FileManagement Class

Cross-platform file IO control without Plug-ins.

eToile

(eToile 2020) V: 2.4

Index

Introduction

Thanks for purchasing `FileManagement`, this class is designed to be simple and lightweight, so you will not need to learn or implement more than its useful interfaces.
This product is a static class, so you can add or remove it from your project without any risks.
You also can access the full source code.
Saving and reading is as fast as the platform allows, because interpretations and parsing are maintained at minimum.
The example scene allows testing most of the `FileManagement` functions.
There is a very important feature in `FileManagement` that merges `PersistentData` and `StreamingAssets` (or embedded resources) as a virtual single drive. This allows a completely safe cross-platform drive access.

Class Description

`FileManagement` is a static class. It means that you don't have to create/instantiate a `FileManagement` object, just write `FileManagement` dot (.) the interface you need.

The `FileManagement` class uses compiler directives to chose the right functions for each platform. So, there are different versions for the next functions: `SaveRawFile()`, `ReadRawFile()` and `DeleteFile()`.

This three functions are platform dependent, so they are used by every other function into this same class.
The three main groups are:

- `UNITY_WINRT`.
- `UNITY_WEBGL`.
- Everything else.

There are not special considerations when exporting to different platforms, nor special considerations when uploading to digital markets.
Just switch platform from "Build settings" dialog on Unity editor and Build.

The `FileManagement` class is designed to be used also on C# environments other than Unity (Mono, Xamarin, Windows Forms), making all your code easier to read and maintain. Your different applications will be coded and will behave in exactly the same way.

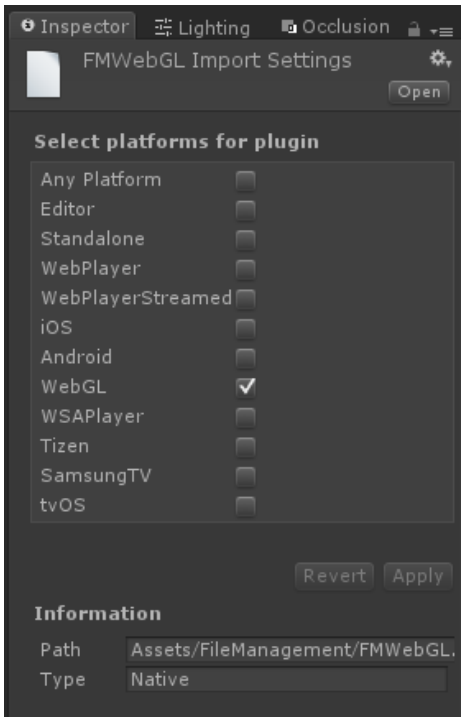
Class Integration

To integrate this class to your Unity project you must include the three main files that are used in this product:

- "FileManagement.cs" (For other platforms than Unity, just include this).
- "FMWebGL.jslib".
- "StreamingAssetsIndexer.cs".

The FileManagement.cs file contains the main `FileManagement` class and the secondary `FM_IniFile` class.
FMWebGL.jslib is a plug-in needed for WebGL exports. This file can be edited with any text editor, it is not a real plug-in, it's some javascript extra functionality needed to save content into browser's data base.

The plug-in importer should look like this:



Note that you don't need to put the plug-in into a "Plugins" folder.

The StreamingAssetsIndexer.cs file runs in Editor only, generating the StreamingAssets index automatically.

PlayerPrefs replacement

`FileManagement` implements the same `PlayerPrefs` functions, so if you are using already `PlayerPrefs` in your project, the only thing you have to do is to replace `PlayerPrefs` with `FileManagement`.

This is the list of equivalent functions:

```
public static void DeleteAll()
public static void DeleteKey(string key)
public static float GetFloat(string key, float defaultValue = 0.0F)
public static int GetInt(string key, int defaultValue = 0)
public static string GetString(string key, string defaultValue = "")
public static bool HasKey(string key)
public static void Save()
public static void SetFloat(string key, float value)
public static void SetInt(string key, int value)
public static void SetString(string key, string value)
```

The **`Save()`** function has no effect in `FileManagement` due to there is no grouping of several values into a single file (If you need this feature, you can use `FM_IniFile` instead). This method exists just to avoid compilation issues when porting an existing application.

This is the list of non standard functions:

```
public static bool GetBool(string key, bool defaultValue = false)
public static double GetDouble(string key, double defaultValue = 0)
public static void SetBool(string key, bool value)
public static void SetDouble(string key, double value)
```

These functions works in the same way that `PlayerPrefs` does, but adding some extra functionality.

Every saved key is always placed relative to the PersistentDataPath. They are saved as separated files, so there is no risk of data loss. All files are saved with the exact provided name, if a sub-directory is provided then it will be created automatically if not exists.

FileManagement Public Methods

This is the complete definition of `FileManagement` public methods.

SaveRawFile()

```
public static void SaveRawFile(string name, byte[] content, bool enc = false,
bool fullPath = false)
```

Use a key, ID, relative or full path to identify your file in **`name`**. The name can include a relative or absolute destination path. If that path doesn't exists it will be created automatically.
The **`enc`** argument enables or disables the encryption functionality.
The **`fullPath`** argument allows to treat the provided **`name`** as absolute path.
NOTE: This function is used by every other function that writes to disk.

This example saves a binary file:

```
byte[] byteArray = {0x10, 0x20, 0x30, 0x40, 0x50};
FileManagement.SaveRawFile("data.bin", byteArray);
```

ReadRawFile()

```
public static byte[] ReadRawFile(string name, bool enc = false, bool checkSA = true,
bool fullPath = false)
```

Use a key, ID, relative or full path to identify your file in **`name`**. It returns empty content if the file do not exists.
The **`enc`** argument enables or disables the encryption functionality.

The **checkSA** argument allows `FileManagement` to search the "StreamingAssets" folder if the requested file is not found. This functionality is disabled with **fullPath** flag.
The **fullPath** argument allows to treat the provided **name** as absolute path.
This example reads an encrypted text file (the content value is decrypted with the internal method):

```
byte[] content = FileManagement.ReadRawFile ("data.txt", true);
```

DeleteFile()

```
public static void DeleteFile(string name, bool fullPath = false)
```

Deletes the **name** file. It fails if the access is denied using **fullPath**.
`FileManagement` will check if the file **name** exists before attempting to delete it, if not exists it don't throws any exception, it prints a message to the console and returns.
The **fullPath** argument allows to treat the provided **name** as absolute path.
This example deletes the file:

```
FileManagement.DeleteFile("data.txt");
```

ImportAudio() [Unity only]

```
public static AudioClip ImportAudio(string file, bool enc = false, bool checkSA = true,
bool fullPath = false, bool stream = false)
```

Imports an `AudioClip` from file. The wav format is the only widely supported by all platforms, for other formats check the Unity documentation (Or try the `FileManagement` test application in your desired platform and check if it works).

The file name without the extension is applied automatically to the returned `AudioClip` name.

IMPORTANT: This function uses `OpenWavParser` for PCM WAV files. This is the only supported format on all platforms, and the only supported format since Unity 2019.

The **checkSA** argument allows `FileManagement` to search the "StreamingAssets" folder if the requested file is not found. This functionality is disabled with **fullPath** flag.
The **fullPath** argument allows to treat the provided **name** as absolute path.

SaveAudio() [Unity only]

```
public static void SaveAudio(string name, AudioClip clip, bool enc = false,
bool fullPath = false, OpenWavParser.Resolution res = OpenWavParser.Resolution._16bit)
```

Saves an AudioClip to file always encoded as PCM WAV (16/24/32bit, mono/stereo, 22K, 44K, etc.).
The `OpenWavParser` class is used by default on all platforms.

The **fullPath** argument allows to treat the provided **name** as absolute path.

