SENSOR MANAGER PLUG-IN SDK

PROGRAMMER'S GUIDE

April 2018

Adobe

Sensor Manager Plug-in SDK, Programmer's Guide.

Contents

1 Overview 1

2 Plug-in components 2

Plug-in manifest elements 2

Manifest example 4

3 Implementing a converter 5

Global initialization 5
- GetPluginIdentifier() 5
- RegisterConverters () 5

Defining a converter 5

SM_PluginBase 7
- convertToMGJSON() 8
- getExtension() 8
- getFilePath() 8
- getMGJSONPath() 8

PluginFileIO 9

4 MGJSON writer interfaces 10

Group 10

Static data 10

Dynamic data 10

Interfaces 11
- IMgjsonRoot 11
- IMgjsonGroup 12
- IMgjsonStaticData 14
- IMgjsonStaticDataString 14
- IMgjsonStaticDataBool 15
- IMgjsonStaticDataNum 16
- IMgjsonStaticDataNumArr 17
- IMgjsonDynamicData 18
- IMgjsonDynamicDataNum 20
- IMgjsonDynamicDataNumArr 21
- IMgjsonDynamicDataString 23

Error handling 25

SM_PluginException 26

Walk-through writer APIs 26

Sample converter 30

1 Overview

This Sensor Manager Plug-in SDK Guide introduces you to the Sensor Manager Plug-in SDK for Adobe Motion Graphics JSON (MGJSON). The Sensor Manager Plug-in SDK enables developers to write plug-ins for converting their file formats to MGJSON.

Adobe Motion Graphics JSON is a structured data format that is specific to motion graphics. It is flexible enough to be able to represent data that represents motion. It differs from standard JSON in that it has a defined self-referential structure. The structure allows for the definition of streams of data that can represent almost any kind of data.

MGJSON is supported natively in Adobe video products - it can be imported just as you would any other type of footage and added to a Composition timeline. The data streams are then accessible and the values can be accessed and used to drive the values for other elements in the Composition.

NOTE: An understanding of JSON and JSON schemas is assumed and necessary in understanding how data is represented with MGJSON.

NOTE: For SDK components and building steps, refer *Sensor Manager Plug-in SDK Overview Guide*.

2 Plug-in components

The plug-in contains a unique identifier or module ID. This is defined in the resource file `MODULE_IDENTIFIER.txt`, and retrieved by the global function `const char* GetPluginIdentifier()`.

You must modify the resource file to contain the unique identifier for your plug-in and implement the retrieval function to return the same ID that is defined in the resource file.

In addition, the file `SMPLUGINUIDS.txt` contains a plug-in manifest (a plug-in resource file) that describes the plug-in content in `JSON` format using the schema present in `docs/Manifest_Schema.json`.

A single plug-in can contain multiple converters. Each converter can handle a different file format. The plug-in file must have `.mgx` extension.

Plug-in manifest elements

The plug-in manifest contains the following elements:

Converter

One plug-in can have multiple converters. Each converter entry corresponds to a single converter and contains the following:

- Name
- Identifier
- Version
- Extensions
- Check format
- Force converter

Name

A set of one or more elements each of which specifies the name of the converter in different locale. Default locale is `en-US` and is mandatory. `localeID` specifies unique identifier for locale. List of valid locale IDs are specified in `Manifest_Schema.json`. The `value` is name of `PluginConverter` in a given locale.

```
"name": [
       {
       "localeID":"en_US",
       "value":"ConverterName"
       }
   ]
```

Identifier

A unique identifying name for the converter.

```
"identifier":"com.adobe.sensor.tsv"
```

Version

The version number for this converter, such as 1.0.

```
"version":1.0
```

Extensions

A set of one or more elements each of which associates the converter with a file name extension. These extensions are used to select the converters.

```
"extensions": ["tsv","txt"]
```

Check format

(Optional) If a file format can be identified by one or more byte sequences at a fixed location within the file, then this information should be specified to identify the format before actually loading the plug-in. If present, the product loads the plug-in if the format matches extension and passes `checkFormat` validation.

