This Guide is a fork of https://gist.github.com/Falki-git/929859d72ed0a5437251213b0925672a .

It's a simple adaptation of the instruction for debugging a KSP2 Mod (the previous versions was for KSP 1 v1.8)

The Mac and linux options and JetBrains Rider option have not been tested and therefore have been removed for the moment.

This Guide applies only to KSP 2 0.1.5 to 1.11 and later. It covers modifying a KSP2 installation to allow :

- In IDE debugging of KSP plugins by using breakpoints, inspecting runtime variables, doing step-by-step execution, etc. Visual Studio and VSCode option are covered.

This guide is extensively tested for a Windows / Visual Studio scenario.

However, it is theoretically possible to make all that work under MacOS or Linux, either with the Rider IDE or Visual Studio for Mac. This guide has limited details about those scenarios. I encourage you to leave a comment if you have additional information / experience.

Modifying KSP for profiling/debugging

Downloading the Unity editor

You will need to replace some of KSP2 files by those provided in the Unity editor. Beside for getting those files, you need the Unity editor if you want to use the Unity profiler :

- For KSP2 **0.1.5**, you need **Unity 2022.3.5f1**

The provided links should get you to the download page. If they don't work, find them on the Unity download archive.

Before you begin, it is highly recommended to make a separate copy of a **clean** install of KSP for this. Just copy the whole KSP folder somewhere else.

Windows instructions

- Open the folder where you installed the Unity editor
- Navigate to the `\Unity 2022.3.5f1\Editor\Data\PlaybackEngines\windowsstandalonesupport\Variations\win64_player_development_mono` folder
- Copy the `UnityPlayer.dll` and `WinPixEventRuntime.dll` files from the `win64_development_mono` folder to the KSP root folder (the one containing `KSP2_x64.exe`). Backup the original files elsewhere if you want to revert that installation to a non-dev build latter.
- From your KSP root folder, open the `KSP2_x64_Data` folder
- In that folder, open the `boot.config` file with a text editor
- Add the following line : `player-connection-debug=1` and save the modifications
- Launch KSP

- The Windows firewall may ask you to authorize KSP to communicate on your network. You **must** allow it. If you previously launched KSP from that install and denied that authorization, you need to go into the Windows firewall settings and authorize it.

Linux instructions (untested)

- Open the folder where you installed the Unity editor
- Navigate to the `Editor\Data\PlaybackEngines\LinuxStandaloneSupport\Variations\linux64_withgfx_development_mono` folder
- Copy the `UnityPlayer.so` and `UnityPlayer_s.debug` files from the `linux64_withgfx_development_mono` folder to the KSP root folder (the one containing `KSP.x86_64`).
  Backup the original files elsewhere if you want to revert that installation to a non-dev build latter.
- From your KSP root folder, open the `KSP_Data` folder
- In that folder, open the `boot.config` file with a text editor
- Add the following line : `player-connection-debug=1` and save the modifications

MacOS instructions (untested)

- Open the folder where you installed the Unity editor
- Navigate to the `Editor\Data\PlaybackEngines\MacStandaloneSupport\Variations\macosx64_development_mono\UnityPlayer.app\Contents\Frameworks` folder, locate the `UnityPlayer.dylib` file.
- Open the root folder of your KSP install, and navigate to the `KSP.app\Contents\Frameworks` folder (you need to right-click on `KSP.app` and select `Show Package Contents` to be able to browse it).
- Replace the `UnityPlayer.dylib` file of your KSP install by the Unity one.
- From the Unity editor root, navigate to `Editor\Data\PlaybackEngines\MacStandaloneSupport\Variations\macosx64_development_mono\UnityPlayer.app\Contents\MacOS` and locate the `UnityPlayer` file.
- From the root KSP folder, navigate to your `KSP.app\Contents\MacOS` folder and locate the `KSP` file.
- Delete the original `KSP` file, copy the `UnityPlayer` file there and rename it to `KSP`.
- From the root KSP folder, navigate to your `KSP.app\Contents\Resources\Data` folder.
- Open the `boot.config` file with a text editor.
- Add the following line : `player-connection-debug=1` and save the modifications