ImportTexture() [Unity only]

```
public static Texture2D ImportTexture(string file, bool enc = false, bool checkSA = true,
bool fullPath = false)
```

Imports a JPG or PNG image file from disk into a Unity `Texture2D`.
The **enc** argument enables or disables the encryption functionality.
The **checkSA** argument allows `FileManagement` to search the "StreamingAssets" folder if the requested file is not found. This functionality is disabled with **fullPath** flag.
The **fullPath** argument allows to treat the provided **name** as absolute path.
This example imports an image file into a texture from the StreamingAssets folder:

```
Texture2D texture = FileManagement.ImportTexture("image.jpg");
```

ImportSprite() [Unity only]

```
public static Sprite ImportSprite(string file, bool enc = false, bool checkSA = true,
bool fullPath = false, Vector2 pivot = new Vector2())
```

Imports a JPG or PNG image file from disk into a Unity sprite.
The **enc** argument enables or disables the encryption functionality.
The **checkSA** argument allows `FileManagement` to search the "StreamingAssets" folder if the requested file is not found. This functionality is disabled with **fullPath** flag.
The **fullPath** argument allows to treat the provided **name** as absolute path.
This example imports an image file into a sprite from the StreamingAssets folder:

```
Sprite sprite = FileManagement.ImportSprite("image.jpg");
```

SaveJpgTexture() [Unity only]

```
public static void SaveJpgTexture(string name, Texture texture, int quality = 75,
bool enc = false, bool fullPath = false)
```

Saves a `Texture` or `Texture2D` into a JPG encoded file. The **quality** parameter determines the file compression, the default file compression is 75.
The **enc** argument enables or disables the encryption functionality.
The **fullPath** argument allows to treat the provided **name** as absolute path.
This examples saves textures into files:

```
FileManagement.SaveJpgTexture("texture.jpg", renderer.material.mainTexture);
FileManagement.SaveJpgTexture("texture.jpg", gameObject.GetComponent<Sprite>().texture);
```

SavePngTexture() [Unity only]

```
public static void SavePngTexture(string name, Texture texture, bool enc = false,
bool fullPath = false)
```

Saves a `Texture` or `Texture2D` into a PNG encoded file.
The **enc** argument enables or disables the encryption functionality.
The **fullPath** argument allows to treat the provided **name** as absolute path.
This examples saves textures into files:

```
FileManagement.SavePngTexture("texture.png", renderer.material.mainTexture);
FileManagement.SavePngTexture("texture.png", sprite.texture);
```

FileExists()

```
public static bool FileExists(string name, bool checkSA = false, bool fullPath = false)
```

Returns `true` if the file exists. The **name** can contain a full or relative path.
The **checkSA** argument allows `FileManagement` to search the "StreamingAssets" folder if the requested file is not found. This functionality is disabled with **fullPath** flag.
The **fullPath** argument allows to treat the provided **name** as absolute path.
This example checks if a file exists:

```
if (FileManagement.FileExists("data.txt"))
        Debug.Log("[FileManagement] This file exists.");
```

SaveFile()

```
public static void SaveFile<T>(string name, T content, bool enc = false,
bool fullPath = false)
```

Saves any C# type of variable, and the following **UnityEngine** types: `Vector2`, `Vector3`, `Vector4`, `Quaternion`, `Rect`, `Color` & `Color32`.
The **enc** argument enables or disables the encryption functionality.
The **fullPath** argument allows to treat the provided **name** as absolute path.
This example saves two types of data:

```
FileManagement.SaveFile("IntData", 12);
FileManagement.SaveFile("StringData", "Example data", true);  // Encrypt
```

ReadFile()

```
public static T ReadFile<T>(string name, bool enc = false, bool checkSA = false,
bool fullPath = false)
```

Reads any C# type of variable, and reads the following **UnityEngine** types: Vector2, Vector3, Vector4, Quaternion, Rect, Color & Color32.
The **enc** argument enables or disables the encryption functionality.
The **checkSA** argument allows FileManagement to search the "StreamingAssets" folder if the requested file is not found. This functionality is disabled with **fullPath** flag.
The **fullPath** argument allows to treat the provided **name** as absolute path.

This example reads two different types of file:

```
int a = FileManagement.ReadFile<int>("IntData");
string a = FileManagement.ReadFile<string>("StringData", true);      // Decrypt
```

SaveArray()

```
public static void SaveArray<T>(string name, T[] content, char separator = (char)0x09,
bool enc = false, bool fullPath = false)
public static void SaveArray<T>(string name, System.Collections.Generic.List<T> content,
char separator = (char)0x09, bool enc = false, bool fullPath = false)
```

Saves any one dimension Array or List of any **ReadFile** supported type. You can specify a custom separator as a **char** argument, default separator is the horizontal tabulator.
The **enc** argument enables or disables the encryption functionality.
The **fullPath** argument allows to treat the provided **name** as absolute path.

This example saves an array of strings separated by a semicolon:

```
string[] myArray = {"one", "two", "three"};
FileManagement.SaveArray("MyArray.txt", myArray, ';');
```

ReadArray()

```
public static T[] ReadArray<T>(string name, char separator = (char)0x09, bool enc = false,
bool checkSA = true, bool fullPath = false)
public static T[] ReadArray<T>(string name, string[] separator, bool enc = false,
bool checkSA = true, bool fullPath = false)
```

Reads any one dimension Array of any **ReadFile** supported type.
You can specify a custom separator as a **char** argument (default is horizontal tabulator) or as an array of **string** values (no default).
The **enc** argument enables or disables the encryption functionality.
The **checkSA** argument allows FileManagement to search the "StreamingAssets" folder if the requested file is not found. This functionality is disabled with **fullPath** flag.
The **fullPath** argument allows to treat the provided **name** as absolute path.

This example reads an array of strings separated by a semicolon:

```
string[] myArray = FileManagement.ReadArray<string>("MyArray.txt", ';');
```

ReadList()

```
public static System.Collections.Generic.List<T> ReadList<T>(string name,
char separator = (char)0x09, bool enc = false, bool checkSA = true, bool fullPath = false)
public static System.Collections.Generic.List<T> ReadList<T>(string name,
string[] separator, bool enc = false, bool checkSA = true, bool fullPath = false)
```

Reads any one dimension List of any **ReadFile** supported type.
You can specify a custom separator as a **char** argument (default is horizontal tabulator) or as an array of **string** values (no default).
The **enc** argument enables or disables the encryption functionality.
The **checkSA** argument allows FileManagement to search the "StreamingAssets" folder if the requested file is not found. This functionality is disabled with **fullPath** flag.

The **fullPath** argument allows to treat the provided **name** as absolute path.

This example reads an array of strings separated by a semicolon:

```
List<string> myArray = FileManagement.ReadList<string>("MyArray.txt", ';');
```

ReadAllLines()

```
public static string[] ReadAllLines(string name, bool enc = false, bool checkSA = true,
bool fullPath = false)
```

Reads all the lines from a text file. It uses `"\r\n"`, `"\n"` and `"\r"` (NewLine) literals for line splitting.
The **enc** argument enables or disables the encryption functionality.
The **checkSA** argument allows `FileManagement` to search the "StreamingAssets" folder if the requested file is not found. This functionality is disabled with **fullPath** flag.
The **fullPath** argument allows to treat the provided **name** as absolute path.

This example reads a file into a string array from StreamingAssets folder:

```
string[] textInLines = FileManagement.ReadAllLines("text.txt");
```

ImportIniFile()

```
public static FM_IniFile ImportIniFile(string name, bool enc = false, bool checkSA = true,
bool fullPath = false)
```

Imports a INI file from disk into a `FM_IniFile` object.
The **enc** argument enables or disables the encryption functionality.
The **checkSA** argument allows `FileManagement` to search the "StreamingAssets" folder if the requested file is not found. This functionality is disabled with **fullPath** flag.
The **fullPath** argument allows to treat the provided **name** as absolute path.

