UnifiedIO: Cross-platform IO for Unity

Version 2.2.x

Daniel Lobo

1 Quick notes

- If you are here for the API documentation, please go to Section 7.
- UnifiedIO works in Windows, Mac and Linux standalones, iOS, Android, Blackberry, Windows Phone and Windows Store Apps (Windows 8, 8.1, 10 and Universal builds). It supports Unity version 4.2.0 onwards (Free and Pro), including Unity 5, and can be called from C# or UnityScript code, though the latter is only fully supported if using Unity 5 or newer (please see Section 6).
- All methods were unit-tested to be consistent and to handle common and difficult cases alike. The library is also successfully used in several published multiplatform games.
- I'd love to hear from you if you find a bug or have a suggestion for UnifiedIO. I must say that I can't be held responsible for critical problems or loss of data when using UnifiedIO, although the danger shouldn't be greater than using common file system functions. That said, if you find any problem, please *do* contact me at daniel.rb.lobo@gmail.com.

2 Introduction

UnifiedIO is a library for Unity that implements common file and directory (IO) operations and compiles for all main platforms (that includes Windows Store Apps and Windows Phone) without the need of implementing platform-dependent code. It allows developers to create, read, write, move, copy, rename, list, etc, files and folders in the directory reserved for their Unity game – `Application.persistentDataPath` – and have it Just Work^{TM} on all platforms.

If you've ever tried building your Unity game for the Windows Store platform – either a Windows Store Apps (WSA) or Windows Phone build – and used some kind of file or folder operation in your code, chances are that you've faced compilation errors. WSA apps use the WinRT API, which includes file system calls that are very different to methods in other platforms[1] and strongly enforce the use of asynchronous methods.

Your game may compile perfectly in most of your target platforms and in the Unity editor (even with WSA set as the active platform), as Unity uses the Mono compiler for each of those, which knows the normal System.IO classes. However, as soon as you press "Build" for WSA, Unity must use a .NET compiler, which enforces the new API, causing new errors to appear at that point.

[1]Microsoft tightened the security in Windows 8. Each app has very limited access to files, and can generally access only a local data folder reserved for it. This means that apps shouldn't use methods that receive a full path to access files/folders anywhere in the system, which has resulted in the removal of some methods and classes from WSA, including `System.IO.File` and `System.IO.Directory`. If you use any of those, compilation will fail.

3 Why UnifiedIO matters

Well, you probably don't want to add a lot of specific code for your game to run on WSA just because of file IO. You also don't want to litter your code with platform dependent compilation statements like `#if NETFX_CORE`. You might also not need or even want async IO methods. Let's see an example of why UnifiedIO is useful.

Imagine you had the following method to save some data, pertaining to a user, inside a certain file in the "Users" folder:

```
public static void SaveUserData(string filename, string content)
{
    string userFolder = Application.persistentDataPath + "/Users";
    if (!System.IO.Directory.Exists(userFolder)) {
        System.IO.Directory.Create(userFolder);
    }
    System.IO.File.WriteAllText(userFolder + "/" + filename, content);
}
```

Doesn't look too bad. But these methods don't exist in WSA! OK, let's try and code the same method according to the WSA API, also including conditional compilation instructions and the previous sample, so that the code compiles on every platform.

```
#if !NETFX_CORE // active when NOT compiling for WSA
    public static void SaveUserData(string filename, string content)
    {
        string userFolder = Application.persistentDataPath + "/Users";
        if (!System.IO.Directory.Exists(userFolder)) {
            System.IO.Directory.Create(userFolder);
        }
        System.IO.File.WriteAllText(userFolder + "/" + filename, content);
    }
#else // active when compiling for WSA
    public static void SaveUserData(string filename, string content)
    {
        return saveUserDataAsync(filename, content).Result;
    }

    public static async Task saveUserDataAsync(string filename, string content)
    {
        StorageFolder rootFolder = ApplicationData.Current.LocalFolder;
        StorageFolder usersFolder = await rootFolder.CreateFolderAsync("Users",
            CreationCollisionOption.OpenIfExists);

        StorageFile file = await usersFolder.CreateFileAsync(filename,
            CreationCollisionOption.ReplaceExisting);
        await FileIO.WriteTextAsync(file, content);
    }
#endif
```

This works, but you'll likely notice that the code just became a whole lot different from what we had before. Oh, and we had to create a new function that simply waits for the result of the async operation of the other, all in order to get an example equivalent to the first (a synchronous one).

