Preprocessor Symbol Definition Files

This package provides an easy way to manage custom `#defines` in a Unity project for multiple platforms by providing dedicated files called: Preprocessor Symbol Definition Files, that allow to handle preprocessor symbols in an intuitive way.

Table of Contents

Preprocessor Symbol Definition Files

Preprocessor Symbol Definition Files are dedicated objects that store and manage custom preprocessor symbols.

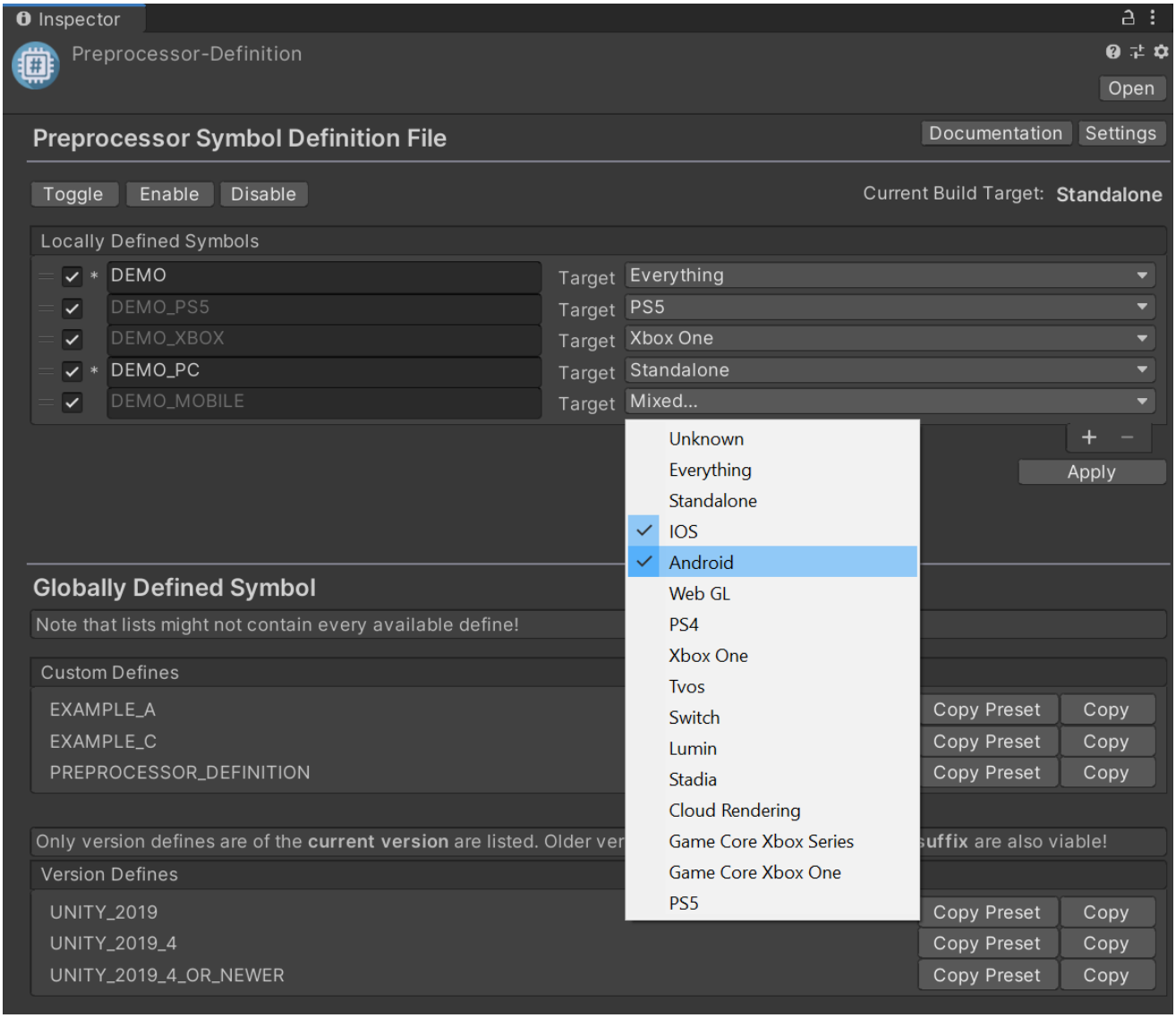
Create Files

Select the target folder then navigate to (menu: `Assets > Create > Preprocessor Definition`) or (rightclick: `Create > Preprocessor Definition`)