This example imports some INI file into a variable:

```
FM_IniFile iniFile = FileManagement.ImportIniFile("Cfg.ini");
```

ImportCsvFile()

```
public static string[][] ImportCsvFile(string name, string[] separator, bool enc = false,
bool checkSA = true, bool fullPath = false)
```

Imports a CSV file from disk into a two-dimensional string array. The first dimension length matches the registers count in the CSV file, and the secondary dimension matches the fields count in a register.
The **enc** argument enables or disables the encryption functionality.
The **checkSA** argument allows `FileManagement` to search the "StreamingAssets" folder if the requested file is not found. This functionality is disabled with **fullPath** flag.
The **fullPath** argument allows to treat the provided **name** as absolute path.

This example imports some CSV file into a variable:

```
FM_string[][] csvFile = FileManagement.ImportCsvFile("Data.csv");
```

SaveCsvFile()

```
public static void SaveCsvFile(string name, string[][] content, char separator = (char)0x09,
bool enc = false, bool checkSA = true, bool fullPath = false)
```

Saves a two-dimensional string array formatted as a csv file. The **separator** parameter is used to "separate" fields. The registers are saved as text lines (separated by NewLine code).
The **enc** argument enables or disables the encryption functionality.
The **checkSA** argument allows `FileManagement` to search the "StreamingAssets" folder if the requested file is not found. This functionality is disabled with **fullPath** flag.
The **fullPath** argument allows to treat the provided **name** as absolute path.

This example imports some CSV file into a variable:

```
FileManagement.SaveCsvFile("Data-Copy.csv", csvFile);
```

AddLine()

```
public static void AddLine(string name, string content, bool enc = false,
bool fullPath = false)
```

This adds a single line of text to an already existing file. If the file not exists, it will be created.
The **enc** argument enables or disables the encryption functionality.
The **fullPath** argument allows to treat the provided **name** as absolute path.

This example adds a text line incrementally:

```
FileManagement.AddLine("MyFile.txt", "This is a new line.");
```

AddLogLine()

```
public static void AddLogLine(string name, string content, bool enc = false,
bool fullPath = false)
```

This adds a single line of text to an existing file, adding also a header with date and time with the format: `"dd/MM/yyyy-hh:mm:ss;"`. If the file not exists, it will be created. This method will lock the file access allowing to be safely used on several threads simultaneously.
The **enc** argument enables or disables the encryption functionality.
The **fullPath** argument allows to treat the provided **name** as absolute path.
IMPORTANT: Try to avoid this method in Windows Store platform, or at least maintain the entries at minimum (It's not well optimized for very big files).

This example adds an error log incrementally:

```
FileManagement.AddLogLine("Register.log", "Error: File not found.");
```

AddRawData()

```
public static void AddRawData(string name, byte[] content, bool enc = false,
bool fullPath = false)
```

This adds a chunk of `byte` data to an existing file. If the file not exists, it will be created.
The **enc** argument enables or disables the encryption functionality.
The **fullPath** argument allows to treat the provided **name** as absolute path.

This examples appends a byte array to an existing file:

```
byte[] byteArray = {0x10, 0x20, 0x30, 0x40, 0x50};
FileManagement.AddRawData("data.bin", byteArray);
```

DirectoryExists()

```
public static bool DirectoryExists(string name, bool checkSA = true, bool fullPath = false)
```

Checks the existence of a directory.
The **checkSA** argument allows `FileManagement` to search the "StreamingAssets" folder if the requested file is not found. This functionality is disabled with **fullPath** flag.
The **fullPath** argument allows to treat the provided **name** as absolute path.

This example checks if a directory exists:

```
if (FileManagement.DirectoryExists ("Test1"))
	Debug.Log("[FileManagement] The Test1 folder exists.");
```

CreateDirectory()

```
public static void CreateDirectory(string name, bool fullPath = false)
```

Creates a new directory. Fails if the access is denied using **fullPath**.
The **fullPath** argument allows to treat the provided **name** as absolute path.

This example creates a new directory:

```
FileManagement.CreateDirectory("Test1");
```

DeleteDirectory()

```
public static void DeleteDirectory(string name, bool fullPath = false)
```

Deletes a directory and its content including sub-directories. Fails if the access is denied using

`fullPath`.
The `fullPath` argument allows to treat the provided `name` as absolute path.
This example deletes an existing directory:
```
FileManagement.DeleteDirectory("Test1");
```

EmptyDirectory()
```
public static void EmptyDirectory(string name = "", bool filesOnly = true,
bool fullPath = false)
```
Deletes the directory content. By default deletes only files. Fails if the access is denied using `fullPath`.
The `fullPath` argument allows to treat the provided `name` as absolute path.
This example deletes the whole content of a folder (including sub-directories):
```
FileManagement.EmptyDirectory("Test1", false);
```

ListFiles()
```
public static string[] ListFiles(string name, bool checkSA = true, bool fullPath = false)
public static string[] ListFiles(string name, string[] filter, bool checkSA = true,
bool fullPath = false, bool subdirectories = false)
```
Returns all of the file names contained in the requested folder. Fails if the access is denied while using `fullPath`.
The file list can be filtered using the `filter` array, make sure to include the dot (.) in the extension names like this:
```
string[] filter = {".wav", ".mp3", ".ogg"};
```
The `checkSA` argument allows `FileManagement` to search the "StreamingAssets" folder if the requested file is not found. This functionality is disabled with `fullPath` flag.
The `fullPath` argument allows to treat the provided `name` as absolute path.
The `subdirectories` argument request a deep search into all subfolders.
This example requests the folder content:
```
string[] fileNames = FileManagement.ListFiles("Test1");
```

ListDirectories()
```
public static string[] ListDirectories(string name, bool checkSA = true,
bool fullPath = false)
```
Returns all of the folder names contained in the requested folder. Fails if the access is denied while using `fullPath`.
The `checkSA` argument allows `FileManagement` to search the "StreamingAssets" folder if the requested file is not found. This functionality is disabled with `fullPath` flag.
The `fullPath` argument allows to treat the provided `name` as absolute path.
This example requests the folder content:
```
string[] folderNames = FileManagement.ListDirectories("Test1");
```

ReadDirectoryContent()
```
public static System.Collections.Generic.List<byte[]> ReadDirectoryContent(string name,
bool enc = false, bool checkSA = true, bool fullPath = false)
```
Returns all of the files contained in the requested folder as a `List` of `byte` arrays. The files are added to the `List` in the same order that are listed with the `ListFiles` function.
Fails if the access is denied using `fullPath`.
The `enc` argument enables or disables the encryption functionality.
The `checkSA` argument allows `FileManagement` to search the "StreamingAssets" folder if the requested file is not found. This functionality is disabled with `fullPath` flag.
The `fullPath` argument allows to treat the provided `name` as absolute path.
This example requests the folder content, it also gets the names of the files in the same order:

```
List<byte[]> files = FileManagement.ReadDirectoryContent("Test1");
string[] fileNames = FileManagement.ListFiles("Test1");
```

CopyFile()

```
public static void CopyFile(string source, string dest, bool checkSA = true,
bool fullPathSource = false, bool fullPathDest = false)
```

Copies a file from **source** to **dest**. The destination path can include a new name for the copied file.
The **checkSA** parameter only affects the **source** path. The **checkSA** argument allows `FileManagement` to search the "StreamingAssets" folder if the requested file is not found. This functionality is disabled with **fullPathSource** flag.
The **fullPathSource**/**fullPathDest** arguments allows to treat the provided **source**/**dest** names as absolute paths independently.
If the **dest** path doesn't exists, It will be created automatically.

This example copies a file:

```
FileManagement.CopyFile("data.txt", "NewFolder/dataCopy.txt");
```

CopyDirectory()

```
public static void CopyDirectory(string source, string dest, bool checkSA = true,
bool fullPathSource = false, bool fullPathDest = false)
```

Copies a folder from **source** to **dest**. The destination path can include a new name for the copied folder. The folder content is copied too, including its sub-directory content.
The **checkSA** parameter only affects the **source** path. The **checkSA** argument allows `FileManagement` to search the "StreamingAssets" folder if the requested file is not found. This functionality is disabled with **fullPathSource** flag.
The **fullPathSource**/**fullPathDest** arguments allows to treat the provided **source**/**dest** names as absolute paths independently.
If the **dest** path doesn't exists, It will be created automatically.

