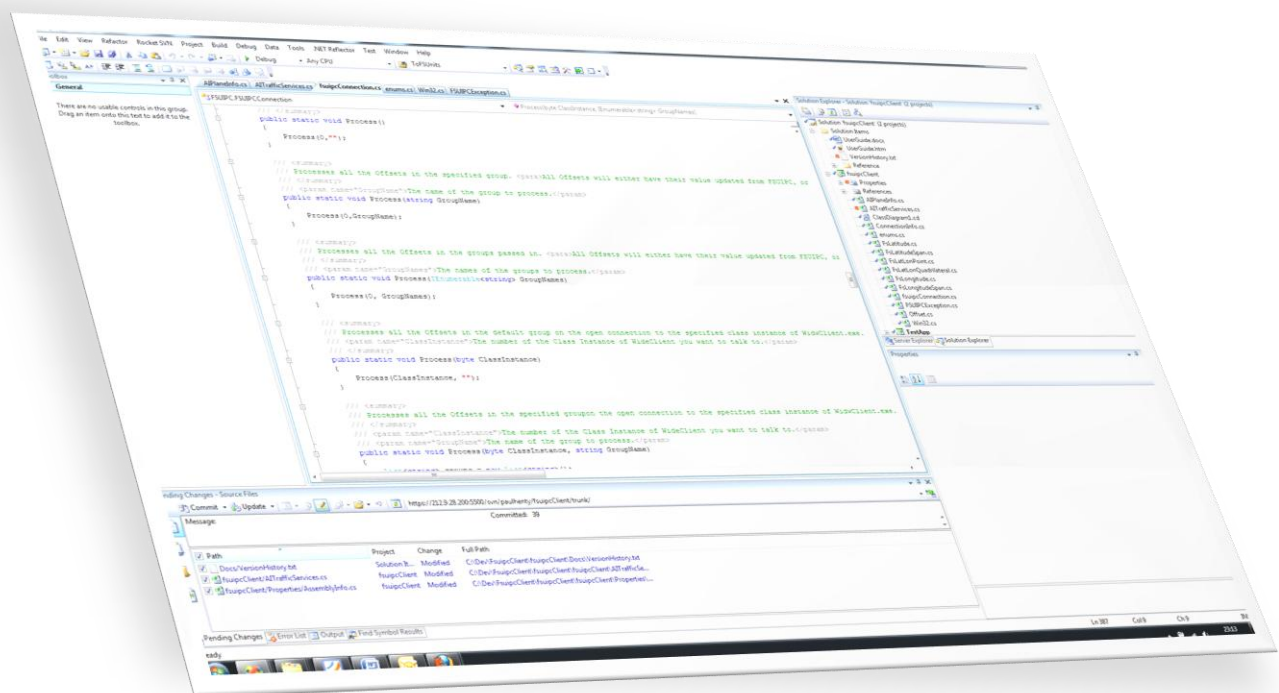


FSUIPC Client DLL for .NET

Version 2.0

by Paul Henty

User Guide



Contents

Introduction	3
The DLL	3
Licence	3
This User Guide.....	3
The Reference Manual.....	4
The Sample Application	4
Distribution	4
.NET Framework Requirements and 64-Bit issues.....	4
Warranty.....	4
Getting Started	5
Installation (Quick).....	5
Installation (Step-by-Step for Visual Studio).....	5
Opening the Connection to FSUIPC	5
Closing the FSUIPC Connection.....	6
Registering your interest in an Offset.....	6
Reading data from an Offset	8
Writing to an FSUIPC Offset.....	8
Handling FSUIPC Errors.....	9
Advanced Features	10
Write-Only Offsets.....	10
Managing Offsets using the Grouping Facilities	10
Managing Offsets Individually	12
Using the BitArray Type for Bit Field Offsets	12
Reading Raw Blocks of Data from FSUIPC	13
Connecting to multiple instances of WideClient	13
References and Garbage Collection Issues.....	14
Longitude/Latitude Helper Classes	15
Reading and Displaying Longitudes and Latitudes	15
Writing Longitudes and Latitudes and Offsetting (Translating) Points	16
Distances and Bearings between Two Points	17
Finding out if a Point is inside an Given Area	18
Finding out if the Player is on a Runway.....	20
AI Traffic.....	22
Getting the AI Traffic Information	22
Filtering AI Traffic.....	23
Getting Extended Identifying Information	23
Finding Active Runways.....	24
Overriding the AI Traffic settings in FSUIPC.INI	26
Writing AI Traffic (TCAS) Information to the FSUIPC Tables.....	26

Introduction

The DLL

This DLL will allow .NET applications to interface with Peter Dowson's FSUIPC or WideFS using an object-oriented .NET class library. It also includes some helper classes to make working with certain FSUIPC data easier.

Licence

The DLL, sample application and documentation is owned by Paul Henty.

You are granted a non-exclusive right to distribute this DLL at no cost and royalty-free as part of a freeware or commercial application.

A credit acknowledging the author of this DLL and its use in your application must appear in the documentation and, if the application has a user-interface, at least one place in the application e.g. on the 'about' screen, version information screen, splash screen etc. The suggested wording is "Uses the FSUIPC Client DLL for .NET by Paul Henty". This applies to both Freeware and Commercial Applications.

You may not sell this DLL, the sample applications, or the documentation, on their own, together, or as part of a toolkit for developing applications.

If you use this DLL in a commercial application and you wish to send a donation to the author, you can do so via paypal to: paul.henty@unitysoftware.net. A license key for your application will also be appreciated if you feel so inclined.

This User Guide

All users should read the "Getting Started" section as it contains the minimum information required to write an application using the DLL.

Users writing more advanced programs or those wanting to know about specific functions of the DLL can then read other relevant sections.

Code examples are given in C# and Visual Basic.NET. These are mainly taken from the example application included in this package. Where the syntax is the same for both languages (except for the terminating semi-colon in C#) only the C# example is given.

When writing applications using FSUIPC you must refer to the documentation contained in the FSUIPC SDK to gain a full understanding of the various offsets and other facilities you want to use. Please do not rely solely on the documentation for this DLL.

The Reference Manual

The HTML reference manual provides the technical documentation for the DLL. All classes in the DLL are explained along with every property, field and method.

The Sample Application

A sample application is provided with full source code in both C# and VB.NET which demonstrates the use of all but the most esoteric features. The solutions were made in Visual Studio 2008 but can be opened in Visual Studio 2010 which will convert them.

The application is a simple windows form with three tabs. The first displays data read from FSUIPC and also demonstrates writing to FSUIPC. The second deals with the new Longitude and Latitude helper classes. The third demonstrates the new AI Traffic features by displaying a simple ATC radar image of the active AI planes.

Distribution

When you build your project, Visual Studio will place a copy of the DLL in your build folder along with your .exe. You need to distribute the DLL with your program for it to work. Some versions of Visual Studio come with tools for building a deployment package or installation program. The requirement for the DLL will be detected and the DLL will be included in the installation package.

.NET Framework Requirements and 64-Bit issues

The DLL targets the .NET 2.0 framework. This is required to be installed for this DLL to work. Your own application can target any version of the .NET framework you require.