And wait! I lied. It still doesn't compile on Windows Phone, as `System.IO.File.WriteAllText` doesn't exist on that platform. We'd have to implement our own. Do you see the problem here?

Forget that.

Let's try "unifying" the original method using UnifiedIO:

```
public static void SaveUserData(string filename, string content)
{
    string userFolder = "Users";
    if (!UnifiedIO.Directory.Exists(userFolder)) {
        UnifiedIO.Directory.Create(userFolder);
    }
    UnifiedIO.File.WriteText(userFolder + "/" + filename, content);
}
```

That's it. Close enough to the first example? Yes. Works in all platforms, including WSA and Windows Phone? Yes. And it has the nice side effect of not making us write `Application.persistentDataPath` anywhere, as that's already where the whole library works!

Yes, if you didn't read the introduction, UnifiedIO works on your app's persistent data directory. This the only place that is available on all platforms for your Unity game to store data, and also the most common folder for data storage in Windows Store apps. If you're not building games for WSA or Windows Phone, you don't need UnifiedIO anyway, as you can use System.IO as normal!

That said, you *can* change the working folder for UnifiedIO for advanced use-cases that I'll cover in Section 5.

4 Platform dependent compilation

As we saw in the example before, UnifiedIO contains two classes: File and Directory. Internally, platform dependent compilation lets the methods in these classes call the correct implementations depending on the platform. When compiling for platforms other than WSA, implementations in the `UnifiedIO_SystemIO` scripts are called; when compiling for WSA, the ones in `UnifiedIO_WSA` are used (which, in turn, use the asynchronous ones in `UnifiedIO_WSAAsync`).

If you don't want to switch all your existing code to use UnifiedIO methods just because of WSA (or eventually Windows Phone), you can also add platform dependent compilation and use UnifiedIO to handle IO only in certain platforms. *Just to be clear, this is not the way I recommend you use UnifiedIO.*

Assume you have the following piece of code:

```
public static byte[] ReadBinaryData(string filename)
{
    return System.IO.File.ReadAllBytes(
        Application.persistentDataPath + "/some␣folder/" + filename);
}
```

This doesn't work in WSA. As we saw before, you could do this, instead, which is the recommended way:

```
public static byte[] ReadBinaryData(string filename)
{
    return UnifiedIO.File.ReadBytes("some␣folder/" + filename);
}
```

It would work on every platform now. However, you don't want to change it, so you can use platform dependent compilation and do this:

```
public static byte[] ReadBinaryData(string filename)
{
#if NETFX_CORE
    return UnifiedIO.File.ReadBytes("some␣folder/" + filename);
#else
    return System.IO.File.ReadAllBytes(Application.persistentDataPath + "/some␣folder/" +
        filename);
#endif
}
```

Now, when compiling for WSA on the .NET compiler, it will use code from UnifiedIO; when compiling for platforms other than WSA, it will use your code. It's more complicated this way, but you have your reasons! :) And it's still better than coding that `ReadBytes` method yourself, with async calls and all. Just beware of different behavior between your code and the UnifiedIO calls!

5 Advanced usage

This section is only for advanced use-cases. The vast majority of users will not need it.

Since version 2.1.0 of UnifiedIO, you can change the working directory of UnifiedIO to something other than `Application.persistentDataPath`. This can be useful if you want to enforce a different path on a certain platform, like saving everything in a custom folder on Android, or using a folder picker in WSA to let the user choose the saving directory. *It's up to you to guarantee that the folder exists and has write permissions for your application and platform!*

On non-WSA platforms, like Android, iOS, desktop, etc, the `UnifiedIO.SystemIO.Internal` namespace contains a class `Path` with a public static variable `BaseFolder`, which is a string and defaults to `Application.persistentDataPath`, as given by Unity. You can change it to some other path and IO will be done there instead.

On WSA, the `UnifiedIO.WSAAsync.Internal` namespace contains a class `Path` with a public static variable `BaseFolder`, which is a StorageFolder and defaults to `ApplicationData.Current.LocalFolder` on the `Windows.Storage` namespace, where `Application.persistentDataPath` points to in Unity.