Add Symbols

Make a new entry in the Locally Defined Symbols list, then press Apply to confirm your changes. Both adding and removing symbols must be applied manually. Pressing Apply forces Unity to refresh the list of script definition symbols and reload the assemblies. Entries contained in the list can be activated and deactivated manually.

The Build Target Group of each symbol can also be set individually. Symbols are only active if their build target group matches the active build target group of the project. Definition files allow you to select multiple build target groups for a symbol simultaneously.



Preprocessor Definition Settings

The settings file is located as an asset in your project and acts not only as a configuration cache, but also as a tool to manage global values and as a tool to locate and manage every definition file in your project. Select this file manually: navigate to (menu: Tools > Preprocessor-Symbol-Definition-File > Settings). Please ensure that there is only on instance of this type file at any time.

Remove Symbols on Delete

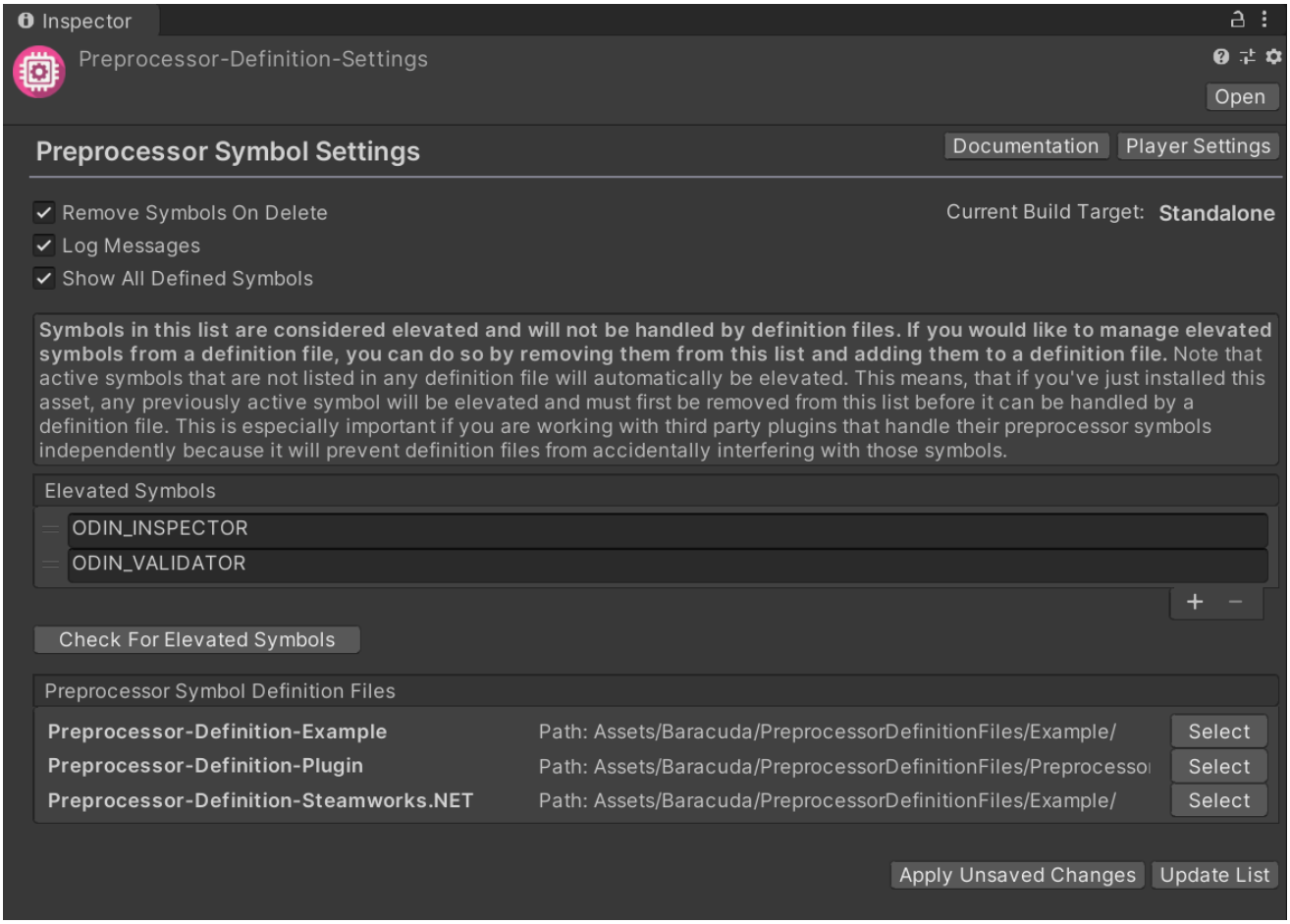
Removes the content of a Preprocessor Symbol Definition File when it is deleted. If this option is not enabled the symbols of a deleted file will be elevated and must be removed manually.

