# Snapshot Library Quick Start Guide

# Table of Contents



## Name

The "Snapshot": A complete, full-fledged and open source, video game, C++ data serialization solution and architecture.

## Description

"Snapshot" is a library written in C++ that provides the infrastructure to build tools capable of collecting and serializing instance data on the runtime of video games and other independent software, while preserving object references.

## Prerequisites

- Operating System: Minimum Windows 10 22H2 x64 19045.4412
- Library Compiler: MSVC v143 VS2022
- Dependencies: CPPPack - nvp_packing branch and C# MessagePack for the Proposed Usage

## Basic Usage

### DLL compilation

Use the provided Visual Studio 2022 solution to compile the library or compile your own .dll with the aforementioned compiler. You can also use the pre-compiled .dll file provided with the complete package of the Snapshot library.

### C# Wrapper

The library comes with a C# wrapper that can be used straight out-of-the box in any C# written software. Every C++ library exposed method is provided through this wrapper. The wrapper provides methods for:

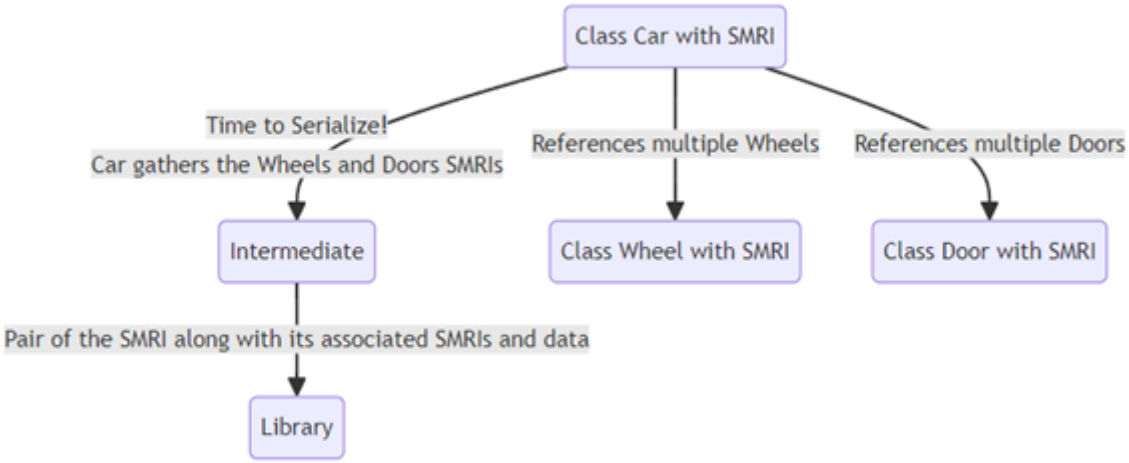
- Save path handling
- SMRI handling
- Data caching and Packing
- Data loading from file
- Memory cleanup

## General usage examples

### What's an SMRI?

**Snapshot Manager Reference Index** or **SMRI** is the main identification aspect that the library uses, to distinguish different class objects and their data. An SMRI should be assigned on each different class that should be saved from the Snapshot library.

A generalized approach of the SMRI-oriented design.




## Proposed Interfaces

> Part of the ProposedArchitecture, which is discussed later.

### ISnapshot

```
/// <summary>
/// Marks a class as ISnapshot-able
/// </summary>
public interface ISnapshot {
    ///<summary>The class should have an SMRI field</summary>
    public uint Smri { get; }

    public void RegisterToSaveManager();
    public void CacheModel();
    public ISnapshotModel ConstructModel();
    public void UnregisterToSaveManager();
    public void LoadModel(ISnapshotModel _model);
    public void RetrieveReferences(int[] _refSmris);
    public Type GetSnapshotModelType();
}
```

### ISnapshotModel

```
public interface ISnapshotModel{
      ///<summary>The class/struct should have an SMRI field</summary>
      public uint Smri {get; set;}
      ///<summary>The class/struct should have an int array to store its references
SMRIs</summary>
      public int[] RefSmris {get; set;}
  }
```

## General methods usage

```
/*
Set the initial save and loading path of the snapshot files
*/

public SaveManager(){
    SnapshotWrapper.SetSavePath(
        Path.Combine(
            Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments),
            "SavesFolder")
        );
}
```

```
/*
Get an SMRI ID for your class instance which
will be later used for reference identification.
*/

List<ISnapshot> _Snapshots = new List<ISnapshot>();

public uint RegisterModel(ISnapshot _snapshot) {
    _Snapshots.Add(_snapshot);

    return SnapshotWrapper.GetSmri();
}
```