This is an example of how to change it, for example at the start of your game/application, or after a folder pick by a user:

```
public static void SetupUnifiedIODirectory()
{
#if NETFX_CORE
    UnifiedIO.WSAAsync.Internal.Path.BaseFolder = /* another StorageFolder */;
#elif UNITY_ANDROID
    UnifiedIO.SystemIO.Internal.Path.BaseFolder = /* a string path for your custom folder */;
#else
    /* something else for the other plaforms; maybe the default, maybe custom directories */
#endif
}
```

You can change the working folder at any time and even in between calls to UnifiedIO methods, if you want to save things in more than one folder.

6 Working with UnityScript

UnifiedIO was programmed using C#, which means that its methods can be called as expected from C# code. If your scripts are written in C#, you can skip this section. That is all there is to it.

If you want to call UnifiedIO methods from UnityScript and create builds for WSA, first be sure to be on at least Unity 5. Now, please *send me an e-mail to daniel.rb.lobo@gmail.com and ask for the UnityScript-compatible DLLs.* I humbly ask that you mention the invoice number for the purchase so that I can verify that you are a legitimate user of UnifiedIO.[2]

6.1 Why this extra step?

I realize this is inconvenient! I apologize. :(This is necessary because I wish to keep UnifiedIO as a source code package, so I couldn't also include DLLs directly without conflicts.

Couldn't we just use the source files with UnityScript, instead of DLLs? Well... In Unity, calling C# code from UnityScript requires the C# code to be compiled first (likewise, calling UnityScript code from C# requires the UnityScript code to be compiled first). The solution for this would be to move the whole "UnifiedIO" folder (which comes in the package) to a "Plugins" or "Standard Assets" folder in your project, as these folders are put into a separate assembly, compiled before most of your remaining folders.[3] This solution is not specific to UnifiedIO and is the common way to handle cross-language calls in Unity.

So, you could do this for non-WSA platforms; however, due to limitations of UnityScript, you would have to activate ".NET Core Partially" in the WSA Player Settings, which in turn requires that code which calls the WinRT API (UnifiedIO!) must be outside of the "Plugins" folder. As you might guess from the previous paragraph, this would make it impossible to call UnifiedIO from UnityScript unless we use a rather cumbersome setup with interfaces in the "Plugins" folder and implementations outside. I decided to use DLLs instead.

[2]To get the invoice number for your purchase, either use the order number in the invoice PDF that Unity Technologies sends you after the asset's purchase, or log into the asset store, access your profile picture, then "Credit Card / PayPal" and get the number in the description for UnifiedIO.

[3]see `http://docs.unity3d.com/Manual/ScriptCompileOrderFolders.html` for more details.

7 API

This section presents the full API of methods available in UnifiedIO. Please start from the preliminary notes for usage details.

7.1 Preliminary notes

- Every single method expects a *relative path* from `Application.persistentDataPath`, so you should *never* prepend `Application.persistentDataPath` to create a full path. Simply write `"some folder/some file.txt"` and it will be as if you used `Application.persistentDataPath + "some folder/some file.txt"` in `System.IO` methods.
- *Don't* start those relative paths with `"/"` (slash), `"./"` (dot and slash) or anything else. If you want to create a folder directly inside of `Application.persistentDataPath` (i.e. at the root) you should simply use `UnifiedIO.Directory.Create("Folder name")`. The same for nested folders, like `UnifiedIO.Directory.Create("Folder name/Child folder")`.
- If you want to enumerate all files at the root, call `UnifiedIO.Directory.EnumerateFiles("")`. Note the empty string. Similarly, use `UnifiedIO.Directory.DeleteContents("")` to clear the root of all files. If you don't like the look of empty strings, for this special case you can also use `"/"` to the same end. Just don't use `"./"`.
- Lastly, when methods are used with invalid paths, when some file/folder is accessed and does not exist or is accessed without permissions, or when another kind of runtime error occurs, you'll get an exception. Unfortunately, exceptions *will* be of different types for WSA and for other platforms, as trying to catch them all in UnifiedIO and re-throwing them as equivalent exceptions would potentially hide important details and be difficult to implement right. It was a decision I made consciously, even though it fails to be part of the whole "Unified" idea. Furthermore, WSA apps like to throw AggregateExceptions, which themselves can contain multiple exceptions inside, making this union even more problematic. If you think something can go wrong, you'll likely have to "catch" the problem in a platform-dependent way, unless you simply catch any `Exception` and don't care about what problem caused it (though that is bad advice and you didn't hear it from me).