Log Messages

When enabled, messages will be logged when symbols are removed, added or elevated.

Elevated Symbols

Symbols in this list are considered elevated and will not be handled by definition files. If you would like to manage elevated symbols from a definition file, you can do so by removing them from this list and adding them to a definition file. Note that active symbols that are not listed in any definition file will automatically be elevated. This means, that if you've just installed this asset, any previously active symbol will be elevated.



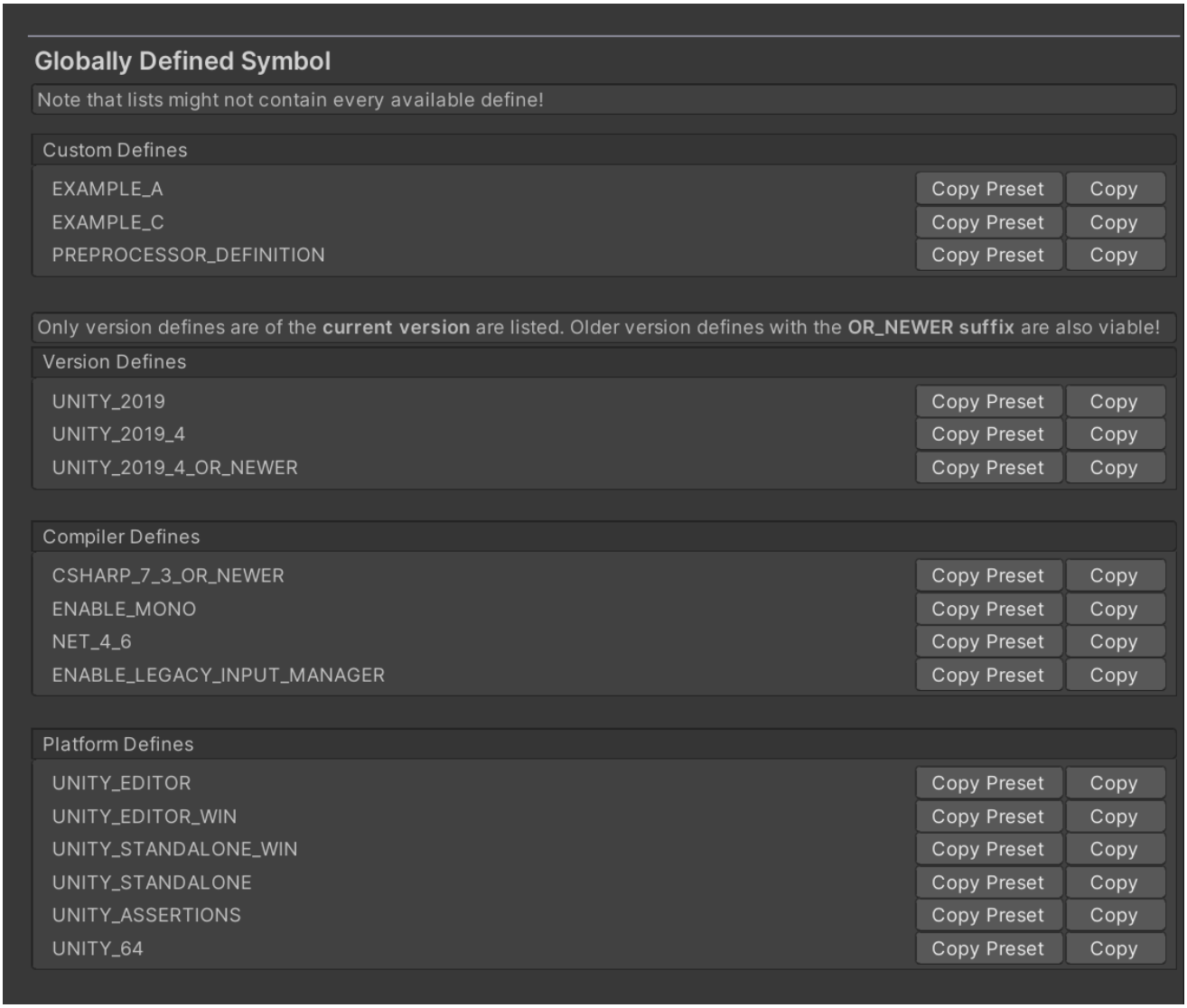
The settings file also displays every definition file, located anywhere in the project and enables each of these files to be selected manually. You can manually check whether the list is complete or if there are some definition files located in the project that are

