iRacing SDK Wrapper tutorial

Version: 1.5

Introduction

In this tutorial I will attempt to explain the structure of the iRacing SDK / API and how you can access the various bits of information. To greatly simplify the SDK usage, I wrote a 'wrapper' project that allows you to use the SDK in an event-driven way, making it both easier and quicker to write.

You can write your application in C# or VB.NET, or possibly any other language that allows you to reference .NET libraries (DLL files).

This tutorial (and the wrapper project) is aimed at beginners and will not go into detail on the stuff going on behind the scenes. I will explain how to use it, not how it works in detail. However, even "pro's" will benefit from using the wrapper in many cases since it just makes everything a lot easier.

All code examples in this document are written in C#, but any decent online converter tool should easily convert them to VB.NET. The included example projects are available in both languages.

http://www.developerfusion.com/tools/convert/csharp-to-vb/

http://converter.telerik.com/

Chapter 1: SDK structure

Before we dive into the SDK usage, let me first explain the structure of the iRacing SDK, which will hopefully make everything a lot more clear.

The iRacing SDK basically consists of two parts: the **live telemetry** and the **session info**. In short, the live telemetry gives you live access to properties that change often, such as the positions of cars on track or the angle of your steering wheel. The SDK outputs this information 60 times per second. The session info on the other hand contains information that does not update as often, such as the drivers on track, the weather, and lap times. The session info is only updated several seconds apart at irregular intervals, mostly when drivers join the server or cross the finish line.

It is important to make the distinction between these two parts of the SDK, since the wrapper makes the same distinction. Before you start writing your application, investigate which information must come from which part, as this will likely influence the structure of your application.

1.1. Live telemetry

The live telemetry is read from something called a 'memory mapped file'. Simply put, a memory mapped file is a piece of memory on your computer that iRacing can share with your application. Your application can read from the memory that iRacing is writing to. This allows very fast communication with the sim.

Details on how values are read from the memory mapped file are not very important at this point, as the wrapper handles everything for you. You can simply ask the wrapper "give me the speed of my car" and it will do that for you. See chapter 3 for more info and an example.

1.2. Session info

The session info does not need to be updated and read extremely fast because it contains information about things that do not change very often. The session info is contained in one big string: the YAML string. YAML is a data format (similar to XML or JSON), if you don't know what YAML is, see a brief explanation in Appendix A.

To extract a value from this large YAML string, you need to form a *query* that points to the data you want. The query simply lists the 'path' you need to walk through the data until you find what you are looking for. You need to understand how to form these queries, as you need them in the wrapper to extract information from the YAML string. Forming YAML queries and reading data using my SDK Wrapper is covered in Chapter 4.

Chapter 2: iRacing SDK Wrapper

The iRacing SDK Wrapper is a class library project that helps you develop your application. It takes most of the boilerplate code out of your hands and allows you to work with the SDK in an event-driven way. This basically means that the SDK will 'notify' you when its information has updated, and you can react accordingly.

To use the Wrapper, the first thing you need to do is add a **Reference** to the SdkWrapper class library in your project. This means your project can reference the objects in the class library as if it was part of your own project. To add a reference from Visual Studio, go to the Solution Explorer, right-click your own project, select Add → Reference, select the 'Solution' tab on the left of the window that opens up and finally check the 'SdkWrapper' project. You may also need to reference the included 'iRSDKSharp' project, which is the actual SDK implementation that my wrapper builds upon.

Next, we must first create an instance of the main class: **SdkWrapper**. This means we are creating a new object of type SdkWrapper and we can then use its methods and events. The SdkWrapper object will raise a number of events, the most important of which are the **TelemetryUpdated** event and the **SessionInfoUpdated** event. We must 'listen' to these events, which basically means that we tell the object which methods it must call when the event is 'raised'. Once the object wants to notify us of the occurrence of an event, it will call the method we told it to and pass along some useful information. In short therefore, when the live telemetry updates, the TelemetryUpdated event will be raised, and when the session info updates, the SessionInfoUpdated event will be raised. Simple huh!

A brief code sample may look like this. In this case, I am creating the SdkWrapper in the constructor of Form1, which is my main Form in a Windows Forms application.

```
    public partial class Form1 : Form
    {
        // Globally declared SdkWrapper object
        private readonly SdkWrapper wrapper;

        public Form1()
        {
            InitializeComponent();

            // Create instance
            wrapper = new SdkWrapper();

            // Listen to events
            wrapper.TelemetryUpdated += OnTelemetryUpdated;
            wrapper.SessionInfoUpdated += OnSessionInfoUpdated;

            // Start it!
            wrapper.Start();
        }

        private void OnSessionInfoUpdated(object sender,
SdkWrapper.SessionInfoUpdatedEventArgs e)
        {
            // Use session info...
        }

        private void OnTelemetryUpdated(object sender,
SdkWrapper.TelemetryUpdatedEventArgs e)
        {
            // Use live telemetry...
        }
    }
```

Let’s now go into some more detail on these two events and what we can do with them.

