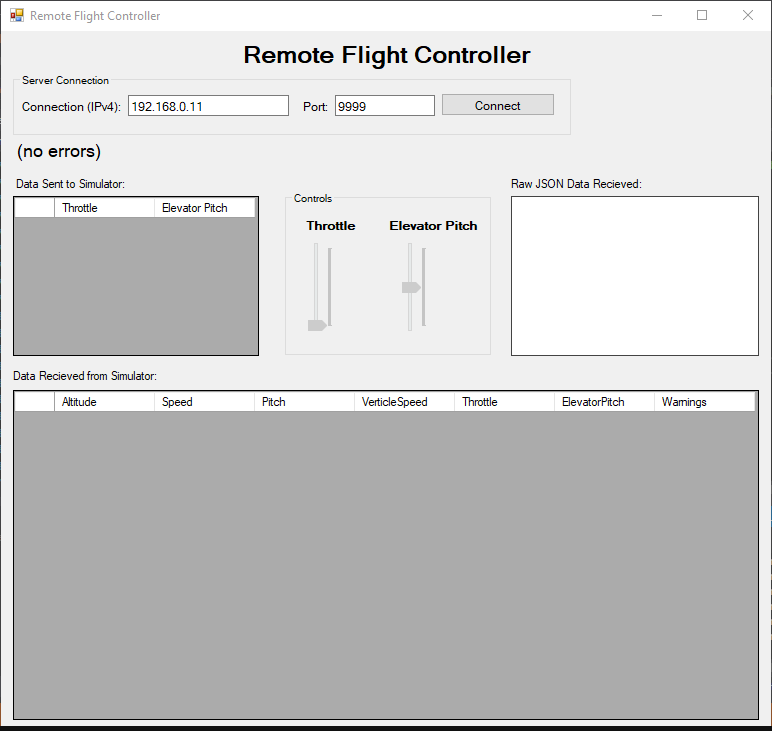
Flight Simulator Controller Report

# Introduction to the Task

I have been tasked with creating a class capable of remotely controlling an aeroplane and receive telemetry data from it. We were provided with a flight simulator program which expected a remote TCP connection to a controller program.

# The Flight Controller

The controller does not directly simulate anything, but remotely connects to a simulator program and sends JSON data to manipulate the controls. The server address can be specified in the text fields at the top of the form (the current machine’s address is automatically entered). Once connected, the user can use the scroll bars in the centre of the form to send control data to a flight simulator (the exact values sent are visible in the “Data Sent to Simulator” table). Each time the value of either of the scroll bars change, the program sends out a JSON packet to the simulator containing the new values.



All incoming data from the simulator is visible here, with the latest data appearing at the top of the table.

When a connection is established, the values of these sliders are sent to the simulator (the exact data sent is visible to the left in the table)

Specify a network connection – when connect is pressed, the program will establish the connection and begin listening for JSON

# Connecting a Simulator

The flight controller requires a TCP connection to a flight simulator program (acting as a server) – the address of the connection can be specified in the text fields at the head of the form. Once a successful connection is established, the text fields will become disabled and the program will listen for incoming plane telemetry.

Data sent to the controller must be in the form of a JSON string. This string represents a data structure with the following fields: Altitude, Speed, Pitch, VerticleSpeed, Throttle, ElevatorPitch, Warnings. While it is not possible to directly send strings through a network stream, it is fine to convert the string to a byte stream using the ASCII module in C#.

The controller uses an infinite loop containing a stream.Read() operation, which blocks the thread it is running on until it receives data. It is running on a dedicated worker thread however, so as long as the incoming data is not being sent faster than the de-serialiser can process the string there is no requirement for how fast the data should be sent.

The controller also sends control information to the simulator. This is not sent consistently, but instead sends an update as soon as the user changes the value of the scroll bar. This is sent in the form of a byte stream, which contains a serialised JSON string containing the following attributes: Throttle, ElevatorPitch. The simulator will need to de-serialise this in order to use the data.

