

User Guide – Remote Flight Controller

Introduction to the Remote Flight Controller

This is a user guide for the Remote Flight Controller program which interacts with the Flight Simulator program. A quick overview of the program's features are that it connects to the Flight Simulator which allows the Remote Controller the ability to read Telemetry Data and display it at runtime as a constant thread in the background and also be able to send control data back to the Flight Simulator (Elevator Pitch and Throttle) in order to control the Flight Simulator. Any control data sent by the user is then stored in a listbox showing the latest message sent at the top.

If the telemetry data received displays a warning code other than 0, a warning message will appear which will contain information on how to circumvent that issue e.g. *"plane is too low to ground, please increase altitude"*.

The Remote Flight Controller also contains an AutoPilot which can be activated by checking/ unchecking the checkbox. The AutoPilot will save the current speed of the Flight Simulator and attempt to maintain that speed by changing the throttle depending on the situation. It also tries to maintain the altitude by changing pitch controls. It should be noted that the AutoPilot should only be used to stay at a constant speed at a comfortable altitude otherwise you risk crashing (AKA the AutoPilot is a not a 'Get out of Jail card' if your plane is in a bad situation!).

Below is an image of the Remote Flight Controller program in action. Similar functions are placed in a group box which clearly display what functions these offer. For example, 'Telemetry Data' displays all telemetry data of the Flight Simulator or 'Control Data' holds functionality that controls the Flight Simulator.

The screenshot shows the 'Remote Flight Controller' application window. It is divided into several sections:

- Connect To Simulator:** A section with a label 'Please enter the IP Address to connect to:'. It contains two input fields: 'IP:' with the value '192.168.0.7' and 'Port:' with the value '9999'. A 'Connect' button is located below these fields.
- Control Data:** A section with two input fields: 'Elevator Pitch:' with the value '2' and 'Throttle:' with the value '35'. Below these is an unchecked checkbox labeled 'AutoPilot' and a 'Send' button.
- Telemetry Data:** A section displaying various flight metrics:
 - Altitude: 5501.44480880114 ft
 - Speed: 186.603587366836 Knts
 - Pitch: 9.18145898753935 degrees
 - Vertical Speed: 3015.25833127582 ft p/m
 - Throttle: 35%
 - Elevator Pitch: 2 degrees
 - Warning Code: 0
- Messages Sent By User:** A listbox containing four entries, each showing 'Elevator Pitch: 2, Throttle: 35' followed by a sequence of three updates where the pitch increases to 3 and the throttle increases to 75.
- Warning:** A section with the text 'All systems functional. No warnings present.'
- NOTE:** A text box at the bottom right stating 'NOTE: AutoPilot should be used to stay at a constant speed only!'

Inside the Program's Code

Before talking about what each part of the Remote Flight Controller does, it is best to talk about how the data is stored and handled via delegates and events first.

To store the control and telemetry data, there are two public structs that hold variables related to their specific usage. These are called 'ControlsUpdate' which holds variables that are used to control the Flight Simulator and 'TelemetryUpdate' which is used to display data about the plane.

```
// struct that holds variables needed to control
// the flight simulator
public struct ControlsUpdate
{
    public double Throttle;
    public double ElevatorPitch;
}
```

```
// struct that holds variables needed to show
// telemetry data to the program
public struct TelemetryUpdate
{
    public double Altitude;
    public double Speed;
    public double Pitch;
    public double VerticalSpeed;
    public double Throttle;
    public double ElevatorPitch;
    public int WarningCode;
}
```

Now there are structs capable of holding all the data stored within the Flight Simulator, we can use delegates to handle this data by passing an object of 'ControlsUpdate' or 'TelemetryUpdate' when needed. Delegates are like pointers in C++ as it allows the user to encapsulate a reference to a method inside a delegate object. Basically, they can be seen as a placeholder for any method that wishes to use its reference type e.g.

```
public delegate void ControlsUpdateHandler(ControlsUpdate newcontroldata);
```

means that this delegate named 'ControlsUpdateHandler' accepts an object of type 'ControlsUpdate' and it will be used each time the delegate is called.