Attributes

`Offset`	The starting offset of the identifying sequence from the beginning of the file, in bytes.
`Length`	The length of the identifying sequence, in bytes.
`ByteSeq`	The specific identifying sequence. Declare the byte sequence either as an ASCII character string or as a hexadecimal number introduced by "0x".

```
"check_format" : [
       {
           "offset":0,
           "length":4,
           "byte_seq": "0xbd22e666"
},
{
"offset":0,
           "length":4,
           "byte_seq": "abcd"
```

```
}
]
```

Force converter

(Optional) It describes whether the converter should overwrite another converter with same identifier and version, if present. Defaults to `false`. In case the other also has `true` for this value, then the first one (as per the loading module of OS) is registered.

```
"force_converter": true
```

Manifest example

A complete manifest file with required fields as defined for the template plug-in is as follows:

```
{
    "converter":[
        {
            "identifier":"com.adobe.sensor.template",
            "name":[
                {
                    "localeID":"en_US",
                    "value":"Template converter"
                }
            ],
            "version":1,
            "extensions":["tmp","temp"]
        }
    ]
}
```

3 Implementing a converter

To implement your own converter, you can modify the provided template to customize the plug-in framework, then define the conversion functionality for each file format.

Global initialization

You must implement these global functions:

- GetPluginIdentifier()
- RegisterConverters ()

GetPluginIdentifier()

Implement this function to return the Module ID as defined in the `MODULE_IDENTIFIER.txt` file. For example:

```
const char* GetPluginIdentifier()
 {
     return "com.adobe.sensor.plugin.template";
}
```

RegisterConverters ()

Implement this function to register your converter classes that are derived from SM_PluginBase. The converter's unique identifier must match the Identifier that you have defined in the manifest. Each converter that a plug-in defines and is intended to be loaded must be registered separately. For example:

```
void RegisterConverters()
 {
   SM_PluginRegistry::registerConverter(
    new SM_PluginCreator<Template_Converter>("com.adobe.sensor.template"));
    SM_PluginRegistry::registerConverter(
    new SM_PluginCreator<Template_Converter2>("com.adobe.sensor.template2"));


}
```

Defining a converter

To implement a converter, derive your converter class from SM_PluginBase and implement the pure virtual methods to provide the functionality of converting native file to MGJSON.

convertToMGJSON()

Implement this method to convert an input file of the format supported by this converter to an MGJSON file. This function should use the WriterAPI Interfaces provided as part of the SDK. These APIs would write the provided data as per the MGJSON schema and produce the output MGJSON file. Return `true` if the conversion for the file is successful, `false` otherwise.

Your class must provide additional static methods that are not defined in the base class:

initialize() and terminate()

In your implementation of these functions, add any initialization and termination code that your converter requires. There is no default initialization or termination behavior.

checkFileFormat()

When the product is looking for a converter to match a file format, it calls the static method `checkFileFormat()`. Your converter must implement this method. Return `true` if the input file matches your file type. The method that does not match your file type should return `false`.

