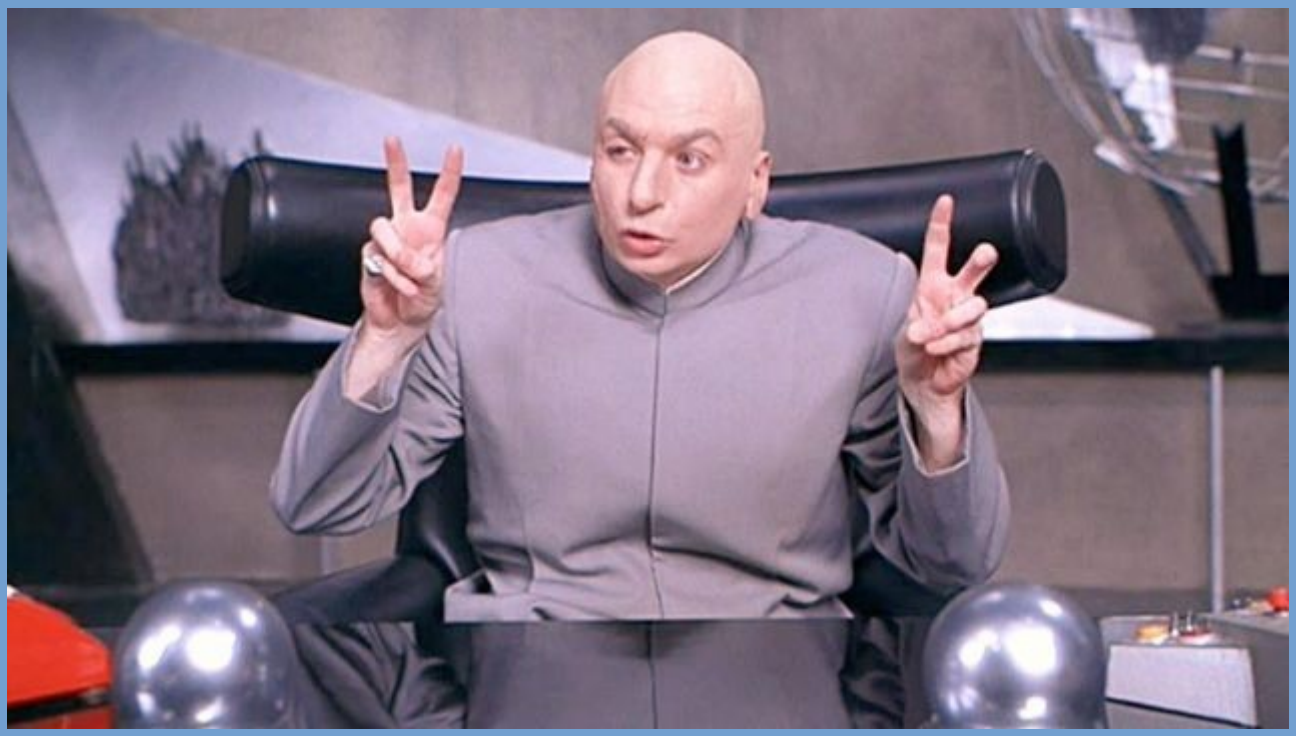
FSUIPC Java SDK



README

https://mouseviator.com/fsuipc-java-sdk

Version 1.0, Last update: 08/14/2020

by Radek Henys (Mouseviator)

Table Of Contents

1) Introduction

This is the brief readme for Java FSUIPC SDK by Radek Henys (Mouseviator) :) I am, like many people I quess, lazy, so instead of writing it again, here is an excerpt from the javadoc:

This whole Java FSUIPC SDK is based upon SDK written by Mark Burton, later amended by Paul Henty for 64 bit environment. But, it has been greatly rewritten and uses different approach.