I would want to use this delegate each time I need to pass any control data as an argument for example when sending control data to the Flight Simulator to control the plane.

```
public delegate void ControlsUpdateHandler(ControlsUpdate newcontroldata); //
public delegate void TelemetryUpdateHandler(TelemetryUpdate newtelemetrydata);
public delegate void WarningUpdateHandler(string warning); // delegate that ac
```

To use these delegates, events are used. Events are used to signal that when a certain event occurs such as a button click or keypress, all subscribers of that event are notified and execute a certain method related to that subscriber. Each subscribed method would receive the same data passed by the event but will all have a different usage for it.

For example, a competitive multiplayer game could have a method which shows the best players in each category at the end of the round but this can lead to cluttered code which can be quite difficult to modify in the future. Instead, there could be an end-of-round event which passes an object holding data such as most kills, most deaths, most flags captured etc. meaning that there

could be subscribers with methods such as 'mvpWithMostKills', 'loserWithMostDeaths' and 'mvpWithMostFlagsCaptured' for example.

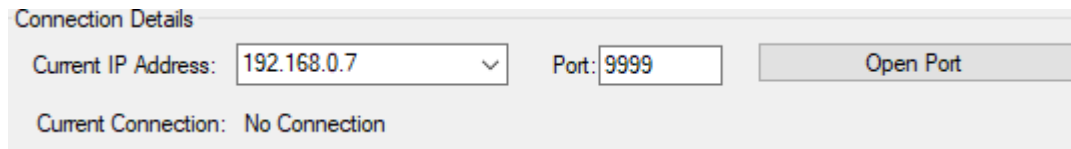
```
// declaring events based on delegate type as given below
public event ControlsUpdateHandler onControlsUpdate;
public event TelemetryUpdateHandler onTelemetryUpdate;
public event WarningUpdateHandler onWarningUpdate;
```

Subscribed/Registered Delegates

```
// registering an event to a delegate on program startup
MyRemoteFlightController.onTelemetryUpdate += new TelemetryUpdateHandler(showTelemetry);
MyRemoteFlightController.onWarningUpdate += new WarningUpdateHandler(showWarning);
MyRemoteFlightController.onControlsUpdate += new ControlsUpdateHandler(showControl); //
```

Connecting to the Flight Simulator & Listening for Telemetry Data

Connecting to the Flight Simulator is simple. All you need is the host's (Flight Simulator Program) IP address and the port number they're using. The specified IP and Port can be found here on the Flight Simulator Program:



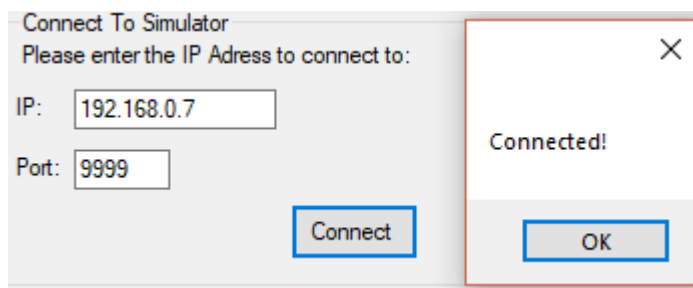
Connection Details

Current IP Address: 192.168.0.7 Port: 9999 Open Port

Current Connection: No Connection

You must also ensure that you click the 'Open Port' button to allow the Remote Controller client to connect to the Flight Simulator.

On the Remote Controller client, enter the IP address and port number found on the Flight Simulator press 'Connect' to establish a connection. You will know if you have successfully connected if a message box appears stating that you've connected. Also, you can find out on the Flight Simulator program as shown below:



Connect To Simulator

Please enter the IP Address to connect to:

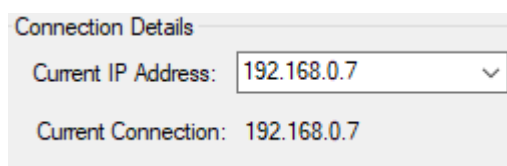
IP: 192.168.0.7

Port: 9999

Connect

Connected!

