S S510E has a Qualcomm Snapdragon MSM8255 CPU running at 1 GHz, which is one of the top of the line processors for smart phones that has come on the market in the past year. Its camera supports resolutions from 480x320 to 960x720 and a frame rate of 22.67 fps [13]. It was hoped that the faster and more advanced processor would eliminate the stuttering issues the LG Vortex had and have a lower latency at a higher resolution. The stuttering issues were nonexistent on the HTC as hoped (See section 7.2 Wireless Camera Performance).

6 Software Design

6.1 Overview

As stated in the project goals, object-oriented programming principles were used to develop the software for the Kinect/RC program. Both Design Patterns and the SOLID principles were applied to reduce coupling between classes and to support a plugin architecture so other programmers can easily modify the software for their needs. Two software programs make up the Kinect/RC program. The first is the software on the Mbed microcontroller and the second is the Kinect/RC program itself.

6.1.1 Mbed Microcontroller

The mbed microcontroller code was written in C and C++. There are two C++ classes and one C file. The C file named RPCFuncs.c holds all the functions that can be called from the serial port. This file uses RPCInterface, a third party library specifically written for easily requesting the execution of functions on the mbed controller over the serial connection [14]. These functions then call the first C++ class, RCController.cpp that facilitates communication with the second C++ class, MCP4261.cpp that directly communicates with the MCP4261 potentiometers, thus isolating communication of the serial

port, the translation of the requested commands and communication to the MCP4261. See section 11.6.2 Code Examples.

This separation of concerns also allows the code handling communication with the MCP4261 to be published separately and for the code in RCController.cpp to light 4 LEDs on the mbed for diagnostic testing. Each LED corresponds with either an increasing or decreasing of the two potentiometers as requested by the user through the Kinect/RC program.

6.1.2 Kinect/RC

Kinect/RC is written in C# using the .NET Framework and Windows Presentation Foundation (WPF) for the user interface. C# was chosen as it is directly supported by the Kinect SDK and for its ease of coding. WPF was chosen for its modern user interface tool set and elimination of boilerplate code for updating the user interface itself. Two additional supporting frameworks were used to create the user interface. The first is PRISM, Microsoft's framework to assist in using the Model View-View Model pattern [15]. The second is Microsoft's Unity Inversion of Control container to assist in dependency injection [2] and to keep the program loosely coupled.

6.2 Frameworks

6.2.1 Windows Presentation Foundation

Windows Presentation Foundation (WPF) is a framework for creating applications for Windows. It uses DirectX as its rendering engine and an XML variant called XAML (Extensible Application Markup Language) to define the user interface. WPF relies on a binding engine that dynamically converts data as needed and displays it on the user interface when the data changes. [16]

6.2.2 Kinect for Windows SDK

The Kinect for Windows software development kit (Kinect SDK) provides the APIs and drivers needed to communicate with the Microsoft Kinect. It provides for the display of the color images from the RGB camera, depth data directly from the Kinect, the skeletal data computed from the depth data, and speech recognition. The API's provide an array called KinectSensors containing all the Kinects connected to the computer. The array also provides an event that is fired when the status of a Kinect sensor changes. [1]

6.2.3 PRISM

PRISM is a toolkit used to assist in the use of the Model View View-Model pattern (see 6.3.1 Model View View-Model for a description of this pattern) and provide additional guidance in building WPF based applications. It provides support for dependency injection (through the Unity IOC container), modularity, event aggregator (mediator pattern), commanding, and navigation. The Kinect/RC program only takes advantage of the Unity, modularity and commanding features. [15]

6.2.4 Unity IOC

Unity is an inversion of control container (See section 6.3.2 Inversion of Control for additional information). It provides instantiation of objects (similar in function to the factory pattern [2]), lifetime management of the objects, and automatic dependency injection. When an object is created, Unity will use reflection on the class to determine the dependencies needed by the object. It then provides the dependencies to the object on creation. It is also possible to manually request a specific type from the container itself. The use of Unity allows the objects to be decoupled from each other through writing classes against interfaces and then having those interfaces fulfilled with objects through the dependency injection. [15]

6.3 Design Patterns

Multiple design patterns are used in the Kinect/RC program to keep it maintainable and support object oriented principles. The following is a general overview of the types of patterns used in the program.

6.3.1 Model View View-Model

The main pattern used to keep the Kinect/RC program maintainable is the Model View View- Model pattern (MVVM). It is based on the popular Model View Controller pattern (MVC) but instead of having a controller that facilitates communication between the View and the Model, it instead takes advantage of the binding engine in WPF. The View and Model are the same as in MVC. The view defines the controls and other strict UI elements and the Model represents the data in the program. The View-Model organizes and presents data as properties that are specific for the View. The view then binds to these properties and updates the data it presents to the user as the data changes in the program. [16][17]

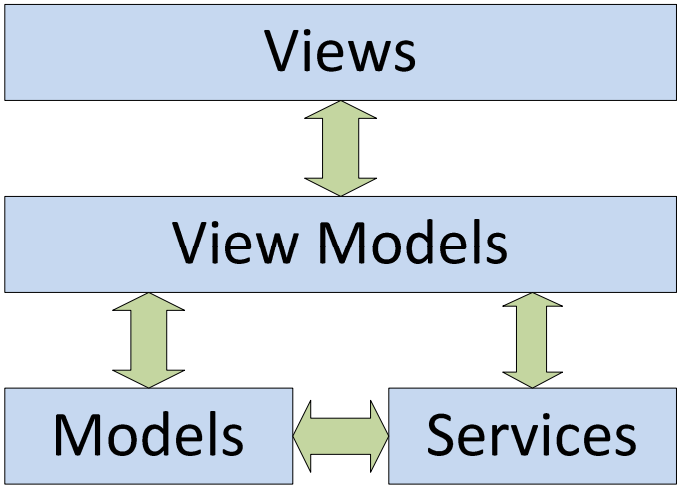

Figure 3 - MVVM Block Diagram

The Kinect/RC program does not define any Models as these are already provided by the services it uses (Kinect, car controller and wireless camera). It also only has one view and one view-model because the user interface does not change its organization as a navigation-based application does. [17]

6.3.2 Inversion of Control

Inversion of control (IOC) is the practice of removing the instantiation of objects that are depended on by a specific class. Normally if a class depends on other classes, these classes are instantiated in the constructor. The original class is then tightly coupled to the instantiated classes. Inversion of control changes this by removing the instantiation of the dependencies from the class. The dependencies are then instantiated outside of the class and then provided as parameters in the constructor of the class. IOC is enabled in Kinect/RC through the use of the Unity container. Thus, the dependency inversion principle of the SOLID principles is used and thus different implementations of the dependencies can be used. [2]

6.3.3 Adapter

The adapter pattern translates an interface of a class into a compatible interface needed by another class [2]. It was required to use the adapter pattern to keep the Kinect/RC program loosely coupled and to make it easier to create unit tests.

The adapter pattern is used in two main areas of the Kinect/RC program. The first is in providing the settings required by the Kinect and Car Controller services. The settings provide the various tolerances or settings needed for the services to work. For example, settings include the threshold needed to define when the user's hands are horizontal to each other. An interface is written against the services and a class is instantiated using the interface. It is then provided to the Unity container for later use in the service. See Figure 39 - Settings Adapter Class Diagram.

The adapter pattern is also used to separate out the external dependencies provided by the .NET framework and the Kinect SDK, such as in the use of the SerialPort class and the KinectSensor class. These classes are not defined by any interface and thus no interface can be written that can later be injected by the required class. An interface that mimics the methods and properties used in the

Kinect/RC program of the SerialPort class is created along with a concrete implementation that wraps the SerialPort (See Figure 38 - SerialPort Adapter Class Diagram). The same is done for the KinectSensor; however, the KinectSensor contains references to two additional classes used by the Kinect/RC program. These are the SkeletonStream and ColorImageStream classes, which do not implement interfaces. Thus, the adapter pattern had to be applied to both of these classes as well. In addition, the concrete implementation of the KinectSensor interface needs concrete implementations of the streams based on the KinectSensor it is wrapping. See Figure 37 - Kinect Adapters Class Diagram.

6.3.4 Command

The command pattern is used to encapsulate the execution of a method from the user interface [2]. The concrete implementation of this pattern is provided by the PRISM framework as the WPF framework only provides an interface for the pattern. The command pattern is utilized in the Connect button to start the Kinect, car controller, and wireless camera services. It is also utilized in the Set Angle button to adjust the elevation angle of the Kinect sensor.

6.4 Business Logic

6.4.1 Custom User Controls

As the Kinect/RC program uses the Kinect and custom hardware, it was necessary to create custom user controls. The custom user controls provide the user with information about the state of the car controller and Kinect. WPF allows the combination of preexisting user controls to make a new user control by declaring the said controls in the wrapping user control then exposing the bindable properties to be used by the parent control, such as a Window control, or another user control. [16]

6.4.1.1 SkeletonUserControl

The SkeletonUserControl is a slightly modified version of the SkeletonUserControl provided in the samples for the Kinect SDK. The code was cleaned up from the version in the samples by removing

the Kinect specific code. The user control originally had a property to bind to the Kinect property itself in the View-Model. This was unnecessary and violated the single responsibility in SOLID. Code was added so the user control could bind to a Skeleton[] array as that is the only data the control needs for operating. The user control was also modified to only display the nearest tracked skeleton. This skeleton represents the user whom the Kinect tracks for controlling the remote control car. See 8 Future Work for improvements desired for this user control.

6.4.1.2 InstrumentsUserControl

The InstrumentsUserControl displays the direction and velocity as a percentage and as a visual representation along with the elevation angle of the Kinect. It also allows the user to modify the angle by changing a slider and pressing the “Set Angle” button. Unlike the SkeletonUserControl, The InstrumentsUserControl is all customized code. See Figure 23 - InstrumentsUserControl for a screenshot.

6.4.1.3 BidirectionalIndicator

It was decided a visual representation of the direction and velocity would be beneficial to the user in addition to the numerical percentages of the direction and velocity of the remote control car. The BidirectionalIndicator is two ProgressBar controls placed next to each other with one transformed 180° from the other. The BidirectionalIndicator binds to a value. When the value is negative, the ProgressBar representing negative values will highlight accordingly. When the value is positive then the positive ProgressBar will highlight accordingly. The user control was straightforward to create. The challenge was modifying the control so that it would work in a vertical orientation. The two ProgressBar controls are by default placed in a 1x2 grid. When the orientation property of the BidirectionalIndicator is changed to Vertical, the second column is removed and a row is added making the grid 2x1. The ProgressBars are then placed in their appropriate cells, their individual orientations are changed and the 180° rotation on the negativeProgressBar is removed. When the orientation is changed to Horizontal,

the second row is removed, a column is added, the ProgressBars are placed in their appropriate cells, their individual orientations are changed and the 180° rotation is added. See Figure 24 - BidirectionalIndicator Orientation Change Method for the code.

6.4.1.4 DeviceStatusUserControl

The DeviceStatusUserControl is used to indicate to the user the status of the Kinect and car controller (the mbed microcontroller). If either device does not have a “Connected” status then a modal dialog with this control will display the status of each control along with an informative message as to what the specific problem is (No power, not connected, etc.). It also displays a green checkmark or red X to help distinguish the type of error (See Figure 22 - DeviceStatusUserControl). Once the problem is corrected, the user then clicks the “Retry” button to try reconnecting the services. They also have the choice of cancelling which closes the dialog. The dialog will also display if an error arises while the services are connected, such as a cable to the Kinect is accidentally disconnected.

The CarCameraService provides a status property as well but the DeviceStatusUserControl is not written to use this property. The reason is that the wireless camera is not required to operate the vehicle. If the program is unable to establish a connection to the camera, then there is no indication to the user aside from the placeholder image remaining on the user interface. See section 8 Future Work for possible solutions.