*Why the rewrite? Well, the main reason for me was performance considerations. After studying FSUIPC C SDK, the wrapper library and the Java SDK by Mark, I found out that the C/C++ wrapper library (**fsuipc_java64.dll**, **fsuipc_java32.dll**), that implements the **FSUIPCWrapper.readData** and **FSUIPCWrapper.writeData** functions, calls the **FSUIPC_Process** function. What is the catch? Well, normally when working with the FSUIPC C/C++ SDK, you call **FSUIPC_Read** and **FSUIPC_Write** functions to tell FSUIPC what data you want to read/write, kind of registering data requests, and than process all of them using the **FSUIPC_Process** function. I have read on the forums that this function is quite heavy on processing time – thus, it is not good to call it very often. But this is how it was implemented in the previous wrapper libraries. The **FSUIPC_Process** function got called every time you called the read or write functions. It may not cause trouble in simple projects. But, when you would read multiple values, continuously, like in the loop, for example for flight monitoring app, I think that might be a performance issue sooner or later. Not mentioning, that your app probably would not be the only one that FSUIPC would have to provides it's services to. That is why I decided to try to write a different SDK, that will try to reflect the approach that we would use when writing the code in C/C++. This SDK thinks of FSUIPC read/writes as of data requests. It allows you to register multiple of them and then process them all via one call of the process function.*

***NOTE:** This SDK is NOT considered to be complete! While I believe it contains solid foundation for developing applications using FSUIPC in Java, it is far away from complete. There are just a couple of helper classes which are more of an examples of what to do with the "data requests", how to write them. It would be nearly impossible for me to cover all, most of FSUIPC offsets with helper classes and also test them. Look on the FSUIPC.jar library source files, on the source of helper classes to see how they are implemented.*

2) Contents of the archive

Depending on what archive you downloaded (full source codes or just pre-compiled SDK distribution…), you should found one or more of these folders:

- **FSUIPC_Java_dist** – the directory with pre-compiled Java package. It should contain the **FSUIPC.jar**, **fsuipc_java32.dll**, **fsuipc_java64.dll** and javadoc folder. These files are ready to use in your project. The FSUIPC library was written in JDK 11 (AdoptOpenJDK 11.0.7.10 Hotspot).
- **FSUIPC** – the directory containing the source code of the **FSUIPC.jar** library. The project is for ***Netbeans*** (Apache Netbeans 12 – to be specific). It was written using JDK 11 , compiled with AdoptOpenJDK 11.0.7.10 Hotspot, and tested also with 32bit JDK 11 (AdoptOpenJDK 11.0.8.10 Hotspot). This folder contains 3 batch files – **Make JavaDoc.cmd** – to make Javadoc (but not needed actually, Netbeans can do that if setup correctly). The **MakeHeaderFiles32.cmd** and **MakeHeaderFiles64.cmd** will create the header file for the FSUIPCWrapper class (that is the one containing native functions) using the 32/64bit JDK. These header files are then used in the **CWrapper32** (fsuipc_java32.dll) and **CWrapper64** (fsuipc_java64.dll) C++ projects, which implements the native functions. Note that all of these batch files contains absolute paths on my system and therefore WILL NEED adjustments for your system.
- **CWrapper32** - Contains the source code for the **fsuipc_java32.dll** – the 32bit library version that implements native functions of the **FSUIPCWrapper** java class. It is written in C++. The project is for ***Visual Studio (C++) 2019***. When you open this project in Visual Studio, it will probably need some settings adjustment, as some paths will be different on your system than on mine, but skill-full developer like you will have not big issues with that, I am sure. For sure you will have to set paths to Java JDK header files, so that C++ knows about them (In VS 2019 this is under project properties → C/C++ → General → Additional Include Directories). The folder is pretty BIG as it contains the packages for boost libraries, that the library uses for logging purposes.
- **CWrapper64** - Contains the source code for the **fsuipc_java64.dll** – the 64bit library version that implements native functions of the **FSUIPCWrapper** java class. It is written in C++. The project is for ***Visual Studio (C++) 2019***. When you open this project in Visual Studio, it will probably need some settings adjustment, as some paths will be different on your system than on mine, but skill-full developer like you will have not big issues with that, I am sure. For sure you will have to set paths to Java JDK header files, so that C++ knows about them (In VS 2019 this is under project properties → C/C++ → General → Additional Include Directories). The folder is pretty BIG as it contains the packages for boost libraries, that the library uses for logging purposes.
- **FSUIPCSimpleTest** – contains simple sample application that shows the usage of some FSUIPC functionality. It shows the basics of “FSUIPC data request”