This example copies a directory:

```
FileManagement.CopyDirectory("Test1", "NewFolder/Test1");
```

Move()

```
public static void Move(string source, string dest, bool fullPathSource = false,
bool fullPathDest = false)
```

Moves a file or folder from **source** to **dest**. The folders are moved with all its content. If the destination folder already exists, it will combine/replace the existing file without prompting the user.
The **fullPathSource**/**fullPathDest** arguments allows to treat the provided **source**/**dest** names as absolute paths independently.
If the **dest** path doesn't exists, It will be created automatically.

This example moves a folder and all its content:

```
FileManagement.Move("NewFolder/Test1", "NewFolder2/Test2");
```

Rename()

```
public static void Rename(string source, string dest, bool fullPathSource = false,
bool fullPathDest = false)
```

This function is the same as **Move()**, you can use it to rename files and folders. It exists just to let you know that you can rename files/folders using **Move()**.

This example moves a folder and all its content:

```
FileManagement.Move("NewFolder/Test2", "NewFolder2/Test2");
```

GetParentDirectory()

```
public static string GetParentDirectory(string path = "")
```

Returns a valid path but removing the file or folder.

This example request a parent folder:

```
string parentDir = FileManagement.GetParentDirectory("Test1/Test2");
```

Now parentDir value is "Test1".

Combine()

```
public static string Combine(string path1, string path2)
```

Returns a valid path combining correctly both paths.

This example combines a path and a file:

```
string path = FileManagement.Combine("Test1\\Test2", "icon2.png");
```

Now path's value is "Test1/Test2/icon2.png".

NormalizePath()

```
public static string NormalizePath(string path)
```

Returns a valid path, correcting possible errors in the string. The `FileManagement` class uses slashes, deletes the ending slash in a path, replaces double slashes, deletes dots and adds slashes at the end automatically if needed.

This example returns a valid directory path:

```
string path = FileManagement.NormalizePath ("Test1\\Test2\\");  // path = "Test1/Test2"
path = FileManagement.NormalizePath ("C:");  // path = "C:/"
```

GetFileName()

```
public static string GetFileName(string path)
```

Returns the last name of the path. It can be a file or a folder.

This example returns the file name only:

```
string name = FileManagement. GetFileName ("Test1\\Test2\\data5.txt");
```

Now name's value is "data5.txt".

GetFileNameWithoutExtension()

```
public static string GetFileNameWithoutExtension(string path)
```

Returns the file name of the path (if any) excluding the last extension. If there is no extension it will return the same file name. If there are more than one extension, it will remove only the last one.

This example returns the filtered file name:

```
string name = FileManagement.GetFileNameWithoutExtension("Test2\\icon2.example.png");
```

Now name's value is "icon2.example".

GetFileExtension()

```
public static string GetFileExtension(string path)
```

Returns the file extension of the path (if any). If there is no extension it will return an empty string. If there are several extensions, it will return the last one only.

This example returns the file extension:

```
string extension = FileManagement. GetExtension ("Test1\\Test2\\icon2.png");
```

Now it's value is ".png".

CustomParser()

```
public static T CustomParser<T>(string content, char separator = (char)0x09)
```

This method is responsible of converting the text data back into the requested data type.
It can convert from `string` to: Every C# type, `Vector2`, `Vector3`, `Vector4`, `Quaternion`, `Rect`, `Color`, `Color32`, `FM_IniFile` & arrays of all supported types (in this case the content must be "separated", it

takes the horizontal tabulator by default).
If a `null` or empty string (`""`) is provided, then the default value for the requested type will be returned or an empty array (except for `string`, it returns always an empty string (`""`) instead of `null`).
If a `byte[]` is requested, then `StringToByteArray` method will be called.

This example parses a string into a float:
```
float val = FileManagement.CustomParser<float>("3.5");
```
This example parses a string into an array of ints:
```
string data = "1;3;5;10;2";
int[] array = FileManagement.CustomParser<int[]>(data, ';');
```

Encrypt()
```
public static byte[] Encrypt(byte[] data, byte[] key)
```
This is the method that makes the encryption of a file content. It only works with byte arrays.
You can use this function to encrypt any data in a byte array.

Decrypt()
```
public static byte[] Decrypt(byte[] data, byte[] key)
```
This is the method that makes the decryption of a file content. It only works with byte arrays.
You can use this function to decrypt any data in a byte array.

ByteArrayToString()
```
public static string ByteArrayToString(byte[] content, byte[] key)
```
This method converts byte arrays into strings depending on the selected conversion mode: `FM_StringMode` stringConversion.

StringToByteArray()
```
public static string StringToByteArray(string content)
```
This method converts strings into byte arrays depending on the selected conversion mode: `FM_StringMode` stringConversion.

FileManagement Private Interfaces

This is the complete definition of `FileManagement` private methods.

CheckNameOnIndex()
```
private static bool CheckNameOnIndex(string name, string type)
```
Checks the name into automatically generated index file. Emulates the `FileExists()` and `DirectoryExists()` methods for the StreamingAssets folder on Android and WebGL builds. There are two slightly different versions of this method for both platforms.

GetNamesOnIndex()
```
private static string[] GetNamesOnIndex(string name, string type)
```
Gets the names from the automatically generated index file. Emulates the `ListFiles()` and `ListDirectories()` methods for the StreamingAssets folder on Android and WebGL builds. There are two slightly different versions of this method for both platforms.

FilterPathNames()
```
private static string[] FilterPathNames(string[] names)
```
This method does the inverse as `GetParentDirectory` because returns the file or folder name removing the parent.

SortPathNames()

```
private static string[] FilterPathNames(string[] names)
```

This method sorts the names by alphabet.
Its use is internal, so it's private.

XorEncryptDecrypt()

```
private static byte[] XorEncryptDecrypt(byte[] data, byte[] key)
```

Performs XOR encryption and decryption.

Experimental methods

The experimental methods are not 100% supported by all platforms. `FileManagement` will print in console if the selected platform lacks the requested functionality.

AesEncrypt()

```
private static byte[] AesEncrypt(byte[] data, byte[] key)
```

Performs AES encryption. Not supported on Windows Store.

AesDecrypt()

```
private static byte[] AesDecrypt(byte[] data, byte[] key)
```

Performs AES decryption. Not supported on Windows Store.

RunCmd()

```
public static void RunCmd(string pathOrCmd)
```

Executes a system command, detecting the provided parameters automatically. This functions works in standalone builds only (Windows, Linux, OSX).

ObjectToByteArray()

```
private static byte[] ObjectToByteArray(object obj)
```

Serializes an object into a byte array allowing to be saved to disk with `SaveRawFile`.

IMPORTANT: Not every variable/class type can be serialized directly (Specially Unity classes). That's the reason why it is not the default save method. Serialized object can't be restored across all platforms or operative systems in most cases, that's why `FileManagement` uses `string` instead.

This method is not supported in Windows Store builds.

ByteArrayToObject()

```
public static object ByteArrayToObject(byte[] arrBytes)
```

Deserializes a byte array into an object allowing to be loaded from disk with `ReadRawFile`.

IMPORTANT: Not every variable/class type can be serialized directly (Specially Unity classes). That's the reason why it is not the default save method. Serialized object can't be restored across all platforms or operative systems in most cases, that's why `FileManagement` uses `string` instead.

This method is not supported in Windows Store builds.

ListLogicalDrives()

```
public static string[] ListLogicalDrives()
```

Get the list of the available logical drives. This method is supported on Windows Standalone only.

InstallLocalApk()

```
public static void InstallLocalApk(string file, bool fullPath = false)
```

Runs the installation of an APK file stored locally in the device.

The installation runs outside the application using the standard Android installer, so the application can update itself. This function is supported on Android builds only.

It only works on devices with Android 8.1 or older.

IsRunningOnMono()

```
public static bool IsRunningOnMono()
```

Simple way to detect if the application is running under Mono runtime (Unity does). Mono and .net can only be differentiated in run time.