Note that the DLL is compiled to target x86 systems only. This means it will run on the WOW64 compatibility layer on x64 systems. It has to be that way otherwise it won't be able to talk to FSUIPC or WideFS as these run as x86 processes.

If you are developing on 64bit operating systems you will have to set your compilation options to also target x86 only. If you do not, your application will not be able to link to the DLL on 64bit systems.

Warranty

The software and documentation are provided 'AS IS' without warranty of any kind.

Getting Started

Installation (Quick)

Just add a reference to this DLL in your .NET application project. The compiler will copy it into the binary build folder when the project is built.

The namespace containing all the classes in the DLL is called `FSUIPC`. The `BitArray` class lives in the `System.Collections` namespace.

Installation (Step-by-Step for Visual Studio)

1. Open your application project, or create a new one.
2. Add a new reference.
 - a) Go to or Open the Solution Explorer window.
 - b) Find the node for your main application project and select it.
 - c) Go to the Project menu and select Add Reference...
 - d) Select the Browse tab.
 - e) Navigate to the `fsuipcClient.dll` and press [OK].
3. In any file you need to access this library from, write a statement at the top of the file to import the `FSUIPC` namespace.

C#

```
using FSUIPC;
```

VB.NET

```
Imports FSUIPC
```

4. If you want to use the `BitArray` class as a data type for the `FSUIPC Client` you also need to import the `System.Collections` namespace if your language doesn't do this by default.

Opening the Connection to FSUIPC

Use the `Open()` method on the static `FSUIPCConnection` class. If anything goes wrong during the connection process this method will throw an `FSUIPCException` with the appropriate error code.

C#

```
try
{
    // Attempt to open a connection to FSUIPC (running on any version of Flight Sim)
    FSUIPCConnection.Open();
    // Opened OK
}
catch (Exception ex)
{
    // Badness occurred - show the error message
    MessageBox.Show(ex.Message, AppTitle, MessageBoxButtons.OK, MessageBoxIcon.Error);
}
```

Visual Basic.NET

```
Try
    ' Attempt to open a connection to FSUIPC (running on any version of Flight Sim)
    FSUIPCConnection.Open()
    ' Opened OK
Catch ex As Exception
    ' Badness occurred - show the error message
    MessageBox.Show(ex.Message, AppTitle, MessageBoxButtons.OK, MessageBoxIcon.Error)
End Try
```

If you need to limit your application to run on a specific version of Flight Sim you can use the overload that takes a `FlightSim` enum.

If FSUIPC is not running inside the specified version, the `FSUIPCConnection` class will throw an `FSUIPCException` with its `FSUIPCErrorCode` property set to `FSUIPC_ERR_VERSION`.

e.g. This tries to open a connection to Flight Simulator 2004.

```
FSUIPCConnection.Open(FlightSim.FS2K4);
```

Closing the FSUIPC Connection

Use the `Close()` method of the static `FSUIPCConnection` class.

```
FSUIPCConnection.Close();
```

This method cleans up the unmanaged memory that is used to communicate with FSUIPC. If you should make sure this method is called before your application quits.

You will also need to call this if your application loses its connection to FSUIPC for some reason. Before you can open a new connection you must close the broken one.

Registering your interest in an Offset

To tell the DLL that you want to either read from or write to an FSUIPC offset you must create an instance of the `Offset` class:

C#

```
Offset<DataType>
```

Visual Basic.NET

`Offset(Of DataType)`

You must specify a type (`DataType`) when creating the instance. This type is the .NET type that the value of the `Offset` will be stored in.

For example, if the FSUIPC offset is an integer and is 4 Bytes long, you would specify a type of `Int32` or a native type specific to your language that is the same as `Int32` (`C# = int`, `VB.NET = Integer`).

In this example we create a new instance of the `Offset` class called 'airspeed'. This is set to read from offset `0x02BC` (Indicated Airspeed) that stores a 4-byte integer. We therefore type this `Offset` instance accordingly:

C#

```
Offset<int> airspeed = new Offset<int>(0x02BC);
```

Visual Basic.NET

```
Dim airSpeed As Offset(Of Integer) = New FSUIPC.Offset(Of Integer) (&H2BC)
```

The following .NET Types can be used as offset types. The table also shows aliases for C# and VB.NET and what kinds of FSUIPC offsets you should use them with.

FSUIPC Offset Size	.NET Framework type	C#	VB.NET
1	<code>System.Byte</code>	<code>Byte</code>	<code>Byte</code>
2	<code>System.Int16</code> <code>System.UInt16</code>	<code>Short</code> <code>Ushort</code>	<code>Short</code> <code>UShort</code>
4	<code>System.Int32</code> <code>System.UInt32</code>	<code>Int</code> <code>UInt</code>	<code>Integer</code> <code>UInteger</code>
4 (Specified as 32-Bit Float)	<code>System.Single</code>	<code>Float</code>	<code>Single</code>
8	<code>System.Int64</code> <code>System.UInt64</code>	<code>Long</code> <code>Ulong</code>	<code>Long</code> <code>ULong</code>
8 (Specified as FLOAT64)	<code>System.Double</code>	<code>Double</code>	<code>Double</code>
Any specified as string	<code>System.String</code>	<code>String</code>	<code>String</code>
Any but only useful for offsets storing bit fields.	<code>System.Collections.BitArray</code>		
Any	Array of <code>System.Byte</code>	<code>byte[]</code>	<code>Byte()</code>

When you create an `Offset` with a type of `String`, `BitArray` or an array of bytes you must also specify the length of the data you want to read from FSUIPC. This is always in Bytes. See the example application for examples of reading different types of data.

Reading data from an Offset

First, create an instance of the `Offset` class. For a detailed explanation of this see [“Registering your interest in an Offset”](#) above.

The next step is to call the `Process()` method on `FSUIPCConnection`.

This will read the value from FSUIPC and store it in the `Value` property of the offset instance. If anything goes wrong during this process, an `FSUIPCException` will be thrown with the appropriate error code. For more details see the section titled [“Handling FSUIPC Exceptions”](#).

In this example, the airspeed Offset is processed and its value converted to Knots and displayed in a textbox.

C#

```
FSUIPCConnection.Process();  
double airpeedKnots = (double)airSpeed.Value / 128d;  
this.txtIAS.Text = airpeedKnots.ToString("f1");
```

Visual Basic.NET

```
FSUIPCConnection.Process()  
Dim airpeedKnots As Double = airSpeed.Value / 128D  
Me.txtIAS.Text = airpeedKnots.ToString("f1")
```

The next time you want to update the value of the airspeed offset you can just call `Process()`. You don't need to create an `Offset` instance again.

Writing to an FSUIPC Offset

Any Offsets you create can be written to.

Most of the time they will be read when you call `Process()`. If you want to write a new value to FSUIPC all you need to do is change the `Value` property of the `Offset` instance.

The next time you call `Process()` the offset's value will be written instead of read. After the write, the offset will go back to reading data again.

When you create an `Offset<>` you can specify it to be a write-only offset. For more details see the section [“Write-Only Offsets”](#).

This example shows the steps involved for turning on the Master Avionics switch by writing 1 to offset 0x2E80. Note that each line of code comes from different parts of an application.