Chapter 3: TelemetryUpdated event

Once the iRacing live telemetry is updated (60 times per second), the Wrapper will raise the TelemetryUpdated event. If we are ‘listening’ to this event (by telling the Wrapper object which method to call), then our OnTelemetryUpdated method will be called at that point. Now that the telemetry has updated, we need to get to the actual telemetry values. In the SDK Wrapper, this is extremely easy. As you may have seen, the OnTelemetryUpdated method contains a parameter ‘e’ of type **TelemetryUpdatedEventArgs**. This object contains all the information you need in the form of two simple properties: **UpdateTime** and **TelemetryInfo**.

UpdateTime is the session time (in seconds) when this piece of the live telemetry was sent out. You can use this to match any telemetry data with the correct time of occurrence (and you should not rely on the system time or any other timing), in case you are doing something that relies on timing.

The **TelemetryInfo** object finally contains all the telemetry data, in the form of properties. There is a property for every type of data, such as Speed, SteeringWheelAngle and Lap. Each of these properties returns another object (of type TelemetryValue), which finally contains the relevant properties. Let’s list these properties with the ‘Speed’ value as an example:

- **Name** returns the name of the property (example: ‘Speed’).
- **Description** returns a short description of the property (example: ‘GPS vehicle speed.’).
- **Unit** returns the unit of measurement of the property (example: ‘m/s’).
- **Value** finally returns the live value of the property as the correct type (floating point number, integer, Boolean, or even arrays of integers, depending on the property of interest).

Sounds complicated? Let’s look at an example. Suppose you want to store the speed and steering wheel angle in some variables (to be used later). You do that by simply accessing the required properties:

```
        private void OnTelemetryUpdated(object sender,
SdkWrapper.TelemetryUpdatedEventArgs e)
        {
            float speed = e.TelemetryInfo.Speed.Value;
            float angle = e.TelemetryInfo.SteeringWheelAngle.Value;
        }
```

Breaking this down is easy:

- **e.TelemetryInfo** returns the TelemetryInfo object that contains the data.
- **eTelemetryInfo.Speed** returns the TelemetryValue object for the Speed data.
- **e.TelemetryInfo.Speed.Value** finally returns the value of this data (the speed).

That’s it! You can now access any live telemetry data with this approach. As you can see, there’s really nothing to it, you just need to access the relevant properties. Now you can retrieve the live telemetry up to 60 times per second!

The update frequency

However, there is a tiny catch here: usually you don’t need to access the telemetry that often. Unless you are doing some seriously complicated physics calculations or something, 60 updates per second is way too much and will bog down your application (unless you start using multi-threading, let's not get into that). For a typical application that just displays the values, updating maybe 4 times per second is plenty. Luckily, you can change the update frequency easily by simply setting the TelemetryUpdateFrequency property of the SdkWrapper object. You can do this as soon as you create the object, so for example:

```
            // Create instance
            wrapper = new SdkWrapper();

            // Only update 4 times per second
            wrapper.TelemetryUpdateFrequency = 4;
```

Reading live telemetry data not covered in the TelemetryInfo object

The variables and data available in the live telemetry change pretty much every season when new features are added (and sometimes existing data is removed). The SDK Wrapper will most likely not be up to date 100% all the time. If there are new variables available in the live telemetry that are not yet available as properties in the TelemetryInfo object, you can still get to them manually by calling the **GetTelemetryValue** method on the SdkWrapper object. You need to specify the type of the value that you expect as well as a type parameter. Example:

```
      var fictionalObject = wrapper.GetTelemetryValue<int>("VariableName");
      var fictionalValue = fictionalObject.Value;
```

Obviously, this particular example will throw an exception because "VariableName" is not a valid telemetry variable.

Chapter 4: SessionInfoUpdated event

When the session info updates, the SessionInfoUpdated event is raised, and (in our example), the OnSessionInfoUpdated method will be called. The event handler method contains a parameter of type **SessionInfoUpdatedEventArgs**. Like the telemetry info, this object contains both the session info (in the form of a YAML string), and the update time.

You can obtain values from the session info YAML in various ways. The easiest way is to form what I like to call a 'YAML query', which is basically a query that points to a certain value in the YAML string. This query can be used to extract that value. As you can see in Appendix A, the YAML string contains a hierarchy of key/value pairs, and your query must follow the path that you need to walk from the root object down to the key/value pair that you need.

The query is made on top of the SessionInfo object using an 'indexer' syntax, kind of like a multi-dimensional array except the indexes you use are the section names instead of numeric indexes:

```
YamlQuery query = e.SessionInfo["WeekendInfo"]["TrackLength"];
```

Example: suppose we have the following YAML string and you want to extract the NumStarters value ("16").

```
WeekendInfo:
   TrackDisplayName: Mid Ohio Sports Car Course
   Category: Road
   WeekendOptions:
      NumStarters: 16
      StartingGrid: 2x2 inline pole on left
```

The path to this value is easy: you start at the root **WeekendInfo**, then down to the subsection **WeekendOptions**, and finally to the key/value pair **NumStarters**. Your query is formed in this order, so your query object becomes:

```
YamlQuery query =
   e.SessionInfo["WeekendInfo"]["WeekendOptions"]["NumStarters"];
```

YAML queries for items in a list

If you want to get values from items in a list, your YAML query must contain a 'search'. You need to specify *which* item in the list you want by specifying the value of the first (unique) identifier of the list item. Specifying which item you want is very similar to the regular way of forming a query, you just specify both the key name (as usual) *and* the value that your desired item has.