The dialog box that is used is from a third-party library that provides a custom user control, called ChildWindow [18]. The ChildWindow mimics a modal dialog box but is a part of the parent user control (in this case the MainView) instead of a separate window. The DeviceStatusUserControl is declared inside this ChildWindow user control.

6.4.2 WMI interface

The Kinect SDK provides an event and property that represents the status of the Kinects attached to the computer. However, the SerialPort class provided by the .NET framework and used by the car controller does not provide a status property to indicate the state the given serial port is in. A status property needed to be written so the Kinect/RC program would work correctly.

```
"SELECT * FROM __InstanceCreationEvent WITHIN 1
WHERE TargetInstance ISA 'Win32_PnPEntity' AND
TargetInstance.DeviceID LIKE
\"%USB\\\\VID_0D28&PID_0204&MI_01%\""
```
Listen to connection event.

```
"SELECT * FROM __InstanceDeletionEvent WITHIN 1
WHERE TargetInstance ISA 'Win32_PnPEntity' AND
TargetInstance.DeviceID LIKE
\"%USB\\\\VID_0D28&PID_0204&MI_01%\""
```
Listen to disconnection event.

```
"SELECT * FROM Win32_PnPEntity WHERE DeviceID LIKE
\"%USB\\\\VID_0D28&PID_0204&MI_01%\""
```
Query for existing hardware.

Figure 4 - WQL Query Strings

It was discovered that the status of the mbed controller's attachment to the computer could be ascertained through the Windows Management Instrumentation (WMI) interface. To determine the status of the mbed controller (whether it was connected or disconnected from the computer), a helper class called WMIHelper was written to listen to the __InstanceCreationEvent provided by the WMI. A query string written in a SQL derivative called WQL (WMI query language) is given to the WMI system with the request. The query string is wrapped by a WqlEventQuery object. The object is then passed into the ManagementEventWatcher object, which provides an event that is fired whenever the query is successful. Part of the query string states how often the query is executed by the WMI system. The query string needs to be specific for the hardware of the mbed microcontroller or else the event would fire when other devices are connected to the computer. [19]

The __InstanceCreationEvent was monitored when the mbed controller was connected to the computer. Ten instances were returned by the system. Looking through these responses a single event was discovered to represent the virtual serial port used by the mbed driver. The event was of a "Win32_PnPEntity" type and returned various properties about the mbed controller such as the Service,

ClassGuid, DeviceID, etc. (see 11.6.3 Mbed WMI __InstanceCreationEvent Examples). All PCI devices have a device id that is assigned based on the hardware device by the manufacturer. So the query string that is used to tell the WMI system what to look for was modified to listen to the creation event of the "Win32_PnPEntity" type that the DeviceID is similar to the one for the mbed controller. Another query string was created for the __InstanceDeletionEvent as well for when the mbed controller is disconnected from the computer. The Kinect/RC program can react appropriately when the mbed microcontroller is connected or disconnected from the computer.

An additional scenario is if the mbed microcontroller is already connected to the computer before the program is started. For this, a third query string is used that only queries for all devices that match the mbed microcontroller DeviceID. This query is wrapped in a WqlObjectQuery object that is then passed to a ManagementObjectSearcher object. The ManagementObjectSearcher returns a collection that contains (if any) all the matches to the query. This is executed when the WMIHelper is first constructed. Based on this fact, the helper will then start the query listener to either the __InstanceCreationEvent or __InstanceDeletionEvent as appropriate.

When the event tied to the ManagementEventWatcher is successful, the status of the mbed controller is updated appropriately and then the ManagementEventWatcher starts listening to the other event.

6.4.3 Pattern for Services

The Kinect/RC program needs to behave correctly when the Kinect or car controller are connected, disconnected, or are in some other state. There also needs to be a uniform pattern used to structure each service (CarCameraService, KinectService and CarControllerService) that represents these devices.

The pattern used is based on how the Kinect SDK solves these problems. It has a Start() and Stop() method, a Status property that is an enumerable type, an event that is fired when this property changes, and a Boolean property that states whether the service is running (Start() was successfully called) or not. The CarCameraService, KinectService and CarControllerService each mimic this same pattern.

The three services vary from each other by having different enumerable types used for the status. The CarCameraService and CarControllerService each have their own enumerable with Connected and Disconnected. The KinectService requires additional enumerables because the Kinect SDK has additional status states, besides the expected connected/disconnected ones. They include Initializing, Error, NotPowered, NotReady, DeviceNotGenuine, DeviceNotSupported, and InsufficientBandwidth [1]. It is possible to have the KinectService change its status to Disconnected if one of these other states happen, but was decided against because some of the statuses provide pertinent information that the user would need to know. An example would be if the Kinect SDK returned a status of “NotPowered”, then the user would need to know this in order to verify the power cord was connected to an outlet and the Kinect itself. The DeviceStatusUserControl provides this information to the user as stated in section 6.4.1.4 DeviceStatusUserControl.

When the Connect button is pressed, the statuses of the services are checked to see if they are set to “Connected” (available for use). If so, then the Start() method for each service is called. If one of the services’ statuses is not set to “Connected” then the DeviceStatusUserControl is displayed to the user and all the services are stopped. Otherwise, the services start as expected. The exception to this situation is the CarCameraService. Its status is not checked to see if it is Connected or not, since connecting to the wireless camera and receiving images from the camera are a part of the same step.

As expected, the Stop() method is called when the user presses the Disconnect button and all services are then stopped. Start() and Stop() do not change the status of the services.

6.4.4 CarCameraService Class

The CarCameraService manages the connection with the wireless camera for the Kinect/RC program. It uses an MJPEG decoding library from the MSDN Coding4Fun website [20]. When the service is started, the library attempts to connect to the camera at the provided URL. If it fails to connect or any exception is raised during the time the service is connected to the camera, then the service shuts down. The only feedback to the user that there was a failed connection is a static image on the user interface. The user is still able to control the remote control like normal.

Once a connection is established, the wireless camera continuously sends back the JPEG images that are captured. An event on the MJPEG decoder is raised when a full image has been received and decoded. The CarCameraService then in turn raises its own event that contains the image. The MainViewModel listens to this event and populates the CarCameraImage property then the image is displayed to the user.

6.4.5 CountdownTimer Class

A countdown timer is needed to assist in verifying that the user has their hands steady before giving control of the remote control car to them. If the timer was not in place then the user could unintentionally start controlling the car just by moving their hands into the play area (above the spine joint point returned by the Kinect). The CountdownTimer class wraps a DispatcherTimer object that is built for use with WPF applications. The reason a specialized class is needed instead of using the DispatcherTimer directly is so that the user can be informed on the user interface of how much time is left until the timer hits zero. The CountdownTimer class takes two TimeSpan object arguments, which are the amount of time for the countdown and the interval to fire an event that returns the current time

left in the countdown. There are two events, the first being the one fired at the interval specified during the construction of the object, and the second is when the countdown finally reaches zero.

6.4.6 ControlLogic Class

The ControlLogic class holds all the business logic that pertains to converting from the skeleton data returned from the Kinect to what is required by the car controller to control the remote control car. It starts a second thread on which the CarControllerService and KinectService run on. That way the CarControllerService and KinectService can run efficiently without interference from the UI updating event loop. The ControlLogic class then listens to the SkeletonFrameReady event on the KinectService, which then drives the business logic associated with calculating the direction and velocity for the remote control car.

6.4.6.1 Threading

The CarControllerService and KinectService indirectly affects the user interface with the Skeleton data, video from the Kinect, and direction and velocity values sent to the car controller. As this data only is modified by their respective services and not by the user interface, there is no need for any mutexes or semaphores. There also is no need to marshal the data manually from the second thread to the user interface thread as this process is taken care of automatically by WPF since version 3.5 [21].

Some additional work was necessary due to a thread affinity issue inside the KinectOnColorFrameReady method in the KinectService. This method processes the color image from the Kinect when the Kinect's ColorFrameReady event fires. The method retrieves the ColorImageFrame and then uses an extension method provided by the Coding4Fun Kinect Library [20] to copy the data to a BitmapSource object. A BitmapSource object inherits from DispatcherObject that is specific to WPF. Any object that inherits from DispatcherObject will verify that any access to it is on the same thread the object was created on. The BitmapSource object in the KinectOnColorFrameReady method is created on

the secondary thread but when the user interface attempts to access the object, the affinity is checked and an exception is thrown since the user interface is on the main application thread and not the secondary thread. To solve this, a method provided by DispatcherObject named Freeze() needs to be called before the object is accessed by other threads [21]. So this call was added to the KinectOnColorFrameReady method before the user interface accesses the BitmapSource object solving the problem.

When the Start() method is called on the ControlLogic object, it starts the second thread that in turn starts the CarControllerService and KinectService via the StartControl() method. When the Stop() method is called, a CancellationTokenSource associated with the second thread stops the thread by calling another method, StopControl() which stops the CarControllerService and KinectService. See Figure 25 - ControlLogic Start/Stop Methods for code example.

6.4.6.2 ControlCar() method

The ControlCar() method is called every time a Skeleton array is returned by the Kinect SDK which is 30 times a second [1]. The method pulls the nearest tracked skeleton that will be used to control the remote control car. It keeps track of the skeleton by the unique ID associated with the Skeleton provided by the Kinect SDK [1]. If the nearest tracked skeleton changes, then control of the remote control car is disabled and the new user will have to start control again.

The ControlCar() method then uses the tracked skeleton to generate three three-dimensional vectors to represent the left and right hands, as well as the midpoint between the two hands. ControlCar() will then call the EnableDisableControl() method which will determine if the countdown timer needs to start for control, or control needs to be disabled due to the hands moving out of the play area. From here the ControlCar() method calls the CalculateDirection() and CalculateVelocity() methods

which calculate and send the direction and velocity values to the CarControllerService based on the position of the hands.

6.4.6.3 EnableDisableControl()

The EnableDisableControl() method is responsible for determining when the user is actually in control of the remote control car. (See Figure 30 - Enable/Disable Control State Diagram and Figure 31 - Enable/Disable Control Timer State Diagram for state diagrams of the EnableDisableControl() Method.) It does this by checking the position of the user's hands (See Figure 32 - HandsStable() and HandsInPlayArea() Method), then acting appropriately. If the user is not currently in control of the car, the method will check to see if the hands are horizontal to each other as well as in the play area, which is defined as a certain height above the spine joint in the Skeleton data. If the hands are in the appropriate position then the countdown timer will start. The user must then keep their hands in position until the timer counts to zero (currently three seconds). Then the user is in control of the remote control car. If the user's hands stray from the checked thresholds then the EnableDisableControl() method will disable control. If the countdown timer was active at the time, then the timer is reset. If the user is currently controlling the car, then the EnableDisableControl() method will check to make sure the user's hands are in the play area as determined by the HandsInPlayArea() method. It will disable control if the hands leave the play area. See Figure 29 - EnableDisableControl() Method for full code. When control is disabled a Reset() method is called on the CarControllerService which in turn calls a Reset() method on the mbed microprocessor that resets the potentiometers to their middle positions stopping the car.

6.4.6.4 CalculateDirection() method

The CalculateDirection() method handles determining the value that will be sent to the car controller for the direction of the remote control car. Direction is calculated as if the user is holding a steering wheel. When the user moves their hands to the left as if holding a steering wheel, then the remote control car's wheels turn accordingly. The method uses the left hand vector and the midpoint vector to calculate the arctangent angle between the two points. The reason for using the left hand is because a negative direction value indicates a direction to the left and the math is set up to use the left hand to calculate this fact. The CalculateDirection() method uses the difference between the left hand Y value and the midpoint Y value as the opposite side of the triangle, and the difference between the X values for the adjacent side of the triangle. The angle is then calculated as the arctangent of these values.