# Explanation of Delegates, Events & Threading

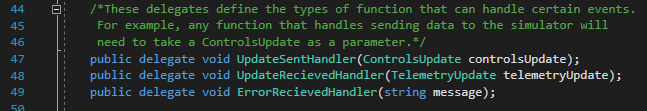
## Delegates

Delegates act like pointers to a function – they hold references to functions / methods that can be changed at runtime. In order to run the functions they reference, they must be “invoked”.



When declaring a delegate, you can imagine it as a template for a function – all functions or methods that this delegate references must have the same return type and parameters. In this case, all functions assigned to the “ErrorRecievedHandler” must return void and take a string value as a parameter. Delegates are declared as public outside of any class, so that all classes can use them to declare events if needed.

My controller program contains three delegates (one for each event). If you wish to add your own methods for handling the events in this class, they must conform to the templates specified here.



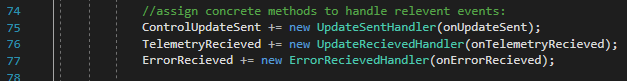
## Events

Events are using to communicate between threads. A method that is running on a certain thread cannot directly call a method on an object running on a different thread. Instead, methods and functions can be assigned to events, which can be invoked on a different thread.

Events are declared within a class and require a delegate as a type specifier. Once declared, methods that conform to the delegate that was given can be added into the event:



The name of the method that is called when the event is invoked



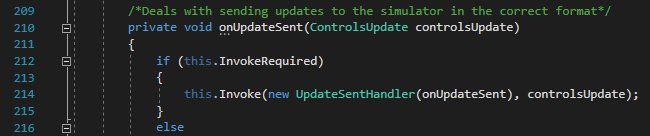
You can add multiple events into the same event, and each will be called when the event is invoked. In order to invoke the event simply call the EventName.Invoke(params) method, as demonstrated below with the TelemetryRecieved event.



## Threading

A normal program executes each instruction in sequence, meaning that the whole program can be blocked by things like infinite loops. Threaded programs can run multiple sections of code seemingly at the same time. If one thread encounters a block, the other threads are not blocked and continue executing.

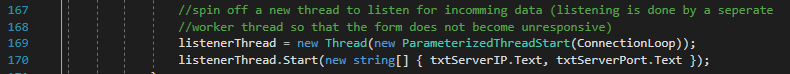
One of the drawbacks to this is that threads cannot easily communicate with each other (as said in the events section). If the call to run a method comes from a thread other than the one that the object is executing on, the method needs to invoke itself before it can access that rest of the object. The InvokeRequired attribute checks whether this step is needed; if it is, we invoke the method directly using delegates. We can invoke an instance of a delegate, which requires two parameters: the method to assign to the delegate (that must conform to the delegate’s template) and a parameter to pass into the method.



My program contains two threads: the main thread and a worker thread that only deals with listening for incoming JSON data from the network stream. The worker thread is defined as an attribute of the controller class, and spun off once a connection has been established.

When a thread is instantiated, it can either take a ThreadStart or a ParameterizedThreadStart object as a parameter. A thread start object itself takes a method as an entry point that is treated like the main function for the thread. The parameterized thread start means that when the start method is called, a single object can be passed to the method. In this case, I pass in an array of string values for the server address.





Inside this thread is an infinite loop that listens for incoming JSON data that would cause the form to become unresponsive if not using threading.

# Reflection and Potential Improvements

Although the program functions as intended, there are a few things I would change / do differently if I was required to continue the project. First, the GUI could have a few extra widgets to display data more explicitly to the user. For example, the scroll bars could have labels next to them indicating the its exact value, and there could be a label to explicitly display whether the program is connected to a simulator.

In addition, the send data event could have been triggered within a second worker thread. This would allow more control over the speed at which data is sent and would not require the use of a Windows Form created event.

I feel that I understand events and delegates a lot more now than I did at the start of the project. I believe that if I was to do this project again now, the solution would have a better user interface and would make use of threading more to control outgoing data.