C#

```
Offset<int> avionics = new Offset<int>(0x2E80);  
...  
avionics.Value = 1;  
...  
FSUIPCConnection.Process();
```


Visual Basic.NET

```
Dim avionics As Offset(Of Integer) = New FSUIPC.Offset(Of Integer) (&H2E80)
...
avionics.Value = 1
...
FSUIPCConnection.Process()
```

Handling FSUIPC Errors

If any errors occur during FSUIPC operations an `FSUIPCException` will be thrown.

You may handle this error by catching a general Exception class, or by catching the specific `FSUIPCException` class.

In both cases the `Message` property of the exception will be a verbose explanation of the error that occurred, including the FSUIPC Error Code from the original C SDK as part of the message string.

If you choose to handle the specific `FSUIPCException` class there is an extra property called `FSUIPCErrorCode` that will return the original C SDK error code separately in the form of an `FSUIPCError` enum. You can use this to check for a specific error if you need to.

`FSUIPCException` is thrown by the `Open()` and `Process()` methods of `FSUIPCConnection`.

In this example we catch the `FSUIPCException` that may be thrown during the `Process()` method. We're looking for a specific error: `FSUIPC_ERR_SENDSMSG` that gets thrown if the connection to FSUIPC is lost. The most likely cause would be that Flight Sim has been closed.

Instead of letting the application crash with an unhandled exception, the example below catches that error and handles it in a more elegant way. It displays a message to the user and enables a 'connect' button so the user can attempt to reconnect. It also closes the current (and now bad) connection.

C#

```
try
{
    FSUIPCConnection.Process();
    // Process OK
}
catch (FSUIPCException ex)
{
    if (ex.FSUIPCErrorCode == FSUIPCError.FSUIPC_ERR_SENDSMSG)
    {
        // Send message error - connection to FSUIPC lost.
        // Show message, relight the
        // connection button:
        // Also Close the broken connection.
        this.btnStart.Enabled = true;
        FSUIPCConnection.Close();
        MessageBox.Show("The connection to Flight Sim has been lost.");
    }
    else
    {
        // not the disconnect error so some other baddness occurred.
        // just rethrow to halt the application
        throw ex;
    }
}
```

Visual Basic.NET

```
Try
    FSUIPCConnection.Process()
    ' Process OK
Catch exFSUIPC As FSUIPCException
    If exFSUIPC.FSUIPCErrorCode = FSUIPCError.FSUIPC_ERR_SENDMSG Then
        ' Send message error - connection to FSUIPC lost.
        ' Show message, relight the
        ' connection button:
        ' Also Close the broken connection.
        Me.btnStart.Enabled = True
        FSUIPCConnection.Close()
        MessageBox.Show("The connection to Flight Sim has been lost.")
    Else
        ' not the disconnect error so some other baddness occurred.
        ' just rethrow to halt the application
        Throw exFSUIPC
    End If
End Try
```

Advanced Features

Write-Only Offsets

When you create an Offset you can set it to be write-only by passing `true` as the `WriteOnly` parameter. This means its value will never be read from FSUIPC when `Process()` is called.

This example shows creating an offset as write-only. The offset is the 'pause' function at 0x0262.

C#

```
Offset<short> pause = new Offset<short>(0x0262, true);
```

Visual Basic.NET

```
Dim pause As Offset(Of Short) = New FSUIPC.Offset(Of Short) (&H262, True)
```

You can also set an Offset to be write-only by setting the `WriteOnly` property of the Offset at any time.

Note also that the values of write-only offsets are only written to FSUIPC when you call `Process()` and the `Value` property has changed since the last `Process()`.

Managing Offsets using the Grouping Facilities

You can use offset groups in your application to control when certain offsets are read from or written to FSUIPC.

Typically applications need to request different sets of data at different rates. For example, the location of the aircraft may be required to be retrieved a number of times per second, whereas weather data might only be needed to update every ten minutes.

By putting the offsets into different groups you can process them at different rates.

When you create an Offset you can optionally specify a 'Group' to put it in. If no Group is specified it gets put in the default group. The default group is used whenever you use an overload that doesn't specify a group.

In this example an Offset is created in a group called "AircraftInfo". The offset requests the aircraft type string from 0x3160.

C#

```
Offset<string> aircraftType = new Offset<string>("AircraftInfo", 0x3160, 24);
```

Visual Basic.NET

```
Dim aircraftType As Offset(Of String) = New FSUIPC.Offset(Of String) ("AircraftInfo", &H3160, 24)
```

To process Offsets in the group, use the overload to specify the group name:

```
FSUIPCConnection.Process("AircraftInfo");
```

You can process multiple groups in one Process() call by using the overload that takes an IEnumerable list of group names.

The following is a simplistic example of multiple groups being processed.

C#

```
private Offset<string> aircraftName = new Offset<string>("info", 0x3D00, 24);
private Offset<int> pitch = new Offset<int>("attitude", 0x0578);
private Offset<int> bank = new Offset<int>("attitude", 0x057c);

public void processGroups()
{
    List<string> groupsToProcess = new List<string>();
    if (needPlaneName)
    {
        groupsToProcess.Add("info");
    }
    if (needPlaneAttitude)
    {
        groupsToProcess.Add("attitude");
    }
    FSUIPCConnection.Process(groupsToProcess);
}
```

VisualBasic.NET

```
Private aircraftName As Offset(Of String) = New Offset(Of String) ("info", &H3D00, 24)
Private pitch As Offset(Of Integer) = New Offset(Of Integer) ("attitude", &H578)
Private bank As Offset(Of Integer) = New Offset(Of Integer) ("attitude", &H57C)

Public Sub processGroups()
    Dim groupsToProcess As List(Of String) = New List(Of String) ()
    If (needPlaneName) Then
        groupsToProcess.Add("info")
    End If
    If (needPlaneAttitude) Then
        groupsToProcess.Add("attitude")
    End If
    FSUIPCConnection.Process(groupsToProcess)
End Sub
```

Alternatively you can use a list of literal group names by creating a new string array:

C#

```
FSUIPCConnection.Process(new string[] { "info", "attitude" });
```

VisualBasic.NET

```
FSUIPCConnection.Process(New String() {"info", "attitude"})
```

Processing multiple groups in a single call to `Process()` is much better than making multiple `Process()` calls. The actual processing (data-exchange with FSUIPC) is pretty slow. The fewer process calls you make, the faster your application and Flight Simulator will run. The amount of data read or written has little impact on the speed of the data-exchange in comparison.

Managing Offsets Individually

You can control when Offsets are read and written on an individual basis.

To prevent the Offset from being read or written, use the `Disconnect()` method.

This unregisters the Offset from the `FSUIPCConnection` class and therefore prevents it from being read and written.

You can optionally make the disconnect happen after the next `Process()` by passing `true` as the `AfterNextProcess` parameter.

If you want to start using the Offset again, you can reconnect it using the `Reconnect()` method:

You can optionally choose to reconnect it for just one `Process()` by passing `true` as the `ForNextProcessOnly` parameter.

Using the BitArray Type for Bit Field Offsets

This DLL will allow you manage bit field type offsets as a `BitArray`, rather than using bit-wise operations and masks etc.