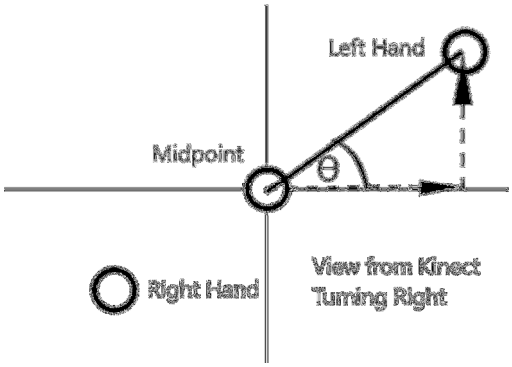

Figure 5 - Graph of Direction Calculation

The angle is checked to see if it is within the acceptable radius that the hands may use. Currently this is Pi/2 radians from center (the trigonometric API's return radians as opposed to degrees). The center is defined as both hands are horizontal to each other. Then the angle is divided by $\frac{\pi}{2}$ to get the percentage and the result is checked to see if it is similar to the value calculated previously. If it is not similar, then the value is sent to the CarControllerService, which then determines the specific value to send to the mbed controller. The value is also sent via an event, to which the user interface (MainViewModel) listens, so it can inform the user of the new direction. See Figure 27 - CalculateDirection() Method for full code.

$$\theta = -\tan^{-1}\frac{leftHandY - midpointY}{leftHandX - midpointY}$$

$$directionPercent = {}^{\theta}/_{(\pi/2)}$$

Figure 6 - Direction Equations

6.4.6.5 CalculateVelocity() method

The CalculateVelocity() method determines the value that will be sent to the car controller for the velocity of the remote control car. The method uses the Z value of the midpoint vector calculated by the ControlCar() method, and the distance the user is from the Kinect. The method first determines the neutral point _neutralDistanceFromBody by finding the distance between the user and the midpoint of their hands. The _neutralDistanceFromBody is used to differentiate between a forward or reverse velocity and their values. This value is saved for the duration of the time the user has control of the remote control car. Using this value as opposed to the distance the user's hands are from the Kinect allows the user to move, if needed, to a better position for controlling the car. The distance between the Kinect and the neutral position is then calculated by finding the distance from the previously calculated _neutralDistanceFromBody and the Kinect itself. After this, the distance the hand midpoint is from the neutral position is calculated. The final velocity percentage is calculated by dividing the distance of the previous hand distance from neutral by a pre-determined distance, MaxVelocityRange. The velocity percentage is then negated as it was calculated with the distance from the Kinect to the neutral position but a velocity in the reverse direction is represented by a negative value. The velocity is then checked to make sure it is not greater than 100%. Otherwise, any percentage greater than 100% would make the car go faster than the settings state. The velocity is also checked to see if it is similar to the previously calculated velocity percentage. If not, then the value is sent to the CarControllerService. The value is also sent via an event, to which the user interface (MainViewModel) listens, so it can inform the user of the new velocity. See Figure 28 - CalculateVelocity() Method for full code.

6.4.7 CarControllerService

The CarControllerService's responsibility is to manage the connection to the mbed controller and to provide a mechanism to translate to the mbed controller specific direction and velocity values from the generalized values used by the Kinect/RC program. The generalized values are percentages

from -100% to 100%. A direction to the left is represented by a negative percentage and a direction to the right is represented by a positive percentage. For velocity, a negative percentage represents a velocity in reverse and a positive percentage represents a velocity in the forward direction.

The generalized values are used to help give the program a plugin architecture. Another programmer who creates his or her own custom car controller may use different hardware. The different hardware most likely will require different values for determining direction and velocity for the remote control car. To keep the main program loosely coupled, the values it uses need to be logically the same across any custom hardware used. See 11.4 CarControllerService Direction & Velocity Methods for code samples of the methods described below.

6.4.7.1 Set/Get Direction

The SetDirection() method takes the direction as a percentage and multiplies it by the range that the direction can be within. The result is then added to the direction midpoint and sent to the mbed controller. The DirectionMidpoint and DirectionRange can be modified in the settings file to calibrate neutral position and limit the turning radius of the remote control car respectively. The GetDirection() method performs the inverse function that is used by SetDirection() on the direction retrieved from the mbed controller and returns a direction percentage.

$$dPercent = \frac{d - dMidpoint}{dRange}$$

$$d = dRange * dPercent + dMidpoint$$

Figure 7 - GetDirection() & SetDirection() Equations

6.4.7.2 Set/Get Velocity

The SetVelocity() method takes the velocity as a percentage and depending on if it is positive or negative (forward or reverse), multiplies it

$$vPercent = \begin{cases} \frac{v - Fmin}{Fmax - Fmin} \; if \; v \geq Fmin \\ \frac{Rmin - v}{Rmax - Rmin} \; if \; v \leq Rmin \end{cases}$$

$$v = \begin{cases} vPercent * (Fmax - Fmin) + Fmin \;\; if \; vPercent \geq 0 \\ vPercent * (Rmin - Rmax) + Rmin \; if \; vPercent < 0 \end{cases}$$

Figure 8 - GetVelocity() & SetVelocity() Equations

by the difference of the maximum and minimum for forward or reverse. The result is then added to the respective minimum and sent to the mbed controller.

The reason for having different maximum and minimum settings for forward and reverse is to first limit how fast the remote control car can travel in either direction, and second, the RC transmitter has a large neutral range in the middle. The different settings allow this range to be reduced as needed to optimize control for the user. The minimum and maximum settings for both directions can be modified in the settings file. The SetVelocity() method performs the inverse function that is used by GetVelocity() on the velocity retrieved from the mbed controller and returns a velocity percentage.

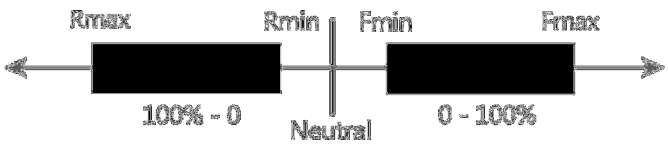

Figure 9 - Velocity Range

7 Test Results

7.1 Controllability

The user is able to fully control the remote control car as intended. The user is able to set the desired direction they wish for the remote control car to travel within its full steering range using the Kinect. The user is also able to regulate the velocity of the remote control car by the distance they move their hands from the neutral position. Since the remote control transmitter is analog-based with potentiometers, there is some initial testing required to verify that the resistances of the circuits are at an acceptable range. According to the remote control car's user manual, this testing is required for before normal use of the car.

The latency from the user's requested direction and velocity via hand motion to the remote control car's reaction is equal to that of using the remote control transmitter itself before modification. No significant delay is noticeable aside from the delay in the remote control car's motor and servo.

The ranges for the remote control car and HTC Desire smart phone were tested outdoors. The range of the remote control car was determined to be about 200 feet. The range of the Desire is limited to the range of the Wi-Fi access point. The access point used for testing is the Linksys WRT54GL. The tested range was about 400 feet. Both ranges are acceptable for controlling the remote control car.

7.2 Wireless Camera Performance

Using a smart phone camera works well to provide a constant video feed to the Kinect/RC program for the user to view, allowing the user to control the remote control car while their vision of the car is blocked. There is a slightly longer latency than would be optimal. The latency is compensated by artificially reducing the maximum velocity of the remote control car via the settings in the program.

The chosen smart phone for the project was the HTC Desire due to its long battery life and the IP Web cam app that supports the MJPEG protocol that the phone supports. Latency of the video from the smart phone to the program averaged 467.17ms for all tested resolutions and quality. Testing the network latency between the smart phone and the test computer, via the ping command, averaged only about three milliseconds. The cause of the latency is most likely a combination of the processing of each video frame from the camera into a JPEG image and the network capacity of the smartphone. Twelve tests were applied using the IP Webcam app on the HTC Desire and the previously used LG Vortex (See 7.2.1 Previous Wireless Cameras Performance for LG Vortex results). The testing of the latency was performed by pointing the smart phone at an Adobe Flash player built digital clock with a millisecond display on a computer screen. Next to the clock on the screen was the output of the MJPEG stream from the smart phone. Both the clock app and the MJPEG stream were displayed in the Chrome web browser. A picture was taken of the digital clock and the MJPEG stream and then the difference in time was calculated. One caveat of this test is at higher resolutions and quality, the frame rate of the MJPEG stream was lower than the frame rate of the digital clock. The result is that some time would pass

between the frames from the MJPEG stream displaying on screen and the capture of the photo of the result, artificially increasing the latency calculated from the photo. This added latency was verified by calculating the difference between latencies with different pictures and the same settings on the smart phones. For example, two identical tests of the HTC Desire at its maximum resolution and MJPEG quality differed by 200 milliseconds from each other.

		Resolution (in pixels)			
		480x320	640x480	960x720	Average
Quality	1%	200	234	400	278
	100%	318	535	1116	656.333
	Average	259	384.5	758	**467.17**

Latency in milliseconds

Figure 10 - HTC Desire Latency Test Results

The latency in processing each frame is a combination of the increase in quality of the image and the increase in resolution, the larger influence being the latter. The latency increased 66.19% on average with the HTC Desire when the resolution increased, but only 52.51% on average when the quality increased. Further testing resulted in using a resolution of 480 x 320 and a quality setting of 50 for the finished project. These settings give the video a decent quality so that the user can easily recognize obstacles while also having a relatively small latency for the video. Latency for this setting was measured at about 318ms.

To compensate for the 318 milliseconds latency in the wireless camera, the top speed of the remote control car is artificially reduced to a maximum of 1.6 m/s. The remote control car requires this speed in order to start moving. However, once the remote control car is moving, the user may then slow the car down to 1.02 m/s. This is the minimum speed the remote control car can travel. If the speed is any slower, then the car stops. The user can change the speed of the remote control car as described in section 4 User Experience. The remote control car will travel between 32.3 cm and 50.88 cm in the time length of the latency.

To calculate the distance travelled during the latency time, the speeds first needed to be calculated. The remote control car was timed to see how long it took to travel a distance of 15 feet. At the faster speed, the remote control car traveled the distance in 3.6 seconds. At the slower speed, it travelled the distance in 4.5 seconds. Thus, the minimum speed required to start the remote control car moving is 1.6 m/s and the slower speed is 1.02 m/s. So then, in 318 milliseconds the car will travel 50.88 cm and 32.3 cm respectively.

7.2.1 Previous Wireless Cameras Performance

The Airlink 101 camera that was initially used provided a maximum resolution of 640 x480. It did provide a setting to change the quality of the video but did not indicate the specific JPEG compression for the settings [7]. Because of this, an accurate comparison in the nature of the test performed on the HTC Desire and Airlink 101 camera cannot be done. However, at the highest quality setting "Highest" and the highest resolution 640x480, the Airlink 101 camera had a latency of about 540ms. At the "Lowest" quality setting and resolution of 160x120 it had a latency of 220ms. Note: the reason this camera was not used is due to the short length of time it could run on battery power before performance dramatically degraded (See 5.4 Wireless Camera).

The LG Vortex smart phone was tested using the same technique as the HTC Desire. An anomaly in the latency for the 175x144 resolution cannot be accounted for. Multiple calculations and restarts of the phone were performed to verify the result.

		Resolution (in pixels)			
		176x144	480x320	640x480	Average
Quality	1%	348	233	483	354.667
	100%	249	635	1001	628.333
	Average	298.5	434	742	**491.5**

Latency in milliseconds

Figure 11 - LG Vortex Latency Test Results

Comparing the latency with the Vortex and that of the HTC Desire shows an overall increase in latency of 5%. When omitting the previous mentioned anomaly, this figure jumps to 26%. This extra

latency compared to the HTC Desire is likely due to the smart phone's slower processor. Note: the reason this smart phone was not used is due to the slower processing of the video and occasional stuttering every 1 to 2 seconds. The HTC Desire does not have this issue and thus was chosen for normal use with the Kinect/RC program.