OK



Connection Details

Current IP Address: 192.168.0.7

Current Connection: 192.168.0.7

When the 'Connect' button is clicked, it calls a method in the RemoteFlightController class called 'connectToFlightSimulator' passing the IP address and port number as an argument.

```
private void btnConnect_Click(object sender, EventArgs e)
{
    MyRemoteFlightController.connectToFlightSimulator(txtIPAddress.Text, txtPing.Text);
}
```

Within the 'connectToFlightSimulator' method, the 'ipaddress' string is converted to an instance of the IPAddress class named 'ip' so that it can be used as an actual IP address. Then a new instance of the TcpClient class is created named 'sender' which allows our client to connect to the host Flight Simulator.

```
IPAddress ip = IPAddress.Parse(ipaddress);
sender = new TcpClient(); // creates a ne
int port = Convert.ToInt16(portnumber); //
```

Now it's possible to connect the client to the host by specifying the IP address and port number required. Once connected, the program will then create a new thread of the method 'listenToFlightSimulator' which endlessly listens for data sent by the Flight Simulator for the Remote Controller client to display the telemetry data given.

Threads allow your program to run more than one operation at a time (multiple processes) in the background whilst you still interact with your program.

```
sender.Connect(ip, port); // connects to the host using the specified ip address and port
MessageBox.Show("Connected!");

// creates a new thread with the method 'listenToFlightSimulator' called 'ListenThread'
// which runs concurrently in the background as a process
Thread ListenThread = new Thread(new ThreadStart(listenToFlightSimulator));
ListenThread.Start(); // starts the 'PilotThread' thread
```

Within the 'listenToFlightSimulator' method, we create a stream from the TCPClient 'sender' which allows data to be sent and received by a network connection. This will be used to read data from the Flight Simulator.

```
NetworkStream stream = sender.GetStream();
```

Once the stream is created, we convert the bytes we've read into a string. However, the string we've converted is in JSON format, so we will need to deserialize it back to its original format which is a telemetry data object.

```
int num_bytes = stream.Read(buffer, 0, 256); // reads the stream from the
string receiveMessage = Encoding.ASCII.GetString(buffer, 0, num_bytes); //

JavaScriptSerializer serializer = new JavaScriptSerializer();
MyTelemetryData = serializer.Deserialize<TelemetryUpdate>(receiveMessage);
```

Then the 'onTelemetryUpdate' event is called passing the telemetry data as its argument. The event then calls the registered subscriber 'showTelemetry' method still referencing the 'MyTelemetryData' object which updates all the labels with the specified telemetry data.

```
onTelemetryUpdate(MyTelemetryData);
```

```
MyRemoteFlightController.onTelemetryUpdate += new TelemetryUpdateHandler(showTelemetry);
```

```
// updates the labels with the telemetry data that has been sent from the Flight Simulator
public void showTelemetry(TelemetryUpdate telemetrydata)
{
    if (this.InvokeRequired)
    {
        this.Invoke(new TelemetryUpdateHandler(showTelemetry), new object[] { telemetrydata });
    }
    else
    {
        lblAltitude.Text = Convert.ToString(telemetrydata.Altitude) + " ft";
        lblSpeed.Text = Convert.ToString(telemetrydata.Speed) + " Knts";
        lblPitch.Text = Convert.ToString(telemetrydata.Pitch) + " degrees";
        lblVerticalSpeed.Text = Convert.ToString(telemetrydata.VerticalSpeed) + " ft p/m";
        lblThrottle.Text = Convert.ToString(telemetrydata.Throttle) + "%";
        lblElevatorPitch.Text = Convert.ToString(telemetrydata.ElevatorPitch) + " degrees";
        lblWarningCode.Text = Convert.ToString(telemetrydata.WarningCode);
    }
}
```

The Remote Controller client should now show telemetry data when the plane is currently active.

Telemetry Data	
Altitude:	5501.44480880114 ft
Speed:	186.603587366836 Knts
Pitch:	9.18145898753935 degrees
Vertical Speed:	3015.25833127582 ft p/m
Throttle:	35%
Elevator Pitch:	2 degrees
Warning Code:	0

'listenToFlightSimulator' also uses a 'onWarningUpdate' event that updates the warning message based on the warning code received. The event passes a string argument stating whether the plane is in a comfortable condition or whether there is an issue such as flying too low or stalling.

```
if (MyTelemetryData.WarningCode != 0)
{
    if (MyTelemetryData.WarningCode == 1)
    {
        onWarningUpdate("Too low! (less than 1000ft)"); // calls the
    }
    else if (MyTelemetryData.WarningCode == 2)
    {
        onWarningUpdate("Stalling!"); // calls the 'onWarningUpdate
    }
}
// if WarningCode is equal to 0, override error message
else if (MyTelemetryData.WarningCode == 0)
{
    onWarningUpdate("All systems functional. No warnings present.");
}
```