This example uses the lights offset at 0x0D0C. Each bit in the two-byte offset represents if a particular light is on or off.

We declare the Offset as follows (pass a length of 2 as this offset is 2 bytes long):

C#

```
Offset<BitArray> lights = new Offset<BitArray>(0x0D0C, 2);
```

Visual Basic.NET

```
Dim lights As Offset(Of BitArray) = New FSUIPC.Offset(Of BitArray) (&HD0C, 2)
```

The `Value` property of this `Offset` is a `BitArray` type. We can access each bit using the indexer. For example – bit 5 (zero-based) represents the Instrument lights. In the following example, the value of bit 5 is used to set the `Checked` property of a `Checkbox` that shows the user the status of the instrument lights:

C#

```
this.chkInstruments.Checked = lights.Value[5];
```

Visual Basic.NET

```
Me.chkInstruments.Checked = lights.Value(5)
```

To turn on the instrument lights we just set the appropriate bit (number 5):

C#

```
this.lights.Value[5] = true;
```

Visual Basic.NET

```
Me.lights.Value(5) = True
```

The lights will be switched on when you next call `Process()`.

Reading Raw Blocks of Data from FSUIPC

For maximum flexibility you can use an array of bytes to read or write any length of data to/from FSUIPC. You may then handle the data in any way you need to. An example would be using the weather facilities in FSUIPC.

Typically you will need to convert the bytes in the array into native data variables. The `BitConverter` class is useful for this. See your .NET help for how to use this class.

The following examples show creating an `Offset` to read 10 bytes from 0x0238. These offsets contain various information about the local date and time.

C#

```
Offset<byte[]> fsLocalDateTime = new Offset<byte[]>(0x0238, 10);
```

Visual Basic.NET

```
Dim fsLocalDateTime As Offset(Of Byte()) = New FSUIPC.Offset(Of Byte())(&H238, 10)
```

You will need to convert the raw byte data yourself. In this case, splitting out the various components of the date and time.

Connecting to multiple instances of WideClient

It is possible to run multiple copies of WideClient on the same machine, each one connecting to a different Flight Simulator (Wideserver) computer. Each copy of WideClient has its own .ini file with the settings pointing to the appropriate Flight Simulator computer.

The .ini file must also include the line

```
ClassInstance=x
```

Where x is a number between 0 and 255. It is this number that you will use when telling the DLL which instance of WideClient (and therefore which Flight Simulator PC) to talk to.

(More details of this feature and how to configure the WideClient ini files can be found in the WideFS documentation.)

In your software you need to use one of the overloaded `FSUIPCConnection.Open()` methods that takes in the `ClassInstance` parameter. You need to call `Open()` for every instance you want to talk to in your software.

From then on, every time you use the `FSUIPCConnection.Process()` method you also need one of the overloads that takes in the `ClassInstance` number.

After calling `Process()` with a class instance, all the offsets processed will now contain data from that specific instance of `WideClient`.

You can call `Close(ClassInstance)` to close a specific instance or just use `Close()` to close them all.

Here is some example code that opens two instances of `WideClient` (0 and 1) and gets the aircraft name from each one:

C#

```
Offset<string> aircraftName = new Offset<string>("aircraft name", 0x3D00, 24);

FSUIPCConnection.Open(0, FlightSim.Any);
FSUIPCConnection.Open(1, FlightSim.Any);

FSUIPCConnection.Process(0, "aircraft name");
string strAircraftName1 = aircraftName.Value;

FSUIPCConnection.Process(1, "aircraft name");
string strAircraftName2 = aircraftName.Value;

FSUIPCConnection.Close();
```

VisualBasic.NET

```
Dim aircraftName As Offset(Of String) = New Offset(Of String) ("aircraft name", &H3D00, 24)

FSUIPCConnection.Open(0, FlightSim.Any)
FSUIPCConnection.Open(1, FlightSim.Any)

FSUIPCConnection.Process(0, "aircraft name")
Dim strAircraftName1 As String = aircraftName.Value

FSUIPCConnection.Process(1, "aircraft name")
Dim strAircraftName2 As String = aircraftName.Value

FSUIPCConnection.Close()
```

References and Garbage Collection Issues

Make sure you keep a reference to the `Offset` instances you create. If they go out of scope you'll not be able to get a reference to them again.

They will still be registered with the `FSUIPCConnection` class however, so they will still be updated during `Process()` calls for the group they are in. Also the garbage collector will not dispose of them because of the reference being held by `FSUIPCConnection`.

If you use a sensible grouping scheme you shouldn't need to worry about `Offsets` going out of scope. If they do then, it's not real problem as you're not likely to ever process the group they were in again. They'll still be taking up a small amount of memory but they'll never get read.

If you really don't want an `Offset` anymore and you want it to be disposed, make sure you disconnect it first before you lose a reference to it.

If your application has a lot of 'one-off' reads you can place them in a group and delete the entire group when you're finished with them:

```
FSUIPCConnection.DeleteGroup(GroupName);
```

Longitude/Latitude Helper Classes

Reading and Displaying Longitudes and Latitudes

There are two helper classes provided by the DLL that make reading and writing Longitude and Latitudes to and from FSUIPC easier.

FSUIPC uses an 8-Byte integer format for longitudes and latitudes. These require conversion to decimal degree values or strings before they are useful to your application.

The `FsLongitude` and `FsLatitude` classes do this for you.

The following example shows reading the player's Latitude and Longitude and displaying them in two text boxes on a form. The Latitudes and Longitudes are created using the overload that takes the raw 8-Byte FS Units returned from FSUIPC. (Some Lat/Lon offsets in FSUIPC use a less accurate 4 Byte value in FS Units. There is also a constructor that understands this value).

These classes are then used to display the value in a text format using the `ToString()` overload. (Note that there is another overload of `ToString()` that lets you specify how the value is formatted, for example, if you want minutes or seconds and the number of decimal places. See the Intellisense or the Reference Manual for details.)

C#

```
private Offset<long> playerLatitude = new Offset<long>(0x0560);
private Offset<long> playerLongitude = new Offset<long>(0x0568);
private void displayCurrentPosition()
{
    // Create new instances of FsLongitude and FsLatitude using the
    // raw 8-Byte data from the FSUIPC Offsets
    FSUIPCConnection.Process()
    FsLongitude lon = new FsLongitude(playerLongitude.Value);
    FsLatitude lat = new FsLatitude(playerLatitude.Value);
    // Use the ToString() method to output in human readable form:
    this.txtLatitude.Text = lat.ToString();
    this.txtLongitude.Text = lon.ToString();
}
```

VisualBasic.NET

```
Dim playerLatitude As Offset(Of Long) = New Offset(Of Long) (&H560)
Dim playerLongitude As Offset(Of Long) = New Offset(Of Long) (&H568)

Private Sub DisplayCurrentPosition()
    ' Create new instances of FsLongitude and FsLatitude using the
    ' raw 8-Byte data from the FSUIPC Offsets
    FSUIPCConnection.Process()
    Dim lon As FsLongitude = New FsLongitude(playerLongitude.Value)
    Dim lat As FsLatitude = New FsLatitude(playerLatitude.Value)
    ' Use the ToString() method to output in human readable form:
    Me.txtLatitude.Text = lat.ToString()
    Me.txtLongitude.Text = lon.ToString()
End Sub
```

Writing Longitudes and Latitudes and Offsetting (Translating) Points

This example shows setting the current location to London Heathrow Airport Runway 27L. This mainly involves setting the current Latitude and Longitude offsets. These offsets accept the data in FS Units (an 8-Byte Integer). Instead of manually converting the Lat/Lon degrees to FS Units, you can use the `ToFSUnits8()` method on the `FsLongitude` and `FsLatitude` classes.