8 Future Work

The following are some optional features that would be beneficial to add to the Kinect/RC program:

- Create an event in the ControlLogic class that fires whenever the tracked user changes. Then have the UI listen to that event and update a property that stores the tracked user id. The SkeletonUserControl could then bind to the property and would not need to call GetFirstTrackedSkeleton() which is also called by the ControlCar() method in the ControlLogic class, duplicating code.
- Add a settings window to the user interface to update the settings easily such as the tolerances and maximums and minimums for direction and velocity of the car.
- It would be appropriate to update the SkeletonUserControl to display the depth data as an image and thus highlight the tracked user from the others.
- Full test cases should be added to verify the correctness of the program.
- If the Kinect SDK is updated to include finger tracking, then it would be logical to separate the velocity and direction hand motions from each other. A user could then indicate that "both hands are on the wheel" by making fists with both hands. When the user wishes to change velocity they would open one of their hands and then make a motion in the direction (Forward or Reverse) they wish for the vehicle to travel. Testing would need to be done to see if this would be better than the current control scheme.

9 References

[1] Microsoft Corporation, "Kinect for Windows SDK," 16 June 2011. [Online]. Available: http://msdn.microsoft.com/en-us/library/hh855347.aspx. [Accessed 16 June 2011].

[2] Elisabeth Freeman, E. Freeman, B. Bates, K. Sierra and E. Robson, Head First Design Patterns, Sebastopol, CA: O'Reilly Media, 2004.

[3] Horizon Hobby, "Electrix RC Boost Instruction Manual," January 2011. [Online]. Available: http://www.ecxrc.com/ProdInfo/Files/ECX3000_manual_English.pdf. [Accessed August 2011].

[4] "MCP4261," 10 December 2008. [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/22059b.pdf. [Accessed 20 July 2011].

[5] mbed, "mbed NXP LCP1768," 21 July 2010. [Online]. Available: http://mbed.org/handbook/mbed-NXP-LPC1768. [Accessed 20 July 2011].

[6] B. Klug, "Microsoft Kinect: The AnandTech Review," 9 December 2010. [Online]. Available: http://www.anandtech.com/show/4057/microsoft-kinect-the-anandtech-review/. [Accessed 21 May 2012].

[7] Airlink 101, "AIC16020W," [Online]. Available: http://airlink101.com/products/aic1620w.php. [Accessed 3 August 2011].

[8] Microsoft Corporation, "MediaElement Class," August 2011. [Online]. Available: http://msdn.microsoft.com/en-us/library/system.windows.controls.mediaelement.aspx. [Accessed 8 August 2011].

[9] ZeBobo5, "VideoLan DotNet for WinForms, WPF & Silverlight 5," 29 September 2011. [Online]. Available: https://vlcdotnet.codeplex.com/. [Accessed 17 October 2011].

[10] Google, "Android SDK," [Online]. Available: https://developer.android.com/sdk/ndk/overview.html. [Accessed 26 February 2011].

[11] P. Khlebovich, Interviewee, *Developer.* [Interview]. 30 April 2012.

[12] Verizon Wireless, "LG Vortex," 18 November 2010. [Online]. Available: https://www.verizonwireless.com/b2c/store/controller?item=phoneFirst&action=viewPhoneDetail&selectedPhoneId=5620. [Accessed 20 February 2011].

[13] HTC Corporation, "HTC Desire S Product Overview," 15 February 2011. [Online]. Available: http://www.htc.com/www/smartphones/htc-desire-s/#specs. [Accessed 30 April 2012].

[14] M. Walker, "RPC Interface Library - Cookbook | mbed," 22 January 2011. [Online]. Available: http://mbed.org/cookbook/RPC-Interface-Library. [Accessed 27 July 2011].

[15] Microsoft Corporation, "Developer's Guide to Microsoft Prism," 12 November 2010. [Online]. Available: http://msdn.microsoft.com/en-us/library/gg406140.aspx. [Accessed 3 June 2011].

[16] Microsoft Corporation, "Windows Presentation Foundation," November 2006. [Online]. Available: http://msdn.microsoft.com/en-us/library/ms754130.aspx. [Accessed 20 June 2011].

[17] J. Smith, "The Model-View-ViewModel (MVVM) Design Pattern," February 2009. [Online]. Available: http://msdn.microsoft.com/en-us/magazine/dd419663.aspx. [Accessed September 2011].

[18] Xceed, "Extended WPF Toolkit," 29 July 2010. [Online]. Available: https://wpftoolkit.codeplex.com/. [Accessed 8 February 2012].

[19] Microsoft Corporation, "Windows Management Instrumentation," 9 March 2012. [Online]. Available: http://msdn.microsoft.com/en-us/library/aa394582.aspx. [Accessed 14 April 2012].

[20] B. Peek, "MJPEG Decoding | Coding4Fun Articles | Channel 9," Microsoft Corporation, 9 February 2011. [Online]. Available: http://channel9.msdn.com/coding4fun/articles/MJPEG-Decoder. [Accessed 23 July 2011].

[21] Microsoft Corporation, "Dispatcher Class," August 2011. [Online]. Available: http://msdn.microsoft.com/en-us/library/system.windows.threading.dispatcher.aspx. [Accessed 2 May 2012].

10 Bibliography

Below are various websites that were extremely helpful in developing this project and have excellent information overall for other projects in general.

Stack Overflow – http://www.stackoverflow.com

Microsoft Developer Network (MSDN) – http://www.msdn.com

mbed microcontroller – http://www.mbed.org

Kinect for Windows – http://www.kinectforwindows.org

MSDN Channel 9 – http://channel9.msdn.com

11 Figures

11.1 Hardware

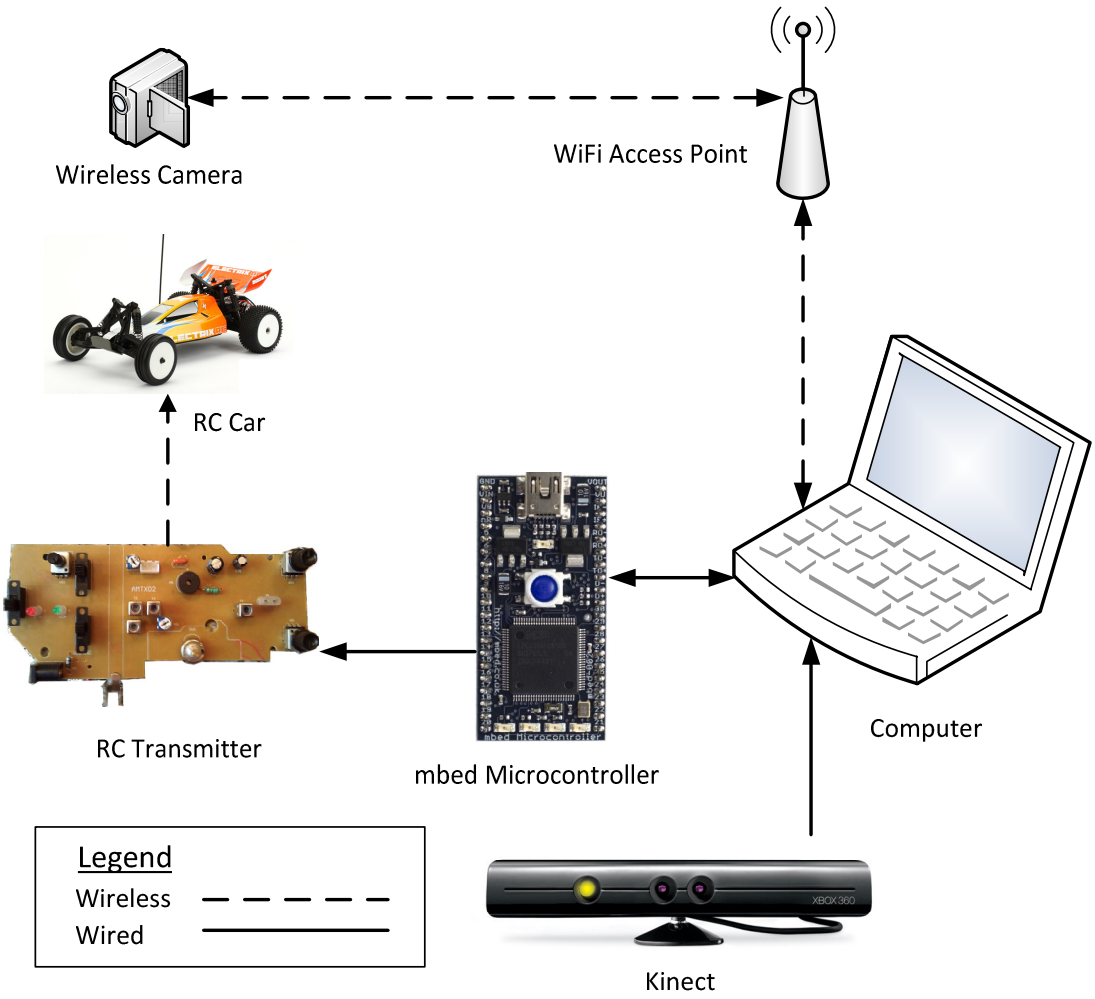

Figure 12 - Block Diagram

Figure 13 – Wi-Fi Access Point WRT54GL

Figure 14 - Remote Control Car

11.1.1 Kinect

Figure 15 - Kinect Sensor

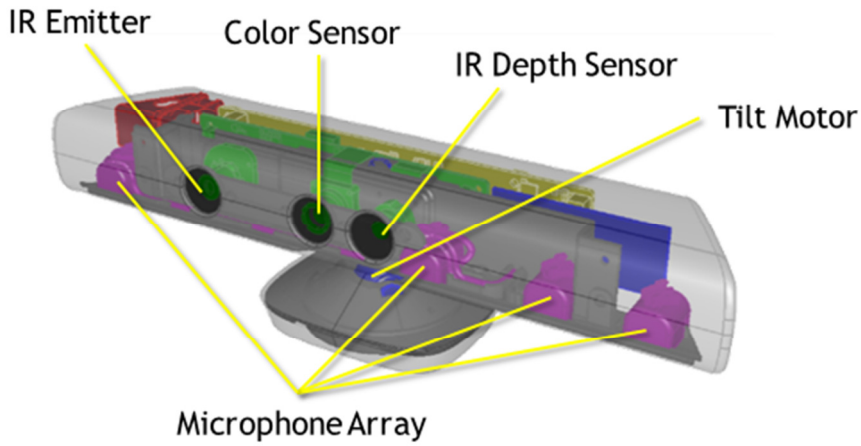

Figure 16 - Kinect Sensor (Components)