7.2 UnifiedIO.Directory class

```
bool Exists(string path)
```

Checks if path exists as a folder (true/false). It will return false if the path does not exist or if it is a file instead of a folder.

```
// check if "some folder" is a folder at the root (Application.persistentDataPath)
bool exists = UnifiedIO.Directory.Exists("some␣folder");
// check if "some folder/some subfolder" exists and is a folder
bool exists = UnifiedIO.Directory.Exists("some␣folder/some␣subfolder");
```

Note: Calling this over the root – using `Exists("")` or `Exists("/")` – has undefined behaviour in UnifiedIO. If no catastrophic error occurs, `Application.persistentDataPath` will always exist because the Unity player creates it for your game, and, in fact, even some Unity services use the folder.

```
void Create(string path)
```

Creates a folder at the given path, as well as all intermediate folders if necessary.

```
// create a folder at the root
UnifiedIO.Directory.Create("simple␣folder");
// create folder named "complete", with "folder" inside, with "path" inside (all are
    folders)
UnifiedIO.Directory.Create("complete/folder/path");
```

```
void Rename(string path, string newName)
```

Renames folder at path with a new name. Fails if a file or folder already exists with that name in the same directory.

Don't try to use Rename as a move operation (i.e. giving it two paths). That is done using the Move method. Rename simply changes the name of the folder.

```
// this will generate the path "folder/new name"
UnifiedIO.Directory.Rename("folder/old␣name", "new␣name");
```

```
void Move(string path, string destinationPath)
```

Changes path of folder to "destinationPath". Intermediate folders of "destinationPath" must already exist. Fails if "destinationPath" already exists.

Don't mistake this function for MoveInside. In MoveInside, the folder at "path" is moved *into* "destinationPath". On the other hand, Move makes "path" *become* "destinationPath".

```
// folder "folder/name" becomes "folder2/name" ("folder2" must exist beforehand)
UnifiedIO.Directory.Move("folder/name", "folder2/name");
// the way to use Move as an equivalent to rename
UnifiedIO.Directory.Move("folder/old␣name", "folder/new␣name");
```

```
void Copy(string path, string destinationPath)
```

Copies folder as "destinationPath". Intermediate folders of "destinationPath" must exist. Fails if "destinationPath" already exists.

Don't mistake this function for CopyInside. In CopyInside, the folder at "path" is copied *into* "destinationPath". On the other hand, Copy creates a copy of "path" *as* "destinationPath".

```
// folder "folder/child" gets copied as "folder/child replica"
UnifiedIO.Directory.Copy("folder/child", "folder/child␣replica");
// folder "folder/child" gets copied as "another folder/child replica" ("another folder"
    must exist beforehand)
UnifiedIO.Directory.Copy("folder/child", "another␣folder/child␣replica");
```

```
void MoveInside(string path, string destinationPath)
```

Moves folder to the inside of “destinationPath”. Fails if a file/folder of the same name already exists inside “destinationPath”.

```
// move folder "name" to the inside of "new folder", creating path "new folder/name"
UnifiedIO.Directory.MoveInside("folder/name", "new␣folder");
```

```
void CopyInside(string path, string destinationPath)
```

Copies folder to the inside of “destinationPath”. Fails if a file/folder of the same name already exists inside “destinationPath”.

```
// copy folder "name" to the inside of "new folder", creating path "new folder/name"
UnifiedIO.Directory.CopyInside("folder/name", "new␣folder");
```

```
void Delete(string path)
```

Deletes the folder at path (including all of its contents). The folder is deleted whether or not it is empty.

```
// delete folder "useless"
UnifiedIO.Directory.Delete("useless");
// delete folder "useless" inside of "parent" ("parent" is kept intact)
UnifiedIO.Directory.Delete("parent/useless");
```

```
void DeleteContents(string path)
```

Empties a folder. All files and folders inside it are deleted, but the folder itself is not.

```
// delete the contents of the root folder (i.e. everything in
    Application.persistentDataPath)
UnifiedIO.Directory.DeleteContents("");
// delete only the contents of folder "useless"
UnifiedIO.Directory.DeleteContents("useless");
// delete only the contents of folder "parent/useless" ("parent" is kept intact)
UnifiedIO.Directory.DeleteContents("parent/useless");
```

```
string[] GetFiles(string path, string searchPattern = null, System.IO.SearchOption
    searchOption = System.IO.SearchOption.TopDirectoryOnly)
```

```
// NOTE: Formerly named EnumerateFiles, which is now deprecated.
```

Gets an array with the names (or relative paths when getting items recursively) of all files inside a folder. If no files exist inside, the array will be empty.