GetAppName()

```
public static string GetAppName()
```

Returns the name of the application under Mono and .net. Access this method for the first time from any `MonoBehaviour` method (Awake(), Start() or Setup()) then from any running thread safely.

AES Encryption

```
#define USE_AES
```

Define or comment this directive to allow AES Encryption.
The AES algorithm uses a fixed length key, it doesn't affects the XOR algorithm due to that algorithm uses any key length.
IMPORTANT: Don't forget to set your own new key!
AES is not supported on Windows Phone platforms.

StreamingAssets indexer

The StreamingAssetsIndexer.cs script works automatically while you are working in the Unity editor and generates a "file+subdirectory" index that allows to navigate the StreamingAssets folder once the application was built for Android and WebGL.

The index file is "FMSA_Index" and is saved automatically into your StreamingAssets folder to be included within your final build.
The StreamingAssetsIndexer.cs script detects modifications in the file system and regenerates always the index to ensure that every existing file or sub-directory can be listed correctly.

The StreamingAssets folder is inaccessible on Android and WebGL builds due to it is not a real folder, that is the main reason that this Index was implemented. The Index interpretation is fully integrated within the `FileManagement` class.
In Android, StreamingAssets is a compressed folder, so accessing it implies file extraction, which is slower than real files.

Do not modify the StreamingAssets folder in your final build, or the index will not match the content resulting in access errors. Do it always from Unity editor, then build.

NOTE: The StreamingAssetsIndexer.cs script is not included in your final build.

Please note that the StreamingAssets folder is read only in some platforms, but you have write access in some others. To do so you have to use `Application.streamingAssetsPath` in `fullPath` mode.

Embedded resources

If you are using `FileManagement` to develop applications for Xamarin, Mono, Windows Forms, etc. (other than Unity), the StreamingAssets folder will be not available, but you can attach files to your final build using "Embedded resources".

The embedded resources are read only since they are, actually, embedded in the main binary. You can navigate the embedded resources and read them just as if you were navigating the StreamingAssets folder in Unity (also considered read-only by `FileManagement`).

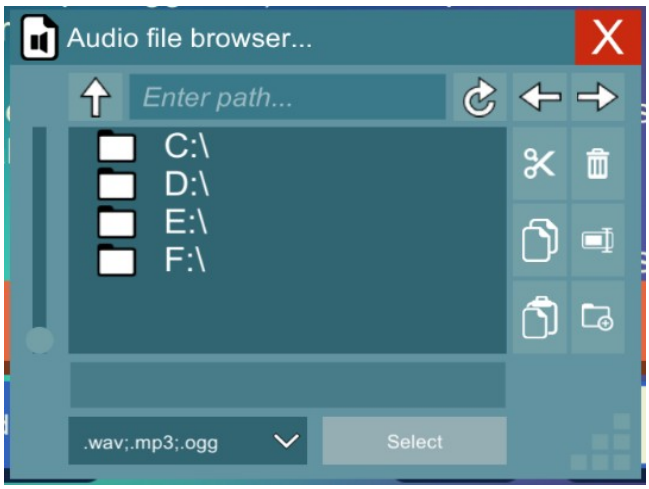
As you may have guessed, `FileManagement` merges the embedded resources with the PersistentData path in the same way as it does in Unity.

This document describes how to embed resources in a Visual Studio project, but the process is similar in most platforms.

FileBrowser

There is a prefab included within this package that allows you to browse the file system and to select a file/folder. This `FileBrowser` is prepared to work also under full 3D environments, making it perfect for VR/AR applications too.

The `FileBrowser` implements some useful functionality that allows it to be used without strong programming skills:



It has built in Back, Fwd, Cut, Copy, Paste, Delete, Rename and New folder buttons.
(The Copy/Cut feature memory is shared by all FileBrowser instances)

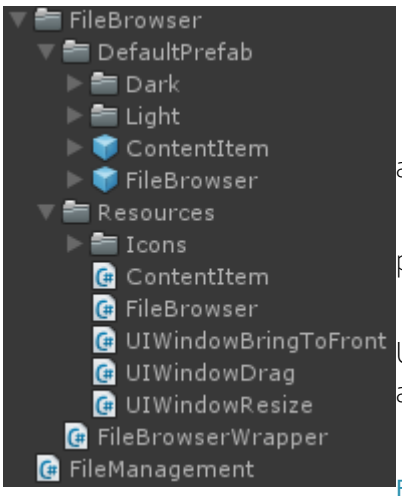
Don't forget to add an EventSystem to your scene or the Unity UI will not be able to receive click/tap events properly (This is done automatically in newer unity versions). You can add it from the GameObject menu: GameObject > UI > EventSystem.

NOTE: If you are using VR/AR make sure to set your particular event system properly for standard Unity UI or you may experiment strange behaviors.

The `FileBrowser` is completely developed in Unity UI, so you can customize it completely at your will using the Editor tools. There are two alternative color schemes for the `FileBrowser` (dark and light) in separated folders.

All image resources used by the `FileBrowser` prefab are contained in the "FileBrowser / Resources / Icons" folder.

This is what you need for the `FileBrowser`:



The `FileBrowser` works always along with `FileManagement`.

The "Icons" folder contains the main image resources.

The ContentItem prefab is used by the `FileBrowser` to list files and folders. Don't forget this when customizing a `FileBrowser`.

The "Dark" and "Light" folders contains alternative `FileBrowser` prefabs with alternative color schemes.

The UIWindowBringToFront, UIWindowDrag and UIWindowResize scripts controls the behavior of the drag, resize and sort order.

The `FileBrowserWrapper` helps to ease the use of the `FileBrowser`.

The `FileBrowser` doesn't open the file, it returns the path to be treated in your application. The path is returned through the event which is fired when the `FileBrowser` closes.

There are two ways to integrate the `FileBrowser`:

1. Through the `FileBrowserWrapper` script, that allows to set its parameters from Editor.
2. Programmatically, that gives you more control over its behavior.

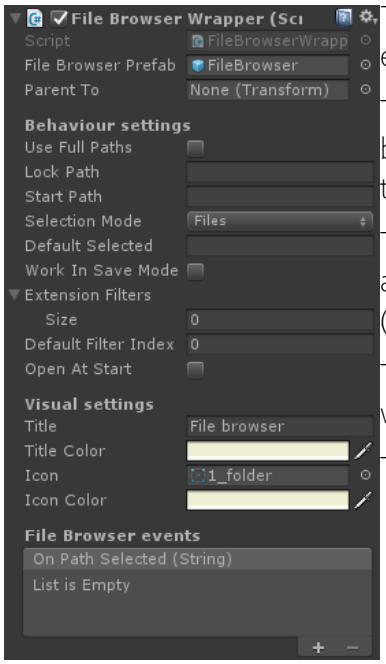
FileBrowserWrapper.cs

The `FileBrowserWrapper` can be attached to any `GameObject` simplifying the process of setting and displaying the `FileBrowser`.

The `FileBrowserWrapper` exposes all available settings, and three methods:

- `Open()`: Creates a FileBrowser and applies the settings automatically.
- `ForceClose()`: Closes the FileBrowser without firing the event.
- `GetInstance()`: Returns the `FileBrowser`'s `GameObject` (if exists) to access it programmatically (refer to the advanced section FileBrowser.cs).

The first two methods can be easily attached to `Button` events.



The available options are self explanatory (You can get some more extended information from the FileBrowser.cs script documentation).

The `_fileBrowserPrefab` parameter contains the `FileBrowser` that will be displayed. If you customize your own copy of `FileBrowser`, you have to set it here in Editor.

The `_parentTo` parameter allows to set the newly created `FileBrowser` as a child of this `Transform`. No action is executed if this is left as None (Transform).

The `_openAtStart` parameter opens the file browser automatically when the application starts (In the `MonoBehaviour`'s `Start()` event).