11.1.2 Car Controller

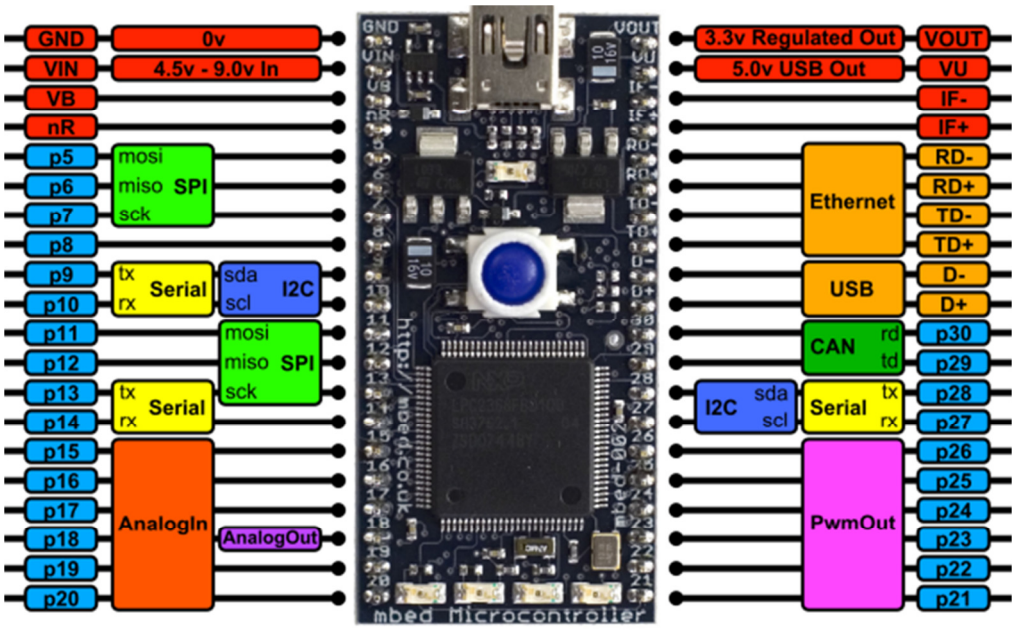

Figure 17 - Mbed Microcontroller Diagram

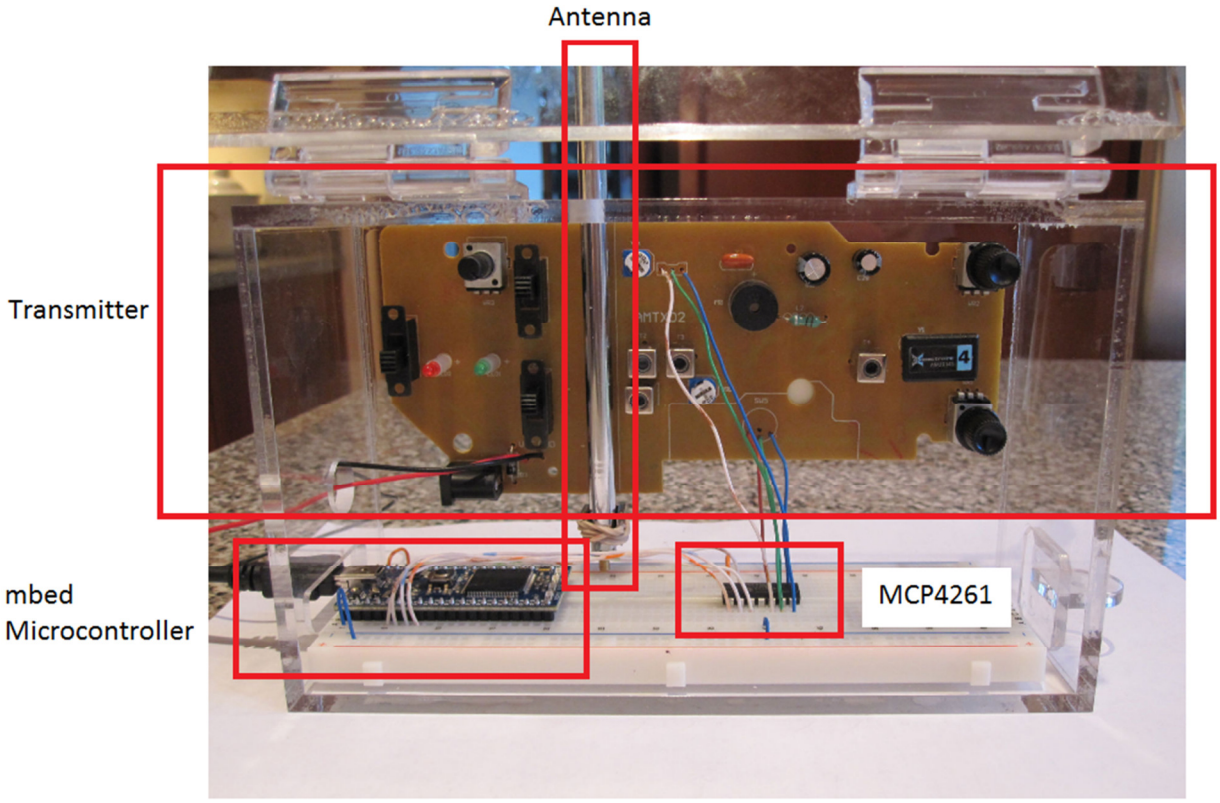

Figure 18 - Car Controller Device

11.1.3 Camera Mounts

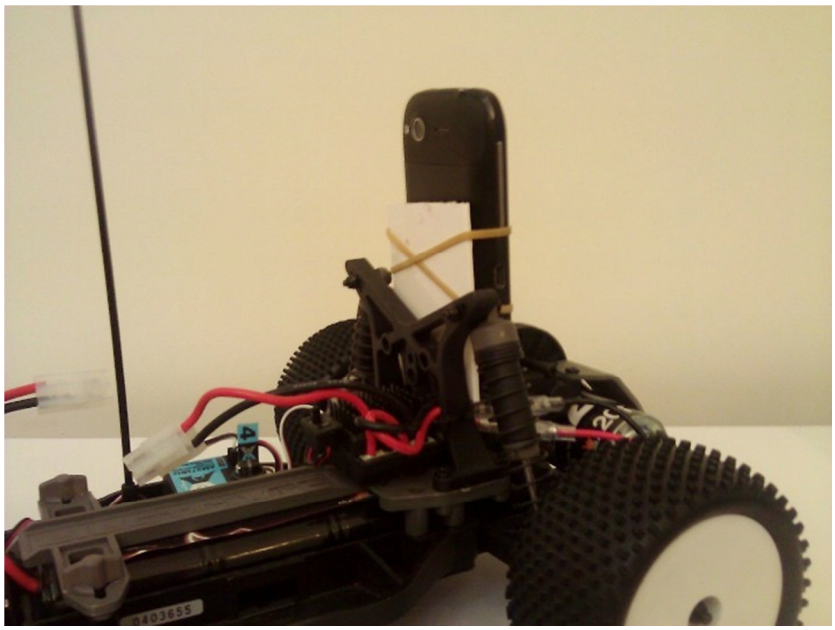

Figure 19 - Remote Control Car Mount with Smart Phone

Figure 20 - Remote Control Car Mount with Airlink 101 1620W Camera

11.2 User Interface

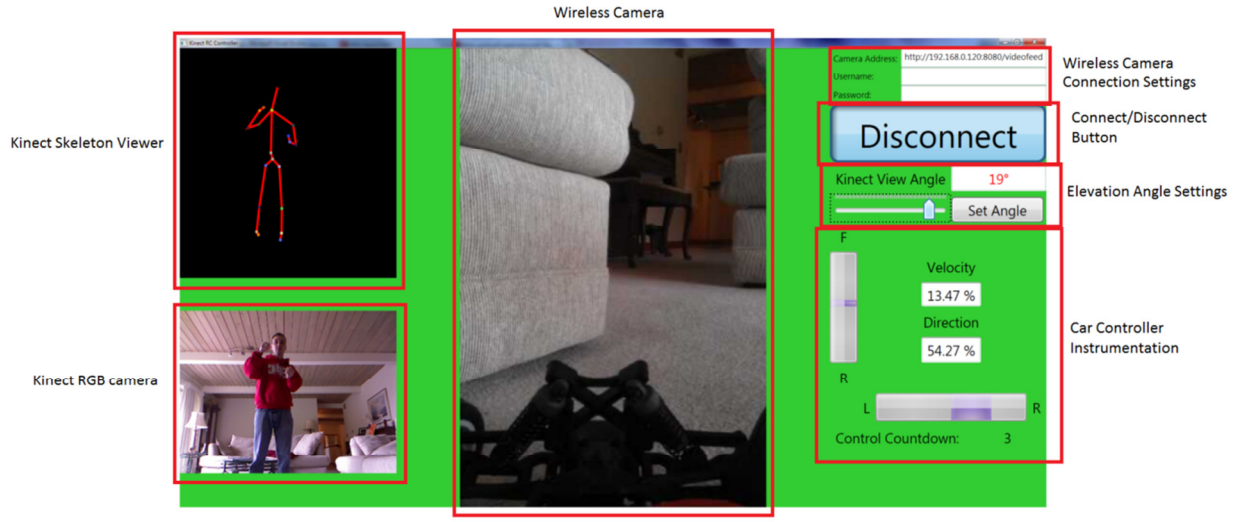

Figure 21 - User Interface

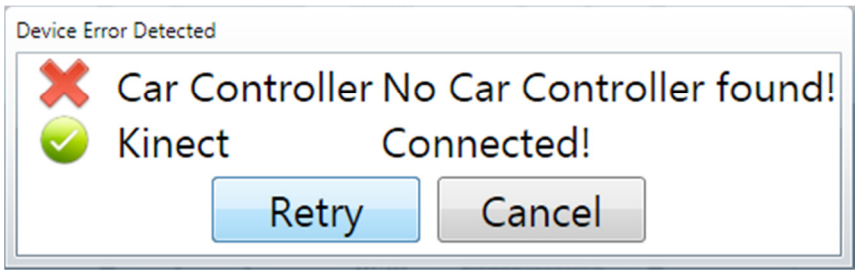

Figure 22 - DeviceStatusUserControl

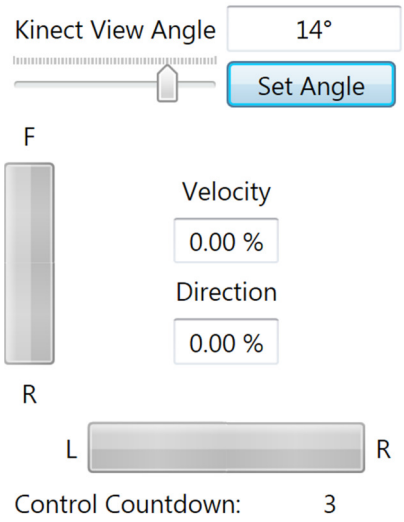

Figure 23 - InstrumentsUserControl

```
private static void OrientationPropertyChanged(DependencyObject d, DependencyPropertyChangedEventArgs e)
{
    BidirectionalIndicator bidirectionalProgressbar = (BidirectionalIndicator)d;
    Orientation orientation = (Orientation)e.NewValue;

    // Don't do anything if the orientation is the same as before.
    if (orientation == (Orientation)e.OldValue)
        return;

    switch (orientation)
    {
        case Orientation.Horizontal:
            // convert the layout grid from 1x2 to 2x1
            bidirectionalProgressbar.layoutRoot.RowDefinitions.RemoveAt(1);
            bidirectionalProgressbar.layoutRoot.ColumnDefinitions.Add(new ColumnDefinition());

            // reposition the progressbars to their appropriate cells
            Grid.SetColumn(bidirectionalProgressbar.positiveProgressBar, 1);
            Grid.SetRow(bidirectionalProgressbar.negativeProgressBar, 0);

            // switch the progressbars' orientations
            bidirectionalProgressbar.negativeProgressBar.Orientation = Orientation.Horizontal;
            bidirectionalProgressbar.positiveProgressBar.Orientation = Orientation.Horizontal;

            // transform the negativeProgressBar by adding a rotation of 180 degrees
            // scoping brackets used to avoid transformGroup name conflict
            {
                TransformGroup transformGroup =
                        (TransformGroup)bidirectionalProgressbar.negativeProgressBar.RenderTransform;
                transformGroup.Children.Add(new RotateTransform(-180));
            }
            break;

        case Orientation.Vertical:
            // convert the layout grid from 2x1 to 1x2
            bidirectionalProgressbar.layoutRoot.ColumnDefinitions.RemoveAt(1);
            bidirectionalProgressbar.layoutRoot.RowDefinitions.Add(new RowDefinition());

            // reposition progressbars to their appropriate cells
            Grid.SetRow(bidirectionalProgressbar.negativeProgressBar, 1);
            Grid.SetColumn(bidirectionalProgressbar.positiveProgressBar, 0);

            // switch the progressbars' orientations
            bidirectionalProgressbar.negativeProgressBar.Orientation = Orientation.Vertical;
            bidirectionalProgressbar.positiveProgressBar.Orientation = Orientation.Vertical;

            // transform the negativeprogressbar by removing the 180 degree rotation
            // scoping brackets used to avoid transformGroup name conflict
            {
                TransformGroup transformGroup =
                        (TransformGroup)bidirectionalProgressbar.negativeProgressBar.RenderTransform;
                transformGroup.Children.RemoveAt(1);
            }
            break;

        default:
            throw new NotImplementedException("Orientation not implemented.");
    }
}
```