concept of this SDK, the connection to FSUIPC and reading one time data requests. The project is for ***Netbeans*** (Apache Netbeans 12 – to be specific). Some settings adjustment will be required after opening the project. I had the **FSUIPC** project set as dependency, you can do the same or you can point it to pre-compiled FSUIPC.jar from **FSUIPC_Java_dist.**

- **FSUIPCSimMonitor** – contains more complex example of FSUIPC library usage. Most SDK are shipped with basic examples, which really does not show you much. Well, I tried to do better here :) This is SWING GUI application, with map. It will wait for successful FSUIPC connection and then show various aircraft and sim data, updating aircraft position on map. Shows even more from the concept of FSUIPC data requests of this SDK – the continual data requests and FSUIPC listener. The project is for ***Netbeans*** (Apache Netbeans 12 – to be specific). Some settings adjustment will be required after opening the project. I had the **FSUIPC** project set as dependency, you can do the same or you can point it to pre-compiled FSUIPC.jar from **FSUIPC_Java_dist.** This example app actually shows also a write requests. You can Pause the sim using the Pause button or toggle the Slew mode using the Slew button.

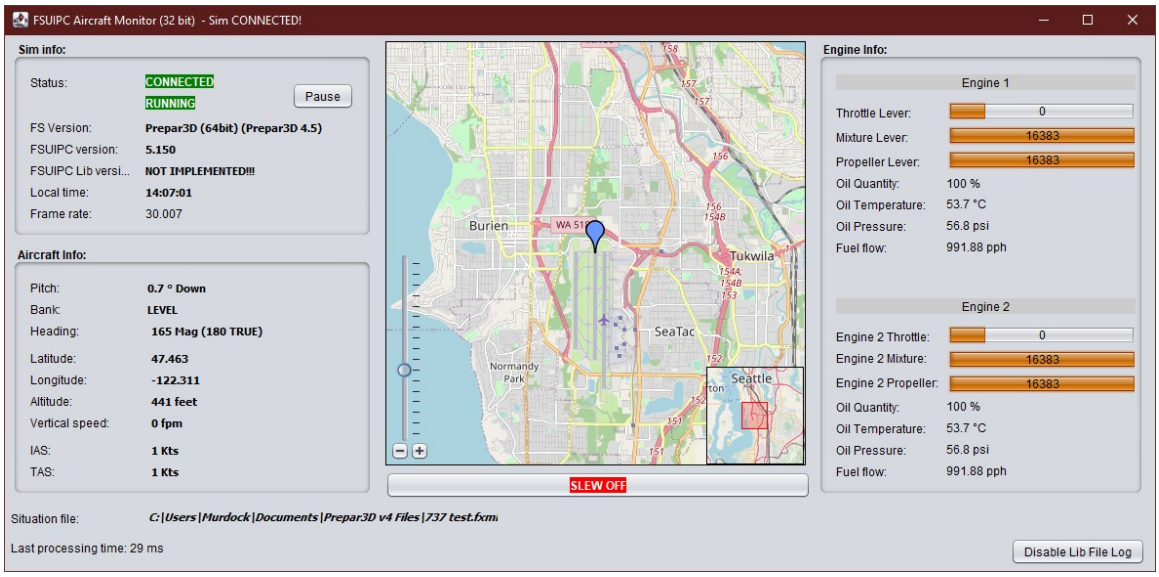


Figure 1: The FSUIPC Aircraft Monitor example app using Java FSUIPC library

- **C++ Memory Validator Reports** – this folder contains exported reports from the C++ Memory Validator software, that I used to monitor the **fsuipc_java32.dll** and **fsuipc_java64.dll** for memory leaks while running the FSUIPC Sim Monitor example for about an hour. There are HTML reports, ans also stored sessions, which you might be able to load in the software, if you own it.