The method accepts a "searchPattern" string to filter the returned items. The pattern can use * wildcards to match zero or more characters, or ? to match zero or one characters. This is *not* a regular expression (regex); it mimics the search patterns used in System.IO.Directory.GetFiles. However, unlike that method, you can pass a null "searchPattern" to get the same effect as passing "*". Beware that, like in the System.IO method, patterns match to the beginning and end of the string, meaning that the pattern "some" will not match "something", but "some*" will. Likewise, "*.c" will not find "file.cpp", but "*.c*" will.

The method also accepts a "searchOption" to specify if the method should only get the direct children of the folder (SearchOption.TopDirectoryOnly) (this is the default) or recursively get all sub-children (SearchOption.AllDirectories). Items found in subfolders will be returned as their relative path from the search folder "path". For example, "a sub folder/another sub folder/file.txt" instead of simply "file.txt".

```
// get an array of file names at the root
string[] files = UnifiedIO.Directory.GetFiles("");
// get an array of file names inside the folder "parent/library"
string[] files = UnifiedIO.Directory.GetFiles("parent/library");
// get an array of file names with the extension "txt" inside the folder "folder"
string[] files = UnifiedIO.Directory.GetFiles("folder", "*.txt");
// get an array of relative paths to files named like "image1.jpg", "image76.jpg" or
    "imageXX.jpg", in all directories and subdirectories of folder "images"
string[] files = UnifiedIO.Directory.GetFiles("images", "image??.jpg",
    System.IO.SearchOption.AllDirectories);
```

```
string[] GetDirectories(string path, string searchPattern = null,
    System.IO.SearchOption searchOption = System.IO.SearchOption.TopDirectoryOnly)
```

```
// NOTE: Formerly named EnumerateDirectories, which is now deprecated.
```

Gets an array with the names (or relative paths when getting items recursively) of all folders inside a folder. If no folders exist inside, the array will be empty.

The method accepts a "searchPattern" string to filter the returned items. The pattern can use * wildcards to match zero or more characters, or ? to match zero or one characters. This is *not* a regular expression (regex); it mimics the search patterns used in System.IO.Directory.GetDirectories. However, unlike that method, you can pass a null "searchPattern" to get the same effect as passing "*". Beware that, like in the System.IO method, patterns match to the beginning and end of the string, meaning that the pattern "some" will not match "something", but "some*" will. Likewise, "*folder" will not find "best folder ever", but "*folder*" will.

The method also accepts a "searchOption" to specify if the method should only get the direct children of the folder (SearchOption.TopDirectoryOnly) (this is the default) or recursively get all sub-children (SearchOption.AllDirectories). Items found in subfolders will be returned as their relative path from the search folder "path". For example, "a sub folder/another sub folder/final" instead of simply "final".

```
// get an array of folder names at the root
string[] folders = UnifiedIO.Directory.GetDirectories("");
// get an array of folder names inside the folder "parent/library"
string[] folders = UnifiedIO.Directory.GetDirectories("parent/library");
// get an array of folder names starting with "user" inside the folder "folder"
string[] files = UnifiedIO.Directory.GetDirectories("folder", "user*");
// get an array of relative paths to folders named like "User1", "User76" or "UserXX", in
    all directories and subdirectories of folder "Users"
string[] files = UnifiedIO.Directory.GetDirectories("Users", "User??",
    System.IO.SearchOption.AllDirectories);
```

7.3 UnifiedIO.File class

```
bool Exists(string path)
```

Checks if path exists as a file (true/false). It will return false if the path does not exist or if it is a folder instead of a file.

```
// check if "some file.txt" is a file at the root (Application.persistentDataPath)
bool exists = UnifiedIO.File.Exists("some␣file.txt");
// check if "some folder/some subfile.txt" exists and is a file
bool exists = UnifiedIO.File.Exists("some␣folder/some␣subfile.txt");
// remember: files can have no extension!
bool exists = UnifiedIO.File.Exists("some␣folder/extensionless-file");
```

```
void CreateEmpty(string path)
```

Creates an empty file at path. Intermediate folders of path must exist. This is just a commodity method for creating an empty file. If you want to create *and* write something to it, see File.WriteText, File.WriteBytes, File.WriteLines, or the methods that let you open a writable Stream to it.

```
// creates an empty file named "SingleInstanceLock.tmp" inside of "folder"
UnifiedIO.File.CreateEmpty("folder/SingleInstanceLock.tmp");
```

```
void Rename(string path, string newName)
```

Renames file at path with a new name. Fails if a file or folder already exists with that name.