Figure 24 - BidirectionalIndicator Orientation Change Method

11.3 Control Logic Code Excerpts

```
public void Start()
{
	if (_carControllerService.Status != CarControllerStatus.Connected ||
		_kinectService.Status != KinectStatus.Connected)
		return;

	_cancellationTokenSource = new CancellationTokenSource();
	_cancellationTokenSource.Token.Register(StopControl);

	_task = new Task(StartControl, _cancellationTokenSource.Token);
	_task.Start();
}

private void StartControl()
{
	_carControllerService.Start();
	_kinectService.Start();
}

private void StopControl()
{
	_kinectService.Stop();
	_carControllerService.Start();
}

public void Stop()
{
	if (_cancellationTokenSource == null ||
		_cancellationTokenSource.IsCancellationRequested)
		return;

	_cancellationTokenSource.Cancel();
}
```

Figure 25 - ControlLogic Start/Stop Methods

```
// Used to keep track of who is controlling the car
private int _trackedId;

// Parse the skeleton for use to control the car
private void ControlCar(Skeleton[] skeletons)
{
        // Get the closest tracked user
        Skeleton trackedSkeleton = skeletons.GetFirstTrackedSkeleton();

        // don't do anything if there is no one to track
        if (trackedSkeleton == null)
        {
                DisableControl();
                return;
        }

        // lost track of the previous user
        // stop controlling the car and start tracking
        // new user
        if (trackedSkeleton.TrackingId != _trackedId)
        {
                DisableControl();

                _trackedId = trackedSkeleton.TrackingId;
        }

        // get a vector representing the hands for easier coding
        Vector3D leftHand = GetJointVector(JointType.HandLeft, trackedSkeleton);

        Vector3D rightHand = GetJointVector(JointType.HandRight, trackedSkeleton);

        // Only continue if the hands are not crossed
        if (leftHand.X >= rightHand.X) return;

        // find the mindpoint between the two hands
        Vector3D handsMidpoint = new Vector3D((rightHand.X + leftHand.X)/2.0,
                                              (rightHand.Y + leftHand.Y)/2.0,
                                              (rightHand.Z + leftHand.Z)/2.0);

        // get the vertical position of the spine
        double spineYPosition = trackedSkeleton.Joints[JointType.Spine].Position.Y;

        // check if it is okay to control the car
        if (!EnableDisableControl(handsMidpoint, spineYPosition, leftHand, rightHand))
                return;

        // calculate the direction for the car
        CalculateDirection(leftHand, handsMidpoint);

        // calculate the velocity of the car
        CalculateVelocity(handsMidpoint.Z, trackedSkeleton.Position.Z);
}
```

Figure 26 - ControlCar() Method

```
private double _previousDirection;

/// <summary>
/// Calculates the Direction to have the car go.
/// The direction is calculated as a percentage of the angle between the
/// position of the <paramref name="leftHand"/> and 0.
/// The calculated angle is then divided by Pi/2 to determine the direction
/// as a percentage.
/// A positive direction is a direction to the right.
/// A negative position is a direction to the left.
/// </summary>
/// <param name="leftHand">The hand to be used to calculate the direction.
/// Note: this must be the lefthand or the calculation will be reversed.</param>
/// <param name="handsMidpoint">The midpoint of the hands to determine if the
/// direction is left or right based on if the <paramref name="leftHand"/> is
/// above the midpoint or below.</param>
private void CalculateDirection(Vector3D leftHand, Vector3D handsMidpoint)
{
        double oppositeLength = leftHand.Y - handsMidpoint.Y;
        double adjacentLength = leftHand.X - handsMidpoint.X;

        // find the angle in degrees the hands are at
        double handTurnRadians = -(Math.Atan(oppositeLength/adjacentLength));

        // if the angle of the hands is with in the threshold
        // then set the direction of the vehicle appropriately
        if (handTurnRadians < -Settings.Default.MaxRadians
                        || handTurnRadians > Settings.Default.MaxRadians)
                return;

        // calculate the direction as a percentage of the angle of the lefthand
        // and Pi/2 (90 degress vertical).
        double direction = handTurnRadians/(Math.PI/2);
        Debug.WriteLine("Direction: {0}", direction);

        // make sure direction isn't larger than 100% either direction.
        // not sure if needed but just in case;
        if (direction > 1.0 || direction < -1.0)
                return;

        // only update the direction if the direction is different by 10%
        // from the previous direction
        if (Math.Abs(direction - _previousDirection) <= _settings.DirectionDeltaThreshold)
                return;

        _previousDirection = direction;
        _carControllerService.SetDirection(direction);
        OnNewDirectionEvent(direction);
}
```

Figure 27 - CalculateDirection() Method

```
// The z position the neutral position is from the body
// set by the users hand position when control is enabled.
private double? _neutralDistanceFromBody;

private double _previousVelocity;

/// <summary>
/// Calculates the velocity of the car.
/// The velocity is calculated as a percentage of the distance the
/// hands are from a set neutral point.
/// The is able to move back and forth as the neutral point is calculated
/// based on the distance the neutral point is from the user.
/// </summary>
/// <param name="handsMidpointZ">The distance the hands are from the Kinect.</param>
/// <param name="positionZ">The distnace the user is from the Kinect.</param>
private void CalculateVelocity(double handsMidpointZ, double positionZ)
{
        // if there is no neutral postion. Set it to be the handsMidpointZ.
        // Should only calculate once.
        // _neutralDistanceFromBody is reset to null in the
        // DisableControl() method.
        if (!_neutralDistanceFromBody.HasValue)
        {
                _neutralDistanceFromBody = positionZ - handsMidpointZ;
        }

        // calculate the position of neutral relative to the Kinect
        double throttleNeutralPosition = positionZ - _neutralDistanceFromBody.Value;

        // calculate distance the hands are from the neutral position
        double handDistanceFromNeutral = handsMidpointZ - throttleNeutralPosition;

        // calculate the velocity based on the ratio of the hand distance from neutral
        // to the maximum velocity range.
        double velocity = handDistanceFromNeutral/Settings.Default.MaxVelocityRange;

        // invert so forward direction is positive
        velocity = -velocity;

        Debug.WriteLine("Velocity: {0}", velocity);

        // Don't update the velocity if the velocity is greater than +-100%.
        if (velocity > 1.0 || velocity < -1.0) return;

        // only continue if the velocity changes more than 1%
        if (Math.Abs(velocity - _previousVelocity) <= _settings.VelocityDeltaThreshold)
                return;

        _carControllerService.SetVelocity(velocity);
        _previousVelocity = velocity;
        OnNewVelocityEvent(velocity);
}
```

Figure 28 - CalculateVelocity() Method

11.3.1 Enable/Disable Control

```
/// <summary>
/// Checks to see if the hands are in the starting position and in the "play area"
/// before enabling control. Control is disabled if the hands drop outside the "play area.
/// </summary>
/// <param name="handsMidpoint">Vector representing the 3D midpoint of the hands.</param>
/// <param name="spineYPosition">The Y position of the spine.</param>
/// <param name="leftHand">Vector representing the 3D position of the left hand.</param>
/// <param name="rightHand">Vector representing the 3D position of the right hand.</param>
/// <returns>Whether the user can control the car or not.</returns>
private bool EnableDisableControl(Vector3D handsMidpoint, double spineYPosition, Vector3D leftHand,
                                  Vector3D rightHand)
{
        // if we are not controlling
        if (!_isControlling)
        {
                // if the hands are horizontal and in the play area
                if (HandsStable(leftHand, rightHand) && HandsInPlayArea(handsMidpoint, spineYPosition))
                {
                        // if the timer hasn't already been started
                        if (!_countdownTimer.IsEnabled)
                        {
                                // start the timer
                                _countdownTimer.Start();
                        }
                }
                        // if the hands are not horizontal or in the play area
                else
                {
                        // reset
                        DisableControl();
                }
        }
        // if we are controlling
        else
        {
                // if the hands are not in the play area
                if (!HandsInPlayArea(handsMidpoint, spineYPosition))
                {
                        // stop controlling
                        DisableControl();
                }
        }
}
```