not in the list by pressing Update List. Pressing Apply Unsaved Changes will check if unsaved changes are present in any of the listed definition files and apply them.

Available Symbols Display

Both, definition files and the settings file can display multiple categories of defined symbols. The contents of those lists, except for the custom defines, might not contain every available symbol. Every symbol defined by Unity is included in these lists. Every entry offers two quick ways to copy its content to the clipboard. Use Copy for the unchanged symbol and Copy Preset to copy the symbol with the following format:

```
#if SYMBOL



#endif!
```



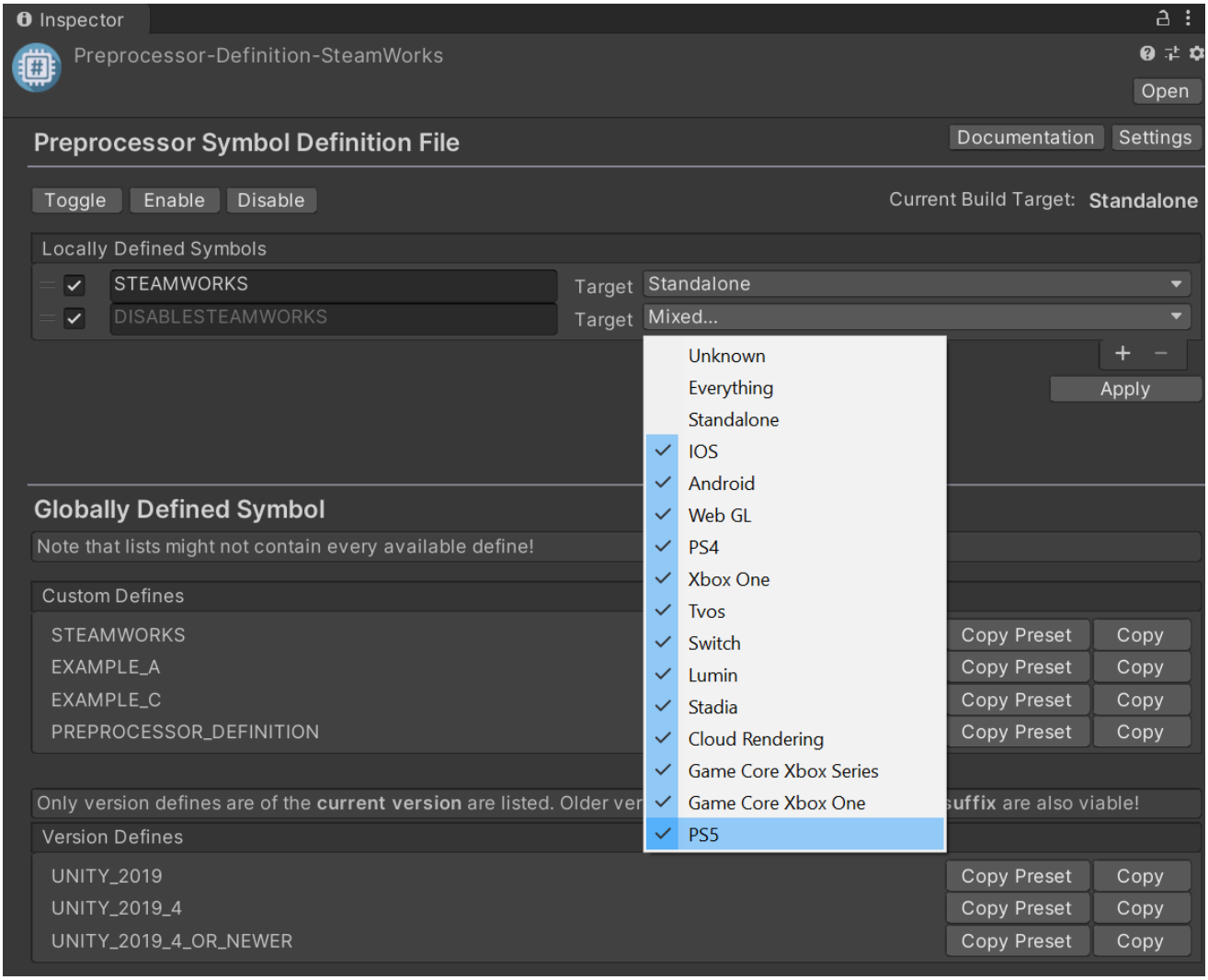
Practical Example Steamworks

Consider the following: we are creating a multi-platform game whilst using Steamworks.NET. If you've ever worked with Steamworks.NET in a multi-platform project, you know that you must manually define: DISABLESTEAMWORKS for the majority of your target platforms to disable it, which must be repeated every time you add a new platform. You also have to wrap your Steamworks API calls in an `#if` compiler directive. Although not the most time-consuming process, it is definitely a very important but sometimes unintelligible process. In general, adding and removing custom symbols for many platforms, especially in large projects with a decent amount of custom symbols can be a very delicate task.

```
using System;