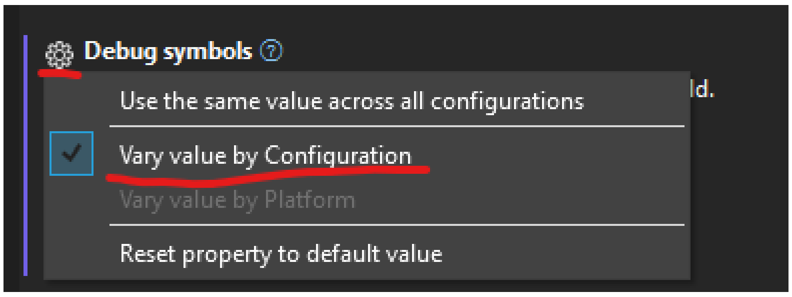
Debugging KSP plugins

Project setup

Visual Studio instructions

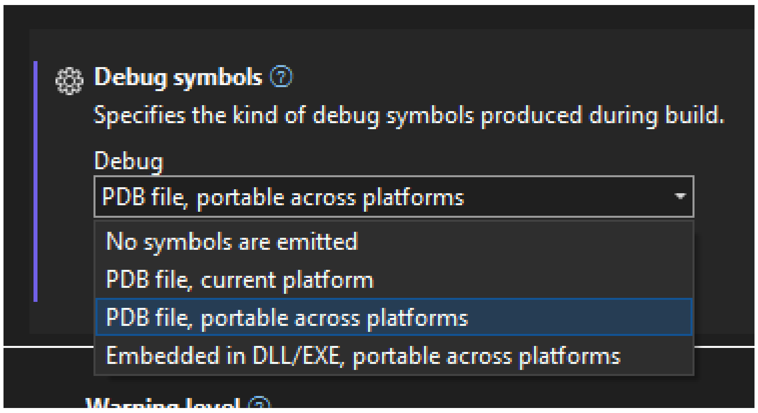
- Install the **Visual Studio Tools for Unity** extension (available from the VS installer or from the VS extension menu in older VS versions).
- In Visual Studio, open your plugin project

- From the solution explorer panel, right-click on the project and select `Properties`
- Go to `Build/General`
- at the end of chapter, find the `Debug Symbols` section and click on the gear icon


Debug symbols
ld.
Use the same value across all configurations
Vary value by Configuration
Vary value by Platform
Reset property to default value

select "Vary Value for Configuration"

- Select for debug "PDB File, portable"



Debug symbols

Specifies the kind of debug symbols produced during build.

Debug

PDB file, portable across platforms

No symbols are emitted
PDB file, current platform
PDB file, portable across platforms
Embedded in DLL/EXE, portable across platforms
Warning level

and no symbol for release

Alternative instructions (other IDE including VSCode and Linux/MacOS)

This is an alternative to the above Visual Studio GUI based instructions :

- Open your `*.csproj` project file in a text editor
- Locate the following section :

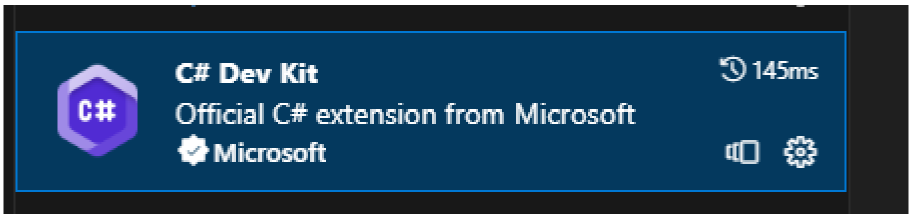
```
<PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Debug|AnyCPU' ">
```

- Under that section (make sure it's the right one, there is a similar one for the `Release` configuration), set the `DebugType` property to `portable` :

```
<DebugType>portable</DebugType>
```