Some FSUIPC offsets for longitude and latitude only take a less-accurate 4-Byte FS Units value. To get this value use the `ToFSUnits4()` method.

The data we have for the runway only gives us the Lat/Lon of threshold. We want to position the plane a little way up the runway, ready for takeoff. We use the `OffsetByMetres()` to calculate this new point 150 metres from the threshold point in the direction of the runway. There are also methods for offsetting by Nautical Miles and Feet. These methods take a bearing (the direction to move from the original point) and a distance to move.

In this example we also set the simulator to slew mode, set the heading and altitude and refresh the scenery.

C#

```
private Offset<long> playerLatitude = new Offset<long>(0x0560);
private Offset<long> playerLongitude = new Offset<long>(0x0568);
private Offset<uint> playerHeadingTrue = new Offset<uint>(0x0580);
private Offset<long> playerAltitude = new Offset<long>(0x0570);
private Offset<short> slewMode = new Offset<short>(0x05DC, true);
private Offset<int> sendControl = new Offset<int>(0x3110, true);
private readonly int REFRESH_SCENERY = 65562;

private void MoveToEGLL()
{
    // Put the sim into Slew mode
    slewMode.Value = 1;
    FSUIPCConnection.Process();
    // Make a new point representing the centre of the threshold for 27L
    FsLatitude lat = new FsLatitude(51.464943d);
    FsLongitude lon = new FsLongitude(-0.434046d);
    FsLatLonPoint newPos = new FsLatLonPoint(lat, lon);
    // Now move this point 150 metres up the runway
    // Use one of the OffsetBy methods of the FsLatLonPoint class
    double rwyTrueHeading = 269.7d;
    newPos = newPos.OffsetByMetres(rwyTrueHeading, 150);
    // Set the new position
    playerLatitude.Value = newPos.Latitude.ToFSUnits8();
    playerLongitude.Value = newPos.Longitude.ToFSUnits8();
    // set the heading and altitude
    playerAltitude.Value = 0;
    playerHeadingTrue.Value = (uint)(rwyTrueHeading * (65536d * 65536d) / 360d);
}
```



```

FSUIPCConnection.Process();
// Turn off the slew mode
slewMode.Value = 0;
FSUIPCConnection.Process();
// Refresh the scenery
sendControl.Value = REFRESH_SCENERY;
FSUIPCConnection.Process();
}

```

Visual Basic.NET

```

Dim playerLatitude As Offset(Of Long) = New Offset(Of Long) (&H560)
Dim playerLongitude As Offset(Of Long) = New Offset(Of Long) (&H568)
Dim playerHeadingTrue As Offset(Of UInteger) = New Offset(Of UInteger) (&H580)
Dim playerAltitude As Offset(Of Long) = New Offset(Of Long) (&H570)
Dim slewMode As Offset(Of Short) = New Offset(Of Short) (&H5DC, True)
Dim sendControl As Offset(Of Integer) = New Offset(Of Integer) (&H3110, True)
Const REFRESH_SCENERY As Integer = 65562

Private Sub MoveToEGLL()
    ' Put the sim into Slew mode
    slewMode.Value = 1
    FSUIPCConnection.Process()
    ' Make a new point representing the centre of the threshold for 27L
    Dim lat As FsLatitude = New FsLatitude(51.464943D)
    Dim lon As FsLongitude = New FsLongitude(-0.434046D)
    Dim newPos As FsLatLonPoint = New FsLatLonPoint(lat, lon)
    ' Now move this point 150 metres up the runway
    ' Use one of the OffsetBy methods of the FsLatLonPoint class
    Dim rwyTrueHeading As Double = 269.7D
    newPos = newPos.OffsetByMetres(rwyTrueHeading, 150)
    ' Set the new position
    playerLatitude.Value = newPos.Latitude.ToFSUnits8()
    playerLongitude.Value = newPos.Longitude.ToFSUnits8()
    ' set the heading and altitude
    playerAltitude.Value = 0
    playerHeadingTrue.Value = rwyTrueHeading * (65536D * 65536D) / 360D
    FSUIPCConnection.Process()
    ' Turn off the slew mode
    slewMode.Value = 0
    FSUIPCConnection.Process()
    ' Refresh the scenery
    sendControl.Value = REFRESH_SCENERY
    FSUIPCConnection.Process()
End Sub

```

Distances and Bearings between Two Points

The `FsLatLonPoint` class also has methods for calculating the distance and bearing to or from another point. This example calculates the distance and bearing (true) to London Heathrow Airport.

C#

```

private Offset<long> playerLatitude = new Offset<long>(0x0560);
private Offset<long> playerLongitude = new Offset<long>(0x0568);

private void showDirectionAndHeadingToEGLL()
{
    // Setup the info for EGLL
    FsLatitude lat = new FsLatitude(51, 28, 39.0d);
    FsLongitude lon = new FsLongitude(0, -27, -41.0d);
    EGLL = new FsLatLonPoint(lat, lon);
    // Next get the point for the current plane position
    lon = new FsLongitude(playerLongitude.Value);
    lat = new FsLatitude(playerLatitude.Value);
    FsLatLonPoint currentPosition = new FsLatLonPoint(lat, lon);
    // Get the distance between here and EGLL
    double distance = currentPosition.DistanceFromInNauticalMiles(EGLL);
}

```

```

// Get the bearing (True)
double bearing = currentPosition.BearingTo(EGLL);
// Write the distance to the text box formatting to 2 decimal places
this.txtDistance.Text = distance.ToString("N2");
// Display the bearing in whole numbers
this.txtBearing.Text = bearing.ToString("F0");
}

```

VisualBasic.NET

```

Dim playerLatitude As Offset(Of Long) = New Offset(Of Long) (&H560)
Dim playerLongitude As Offset(Of Long) = New Offset(Of Long) (&H568)

Private Sub showDirectionAndHeadingToEGLL()
    ' Setup info for EGLL
    Dim lat As FsLatitude = New FsLatitude(51, 28, 39D)
    Dim lon As FsLongitude = New FsLongitude(0, -27, -41D)
    EGLL = New FsLatLonPoint(lat, lon)
    ' get current plane position
    lon = New FsLongitude(playerLongitude.Value)
    lat = New FsLatitude(playerLatitude.Value)
    Dim currentPosition As FsLatLonPoint = New FsLatLonPoint(lat, lon)
    ' Get the distance between here and EGLL
    Dim distance = currentPosition.DistanceFromInNauticalMiles(EGLL)
    ' Get the bearing (True)
    Dim bearing As Double = currentPosition.BearingTo(EGLL)
    ' Write the distance to the text box formatting to 2 decimal places
    Me.txtDistance.Text = distance.ToString("N2")
    ' Display the bearing in whole numbers and tag on a degree symbol
    Me.txtBearing.Text = bearing.ToString("F0") & Chr(&HB0)
End Sub

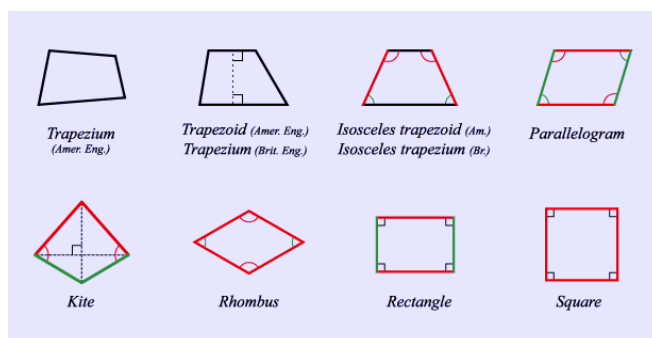
```