Figure 29 - EnableDisableControl() Method

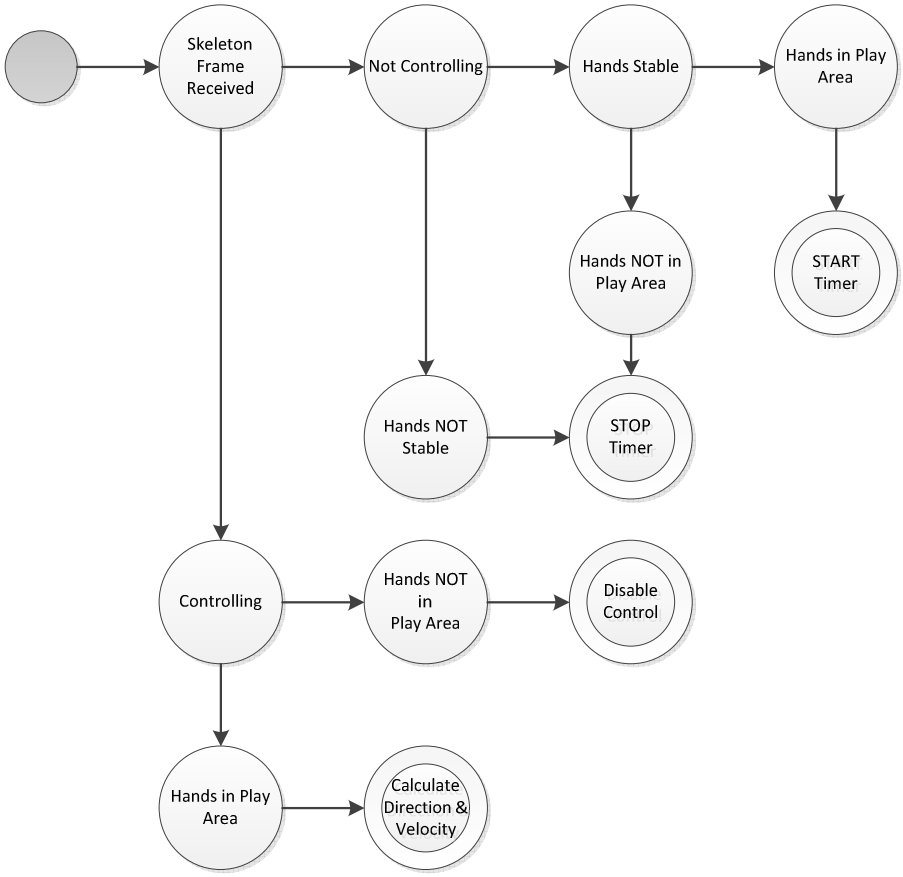

Figure 30 - Enable/Disable Control State Diagram

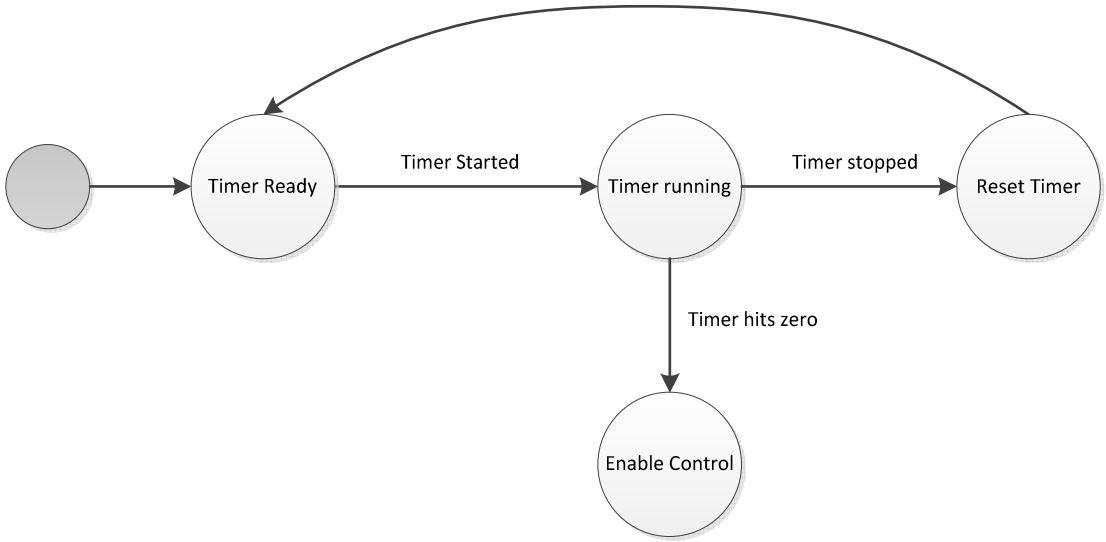

Figure 31 - Enable/Disable Control Timer State Diagram

```
// Checks to see if the hands have the same horizontal position
// and distance from the kinect.
private bool HandsStable(Vector3D leftHand, Vector3D rightHand)
{
        // are the hands horizontal
        bool areHorizontal = Math.Abs(leftHand.Y - rightHand.Y) < _settings.HorizontalHandThreshold;

        // are the hands depth equal
        bool areZStable = Math.Abs(leftHand.Z - rightHand.Z) < _settings.ZHandThreshold;
        return areHorizontal && areZStable;
}

// Checks if the hands are in the play area, defined as being
// a distance above the Y position of the spine.
private bool HandsInPlayArea(Vector3D handsMidpoint, double spinePositionY)
{
        // Check if the both hands are below the base plane

        //defined as above the spine Y point that is the bottom of the play area
        double basePlane = spinePositionY + _settings.DistanceAboveSpineY;

        // Are the hands above the base plane of the play area
        bool handsAbovePlayAreaPlane = handsMidpoint.Y > basePlane;

        return handsAbovePlayAreaPlane;
}
```

Figure 32 - HandsStable() and HandsInPlayArea() Method

11.4 CarControllerService Direction & Velocity Methods

```
public double GetDirection()
{
        int direction = Read(ControlPot.Direction);
        double directionPercent;

        directionPercent = ((double)direction –
                _settings.DirectionMidpoint)/_settings.DirectionRange;

        return directionPercent;
}
```

Figure 33 - GetDirection() Method

```
public void SetDirection(double directionPercent)
{
	if (directionPercent < -1.0 || directionPercent > 1.0)
		throw new ArgumentOutOfRangeException("directionPercent", "Must be a percentage
			between -1 and 1.");

	int directionValue = Convert.ToInt32(_settings.DirectionRange * directionPercent);

	directionValue = _settings.DirectionMidpoint + directionValue;

	Write(ControlPot.Direction, directionValue);
}
```

Figure 34 - SetDirection() Method

```
public double GetVelocity()
{
	int velocity = Read(ControlPot.Velocity);
	double velocityPercent;

	if (velocity >= _settings.ForwardMinimum)
		velocityPercent = ((double)velocity - _settings.ForwardMinimum) /
					(_settings.ForwardMaximum - _settings.ForwardMinimum);

	else if (velocity <= _settings.ReverseMinimum)
		velocityPercent = ((double)_settings.ReverseMinimum - velocity) /
					(_settings.ReverseMaximum - _settings.ReverseMinimum);
	else
		velocityPercent = 0.0;

	return velocityPercent;
}
```

Figure 35 - GetVelocity() Method

```
public void SetVelocity(double velocityPercent)
{
        if (velocityPercent < -1.0 || velocityPercent > 1.0)
                throw new ArgumentOutOfRangeException("velocityPercent", "Must be a percentage
                                                                between -1 and 1.");

        int velocityValue;

        if (velocityPercent >= 0)
        {
                velocityValue = Convert.ToInt32((_settings.ForwardMaximum –
                                                        _settings.ForwardMinimum)*velocityPercent);

                velocityValue = velocityValue + _settings.ForwardMinimum;
        }
        else
        {
                velocityValue = Convert.ToInt32((_settings.ReverseMinimum –
                                                        _settings.ReverseMaximum)*velocityPercent);

                velocityValue = velocityValue + _settings.ReverseMinimum;
        }

        Write(ControlPot.Velocity, velocityValue);
}
```