using System.Collections.Generic;

using UnityEngine;



#if !DISABLESTEAMWORKS

using Steamworks;

#endif



public class PlayerManager : MonoBehaviour

{

    private void Awake()

    {

#if !DISABLESTEAMWORKS

        // Add steam logic here...

#endif

    }

}
```

If we use this asset instead, we can handle this scenario much more elegantly, simply by creating a Preprocessor Symbol Definition File and adding a new entry to define DISABLESTEAMWORKS. We then select the platforms on which we want the symbol to be active and press apply. Every time we switch platforms, the definition file will automatically check if either needs to define or un-define the symbol.

> I've defined another symbol for this example: STEAMWORKS is an optional symbol that will be activated when DISABLESTEAMWORKS is disabled. We can then wrap our Steamworks API calls in a `#if STEAMWORKS` block instead of an `#if !DISABLESTEAMWORKS` which will improve the clarity of our code and enable us to individually activate/deactivate our steam code without affecting the internals of the Steamworks API itself.



Support Me

I spend a lot of time working on this and other free assets to make sure as many people as possible can use my tools regardless of their financial status. Any kind of support I get helps me keep doing this, so consider leaving a star ⭐ making a donation or follow me on my socials to support me ❤

- Donation (PayPal.me)

- Linktree
- Twitter
- Itch