Finding out if a Point is inside an Given Area

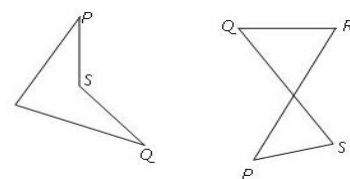
The `FsLatLonQuadrilateral` class lets you define a quadrilateral area on the Earth. The area is specified by the position of the four corners.

The area is then considered to be bounded by the four straight lines that join the corners. Note that the shape must be convex. Concave shapes will not work.

These shapes are OK:



These shapes are NOT OK:



To create a new `FsLatLonQuadrilateral` pass in the four `FsLatLonPoint` classes that define the corners. The corners do not have to be given in a certain order – they will be sorted internally into which is the North-East, South-West, etc.

This example makes an `FsLatLonQuadrilateral` that represents the four corners of London Heathrow Airport (EGLL).

C#

```
// First corner
FsLatitude lat = new FsLatitude(51,27.34);
FsLongitude lon = new FsLongitude(0,-26.97);
FsLatLonPoint point0 = new FsLatLonPoint(lat, lon);

// Second corner
lat = new FsLatitude(51, 29.03);
lon = new FsLongitude(0, -24.48);
FsLatLonPoint point1 = new FsLatLonPoint(lat, lon);

// Third corner
lat = new FsLatitude(51, 29.02);
lon = new FsLongitude(0, -29.66);
FsLatLonPoint point2 = new FsLatLonPoint(lat, lon);

// Forth corner
lat = new FsLatitude(51, 27.61);
lon = new FsLongitude(0, -29.65);
FsLatLonPoint point3 = new FsLatLonPoint(lat, lon);

// Create Quadrilateral
FsLatLonQuadrilateral quad = new FsLatLonQuadrilateral(point0, point1, point2, point3);
```

Visual Basic.NET

```
' First corner
Dim lat As FsLatitude = New FsLatitude(51, 27.34)
Dim lon As FsLongitude = New FsLongitude(0, -26.97)
Dim point0 As FsLatLonPoint = New FsLatLonPoint(lat, lon)

' Second corner
lat = New FsLatitude(51, 29.03)
lon = New FsLongitude(0, -24.48)
Dim point1 As FsLatLonPoint = New FsLatLonPoint(lat, lon)

' Third corner
lat = New FsLatitude(51, 29.02)
lon = New FsLongitude(0, -29.66)
Dim point2 As FsLatLonPoint = New FsLatLonPoint(lat, lon)

' Forth corner
lat = New FsLatitude(51, 27.61)
lon = New FsLongitude(0, -29.65)
Dim point3 As FsLatLonPoint = New FsLatLonPoint(lat, lon)

' Create Quadrilateral
Dim quad As FsLatLonQuadrilateral = New FsLatLonQuadrilateral(point0, point1, point2, point3)
```

The main use of this class is to find out if a given point is inside the defined area. For example, by using the quadrilateral we made above, we can tell if the player is with the bounds of London Heathrow (EGLL).

C#

```
lat = new FsLatitude(playerLatitude.Value);
lon = new FsLongitude(playerLongitude.Value);
FsLatLonPoint playerPosition = new FsLatLonPoint(lat, lon);

if (quad.ContainsPoint(playerPosition))
{
    // Player is inside the boundary of EGLL
    // Do stuff
}
```

Visual Basic.NET

```
lat = New FsLatitude(playerLatitude.Value)
lon = New FsLongitude(playerLongitude.Value)
Dim playerPosition As FsLatLonPoint = New FsLatLonPoint(lat, lon)

If quad.ContainsPoint(playerPosition) Then
    ' Player is inside the boundary of EGLL
    ' Do stuff
End If
```

Finding out if the Player is on a Runway

If you know the Latitude and Longitude of the four corners of the runway you can simply make an `FsLatLonQuadrilateral` and test if the plane is within that area. Details of this are in the [previous section](#).

It is more common however to have a different set of information about a runway. For example, Pete Dowson's Make Runways program generates text files with data such as the Lat/Lon of the runway threshold, heading, length and width.

There is a static helper method on the `FsLatLonQuadrilateral` class that lets you create a new quadrilateral from the following information:

- Threshold Centre Lat/Lon
- Width in Feet
- Length in Feet
- True Heading

This method returns a normal `FsLatLonQuadrilateral` which you can use as described in the [previous section](#).

The following example detects if the player is on Runway 27L of London Heathrow (EGLL). The information about the runway (heading, width etc) is hard-coded in the example. I got this information from one of the text files generated by Make Runways. A real application that has to cope with many/all runways would need to get this information from a database or parse one of the Make Runways text files.

In a real application you won't want to be generating the Quadrilateral from the runway data every time you test. Generate it once when you know the runway you want to test for and then use the same `FsLatLonQuadrilateral` object to do the testing. The sample application provided demonstrates this technique.

C#

```
private Offset<long> playerLatitude = new Offset<long>(0x0560);
private Offset<long> playerLongitude = new Offset<long>(0x0568);
private Offset<short> onGround = new Offset<short>(0x0366);

private void CheckPlayerIsOnRunway()
{
    FsLongitude lon = new FsLongitude(playerLongitude.Value);
    FsLatitude lat = new FsLatitude(playerLatitude.Value);
    // Get the point for the current plane position
    FsLatLonPoint currentPosition = new FsLatLonPoint(lat, lon);
    // Now define the Quadrangle for the 27L (09R) runway.
```

```

// We could just define the four corner Lat/Lon points if we knew them.
// In this example however we're using the helper function to calculate the points
// from the runway information. This is the kind of info you can find in the output files
// from Pete Dowson's MakeRunways program.
FsLatitude rwyThresholdLat = new FsLatitude(51.464943d);
FsLongitude rwyThresholdLon = new FsLongitude(-0.434046d);
double rwyMagHeading = 272.7d;
double rwyMagVariation = -3d;
double rwyLength = 11978d;
double rwyWidth = 164d;

// Call the static helper on the FsLatLonQuarangle class to generate
// the Quadrilateral for this runway...
FsLatLonPoint thresholdCentre = new FsLatLonPoint(rwyThresholdLat, rwyThresholdLon);
double trueHeading = rwyMagHeading + rwyMagVariation;
runwayQuad = FsLatLonQuadrilateral.ForRunway(thresholdCentre, trueHeading, rwyWidth,
rwyLength);

// Now check if the player is on the runway:
// Test is the plane is on the ground and if the current position is in the bounds of
// the runway Quadrangle we calculated in the constructor above.
if (this.onGround.Value == 1 && runwayQuad.ContainsPoint(currentPosition))
{
    // Player is on the runway
    // Do Stuff
}
}