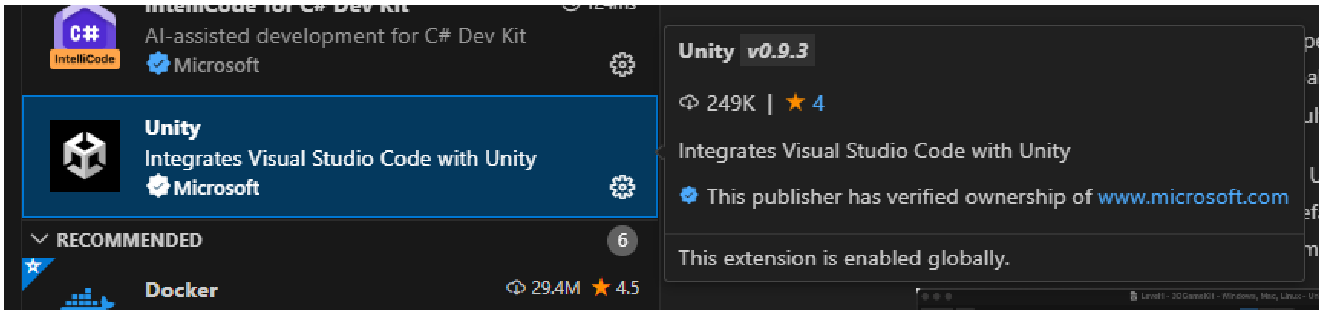
Additional tips for VSCode

- Install the C# Dev Kit

- 

C# Dev Kit
Official C# extension from Microsoft
Microsoft
145ms

- Install the Unity Engine tool



IntelliCode for C# Dev Kit
AI-assisted development for C# Dev Kit
Microsoft

Unity
Integrates Visual Studio Code with Unity
Microsoft

RECOMMENDED 6

Docker 29.4M ★ 4.5

Unity v0.9.3
249K | ★ 4
Integrates Visual Studio Code with Unity
This publisher has verified ownership of www.microsoft.com
This extension is enabled globally.

Add build tasks :
`F1 - >task` select `configure tasks` or open `.vscode/task.json`

add Debug and Release task like this :

```
    "tasks": [
        {
               "label": "build release",
               "command": "dotnet",
               "type": "shell",
               "args": [
                   "build",
                   "-c",
                   "Release",
                   "/property:GenerateFullPaths=true",
                   "/consoleloggerparameters:NoSummary"
               ],
               "group": {
                   "kind": "build",
                   "isDefault": true
               },
               "presentation": {
                   "reveal": "silent"
               },
               "problemMatcher": "$msCompile"
           },
           {
               "label": "build debug",
               "command": "dotnet",
               "type": "shell",
               "args": [
                   "build",
                   "-c",
                    "Debug",
                    "/property:GenerateFullPaths=true",
                    "/consoleloggerparameters:NoSummary"
                ],
                "group": {
                    "kind": "build",
                    "isDefault": true
                },
                "presentation": {
                    "reveal": "silent"
                },
                "problemMatcher": "$msCompile"
            },
        ]
}
```

Build using the standard build tasks (Ctrl+Shift+B by default)

The Intellisense code completion and navigation is quite long to work fully. I've has to wait for quite long to have a fully working version on my first try. But I've not made any change in configuration files to make it work properly. Just be patient.

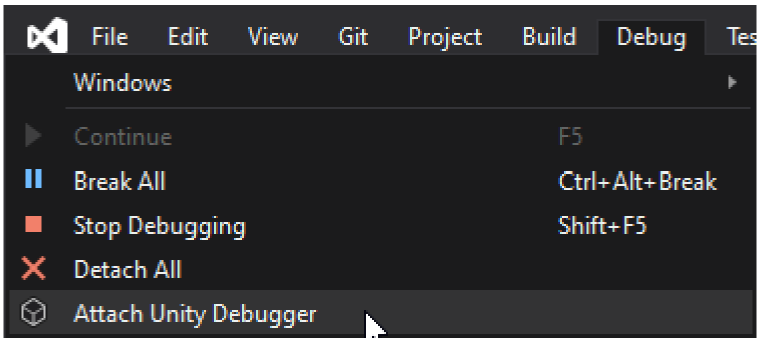
Decompiled classes from KSP2 dll is not as detailled as in VisualStudio and I come back to this primal IDE from time to time to dig into actual KSP2 behaviour.

Building and debugging

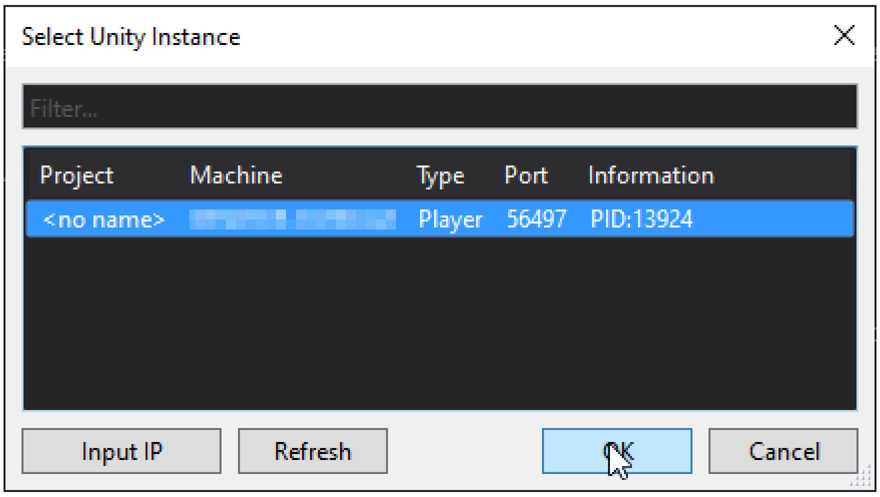
When building your plugin assembly in the `Debug` configuration, a `*.pdb` debug symbols file will be generated alongside your `*.dll` file. You need to copy that file alongside the dll in the GameData subfolder of your plugin.