Suppose you have the following YAML with a list of drivers:

```
DriverInfo:
   DriverCarIdx: 10
   DriverCarRedLine: 6400.000
   Drivers:
      - CarIdx: 0
        UserName: Driver A
        CarNumber: 1
      - CarIdx: 1
        UserName: Driver B
        CarNumber: 2
      - CarIdx: 2
        UserName: Driver C
        CarNumber: 3
```

If you want to get the **UserName** of the driver with a **CarIdx** of 1, you would form your query as such:

```
YamlQuery query =
   e.SessionInfo["DriverInfo"]["Drivers"]["CarIdx", 1]["UserName"];
```

The relevant part is in bold text: **`["CarIdx", 1]`** specifies that you are looking for the item in the list where CarIdx is 1.

Obtaining a value from your query

Once you have your query object, you can get the value by simply calling the Value property. Of course you can also do that inline directly:

```
string value = e.SessionInfo["WeekendInfo"]["TrackLength"].Value;
```

Note that the **Value** property will throw an exception if the resulting value is null (most likely your query is wrong or does not point to an existing key/value pair). If you want to prevent exceptions, you have two more methods available, called **GetValue** and **TryGetValue**. If you do not pass any parameters to **GetValue**, it will return the value if successful, or **null** if an exception occurred (but it will not throw the exception). You can also pass a default value to the function in which case it will return that default value in case of an error.

The **TryGetValue** method allows you to check if the query was successful or not. You can pass an uninitialized string as an 'out' parameter (an 'out' parameter indicates that the parameter will be modified by the method when it has returned). In that case, the **TryGetValue** method will return true or false depending on the validity of the query. Here is an example of that last method:

```
string length;
YamlQuery query = e.SessionInfo["WeekendInfo"]["TrackLength"];

if (query.TryGetValue(out length))
{
    // Success! You can use the 'length' variable now
}
else
{
    // Failed...
}
```

Invalid queries

If your YAML query is invalid (meaning: the path to your data does not exist or you use a key that does not exist), the resulting query will be an 'error query'. You can still attempt to retrieve a value from your query but it will fail, or return null, depending on which value retrieval method you use (see next section). If your query path is invalid, you can inspect the resulting exception for the offending section of the path.

Here is an example of an invalid query (the key "TypoHere" does not exist).

```
var invalidQuery = sessionInfo["DriverInfo"]["TypoHere"]
["CarIdx", 1]["UserName"];
```

When we attempt to get the Value property of this query, an exception will be thrown:

```
var value = invalidQuery.Value;
Exception: The YAML query path is incorrect: DriverInfo:TypoHere:
```

To prevent the exception, you can use **GetValue** instead. It will return null because the query is invalid, but it will not throw an exception.

```
var value = invalidQuery.GetValue();
No exception, but 'value' is null.
```

Appendix A: The YAML format

YAML is a data serialization format, like XML or JSON, which is easy to read by humans as well as computers. An example of the session info YAML is included in the wrapper (see ‘sessioninfo_example.txt’). The YAML contains several sections of data (WeekendInfo, CameraInfo, DriverInfo, etc), each of which contain properties and/or sub-sections of more data. There are also lists of data available, such as the list of drivers.

Subsections

A section in the YAML string can contain subsections (which in turn can contain their own subsections, etc). A subsection is identified by some extra indentation on the left of the text. When a block of text jumps to the right, that means this text is part of a new subsection (and the line preceding it is the ‘header’ or name of this subsection).

Example:

```
WeekendInfo:
   TrackDisplayName: Mid Ohio Sports Car Course
   Category: Road
   WeekendOptions:
      NumStarters: 16
      StartingGrid: 2x2 inline pole on left
```

In the above example, we first find the WeekendInfo section. The following lines of data are indented more, which indicates that this data is a subset of the WeekendInfo section. The TrackDisplayName and Category lines are two properties of the WeekendInfo section; both contain values.

The WeekendOptions line next does not contain a value, instead it is followed by a new block of data which is indented one extra time. The WeekendOptions section is thus a subsection of WeekendInfo and NumStarters and StartingGrid are properties of WeekendInfo.

Lists

A YAML string can contain lists (or arrays) of data. Each item in the list is identified by a dash in front of the first property. Example:

```
DriverInfo:
   DriverCarIdx: 10
   DriverCarRedLine: 6400.000
   Drivers:
      - CarIdx: 0
        UserName: Driver A
        CarNumber: 1
      - CarIdx: 1
        UserName: Driver B
        CarNumber: 2
      - CarIdx: 2
        UserName: Driver C
        CarNumber: 3
```

In the above example, the Drivers line is a list (not a subsection) because the next line starts with a dash, indicating an item in the list. In this case, the Drivers list contains three items (drivers).