```

Visual Basic.NET

```

Dim playerLatitude As Offset(Of Long) = New Offset(Of Long) (&H560)
Dim playerLongitude As Offset(Of Long) = New Offset(Of Long) (&H568)
Dim onGround As Offset(Of Short) = New Offset(Of Short) (&H366)

Private Sub CheckPlayerIsOnRunway()
    Dim lon As FsLongitude = New FsLongitude(playerLongitude.Value)
    Dim lat As FsLatitude = New FsLatitude(playerLatitude.Value)
    ' Get the point for the current plane position
    Dim currentPosition As FsLatLonPoint = New FsLatLonPoint(lat, lon)
    ' Now define the Quadrangle for the 27L (09R) runway.
    ' We could just define the four corner Lat/Lon points if we knew them.
    ' In this example however we're using the helper function to calculate the points
    ' from the runway information. This is the kind of info you can find in the output files
    ' from Pete Dowson's MakeRunways program.
    Dim rwyThresholdLat As FsLatitude = New FsLatitude(51.464943D)
    Dim rwyThresholdLon As FsLongitude = New FsLongitude(-0.434046D)
    Dim rwyMagHeading As Double = 272.7D
    Dim rwyMagVariation As Double = -3D
    Dim rwyLength As Double = 11978D
    Dim rwyWidth As Double = 164D

    ' Call the static helper on the FsLatLonQuarangle class to generate
    ' the Quadrilateral for this runway...
    Dim thresholdCentre As FsLatLonPoint = New FsLatLonPoint(rwyThresholdLat, rwyThresholdLon)
    Dim trueHeading As Double = rwyMagHeading + rwyMagVariation
    runwayQuad = FsLatLonQuadrilateral.ForRunway(thresholdCentre, trueHeading, rwyWidth,
rwyLength)

    ' Now check if the player is on the runway:
    ' Test is the plane is on the ground and if the current position is in the bounds of
    ' the runway Quadrangle we calculated in the constructor above.
    If onGround.Value = 1 And runwayQuad.ContainsPoint(currentPosition) Then
        ' Player is on the runway
        ' Do Stuff
    End If
End Sub

```

AI Traffic

Getting the AI Traffic Information

The AI Traffic Information is supplied by the `AITrafficServices` class. An instance of this is available from the `FSUIPCConnection` class. If you want to use the AI Traffic services I recommend you grab a reference to this instance at the start of your application and use that. Note that the `AITrafficServices` instance is not initialised until the connection is open.

This isn't required but it makes the code easier to read and there is less to type. Most of the examples in this guide however use the full syntax so as to explain the object model more clearly.

To read the latest AI Traffic from FSUIPC you just need to call `RefreshAITrafficInformation()` on `AITrafficServices`. Note that you don't need to call `Process()`, the traffic services operate independently of any `Process()` calls your application makes.

If you are just working with Ground or Airborne traffic then there is an overload which lets you choose which of these are refreshed.

Once you have called `RefreshAITrafficInformation()` you then have three strongly-typed lists of `AIPlaneInfo` instances available. These are:

```
AITrafficInformation.AirbourneTraffic  
AITrafficInformation.GroundTraffic  
AITrafficInformation.AllTraffic
```

You can iterate through each of these lists and work on each AI plane. The planes are ordered by distance from the player, the closest being at index 0. If you are particularly interested in one or more AI Planes, you can hold a reference to the relevant `AIPlaneInfo` objects. They will be updated when you next call `RefreshAITrafficInformation()`.

The following example shows adding the `AtcID` and status of each plane to a list box. There are many other properties available in the `AIPlaneInfo` class. See the reference manual or Intellisense for details.

C#

```
private void readAI()  
{  
    // Get the AI Traffic info back from FSUIPC  
    // This happens immediatly, no need to call Process()  
    FSUIPCConnection.AITrafficServices.RefreshAITrafficInformation();  
    // The following code iterates through the list and adds each plane's ATCid to a listbox  
    // along with it's state in brackets. (Landing, enroute etc..)   
    this.listBox1.Items.Clear();  
    foreach (AIPlaneInfo plane in FSUIPCConnection.AITrafficServices.AllTraffic)  
    {  
        this.listBox1.Items.Add(plane.ATCIdentifier + " (" + plane.State.ToString() + ")");  
    }  
}
```

Visual Basic.NET

```
Private Sub readAI()  
    ' Get the AI Traffic info back from FSUIPC  
    ' This happens immediatly, no need to call Process()  
    FSUIPCConnection.AITrafficServices.RefreshAITrafficInformation()  
    ' The following code iterates through the list and adds each plane's ATCid to a listbox  
    ' along with it's state in brackets. (Landing, enroute etc..)  
    this.listBox1.Items.Clear()  
    For Each Plane As AIPlaneInfo In FSUIPCConnection.AITrafficServices.AllTraffic  
        Me.listBox1.Items.Add(Plane.ATCIdentifier & " (" & Plane.State.ToString() & ")")  
    Next Plane  
End Sub
```

Filtering AI Traffic

After refreshing the traffic information you can optionally apply a filter to delete planes you are not interested in. You can filter planes that do not fall between an altitude range, are not within a certain distance from the player or those that do not fall between two bearings from the player.

You can also choose whether to filter the ground traffic, airborne traffic or both.

To apply the filter you must call the `ApplyFilter()` method after each `RefreshAITrafficInformation()` call. See the Intellisense or reference manual for information about the parameters you need to pass to this method.

The distance filtering here is done by the DLL. FSUIPC also has a filtering facility for distance that is controlled by the INI file. The filtering here does not affect the FSUIPC filter. If the FSUIPC filter is to 30NM then no traffic further away than that will be present, even if you set the `WithinDistance` parameter of `ApplyFilter` to 40NM.

By default the FSUIPC filter is set to a maximum of 40NM. You can override this INI setting if you want using this DLL – see the section later called: [Overriding the AI Traffic settings in FSUIPC.INI](#)

The example below applies a filter to the airborne AI Traffic. It filters out planes over 10,000ft and more than 30 nautical miles away.

C#

```
FSUIPCConnection.AITrafficServices.ApplyFilter(false, true, 0, 360, null, 10000d, 30d);
```

Visual Basic.NET

```
FSUIPCConnection.AITrafficServices.ApplyFilter(False, True, 0, 360, Nothing, 10000D, 30D)
```

Getting Extended Identifying Information

The following properties on the `AIPlaneInfo` class are not filled in by default.

```
TailNumber  
AirlineAndFlightNumber  
AircraftTypeAndModel  
AircraftTitle
```

The reason is that this information takes a long time to retrieve from FSUIPC (about one second per plane for all four fields). This information never changes however, so the performance hit only needs to be taken once per plane.

There are two ways to get this information.