Debugging with Visual Studio

- In Visual Studio, from the `Debug` menu, select `Attach Unity Debugger` :



- In the popup, select your running KSP instance (it can sometimes take a while for it to appear) :



Project	Machine	Type	Port	Information
<no name>	[redacted]	Player	56497	PID:13924

- Visual Studio will switch to the debugging context and you can now set a breakpoint in your code. It might say "The breakpoint will not currently be hit" initially, but that should change once your class is actually instantiated and the code executed.

Debugging with VSCode

It only work if Intelisense is fully ready to work.

- run the `Attach Unity Debugger` command
- select the KSP2

Troubleshooting

If the running KSP instance isn't listed in the `Select Unity instance` popup :

- Make sure you followed the **Modifying your game for profiling/debugging** instructions correctly.
- The Unity executable communicate with the debugger/profiler over a network interface, so :
  - Make sure both KSP and your IDE aren't firewalled (by the Windows firewall or an antivirus/firewall software if you have one).
  - Uncommon network setups like a VPN or a network bridge will often interfere. If you have multiple network interfaces (ex : ethernet + wifi), keep only one enabled.

As a last resort, you can try inputting the IP/port manually. Launch KSP and open the `Player.log` file (on windows it will be here : `%USERPROFILE%\AppData\LocalLow\Intercept Games\Kerbal Space Program 2\Player.log`).
You need to find the IP adress and port Unity is listening on. They should be on two separate lines near the beginning of the file :

- `Multi-casting "[IP] 192.168.1.81` (truncated) : IP adress to use
- `Starting managed debugger on port 56061` : port to use

Then in the `Select Unity instance` popup, click on the "Input IP" button and try using the above IP/port.

See also the troubleshooting debugging Unity players page from the Rider documentation (this has useful additional information, especially for MacOS).

Profiling with the Unity editor profiler (Not Tested)

The Unity profiler is a frame based profiler that allow to see all calls to the default MonoBehaviour message methods (Update, FixedUpdate, etc), and to profile arbitrary sections of your code. You can measure call count, time taken and see call hierarchy and managed memory usage. See the Profiler Window documentation.

Running the profiler

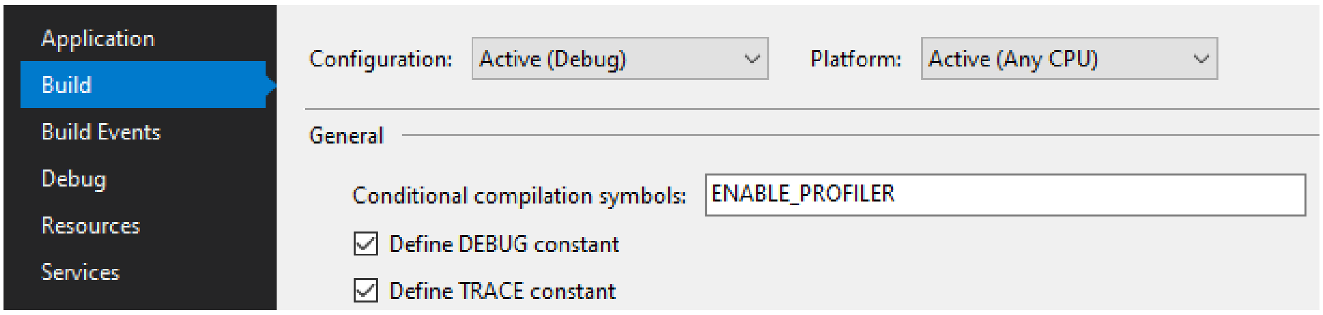
- Launch KSP
- Launch the Unity editor, and create/open a blank project
- From the editor menu, go to Window > Analysis > Profiler
- At the top of the profiler window, you should see a dropdown list where you can connect to you KSP instance. If that doesn't work, read the above **Troubleshooting** section.