> The library accepts byte[] so an external serializer is used for convenience. You can either use and external serializer or create your own serialization technique using .NET tools.
>
> Note: You should also pass an array of the SMRIs this particular class instance holds reference of.

```
/*
A full saving sequence of the saved classes of an application.
*/

public void Save() {
    byte[] bytes;
    int[] refSmris;

    for (int i = 0; i < _Models.Count; i++) {
        bytes = MessagePack.MessagePackSerializer.Serialize<object>(_Models[i]);
        refSmris = _Models[i].RefSmris;

        SnapshotWrapper.CacheData(_Models[i].Smri, bytes, refSmris);
    }

    _Models = new List<ISnapshotModel>();

    SnapshotWrapper.PackData();
}
```

## Loading sequence

```
/*
A full loading sequence of the serialized classes of an application.
Again, an external serializer is used for convenience.
*/

public void LoadSaveFile(string _fileName) {
    if (SnapshotWrapper.SetLoadFileName(_fileName)) {
        SnapshotWrapper.UnpackData();

        byte[] bytes;
        ISnapshotModel model;
        foreach (ISnapshot snapshot in _Snapshots) {
            bytes = SnapshotWrapper.GetData(snapshot.Smri);
            model =
(ISnapshotModel)MessagePack.MessagePackSerializer.Deserialize(snapshot.GetSnapshotMode
lType(), bytes);
            snapshot.LoadModel(model);
        }

        //Instructs the created classes to gather their old referenced objects.
        foreach (ISnapshot snapshot in _Snapshots) {
            snapshot.RetrieveReferences(SnapshotWrapper.GetRefSmris(snapshot.Smri));
        }
    }
}
```

> Memory cleanup

```
/*
Memory cleanup sequence
*/
public bool Cleanup() {
    return SnapshotWrapper.ResetCache() && SnapshotWrapper.ResetSmri();
}
```

# Usage in Unity

## Unity Package

Select the "Snapshot" Unity package from the latest Snaspshot distribution and import it into any kind of Unity project you want to use it on.

The Snapshot Unity package includes the following:

- Pre-compiled C++ library for both x86 and x64 systems
- C# Message Pack from dependencies
- C# Snapshot wrapper
- The Proposed Usage source files
- Proposed Usage scenes
- Proposed Usage prefabs
- Proposed Usage Dummy scripts.

The main entry point of the Proposed usage showcase is the:
*Snapshot/Tests/Scenes/ProposedUsage* scene.

## Compatibility

In case you keep the **C# MessagePack serialization** usage from the Unity Package it is advised to switch the Unity project from **.NET Standard 2.1** to **.NET Framework**. This can be done through *Project Settings/Player/Other Settings/Api Combatibility Level*

# Proposed Architecture

The below files can be found at: *Snapshot/Runtime/Source/Mad/ProposedUsage*

> Disclaimer: The Proposed Architecture is here to guide the developer throughout the parts of the package and show him a way that he can utilize the library. Everything can be changed at will.

## General Information

The library can be used as is in general but for the optimal level of usage and efficiency a proposed architecture has beed designed along with it. Below comes a general approach on how to implement and use the library to its full potential.