The OnPathSelected event allows custom methods like this one:

```
void OnPathSelected(string path)
{
	// path contains the user selection.
}
```

FileBrowser.cs

The way you create a `FileBrowser` is as follows:

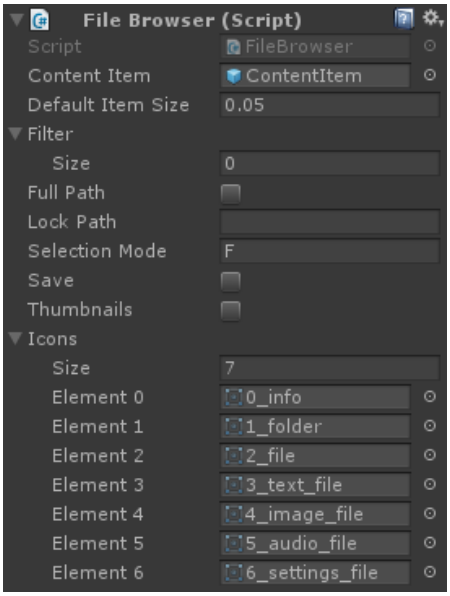
```
public GameObject fileBrowser;  // Drag the FileBrowser prefab here (in the editor).

// Create a FileBrowser window (with default options):
public void OpenFileBrowser()
{
    GameObject browserInstance = GameObject.Instantiate(fileBrowser);
    browserInstance.GetComponent<FileBrowser>().SetBrowserWindow(OnPathSelected);
    // Add file extension filters (optional):
    string[] filter = { ".wav", ".mp3", ".ogg" };
    browserInstance.GetComponent<FileBrowser>().SetBrowserWindowFilter(filter);
}

// You should use this function signature in order to receive properly:
void OnPathSelected(string path)
{
    // Do something with the returned path.
}
```

The `FileBrowser` destroys itself once a file is selected or the window is closed. You can access some public functions through the `browserInstance` variable.

You can double-click or double-tap the items to navigate and chose.

The UIWindowBrigToFront.cs, UIWindowDrag.cs and UIWindowResize.cs are independent scripts that allows the `FileBrowser` window to be dragged, resized and rearanged (They are generic MonoBehaviour classes).

There are a few parameters that can be customized directly from editor in the FileBrowser.cs script:



`public GameObject _ContentItem`: This is the reference to the item prefab. This prefab is used to render every item in the `FileBrowser` (files, folders and messages).

`float _defaultItemSize`: The default size for items if the size slider is disabled or deleted (percentage of the canvas height in units).

`string[] _filter`: This `string List` defines the file extensions to be shown in the file browser. Any other file extension will be discarded. You can set this property by coding through the `SetBrowserWindowFilter()` method (it has several overloads).

`bool _fullPath`: This flag indicates that the provided paths are absolute, allowing to browse any accessible drive location.

`string _lockPath`: Set the lock path to avoid navigation outside this directory.

`string _selectionMode`: The selection mode allows only two values: "F" for files, and "D" for directories or folders.

`bool _save`: This flag sets the browser to work in "save" mode.

`bool _thumbnails`: This flag enables the thumbnails for supported image files.

`Sprite[] _icons`: This array of icons are used by the ContentItem.cs script.

FileBrowser.SetBrowserWindow()

```
public void SetBrowserWindow(OnPathSelected selectionReturn, string iniPath = "",
bool fullPath = false, string selectionMode = "F", bool save = false, string lockPath = "",
string defaultSelection = "", bool thumbnails = false)
```

Use this method to set the `FileBrowser` behavior in real time.

The `OnPathSelected` delegate has a specific signature in order to be called properly:

```
void MyFunction(string)
```

The `ReturnSelectedFile()` function executes the delegate, returning the selected path/file and closes the windows.

The `iniPath` argument sets the first folder to be shown when the window becomes visible.

The `fullPath` argument allows the browser to navigate outside the PersistentData (+StreamingAssets) path without restrictions.

If you navigate until the root in `fullPath` mode, the Windows standalone builds will show the available logical drives.

The `selectionMode` argument allows the browser to select files or folders depending on its value: `"F"` for files, `"D"` for directories or logical drives (please note that in `"D"` mode files are not shown).

The `save` argument allows the `InputField` of the selected item to allow writing a custom file or folder name.

The `lockPath` argument forces the browser to stay always into that directory and sub-directories not allowing further navigation (no matter what navigation mode was selected).

The `defaultSelection` argument allows the browser to select automatically some existing file/folder that matches the provided name. In `save` mode it also sets the default name for the item to be saved.

The `thumbnails` argument enables the presentation of thumbnails for supported image files.

FileBrowser.SetBrowserWindowFilter()

```
public void SetBrowserWindowFilter(List<string> newFilter, int set = 0)
public void SetBrowserWindowFilter(string[] newFilter, int set = 0)
public void SetBrowserWindowFilter(string newFilter, int set = 0)
```

You can use any of the three overloads to set the file extension list to filter the rendered content. You can set the filter dynamically at any time and it will also set the file extension `Dropdown` for you.

File extensions should include the dot before the extension, and extensions can have any length: `".jpg"`, `".unity"`. You should provide the filter as a string like this: `".jpg;.unity"`.

The first filter option includes all provided filters at once.

Special characters as `"*"` or `"?"` are not allowed.

The `set` argument, allows the browser to select any of the provided filters as the default.

FileBrowser.SetBrowserCaption()

```
public void SetBrowserCaption(string title)
public void SetBrowserCaption(string title, Color32 colour)
```

This method sets the browser caption text and color.

You can use any of the overloads to set this parameters in real time. You can set this parameter even while the `FileBrowser` is already open.

FileBrowser.SetBrowserIcon()

```
public void SetBrowserIcon(Sprite icon)
public void SetBrowserIcon(Sprite icon, Color32 colour)
```

This method sets the browser caption icon and color.

You can use any of the overloads to set this parameters in real time. You can set this parameter even while the `FileBrowser` is already open.

Other useful methods in FileBrowser.cs

`Cut()`: This method is called by the "Cut" button. It remembers the file/folder to cut. It shows a warning a message if the file/folder is read only.

`Copy()`: This method is called by the "Copy" button. It remembers the file/folder to copy.

`Paste()`: This button calls copying or moving the previously selected file or folder and all its content.

`PromtDeleteSelection()`: This method is called by the "Delete" button, allowing to delete a file/folder. It will show a warning a message if the file/folder is read only.

`PromptForRename()`: This method is called by the "Rename" button, allowing to rename a file/folder. It will show a warning a message if the file/folder is read only.

`PromtNewFolderName()`: This method is called by the "New folder" button, allowing to create a new folder. You can create nested folders using a path notation (`"Folder1/Folder2"`).

NOTE: The copy/cut/paste features don't uses the system clipboard.

NOTE: The copy/cut/paste paths are shared across all `FileBrowser` instances through `FileManagement`.

ContentItem.cs

This script controls the behavior of the rendered items (files, folders or messages).

`ContentItem` gets the icons from `FileBrowser._icons`, you should add your new icons there in order to be usable.

The icons rendered depending on the file extension are selected by the `SetItem()` method in this script. If you have more custom icons you should code them here.

There aren't configurable parameters in this script.

Customize the browser window

The file browser is completely built with the Unity UI, so you can use the standard tools to customize the window at your will.

IMPORTANT: Before start customizing the prefab, please duplicate it with Ctrl+D and rename or move as needed. Duplicate the ContentItem prefab too, and once renamed or moved, assign its reference to the `ContentItem` variable into your newly created FileBrowser prefab.

There are no scripts attached to the elements of the browser window, everything is controled from FileBrowser.cs.

You just have to keep in mind to use some special names in its hierarchy to allow the FileBrowser.cs script to find the elements that it uses to work properly.

Here are the items you must keep their names:

`BrowserWindow`: This is the main container panel. Everything is contained by this element (it is the window itself).

`Caption`: The title of the browser window. It can be modified through the right methods.

`Icon`: this is the icon at the left of the window caption. It can be modified through the right methods. Can be deleted if not used.

`InputCurrentPath`: This `InputField` contains the current directory being rendered. It can be edited at

any time.