Profiling your own code

By default, the profiler will only show calls to the message type MonoBehaviour methods, which isn't very helpful for profiling KSP plugins. To profile specific parts of your code, you need to compile your plugin with the `ENABLE_PROFILER` compilation constant :

Project setup with Visual Studio :

Right click your project in the solution explorer panel, select the `Properties` option.
Make sure the `Debug` configuration is selected, then in the `Build` tab add `ENABLE_PROFILER` in the `Conditional compilation symbols` box. Note that if there is already a value there, you should separate additional values with a semicolon (`;`).



Configuration: Active (Debug)
Platform: Active (Any CPU)
General
Conditional compilation symbols: ENABLE_PROFILER
☑ Define DEBUG constant
☑ Define TRACE constant

Manual project setup

This is an alternative to the above Visual Studio GUI based instructions :

- Open your `*.csproj` project file in a text editor
- Locate the following section :

```
<PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Debug|AnyCPU' ">
```

- Under that section (make sure it's the right one, there is a similar one for the `Release` configuration), append `ENABLE_PROFILER` separated with a semicolon (`;`) to the `DefineConstants` property :

```
<DefineConstants>DEBUG;TRACE;ENABLE_PROFILER</DefineConstants>
```

Adding profile markers to your code

To profile an arbitrary section of your code, enclose that code between a `Profiler.BeginSample("NameShownInTheProfiler")` and `Profiler.EndSample()` call. See the Profiler class documentation.

For this to work reliably, you must ensure that `Profiler.EndSample()` is called before every return point of your method.

The `Profiler` calls are executing only if your plugin is compiled with the `ENABLE_PROFILER` constant, so you can leave them around, they won't do anything and won't cause a performance overhead once you plugin is compiled in the `Release` configuration.

Note that ideally, you want to do performance profiling in the `Release` configuration to get a more accurate picture (debug builds are quite noticeably slower). To achieve that, you can create a third build configuration in your project (based on the `Release` configuration), but with the `ENABLE_PROFILER` constant added.

Note that the `Profiler.*Sample()` methods have a small overhead which is insignifant in usual cases, but if you want to profile "hot paths" (small blocks of code executed many times in a loop for example) or to nest profiler calls, it is preferable to use the ProfilerMarker methods. Also note that contrary to the legacy `Profiler` methods, `ProfilerMarker` allow profiling `Jobs` based code.

Profile analyzer

While the frame profiler is a handy and straightforward tool for fixing obvious performance issues, it doesn't give you a full picture since there will be quite some variance from one frame to another, which can be very misleading when trying to compare results from previous runs.

To get more reliable results, you can use the profile analyzer, which does advanced statistical analysis of profiling sessions, as well as saving and comparing different session results.

The profile analyzer is available as an additional Unity package. To install it, open the empty unity editor project you're using for profiling, go to the `Window > Package manager` menu option, and install it. It should then appear in the `Window > Analysis > Profile Analyzer` menu.

Notes on performance analysis

In a modified for debugging/profiling install, performance will be vastly degraded compared to an untouched KSP :

- There is a lot of extra debugging/error checking code executed on the native side of Unity, so internal Unity methods will execute slower, affecting almost every aspect of the engine (physics, rendering, game logic...).

- The Mono runtime will run with many JIT compiler optimizations disabled. This will induce a very variable and sometimes quite significant performance impact on the C# code from KSP and plugins.
- Remember that building code in the `Debug` configuration will a induce a large and variable performance penalty. To avoid that, use the same build parameters as for `Release`, but with the `ENABLE_PROFILER` constant added.
- Attaching the profiler will cause a large overhead to your frame time.

For these reasons, profiler results should be taken with a grain of salt. The performance difference between profiler results and a regular KSP install isn't uniform : some pieces of code will run hugely slower, other won't be affected much. The profiler is still very useful to get an overall picture, comparing different implementations and identifying bottlenecks as well as GC pressure issues, but it isn't reliable at all for micro-optimization.

To perform entirely reliable performance measurements, it is necessary to implement them manually in your plugin (for example with a `StopWatch`), and to run the plugin compiled in the `Release` configuration in a non-modified KSP install.