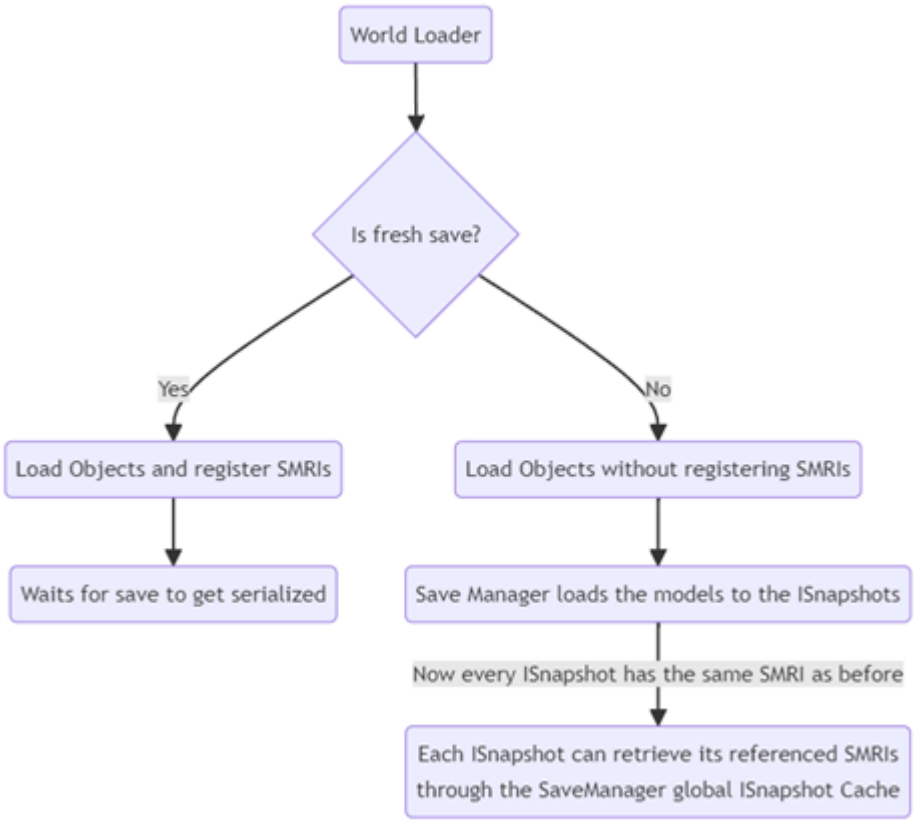
## Architectural Design

The design follows the principle of Default-state loading of a world which then gets filled with the loaded data of a save file, thus a *World Loader* should be created that handles the sequencial loading of the game world.

Each class that should be serialized should inherit from the **ISnapshot** interface and implement the needed methods. A *model* should be created that will cache the needed class data and should inherit from the **ISnapshotModel** interface. When it's time to create a save file, each ISnapshot class should gather the SMRIs of its needed referenced instance objects and pass them along to its own ISnapshotModel instance.

For this exact purpose, the *SaveManager* was created, which works with an event approach design for maximum decoupling.

> Note: For the purpose of the example the simplest form of world loading and saving was used.




## ISnapshot Interface

The ISnapshot is an interface that the inherited members should be able to interact with the SaveManager that they are registered into.

Each member should register to the TakeSnapshot event of their corresponding SaveManager. Should the event is triggered then the member of such an interface must collect and deliver the desired data through their respective *ISnapshotModel* class or struct.

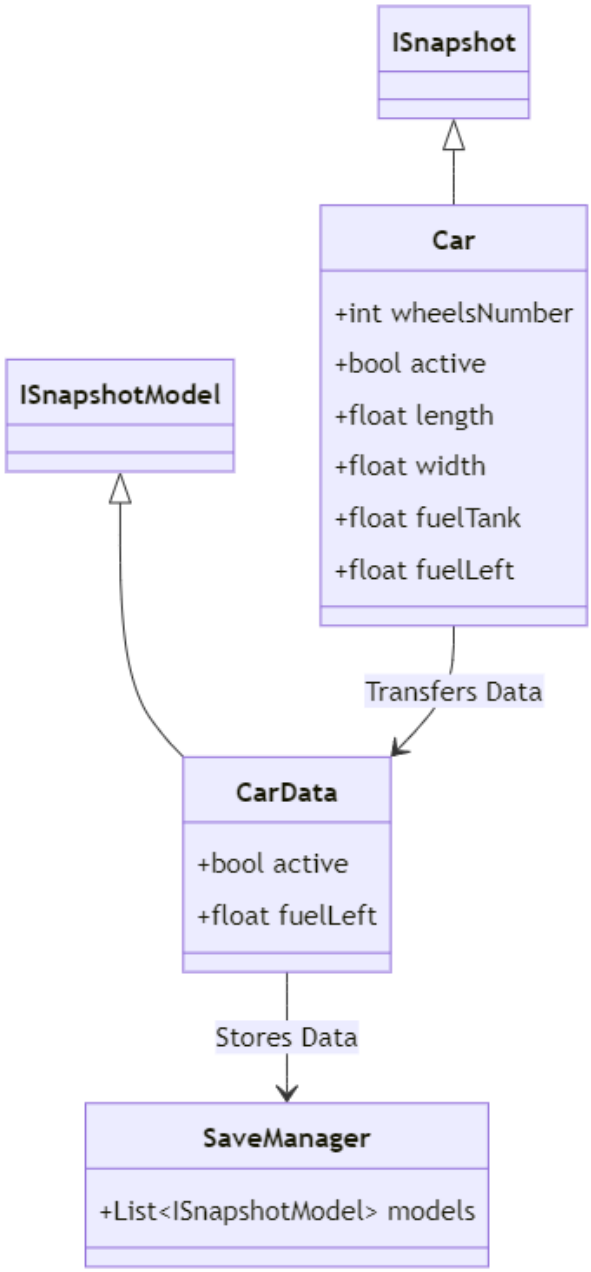
```
namespace ProposedArchitecture {

    /// <summary>
    /// Marks a class as ISnapshot-able
    /// </summary>
    public interface ISnapshot {
        ///<summary>The class should have an SMRI field</summary>
        public uint Smri { get; }

        public void RegisterToSaveManager();
        public void CacheModel();
        public ISnapshotModel ConstructModel();
        public void UnregisterToSaveManager();
        public void LoadModel(ISnapshotModel _model);
        public void RetrieveReferences(int[] _refSmris);
        public Type GetSnapshotModelType();
    }
}
```