1. You can use the `GetExtendedPlaneIdentifiers()` method of the `AIPlaneInfo` class. This lets you specify which of the above fields you want populating for that particular AI Plane.
2. You can use the `UpdateExtendedPlaneIdentifiers()` method of the `AITrafficServices` class. This lets you specify which of the above fields you want populating for *all* AI Planes. When a new plane is discovered the requested fields will be populated. Depending on how many fields you are requesting there may be up to a second delay. This only ever happens once though.

The main problem will come with the first ever call to `RefreshAITrafficInformation()` when all the planes will need this information filling in. If your user has heavy AI traffic (e.g. 60 planes) and you request all 4 field your application could take over a minute to return from the `Refresh()` call.

Care should be used when working with this information.

It is possible to change the ATC Identifier of the AI Planes to contain one of these extended pieces of information. This has no performance impact; however there are two things to note:

1. The ATC Identifier is only 15 characters long in FSUIPC, so the information may be truncated. (The FSUIPC documentation seems to suggest that such truncation would be rare however).
2. Other programs using FSUIPC may expect and require the ATC Identifier to be the in default format. Therefore if you change this you may stop other programs from working.

To change the ATC Identifier see the section below called: [Overriding the AI Traffic settings in FSUIPC.INI.](#)

Finding Active Runways

This DLL can give you a list of runways in use for a particular airport. The airport must have active AI traffic taking off and landing there. This data is only as good as the AI Traffic Information. To get the up-to-date runway info you must first call `RefreshAITrafficInformation()`.

The more AI traffic there is, the more likely that active runway information will be available for a particular airport.

The DLL gets this data internally; it does not use the FSUIPC facilities at offset D000 because it takes ages to get this information from FSUIPC. The DLL can give it to you instantly. However, unlike the FSUIPC facility, the DLL can only get runway information from planes within the range setting in the FSUIPC.INI file.

Use the two methods provided on the `AITrafficServices` class:

```
GetDepartureRunwaysInUse (AirportICAOCode)  
GetArrivalRunwaysInUse (AirportICAOCode)
```

You need to pass in the ICAO code of the airport you are interested in. The methods will return you a list of `FSRunway` objects which you can iterate through using a For Each loop. The `FSRunway` class has properties to tell you the number of the runway and the designator (Left, Centre, Right, Water). There is also a handy `ToString()` method that will give you a string representation of the runway number and designator.

The following example displays the arrival and departure runways in use at EGLL in a list box.

C#

```
private void getActiveRunways()  
{  
    // Get a reference to the AITrafficServices class (saves typing)  
    AITrafficServices AI = FSUIPCConnection.AITrafficServices;  
    // Refresh the traffic information  
    AI.RefreshAITrafficInformation();  
    // Get the arrival runways in use  
    List<FSRunway> runways = AI.GetArrivalRunwaysInUse("EGLL");  
    // Display in the listbox  
    this.lstArrival.Items.Clear();  
    foreach (FSRunway rw in runways)  
    {  
        this.lstArrival.Items.Add(rw.ToString());  
    }  
    // same for departure runways  
    runways = AI.GetDepartureRunwaysInUse("EGLL");  
    this.lstDeparture.Items.Clear();  
    foreach (FSRunway rw in runways)  
    {  
        this.lstDeparture.Items.Add(rw.ToString());  
    }  
}
```

Visual Basic.NET

```
Private Sub getActiveRunways()  
    ' Get a reference to the AITrafficServices class (saves typing)  
    Dim AI As AITrafficServices = FSUIPCConnection.AITrafficServices  
    ' Refresh the traffic information  
    AI.RefreshAITrafficInformation()  
    ' Get the arrival runways in use  
    Dim runways As List(Of FSRunway) = AI.GetArrivalRunwaysInUse("EGLL")  
    ' Display in the listbox  
    Me.lstArrival.Items.Clear()  
    For Each rw As FSRunway In runways  
        Me.lstArrival.Items.Add(rw.ToString())  
    Next rw  
    ' same for departure runways  
    runways = AI.GetDepartureRunwaysInUse("EGLL")  
    Me.lstDeparture.Items.Clear()  
    For Each rw As FSRunway In runways  
        Me.lstDeparture.Items.Add(rw.ToString())  
    Next rw  
End Sub
```

Overriding the AI Traffic settings in FSUIPC.INI

The FSUIPC.INI file contains settings that affect how FSUIPC reports the AI Traffic. Usually this is left up to the user to set according to their tastes. You can however override these settings. This does not change the INI file, it just changes the way FSUIPC responds while your application is running.

The following settings can be overridden:

- The distance limit for Airborne AI traffic
- The distance limit for Ground AI traffic when the player is in the air
- The distance limit for Ground AI traffic when the player is on the ground
- To give priority to the Ground AI traffic in 'Active' states, over ones that are sleeping or starting up
- The format of the ATC identifier. This can be one of the following:
 - Tail Number
 - Airline and flight number (or tail number for GA Planes)
 - Aircraft Type
 - Aircraft Title
 - Aircraft Type plus last 3 digits of tail number
 - Aircraft Model

To change these INI setting use one of the following two methods on the `AITrafficServices` class as appropriate:

```
OverrideAirborneTrafficINISettings  
OverrideGroundTrafficINISettings
```

Note that these overrides will be cancelled by FSUIPC after 20 seconds. To keep them active you must call these methods at regular time intervals before the 20 seconds is up. The FSUIPC documentation recommends every 5 seconds.

Writing AI Traffic (TCAS) Information to the FSUIPC Tables

The DLL also has a facility for writing data into the FSUIPC AI Traffic tables. This does NOT create AI Planes inside Flight Simulator. In fact Flight Simulator never knows anything about these created planes.

The main use for this is to inject AI Planes into FSUIPC so they are read by other FSUIPC applications (or gauges). These are most commonly TCAS radars/displays, but there could be other applications like departure/arrival boards. So as not to give the impression that the DLL creates actual AI traffic the DLL refers to these created entities as TCAS targets.

Writing TCAS targets is a two-stage process using two methods on the `AITrafficServices` class. First you use the `AddTCASTarget()` method to define the information about the new plane. There are many parameters on this method. If you don't know some of the information you can just send

0 values. For example if you don't know the vertical speed you can just send 0. FSUIPC does nothing with most of these values except pass them on to the application requesting AI Traffic Information.

Obviously the data you write must be useful for the intended recipient. For example if you are writing targets for a TCAS gauge to display then you need at minimum the Longitude, Latitude and Altitude.

The important parameters that are required by FSUIPC are the ID and the ATC Identifier. For more information about this method and its parameters see the reference manual or the Intellisense.

You call `AddTCASTarget ()` once for each plane you want to inject.

Once you have added all the planes you need, the second stage is to call `SendTCASTargets()` to actually send the data to FSUIPC. You don't need to call `Process()`.

Note that FSUIPC will erase these planes after 8-12 seconds. Therefore you need to send this data on a regular basis (probably every 5 seconds at least. It could of course be more often).

Each time you send the data you must write the new data using the two stage process of multiple `AddTCASTarget()` calls followed by `SendTCASTargets()`. The DLL does not remember TCAS targets from previous `SendTCASTargets()` calls.