`ContentWindow`: This `ScrollRect` is the list controller of rendered items.

`ContentWindow>ViewPort>Content`: This is the main container of the rendered items. The ContentItem prefabs are added automatically to its `Transform`. It has a `VerticalLayoutGroup` component to arrange items.

`InputSelection`: This `InputField` contains the name of the selected item. When in **save** mode, this `InputField` can be edited to set a custom file or folder name.

`ButtonSelect`: This is the selection `Button`, it can be enabled or disabled depending on the current selection.

`ButtonSelect>Text`: the `Text` of the `ButtonSelect` changes depending on the action that can be executed: `"Select"`, `"Open"` or `"Save"`.

`Confirmation`: This is the confirmation pop-up with a `Label` and two buttons: `ButtonOk` and `ButtonCancel`.

`NewName`: This is the text input pop-up with a `Label`, an `InputField` called `InputNewName` and two buttons: `ButtonOk` and `ButtonCancel`.

`ErrorMessage`: This is the message pop-up with a `Label`, an `Image` and a button: `ButtonOk`.

`SizeSlider`: This `Slider` allows modifying the item size dynamically. Can be deleted if not used.

`FilterDropdown`: This `Dropdown` lists the available file extensions to be rendered in the content view. The first item is always a list with all the available extensions. The extensions will be added separately in the successive items. Can be deleted if not used (filters will still working internally).

Excepting `ButtonSelect`, every button or interactive element has an event that points to a method contained in FileBrowser.cs, so you can delete them if they are not needed.

Set the FileBrowser for 3D environments

To set the browser in 3D world you have to adjust physically the `FileBrowser`, then save the prefab parameters to be properly instantiated.

1. Duplicate the original `FileBrowser` prefab and rename it conveniently (Let's say "MyBrowser").
2. Set the **RenderMode** parameter (in the `Canvas` element) as "World Space".
3. Drag your "MyBrowser" prefab into your scene, and place it in your desired target position, adjusting also size and scale. Adjust the scale of the `RectTransform` that contains the `Canvas`, do not alter the scale of its child UI elements.
4. You can place the "MyBrowser" instance as a child of some other 3D object to check it's OK (in the example scene it's a cube).
5. Once you are happy with the result, Save or Apply the prefab changes. If your "MyBrowser" is a child of some 3D object, do NOT save its parent in the prefab.
6. Now drag the "MyBrowser" prefab to your FileBrowserWrapper.cs or your custom script if you are instantiating the `FileBrowser` programmatically.

If you need the "MyBrowser" to have a parent, then here is how you do that programmatically:

```
GameObject _instance = Instantiate(browser);
_instance.transform.SetParent(parent, false);
```

To assign a parent using FileBrowserWrapper.cs, just drag it to the `_parentTo` parameter.

You can take a look at the 3D example to see how it works.

INI file management

The class `FM_IniFile` is created to manage INI files in memory and then save them to disk using automatic formatting.

INI files are intended to save multiple parameters called "Keys" structured in groups called "Sections". Ini files are widely used to save configurations because they are very easy to read and modify manually.

The standard INI file structure is as follows:

```
; This is a comment.
KeyIntoDefaultGroup = 10          ; Another comment

[Section1]
KeyIntoSection1 = Hello world

[Section2]
KeyIntoSection2 = 3.14
```

Blank lines and comments are discarded.

Key names can be repeated in different sections, they are in fact independent values:

```
Key = 20
[Section1]
Key = 30
```

If a section is repeated, then the values contained are also interpreted as part of the same section.

```
[Section1]
KeyIntoSection1 = Hello world
[Section2]
KeyIntoSection2 = 3.14
[Section1]
_KeyIntoSection1 = Message
```

Is the same as:

```
[Section1]
KeyIntoSection1 = Hello world
_KeyIntoSection1 = Message
[Section2]
KeyIntoSection2 = 3.14
```

If a section+key name is repeated, then the key is not duplicated but updated, so the last one has the highest priority.

```
[Section1]
KeyIntoSection1 = Hello world
[Section2]
KeyIntoSection2 = 3.14
[Section1]
KeyIntoSection1 = Message
```

Is the same as:

```
[Section1]
```

```
KeyIntoSection1 = Message
[Section2]
KeyIntoSection2 = 3.14
```

The FM_IniFile class

`FileManagement` includes the `FM_IniFile` class that allows an easy manipulation of INI files.

You can read, modify and save them with a few simple methods.

The class is empty when it's created, you have a few ways to load an INI file in a `FM_IniFile` object.

Through the constructor:

```
FM_IniFile iniFile = new FM_IniFile("ExampleCfg.ini");
```

This constructor loads the provided INI file if it exists, otherwise an empty `FM_IniFile` will be created.

Calling the `LoadNewIniFile` method:

```
FM_IniFile iniFile = new FM_IniFile();          // Creates an empty INI object.
iniFile.LoadNewIniFile("ExampleCfg.ini");       // Fills the INI object.
```

Parsing a `string`:

```
FM_IniFile iniFile = FM_IniFile.Parse(iniFileContent);
```

If the provided `string` can't be parsed, an empty `FM_IniFile` will be created.

Calling the `ImportIniFile` method in the `FileManagement` class:

```
FM_IniFile iniFile = FileManagement.ImportIniFile("ExampleCfg.ini");
```

FM_IniFile constructor

```
public FM_IniFile()
public FM_IniFile(FM_IniFile ini)
public FM_IniFile(string name, bool enc = false, bool checkSA = true, bool fullPath = false)
```

There are three overloads of the constructor, the one without parameters creates an object without initialization, you'll have to load a file manually. The one accepting an `FM_IniFile` makes a fast copy. The constructor with parameters tries to load the requested file in the new object, if the file not exists the `FM_IniFile` will be empty.

Parse()

```
public static FM_IniFile Parse(string content, char separator = (char)0x00)
public static FM_IniFile Parse(byte[] content, char separator = (char)0x00)
```

This method parses the provided `string` or `byte[]` into a new `FM_IniFile`.

The `separator` parameter determines where the parsed text lines ends. If none is provided, the null character `(char)0x00` will default to the NewLine codes (EOL: `"\r\n"`, `"\n"`, `"\r"`).

LoadNewIniFile()

```
public bool LoadNewIniFile(string name, bool enc = false, bool checkSA = true,
bool fullPath = false)
```

This method loads the requested file into the `FM_IniFile` from disk. The older content will be entirely replaced.

It returns `true` if the process succeeded or `false` otherwise.

LoadNewIni()

```
public bool LoadNewIni(string content, char separator = (char)0x00)
public bool LoadNewIni(string[] content)
```

This method loads the provided INI `content` into the `FM_IniFile`. The older content will be entirely replaced.

It returns `true` if the process succeeded or `false` otherwise.

ToString()

```
public string ToString(char separator = (char)0x00, bool sort = false)
```

This method converts the `FM_IniFile` into a string separated by the `separator` character.

If no `separator` is provided, the null character `(char)0x00` will default to the NewLine (EOL) code.

The `sort` parameter sorts the content alphabetically.

GetKey()

```
public T GetKey<T>(string key, T defaultValue, string section = "",
char separator = (char)0x09)
```

This method returns a value stored in `key`. The method allows type conversion, it uses the `CustomParser()`. The `separator` is passed to the `CustomParser()` in order to parse arrays.

The `defaultValue` parameter is the returned value in case the requested `key`+`section` not exists.

This example returns an int value:

```
FM_IniFile iniFile = FileManagement.ImportIniFile("ExampleCfg.ini");
int value = iniFile.GetKey<int>("Int", 9, "Group1");
```

Now value is "5678" (from the example file) in it, instead of "9".

SetKey()

```
public void SetKey<T>(string key, T value, string section = "", char separator = (char)0x09)
```

Updates a `key` in the `FM_IniFile`. No action is done if the `key` is not found.

You can pass any supported type to this method, it supports the same types as `CustomParser()`. The `separator` is used in the case that an arrays is being saved in this key. The default `separator` is the horizontal tabulator.