## ISnapshotModel Interface

The ISnapshotModel is a **data carriage** interface. The inherited members verify to the SaveManager their role, which is data delivery between the SaveManager and an ISnapshot interface class instance.




A simple ISnapshotModel of a ISnapshot class named Inventory:

```
namespace ProposedArchitecture {

    [MessagePackObject]
    public struct SInventory : ISnapshotModel {
        [Key(0)]
        public uint Smri { get; set; }
        [Key(1)]
        public int[] RefSmris { get; set; }
        [Key(2)]
        public int _MaxItems;
        [Key(3)]
        public Vector3 _Position;
        [Key(4)]
        public Quaternion _Rotation;
    }
}
```

The typical use of a model construction when its time to save, from its ISnapshot class is as follows:

```
/// <summary>
/// Returns an ISnapshotModel with the inventory needed data.
/// </summary>
public ISnapshotModel ConstructModel() {
    SInventory temp = new SInventory() {
        Smri = this.Smri,
        RefSmris = new int[_Weapons.Count],
        _MaxItems = this._MaxItems,
        _Position = transform.position,
        _Rotation = transform.rotation,
    };

    //Cache the SMRIs here
    for (int i = 0; i < temp.RefSmris.Length; i++) {
        temp.RefSmris[i] = (int)_Weapons[i].Smri;
    }

    return temp;
}
```

## Fresh save or Load

The **WorldLoader** distinguishes between a fresh save or a load sequence:

```
        void Start() {
            if (!_FromLoad) {
                DefaultStart();
            } else {
                FromLoadStart();
            }
        }

        ///<summary>Loads the game objects normally</summary>
        void DefaultStart() {
            GameObject temp;
            for (int i = 0; i < _Objects.Length; i++) {
                temp = Instantiate(_Objects[i], _Objects[i].transform.position,
_Objects[i].transform.rotation);
                if (temp.TryGetComponent<ISnapshot>(out ISnapshot snapshot)) {
                    snapshot.RegisterToSaveManager();
                }
            }
        }

        ///<summary>Loads the objects and then calls the SaveManager unpacking method.
</summary>
        void FromLoadStart() {
            GameObject temp;
            for (int i = 0; i < _Objects.Length; i++) {
                temp = Instantiate(_Objects[i], _Objects[i].transform.position,
_Objects[i].transform.rotation);
                if (temp.TryGetComponent<ISnapshot>(out ISnapshot snapshot)) {
                    snapshot.RegisterToSaveManager();
                }
            }

            //Fires off the loading sequence
            Common.Instance.SaveManager.LoadSaveFile(_SaveFileName);
        }
```

## Loading a model

When an ISnapshot should load a deserialized models the following methods are called from the SaveManager loading sequence to instruct it to load its own ISnapshotModel and then retrieve its referenced ISnapshots.

```
/// <summary>
/// Sets the playe fields from the incoming deserialized model
/// </summary>
/// <param name="_model">SInventory model containing the deserialized data</param>
public void LoadModel(ISnapshotModel _model) {
    SInventory model = (SInventory)_model;
    _MaxItems = model._MaxItems;
    transform.position = model._Position;
    transform.rotation = model._Rotation;
}

/// <summary>
/// Sets any reference the inventory may have, like its weapons.
/// </summary>
public void RetrieveReferences(int[] _refSmris) {
    _Weapons = new List<Weapon>();
    for (int i = 0; i < _refSmris.Length; i++) {
        _Weapons.Add((Weapon)Common.Instance.SaveManager.Snapshots[_refSmris[i]]);
    }
}

/// <summary>
/// Returns the type of the ISnapshotModel this ISnapshot's data get represented.
/// </summary>
public Type GetSnapshotModelType() {
    return typeof(SInventory);
}
```

## Documentation

Full documentation can be found at the provided package Docs folder.