Figure 36 - SetVelocity() Method

11.5 Adapter Pattern Class Diagrams

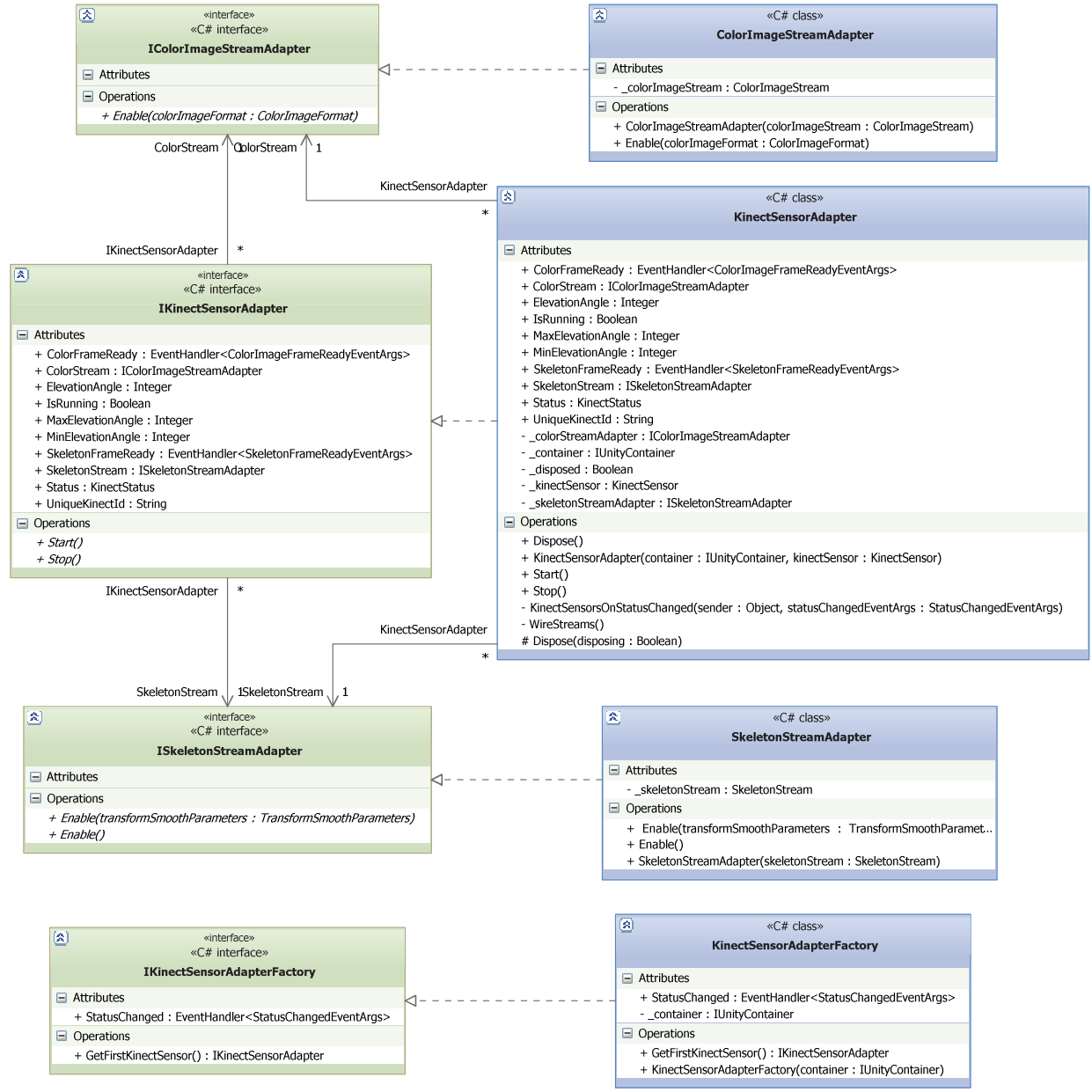

Figure 37 - Kinect Adapters Class Diagram

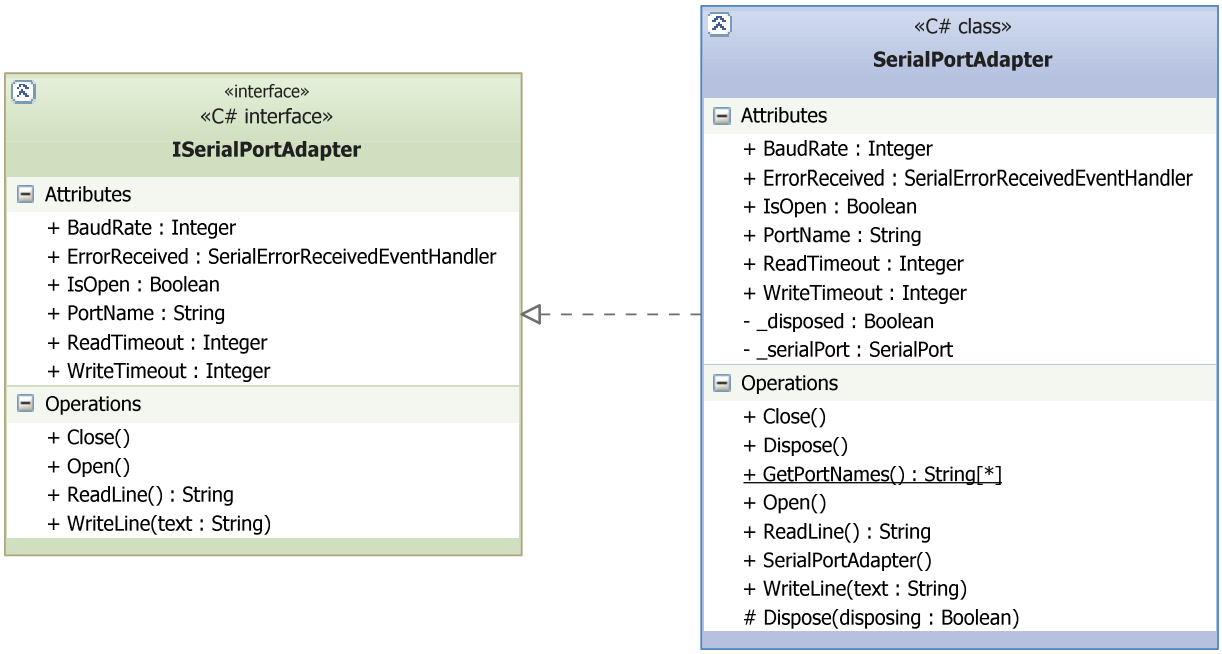

Figure 38 - SerialPort Adapter Class Diagram

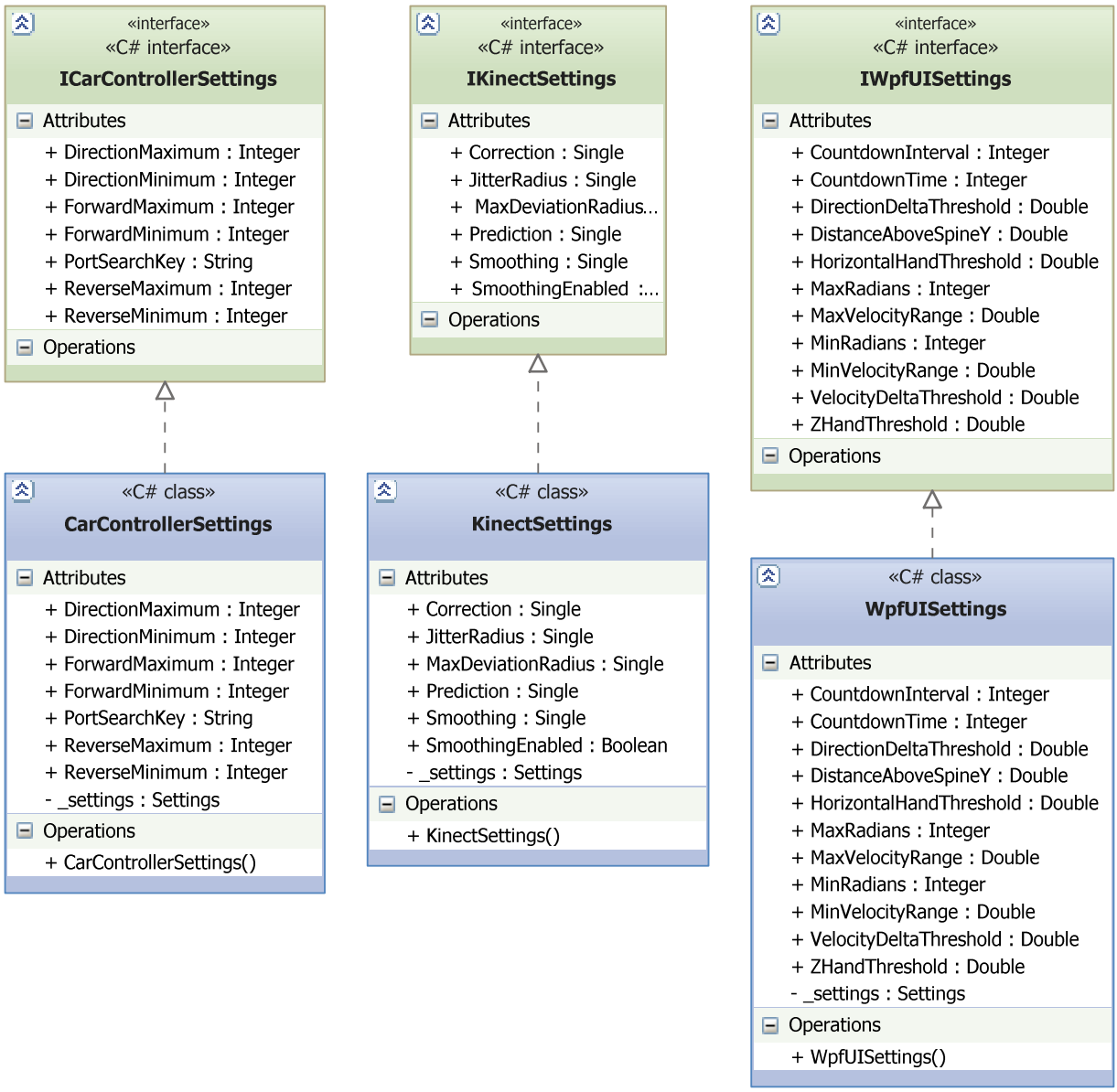

Figure 39 - Settings Adapter Class Diagram

11.6 Mbed

11.6.1 Transmission Byte Formats

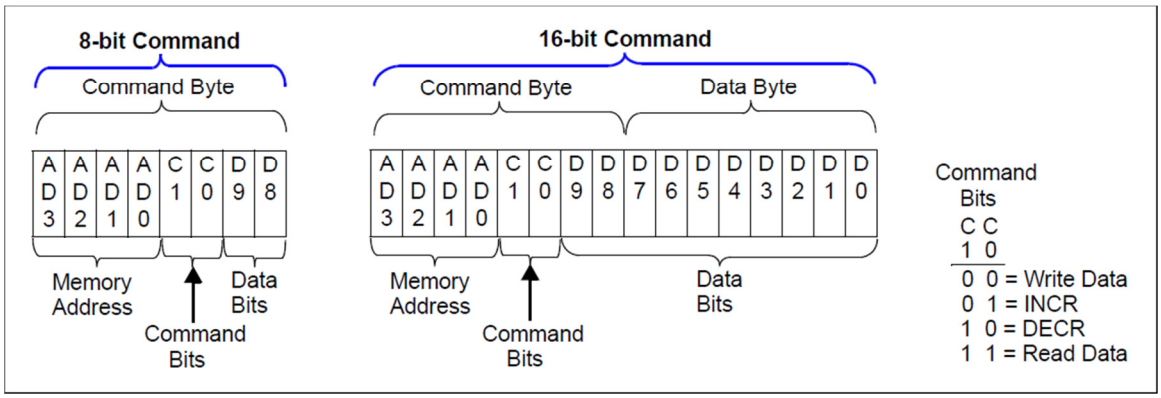

Figure 40 - Send Bytes Format

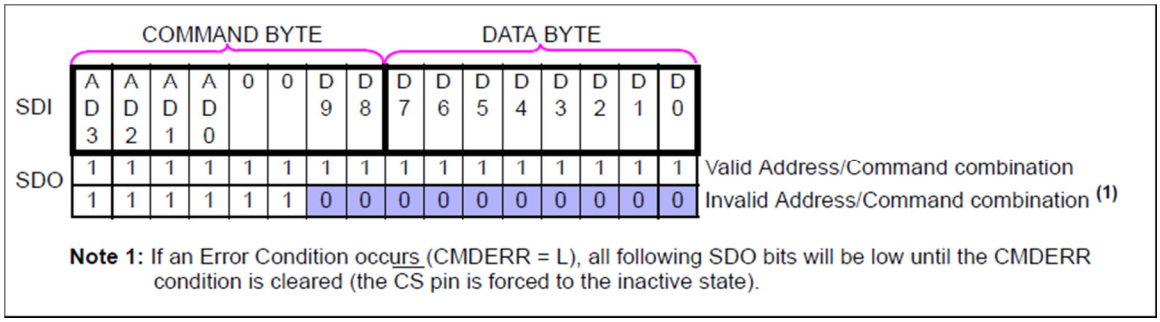

Figure 41 - Receive Bytes Format

11.6.2 Code Examples

```
int MCP4261::Read(Address address)
{
    int response = 0;

    SetChipSelect(true);

    if (CheckAddress(address))
    {
        int command;

        command = address << 2;
        command = (command | 3) << 10;

        _spi->format(16);

        response = _spi->write(command);
    }

    if (!GetIsContinuous())
        SetChipSelect(false);

    bool isValid = CheckValidResult(response, CMDERR_16);

    if(isValid)
        return response - 65024;
    else
        return -1;
}

bool MCP4261::Write(Address address, int data)
{
    int response = 0;

    SetChipSelect(true);

    if (CheckAddress(address) && data >= 0 && data <= 256)
    {
        int command;

        command = address << 2;
        command = (command | 0) << 10;
        command = command | data;

        _spi->format(16);
        response = _spi->write(command);

        if (!GetIsContinuous())
            SetChipSelect(false);
    }

    bool isValid = CheckValidResult(response, CMDERR_16);

    return isValid;
}
```

Figure 42 - MCP4261.cpp Read() & Write() Methods

```
RPCFunction Read(&read, "Read");
RPCFunction Write(&write, "Write");

void read(char * input, char * output) {
     MCP4261::Address address = (MCP4261::Address) atoi(input);

     int response = _rcController.Read(address);

     sprintf(output,"%d",response);
}

void write(char * input, char * output) {

     MCP4261::Address address = (MCP4261::Address) atoi(strtok(input, " "));

     int data = atoi(strtok(NULL, " "));

     int response = _rcController.Write(address, data);

     sprintf(output,"%d",response);
}
```

Figure 43 - RPCFuncs.c Read() & Write() Methods

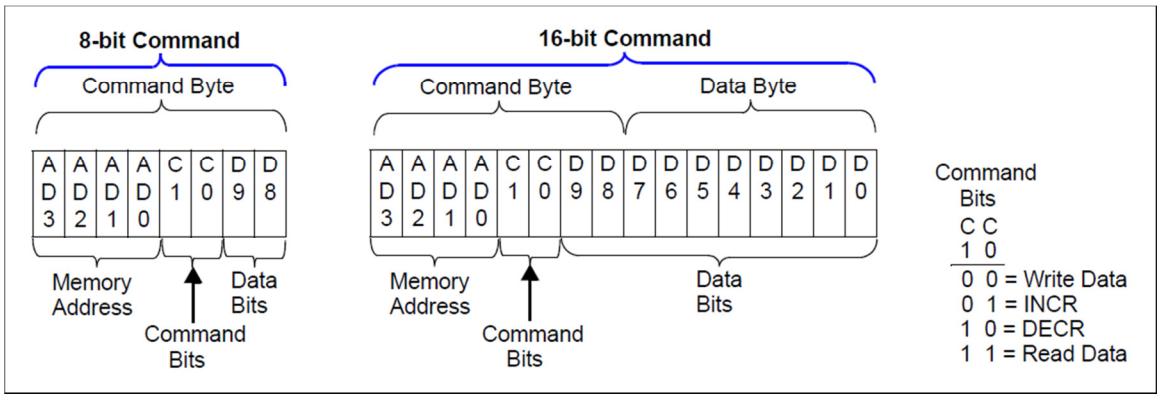

Figure 44 - Command Byte Format

11.6.3 Mbed WMI __InstanceCreationEvent Examples

```
instance of Win32_PnPEntity
{
        Caption = "mbed Serial Port (COM3)";
        ClassGuid = "{4d36e978-e325-11ce-bfc1-08002be10318}";
        CompatibleID = {"USB\\CLASS_02&SUBCLASS_02&PROT_01", "USB\\CLASS_02&SUBCLASS_02",
                "USB\\CLASS_02"};
        ConfigManagerErrorCode = 0;
        ConfigManagerUserConfig = FALSE;
        CreationClassName = "Win32_PnPEntity";
        Description = "mbed Serial Port";
        DeviceID = "USB\\VID_0D28&PID_0204&MI_01\\1010000000000000000000002F7F06B26";
        Manufacturer = "mbed";
        Name = "mbed Serial Port (COM3)";
        PNPDeviceID = "USB\\VID_0D28&PID_0204&MI_01\\1010000000000000000000002F7F06B26";
        Service = "mbedSerial";
        Status = "OK";
        SystemCreationClassName = "Win32_ComputerSystem";
        SystemName = "MAIN";
};
```

Figure 45 - Mbed WMI __InstanceCreationEvent x86 Example

```
instance of Win32_PnPEntity
{
        Caption = "mbed Serial Port (COM3)";
        ClassGuid = "{4d36e978-e325-11ce-bfc1-08002be10318}";
        CompatibleID = {"USB\\CLASS_02&SUBCLASS_02&PROT_01", "USB\\CLASS_02&SUBCLASS_02",
                "USB\\CLASS_02"};
        ConfigManagerErrorCode = 0;
        ConfigManagerUserConfig = FALSE;
        CreationClassName = "Win32_PnPEntity";
        Description = "mbed Serial Port";
        DeviceID = "USB\\VID_0D28&PID_0204&MI_01\\1010000000000000000000002F7F06B26";
        HardwareID = {"USB\\VID_0D28&PID_0204&MI_01"};
        Manufacturer = "mbed";
        Name = "mbed Serial Port (COM3)";
        PNPDeviceID = "USB\\VID_0D28&PID_0204&MI_01\\1010000000000000000000002F7F06B26";
        Service = "mbedSerial_x64";
        Status = "OK";
        SystemCreationClassName = "Win32_ComputerSystem";
        SystemName = "MAIN";
};
```

Figure 46 - Mbed WMI __InstanceCreationEvent x86 Example