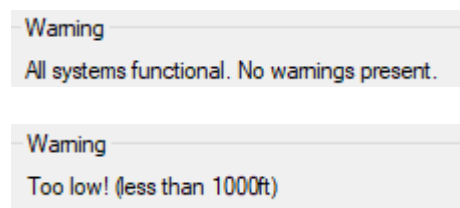
```
MyRemoteFlightController.onWarningUpdate += new WarningUpdateHandler(showWarning);
```

```

public void showWarning(string warning)
{
    if (this.InvokeRequired)
    {
        this.Invoke(new WarningUpdateHandler(showWarning), warning);
    }
    else
    {
        lblWarningMessage.Text = warning;
    }
}

```

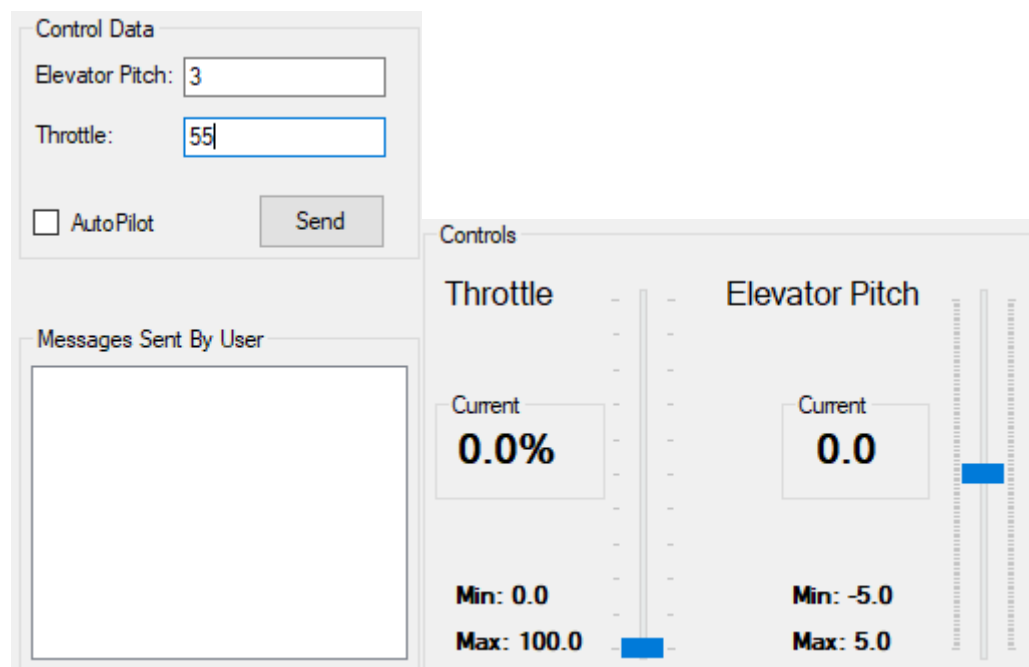
On the Remote Controller client, it looks like this depending on the current situation of the plane:



Sending Control Data to the Flight Simulator

The Control Data panel located on the Remote Controller client gives the ability to send control information to the Flight Simulator as shown below:

Before



After

This is achieved by converting the inputted text strings for throttle and elevator pitch to their respective data types (double) when clicking the 'Send' button. This allows the 'btnSend_Click' method to use the converted control data object as an argument when being called with the 'sendControlEvent' method.

```
// if pitchcheck and throttle check are true, then send both control data to the FlightSimulator
if (pitchcheck == true && throttlecheck == true)
{
    MyControlsUpdate.ElevatorPitch = double.Parse(txtSendElevatorPitch.Text);
    MyControlsUpdate.Throttle = double.Parse(txtSendThrottle.Text);
    MyRemoteFlightController.sendControlEvent(MyControlsUpdate);
}
```