Example class declaration

```
class My_Converter : public SM_PluginBase
    {
    public:
        My_Converter (const SM_NAMESPACE::SM_FileExtension& inExt,
               const SM_NAMESPACE::SM_UTF8String& inFilePath,
               const SM_NAMESPACE::SM_UTF8String& inMGJSONPath);

        virtual ~ My_Converter () {}

        // ----------------------------------------------------------------------
        /// @brief \c convertToMGJSON() convert the native file to mgjson file.
        ///
        /// @return true if success otherwise false.
        // ----------------------------------------------------------------------
        virtual bool convertToMGJSON();

        // ----------------------------------------------------------------------
        /// @brief \c This function is called to initialize the file converter.
        /// It may be an empty function if nothing needs to be initialized.
        ///
        /// @return true if success otherwise false.
        ///
```

```
        // ------------------------------------------------------------------------
        static bool initialize();


        // ------------------------------------------------------------------------
        /// @brief \c This function is called to terminate the file converter.
        ///
        /// @return true if success otherwise false.
        ///
        // ------------------------------------------------------------------------
        static bool terminate();


        // ------------------------------------------------------------------------
        /// @brief \c The following function need to be implemented to check the format.
        ///
        /// @return true if success otherwise false.
        ///
        // ------------------------------------------------------------------------
        static bool checkFileFormat(const SM_NAMESPACE::SM_UTF8String& inFilePath);


    };
```

SM_PluginBase

All new converters must derive from `SM_PluginBase` base class. Some of the methods must or can be specialized to provide your converter functionality, and some provide supporting functionality. The base class defines the following methods:

Method	Description	Implement in plug-in
`convertToMGJSON()`	Implement this method to convert a file of a native format to MGJSON file.	Required
`getExtension()`	Retrieves the extension of the native file to be converted to MGJSON.	No

`getFilePath()`	Retrieves the file path of the native file to be converted.	No
`getMGJSONPath()`	Retrieves the path of the converted MGJSON file.	No

convertToMGJSON()

Implement this method to convert an input file of the format supported by this converter to an MGJSON file. This function should use the WriterAPI Interfaces provided as part of the SDK. These APIs would write the provided data as per the MGJSON schema and produce the output MGJSON file.

```
bool convertToMGJSON();
```

Returns

A boolean value. `true` if the conversion is successful, `false` otherwise.

getExtension()

Retrieves the file extension of the native file to be converted to MGJSON file.

```
const SM_NAMESPACE:SM_FileExtension& getExtension() const;
```

Returns

The file extension.

getFilePath()

Retrieves the file path of the native file to be converted to MGJSON.

```
const SM_NAMESPACE::SM_UTF8String& getFilePath() const;
```

Returns

The absolute path string as UTF8 String.

getMGJSONPath()

Retrieves the path of the output MGJSON file.

```
const SM_NAMESPACE::SM_UTF8String& getMGJSONPath() const;
```

Returns

The absolute path string as UTF8 String.

PluginFileIO

`SM_PluginFileIO.h` provides some of the helper APIs to interact with files. You can use these to parse and process the native file.

4 MGJSON writer interfaces

This section describes the MGJSON writer interfaces that you can use to serialize the data to MGJSON format.

Any data which needs to be converted to MGJSON can be of two types, static and dynamic. The associated static and/or dynamic data can be grouped together and can be nested to multiple levels as per the requirement of hierarchical data representation.

Group

A group can be used to represent any logical grouping of Static and Dynamic data.

Static data

With reference to MGJSON, static data is used to represent non-temporal data. For example, Event Names, Race Number.

```
"displayName": "Event",
      "matchName": "Event",
       "value": {
           "str": "Italian Grand Prix  ",
           "length": 18
       }
```

Dynamic data

With reference to MGJSON, dynamic data is used to represent temporal/time-based data. For example: RPM, Speed, LeanAngle, GPS, etc.

```
"dataDynamicSamples": [{
       "sampleSetID": "00001",
       "samples": [{
           "time": "2017-06-02T12:40:25.390Z",
           "value": "16009"
       }, {
           "time": "2017-06-02T12:40:25.437Z",
           "value": "16167"
       }, {
```

```
            "time": "2017-06-02T12:40:25.453Z",
            "value": "16187"
        }]
}]
```

For more details on static, dynamic and group, refer to `MGJSON_Schema2.0.0.json` in `public/docs`.

Interfaces

In order to serialize the data outline and MGJSON file, you should use IMgjsonRoot. The root represents the collection of static data, dynamic data, group and file level information. If the dynamic data is present, client needs to provide this information in creation of root itself.