3) API documentation

You will find that inside the **javadoc** folder in the **FSUIPC_Java_dist** folder. Read especially the documentation of the **FSUIPC** class in the **com.mouseviator.fsuipc** package, as it is the core class and the doc has brief samples in it.

4) License?

Well, the API from Mark Burton and Paul Henty were given for free, this one is also FREE. If you want to be specific, than say it is LGPL.

5) Bugs?

Well, it is most likely that some have found their way in the SDK. If you find any, I will be happy if you let me know. See the Contact chapter. Because honestly, this is “not a simple project”, at least not for me. It is not a hard hard, but neither simple. I am a seasonal C++ programmer, the times C++ was my second language are gone for a while. JNI (Java Native Interface), well, I did not read the whole documentation either. So what I was afraid of were memory leaks. Because, to be honest, the way this works is a little bit crazy so to speak.

How is it? Well, when you call **FSUIPC_read** or **FSUIPC_write** function from the FSUIPC C/C++ SDK to read/write to/from some FSUIPC offset, it will store that request to a memory. You can call these functions numerous times, the requests will get stores, but you will get no values (will not write any to the sim), until you call **FSUIPC_Process** function. That will trigger the inter process communication, send the stored data requests to the FSUIPC library running within the simulator, which will perform the reading and writing of required data, and send back results (actually, is stores them in the memory at place as indicated by the data requests that were send). Don't get it? In other words, the FSUIPC_Process function will tell the FSUIPC library: „here are the data that we want to read/write, store the results to this address….“. This is still the C/C++ FSUIPC SDK, where is the Java part you ask?

Well, the FSUIPCWarpperjava class implements some native methods. These reflect the FSUIPC_read, FSUIPC_write, FSUIPC_Process functions (and some others). In a way, we are exchanging data here between Java and C/C++ code. That is, what Java JNI (Java Native Interface is for). And this is the side from FSUIPC C/C++ SDK to Java code. The problem is how to keep reference to the Java variable after the native function call ended… to be able to return values after process function. Because, we cannot know when developer will call it. Java has no pointers as C/C++ and that is, I believe, why the original SDK called FSUIPC_Process in each call of the FSUIPC_read and FSUIPC_write implementation within the FSUIPCWrapper class (otherwise, it would lead to memory exception). I modified the libraries so that these functions does not do that and they behave like the FSUIPC SDK C/C++ read and write functions. For this to work, the library must remember the memory addresses where FSUIPC stores results for each request made, and also memory that acts as inter-changer between C/C++ and respective Java variable. Have no clue what I am writing about? Ok, no problem. In other words, we are doing quite a things with memory here :) Not that it would that complex at the end of the day. If you look at the code, it is pretty simple. But, as I wrote before, I am not a JNI expert, so the risk of unintentional memory leak is there.

But, I tested both versions of the libraries (**fsuipc_java32.dll** and **fsuipc_java64.dll**) with the software called C++ Memory Validator, which is designed to find such as issues. Both times, the sample **FSUIPC Sim Monitor** app monitored the sim (Prepar 3D) for about an

hour. It is an excellent program by the way. But why I am writing about it. The result indicate that there were about 980-1800 bytes (28 potential memory leaks) of unreleased memory. These were all originated from 3rd party source files, and I think are potential, because I was only monitoring the said libraries, so the freeing of memory indicated as these memory leaks simply may not have been recorded. If you want, you can examine the reports. The exports are stored in the **C++ Memory Validator Reports** folder.

6) Contact

If you want to contact me in regard of this JAVA FSUIPC SDK, you can do so via an email: admin@mouseviator.com, or leave a post in the forums at https://forums.mouseviator.com. I will gladly hear from you any suggestions, bug reports etc.

7) Thanks

Many people behind the flight simulator platforms we use, Pete and John Dowson for FSUIPC, Mark Burton and Paul Henty for their SDK, as it was great starting point for me.