Don't try to use Rename as a move operation (i.e. giving it two paths). That is done using the Move method. Rename simply changes the name of the file given by the path.

```
// this will generate the path "folder/new name.txt"
UnifiedIO.File.Rename("folder/old␣name.txt", "new␣name.txt");
```

```
void Move(string path, string destinationPath)
```

Changes path of file to be "destinationPath". Intermediate folders of "destinationPath" must exist. Fails if "destinationPath" already exists.

Don't mistake this function for MoveInside. In MoveInside, the file at "path" is moved to the inside of "destinationPath". On the other hand, Move makes "path" *become* "destinationPath".

```
// file "folder/old name.txt" is now "new parent/name.tx"
UnifiedIO.File.Move("folder/old␣name.txt", "new␣parent/name.txt");
// the way to use Move as a rename
UnifiedIO.File.Move("folder/old␣name.txt", "folder/new␣name.txt");
```

```
void Copy(string path, string destinationPath)
```

Copies file as “destinationPath”. Intermediate folders of “destinationPath” must exist. Fails if “destinationPath” already exists.

Don’t mistake this function for CopyInside. In CopyInside, the file at “path” is copied to the inside of “destinationPath”. On the other hand, Copy creates a copy of “path” *as* “destinationPath”.

```
// file "folder/old name.txt" gets copied as "new parent/name.txt"
UnifiedIO.File.Copy("folder/old␣name.txt", "new␣parent/name.txt");
```

```
void MoveInside(string path, string destinationPath)
```

Moves file to the inside of “destinationPath”. Fails if a file/folder of the same name already exists inside “destinationPath”.

```
// move file "name.txt" to the inside of "new parent", creating path "new parent/name.txt"
UnifiedIO.File.MoveInside("folder/name.txt", "new␣parent");
```

```
void CopyInside(string path, string destinationPath)
```

Copies file to the inside of “destinationPath”. Fails if a file/folder of the same name already exists inside “destinationPath”.

```
// copy file "name" to the inside of "another folder", creating path "another
    folder/name.txt"
UnifiedIO.File.CopyInside("folder/name.txt", "another␣folder");
```

```
void Delete(string path)
```

Deletes the file at path.

```
// delete file "useless.txt" at the root
UnifiedIO.File.Delete("useless.txt");
// delete file "useless.txt" inside of "folder" ("folder" is kept intact)
UnifiedIO.File.Delete("folder/useless.txt");
```

`Stream GetReadStream(string path)`

Creates a read stream from the file at path.

```
// open a read stream over a file and create a BinaryReader to read a few values from it
Stream stream = UnifiedIO.File.GetReadStream("folder/some␣file.txt");

using (var reader = new BinaryReader(stream))
{
    float f = reader.ReadSingle();
    string s = reader.ReadString();
    int i = reader.ReadInt32();
    // do something to the values here
}
```

`Stream GetWriteStream(string path)`

Creates a write stream from the file at path. Intermediate folders of path must exist. This will empty the contents of the file and start writing at its beginning. To append to the file, open an append stream with GetAppendStream.

```
// open a write stream over a file and create a BinaryWriter to write a few values to it
Stream stream = UnifiedIO.File.GetWriteStream("folder/some␣file.txt");

using (var writer = new BinaryWriter(stream))
{
    writer.Write(123f);
    writer.Write("a␣string");
    writer.Write(10);
}
```

`Stream GetAppendStream(string path)`

Creates an append stream over the file at path. Intermediate folders of path must exist. This will append to the end of the file. To write from the beginning, open a write stream with GetWriteStream.

```
// open an append stream over a file and create a BinaryWriter to write a few values to it
Stream stream = UnifiedIO.File.GetAppendStream("folder/some␣file.txt");

using (var writer = new BinaryWriter(stream))
{
    writer.Write(false);
    writer.Write(3.14);
    writer.Write('c');
}
```

```
byte[] ReadBytes(string path, int position = 0, int nBytes = 0)
```