In order to serialize Groups containing static data, dynamic data and groups, you should use the IMgjsonGroup. The static, dynamic and group nodes belonging to a particular group node can be added to this object.

In order to serialize the static data, you should use the IMgjsonStaticDataNum, IMgjsonStaticDataBool, IMgjsonStaticDataNumArr and IMgjsonStaticDataString interfaces. The four interfaces are derived from IMgjsonStaticData. Objects of these four classes allow you to create static data objects with properties of the respective types and finally serialize them to MGJSON file.

In order to serialize the dynamic data and samples, you should use the IMgjsonDynamicDataNum, IMgjsonDynamicDataNumArr and IMgjsonDynamicDataString interfaces. The three interfaces are derived from IMgjsonDynamicData. Objects of these three classes allows you to create dynamic data objects with properties of the respective types, add samples to these objects and finally serialize them to MGJSON file.

IMgjsonRoot

This allows creation of root node. Its creation must be the first call among writer APIs. The Commit() must be the last call once all the static data, dynamic data with all the dynamic samples and groups has been added. This interface defines the following methods:

CreateRoot()

Creates a root node. It must be the first and mandatory call among writer APIs. Root is singleton for a converter instance. For one converter instance, `MGJSON path` and `DynamicFileInfo` cannot be changed.

```
spIMgjsonRoot CreateRoot (ConverterRef inConverter,
                SM_NAMESPACE::SM_UTF8FilePath inMGJSONpath,
                const SM_NAMESPACE::SM_DynamicFileInfo* inDynamicFileInfo = NULL );
```

PARAMETERS:

`inConverter`	A pointer of the converter instance.
`inMGJSONpath`	Path of the output MGJSON file
`inDynamicFileInfo`	Structure containing dynamic file info associated with this MGJSON file. This is required only when MGJSON contains dynamic data. In such a case, `SM_TimeInfo` is mandatory and `SM_VideoSyncTimeList` is optional. For more details, see `public/include/SM_PluginTypes.h`.

Returns

A shared pointer to a `SM_PLUGIN::IMgjsonRoot` object.

On Error

`SM_PluginException` is thrown in following cases:

- `inConverter` is NULL.
- `inMGJSONpath` is NULL or empty.

Commit()

Serializes the MGJSON file. This must be the last and mandatory call once all static data, dynamic data with all dynamic samples and groups have been added. Any changes done to the root after this API call would not take any effect. This should be called only once.

```
bool Commit ( );
```

Returns

A boolean value. `true` in case the commit is successful, `false` in case the node was already committed.

On Error

`SM_PluginException` is thrown in the following case:

- There is a failure in serialization of file.

SetCreator()

Sets the value of creator field in MGJSON.

```
void SetCreator ( const SM_NAMESPACE::SM_UTF8String& inVal ) ;
```

PARAMETERS:

`inVal`	Value of the creator field.

IMgjsonGroup

Group can contain static data, dynamic data with sample information and groups. This interface defines the following methods:

CreateGroup()

Creates a group node. The node must be later added to an `MGJSONGroup` or `MGJSONRoot` to be serialized in the MGJSON file.

```
spIMgjsonGroup CreateGroup (ConverterRef inConverter,
                            const SM_NAMESPACE::SM_AsciiString& inMatchName ,
                            const SM_NAMESPACE::SM_UTF8String& inDisplayName );
```

PARAMETERS:

`inConverter`	A pointer of the converter instance.
`inMatchName`	A unique identifier with ASCII, alphanumeric characters, and no leading numerals.
`inDisplayName`	A label used for the stream.

Returns

A shared pointer to a `SM_PLUGIN::IMgjsonGroup` object.

On Error

`SM_PluginException` is thrown in the following cases:

- `inConverter` is NULL.
- `inMatchName` is not valid ASCII.