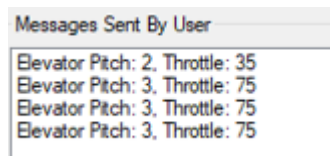
'sendControlEvent' then raises the event 'onControlsUpdate' passing the controldata object as an argument. The event then calls two registered subscribers 'showControl' and 'sendControlData' where both methods are still referencing the control data object.

```
public void sendControlEvent(ControlsUpdate newcontrolsupdate)
{
    onControlsUpdate(newcontrolsupdate);    // calls all events
}
```

```
MyRemoteFlightController.onControlsUpdate += new ControlsUpdateHandler(showControl);
```

```
onControlsUpdate += new ControlsUpdateHandler(sendControlData);
```

The 'showControl' method is used to keep an update on the messages sent to the Flight Simulator by inserting the control data into a listbox as shown below:



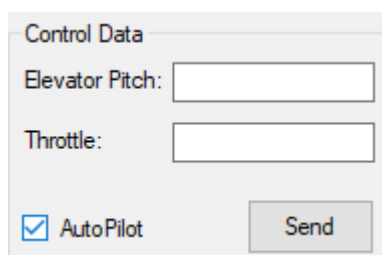
Messages Sent By User	
Elevator Pitch:	2, Throttle: 35
Elevator Pitch:	3, Throttle: 75
Elevator Pitch:	3, Throttle: 75
Elevator Pitch:	3, Throttle: 75

Whereas the 'sendControlData' method serialises the control data into JSON format later converting into bytes before writing it to the stream so that the Flight Simulator can read the JSON formatted message and update the controls with the specified control data.

```
public void sendControlData(ControlsUpdate newcontrolsupdate)
{
    JavaScriptSerializer serializer = new JavaScriptSerializer();
    string jsonmessage = serializer.Serialize(newcontrolsupdate);
    NetworkStream stream = sender.GetStream(); // creates a stream
    byte[] buffer = Encoding.ASCII.GetBytes(jsonmessage); // encode
    stream.Write(buffer, 0, buffer.Length); // sends/writes data to
}
```

AutoPilot – Enhancement

There is also AutoPilot functionality included that is activated/deactivated when the checkbox is unchecked/checked. The AutoPilot will save the current speed of the Flight Simulator and attempt to maintain that speed by changing the throttle depending on the situation. It also tries to maintain the altitude by changing pitch controls. It works by utilising the same 'onControlsUpdate' event (updating 'showControl' listbox and 'sendControlData') as the manual sending control data button which cuts down on code whilst also maintaining reusability.



Control Data	
Elevator Pitch:	<input type="text"/>
Throttle:	<input type="text"/>
<input checked="" type="checkbox"/> AutoPilot	<input type="button" value="Send"/>

When the AutoPilot checkbox is checked, it creates a new thread calling the 'activateAutoPilot' method which runs concurrently in the background as a process or when the AutoPilot checkbox is unchecked, it aborts the thread essentially terminating it.

```

private void chkAutoPilot_CheckedChanged(object sender, EventArgs e)
{
    if (chkAutoPilot.Checked)
    {
        // creates a new thread with the method 'activateAutoPilot' called 'PilotThread'
        // which runs concurrently in the background as a process
        PilotThread = new Thread(new ThreadStart(MyRemoteFlightController.activateAutoPilot));
        PilotThread.Start();    // starts the 'PilotThread' thread
    }
    else if (!chkAutoPilot.Checked)
    {
        PilotThread.Abort();    // terminate the 'PilotThread' thread
    }
}

```

The AutoPilot code works by checking the speed saved when the AutoPilot was activated against the current speed of the Flight Simulator plane. If the plane is going faster than intended, it decreases the throttle to 25, otherwise, if the plane is going slower than intended, it increases the throttle to 75. The AutoPilot also tries to maintain altitude by constantly changing the pitch until it eventually balances out by lowering the elevator pitch when it's too high and vice-versa.

The 'onControlsUpdate' event is then called passing the control data as an argument which both updates the messages sent on the listbox whilst also sending control data to the Flight Simulator via the two subscribed methods.

To stop the thread from sending controls too quickly, I used a sleep function which suspends the thread for 0.2 seconds.

Below shows the AutoPilot functioning with the Flight Simulator by sending controls whilst updating the listbox.

3. [Download the software](#)

File Edit View Options Window Help

אני מודה לך על כל המאמצים והעזרה.