Reads the full content of the file at path as an array of bytes. If “position” (offset in number of bytes) is specified, it reads only content from that position onward. If “nBytes” is specified, it reads only that number of bytes.

```
// reads all bytes from the file at "folder/file.dat"
byte[] content = UnifiedIO.File.ReadBytes("folder/file.dat");
// reads 5 bytes from the beginning of file at "folder/file.dat"
byte[] content = UnifiedIO.File.ReadBytes("folder/file.dat", 0, 5);
// reads 1024 bytes after position 16 of file at "folder/file.dat"
byte[] content = UnifiedIO.File.ReadBytes("folder/file.dat", 16, 1024);
// reads all bytes after position 20 of file at "folder/file.dat"
byte[] content = UnifiedIO.File.ReadBytes("folder/file.dat", 20);
```

```
void WriteBytes(string path, byte[] content)
```

Creates/overwrites a file at path with the bytes in “content”. Intermediate folders of path must exist.

```
// create or overwrite the file at "folder/file.dat" with the given bytes
UnifiedIO.File.WriteBytes("folder/file.dat", new byte[] { 1, 2, 3, 4, 5 });
```

```
void WriteBytes(string path, byte[] content, int position)
```

Creates/overwrites part of a file at path with the bytes in “content” at offset “position” (in number of bytes). Intermediate folders of path must exist. The previous contents of the file will remain the same, except for the overwritten part. This contrasts with the WriteBytes call that uses no positional argument, where the full file is overwritten. Calling this method for a non-existent file creates it and writes 0s before the specified position.

```
// create or overwrite the file at "folder/file.dat" after offset 3 with the given bytes
UnifiedIO.File.WriteBytes("folder/file.dat", new byte[] { 1, 2, 3, 4, 5 }, 3);
// if the file didn't exist previously, the full contents will be {0, 0, 0, 1, 2, 3, 4, 5}
// if the file did exist, the full contents will be {b, b, b, 1, 2, 3, 4, 5, b, b, b, ...},
    where "b" is a byte already in the file
```

```
void AppendBytes(string path, byte[] content)
```

Creates or appends to a file at path with the bytes in “content”. Intermediate folders of path must exist.

```
// create or append to file at "folder/file.dat" with the given bytes
UnifiedIO.File.AppendBytes("folder/file.dat", new byte[] { 1, 2, 3, 4, 5 });
```

`string ReadText(string path)`

Reads the content of the file at path as a string.

```
// reads all text from the file at "folder/file.txt" as a string
string content = UnifiedIO.File.ReadText("folder/file.txt");
```

`void WriteText(string path, string content)`

Creates/overwrites a file at path with the text in the string “content”. Intermediate folders of path must exist.

```
// create or overwrite the file at "folder/file.txt" with the given string
UnifiedIO.File.WriteText("folder/file.txt", "a␣string,␣possibly␣with␣unicode␣chars");
```

`void AppendText(string path, string content)`

Creates or appends to a file at path with the text in the string “content”. Intermediate folders of path must exist.

```
// create or append to file at "folder/file.txt" with the given string
UnifiedIO.File.AppendText("folder/file.txt", "a␣string,␣possibly␣with␣unicode␣chars");
```

`IList<string> ReadLines(string path)`

Reads the content of the file at path as lines of text (strings).

```
// reads all lines from the file at "folder/file.txt" as strings
IList<string> content = UnifiedIO.File.ReadLines("folder/file.txt");
foreach (string line in content) {
   // do something to line
}
```

`void WriteLines(string path, IEnumerable<string> content)`

Creates/overwrites a file at path, containing the lines of text in “content” (it can be a `string[]`, a `List<string>` or, more generally, `IEnumerable<string>`). Intermediate folders of path must exist.

```
// create or overwrite the file at "folder/file.txt" with the given lines of text
UnifiedIO.File.WriteLines("folder/file.txt",
   new List<string> { "line␣one", "another␣line", "final␣line"});
```

```
void AppendLines(string path, IEnumerable<string> content)
```

Creates or appends to a file at path, containing the lines of text in “content” (it can be a `string[]`, a `List<string>` or, more generally, `IEnumerable<string>`). Intermediate folders of path must exist.

```
// create or append to file at "folder/file.txt" with the given lines of text
UnifiedIO.File.AppendLines("folder/file.txt",
    new List<string> { "line␣one", "another␣line", "final␣line"});
```