This example updates an existing key to the INI object:

```
iniFile.SetKey("Int", 321, "Group1");
```

AddKey()

```
public void AddKey<T>(string key, T value, string section = "", char separator = (char)0x09)
```

Add a new `key` to the `FM_IniFile`. When adding a new `key`, also can be defined a new `section`. If `section` is not provided, the key will be added to the default section.

You can pass any supported type to this method, it supports the same types as `CustomParser()`. The `separator` is used in the case that an arrays is being saved in this key. The default `separator` is the horizontal tabulator.

This example adds a new key to the INI object:

```
iniFile.AddKey("MyKey", 3,14, "MySection");
```

RemoveKey()

```
public void RemoveKey(string key, string section = "")
```

Removes a `key` from the `FM_IniFile`.

This example removes a key from the INI object:

```
iniFile.RemoveKey("MyKey", "MySection");
```

CopySection()

```
public void CopySection(FM_IniFile source, string section = "")
```

Copies an entire `section` from the provided `source`. If the requested section already exists, it will be merged and updated.

This example copies the default section from another INI object:

```
iniFile.CopySection(sourceIniFile, "");
```

RemoveSection()

```
public void RemoveSection(string section = "")
```

Removes a `section` from the `FM_IniFile`, including all its keys.

This example removes a section from the INI object:

```
iniFile.RemoveSection("MySection");
```

KeyExists()

```
public bool KeyExists(string key, string section = "")
```

Checks if `key` exists in the `FM_IniFile`.

SectionExists()

```
public bool SectionExists(string section)
```

Checks if `section` exists in the `FM_IniFile`.

GetSectionList()

```
public string[] GetSectionList(bool sort = false)
public string[] GetSectionList(string startsWith, bool sort = false)
```

Gets the list of all the sections in the `FM_IniFile`.

The first method returns all available sections. The overload filters the sections by matching the names with the provided string `startsWith`.

The `sort` parameter sorts the list alphabetically.

This example gets the list of sections that begins with "DATA_" and sort alphabetically:

```
string[] dataSections = iniFile.GetSectionList("DATA_", true);
```

GetKeyList()

```
public string[] GetKeyList(string section = "", bool sort = false)
```

Gets the list of all the keys in the provided `section`.

The `sort` parameter sorts the list alphabetically.

This example gets the list and sort alphabetically:

```
string[] keys = iniFile.GetKeyList(true);
```

Save()

```
public void Save(string name, bool sort = false, bool enc = false, bool fullPath = false)
```

Saves the content of the `FM_IniFile` to disk. It uses the standard formatting mode (comments are discarded).

The `sort` parameter sorts sections and keys alphabetically.

This example saves the ini file in PersistentData folder:

```
iniFile.SaveIniFile("MyIniFile.ini", true);
```

Merge()

```
public void Merge(FM_IniFile source)
```

Merges the content of the INI file from the parameter `source` to the local INI file object.

Duplicated values are updated with the `source` values.

This example merges two ini files:

```
FM_IniFile otherFile = new FM_IniFile("Cfg.ini");
```

```
iniFile.Merge(otherFile);
```

Clear()

```
public void Clear()
```

Deletes the whole content of the INI file.

This example deletes the content of iniFile:

```
iniFile.Clear();
```

The FM_Average class

`FileManagement` includes the `FM_Average` class that helps with incremental average calculations. The class starts with zero values and they can be added progressively.

Very useful for real time values from any sensors (FPS, RMP, temperature, voltage, density, transfer-rate, etc.) to make constant measurements more readable and stable to the eye.

FM_Average constructor

```
public FM_Average(uint size)
```

The **size** parameter sets the maximum count of values to be kept internally from where get the calculations.

This example creates a new object setting a maximum average count of 100:

```
FM_Average _rateAverage = new FM_Average(100);
```

AddSample()

```
public void AddSample(double newSample)
```

This method adds a new value to the internal list up to a maximum set by the constructor.

Clear()

```
public void Clear()
```

This method clears the internal values.

_result (read only)

```
public double _result { get }
```

This property is read only and computes the average with the current internal values.

You don't need to fill the entire list of samples to get the calculation, you can get the average at any moment.

Multi Threading

IMPORTANT: The multi-threading method discussed in this section is not supported in Windows Store builds.

To add multi-threading tasks to your application easily you have to create a `Thread` object, then assign a function to be executed and finally start the execution like this:

```
using System.Threading;
using UnityEngine;

public class SaverTest : MonoBehaviour
{
    readonly object _fileLock = new object();   // Locks file access.

    // This method sets and starts the thread:
    void Start()
    {
        // Start the thread:
        ThreadPool.QueueUserWorkItem(Saver);
    }

    // This is the method executed as a parallel thread:
    void Saver(object state)
    {
        lock (_fileLock)
        {
            // Save some text:
            FileManagement.SaveFile("MyNewFile.txt", "Hello world!");
        }
    }
}
```

You can't stop a running `Thread`, you can only wait until it ends, so beware of infinite loops.

Use `Thread` to make long tasks, so allow your application to be responsive while the task is being performed.

To prevent many threads from accessing the same file at once, this example uses the `lock` statement.

Most Unity classes and methods can't be used into a `Thread`, that's the reason why some methods in `FileManagement` can't be used into a `Thread` neither.

Those methods are:

- `ImportAudio`: Excepting for WAV files platform (it uses OWP).
- `ReadRawFile`: When accessing StreamingAssets in Android.
- `FileExists`: When accessing StreamingAssets in Android.
- `DirectoryExists`: When accessing StreamingAssets in Android.
- `ImportTexture`: It uses `Texture2D`.
- `ImportSprite`: It uses `Texture2D`.

String conversion and interpretation

In order to allow the most widely supported string format, `FileManagement` includes a way to select how it converts a `string` into `byte[]` (array) and vice-versa.

To do this you have to assign the selected value to the `FM_StringMode` stringConversion parameter like this:

```
FileManagement.stringConversion = FM_StringMode.UTF8;
```

The `FM_StringMode` enumeration has the following values:

`UTF8`: Designed to 8bit environments. This is the default mode, it covers all user cases.

`Fast`: Custom straight conversion from 8bit char to byte. This mode is recommended if you are not using extended ASCII characters.

`ASCII`: 7bit character set.

`BigEndianUnicode`: UTF16 with big endian byte order.

`Unicode`: UTF16 with little endian byte order.

`UTF32`: UTF32 with little endian byte order.

`UTF7`: For 7bit environments.

Known Issues

- There is no straight compatibility with the legacy interfaces used for reading and writing cookies in Web Browsers. Nevertheless you will find those interfaces as `private` in the `FileManagement` class so you can still using them if needed (WebGL only).

- To grant SDCard access in Android you must enable the option: BuildSettings>PlayerSettings>OtherSettings>WriteAccess = External (SDCard).

- Android file system is case sensitive.

- The `InstallLocalApk()` method only works in Android 8.1 and older.

- Listing logical drives is not supported across all platforms.

- Only WAV files are supported since Unity 2019.

- To import OGG and MP3 audio files on Unity 2019 and above, use `UnityWebRequestMultimedia`. Please check the Unity documentation: https://docs.unity3d.com/ScriptReference/Networking.UnityWebRequestMultimedia.GetAudioClip.html

- If you see "The type initializer for 'FileManagement' threw an exception.", just print `FileManagement.persistentDataPath` on any `MonoBehaviour` script at Start() or Awake().

- FileBrowser: Adjust the parameter "EventSystem (script) > Drag Threshold" increasing its value if the item selection fails (specially for VR and mobile implementations).

Contact

If you need some new features or if you find some errors in this documentation or the asset, please don't hesitate on sending me an email: jmonsuarez@gmail.com

I you find that the test version is not the same version that you downloaded from the AssetStore, please send me your invoice number and I will send you back the last `FileManagement` version (new version normally is into approval process).

Please, once you have tested this product, take a minute of your time to write a good review in the Unity Asset Store, so you will help to improve this product:

See File Management in the Asset Store.

Thanks.