AddDynamicData()

Adds the dynamic data node to `MGJSONGroup` or `MGJSONRoot`.

```
void AddDynamicData ( const spIMgjsonDynamicData& inDynamic ) ;
```

PARAMETERS:

`inDynamic`	A shared pointer to the `SM_PLUGIN::IMgjsonDynamicData` object.

On Error

`SM_PluginException` is thrown in the following cases:

- `inDynamic` is empty.
- A dirty node is added.

AddStaticData()

Adds the static data node to `MGJSONGroup` or `MGJSONRoot`.

```
void AddStaticData ( const spIMgjsonStaticData& inStatic ) ;
```

PARAMETERS:

`inStatic`	A shared pointer to the `SM_PLUGIN::IMgjsonStaticData` object.

On Error

`SM_PluginException` is thrown in the following case:

- `inStatic` is empty.

AddGroup()

Adds the group node to `MGJSONGroup` or `MGJSONRoot`.

```
void AddGroup ( const spIMgjsonGroup& inGroup ) ;
```

PARAMETERS:

`inGroup`	A shared pointer to the `SM_PLUGIN::IMgjsonGroup` object.

On Error

`SM_PluginException` is thrown in the following case:

- `inGroup` is empty.

IMgjsonStaticData

All the four static data classes are derived from this class. Each instance of a derived class would correspond to a static data value. It contains the methods which can be called on objects of all the four static data classes. This interface defines the following method:

GetType()

Returns the type of `StaticData` node.

```
SM_NAMESPACE::eDataValueType GetType() const;
```

Returns

The type of `StaticData` node. `eDataValueType` can be `kSM_ValueType_NUMBER`, `kSM_ValueType_STRING`, `kSM_ValueType_NUMBER_ARRAY`, or `kSM_ValueType_BOOLEAN`.

IMgjsonStaticDataString

This class is derived from IMgjsonStaticData. It contains static data value of type string. The interface defines the following method:

CreateStaticDataString()

Creates a `StaticData` node of type string. The node must be later added to `MGJSONGroup` or `MGJSONRoot` to be serialized in the MGJSON file.

```
spIMgjsonStaticDataString CreateStaticDataString (ConverterRef inConverter,
              const SM_NAMESPACE::SM_AsciiString& inMatchName ,
              const SM_NAMESPACE::SM_UTF8String& inDisplayName,
              const SM_NAMESPACE::SM_UTF8String& inValue,
              const SM_NAMESPACE::SM_StringProperties& inStringProp =
                     SM_NAMESPACE::SM_StringProperties());
```

PARAMETERS:

`inConverter`	A pointer of the converter instance.
`inMatchName`	A unique identifier with ASCII, alphanumeric characters, and no leading numerals.
`inDisplayName`	A label used for the stream.
`inValue`	The value to be serialized as string.
`inStringProp`	*(Optional)* Structure containing `StringProperties` associated with this node. The properties would be calculated if not provided. Otherwise, the provided properties are used and serialized. For more details, refer `public/include/SM_DataTypesCommon.h`.

Returns

A shared pointer to the `SM_PLUGIN::IMgjsonStaticDataString` object.

On Error

`SM_PluginException` is thrown in the following cases:

- `inConverter` is NULL.
- `inMatchName` is not valid ASCII.
- One of `mMaxLen` or `mMaxLenValDigit` is not provided.

IMgjsonStaticDataBool

This class is derived from IMgjsonStaticData. It would contain static data value of type boolean. The interface defines the following method:

CreateStaticDataBool()

Creates a `StaticData` node of type boolean. The node must be later added to `MGJSONGroup` or `MGJSONRoot` to be serialized in the MGJSON file.

```
spIMgjsonStaticDataBool CreateStaticDataBool (ConverterRef inConverter,
              const SM_NAMESPACE::SM_AsciiString& inMatchName ,
              const SM_NAMESPACE::SM_UTF8String& inDisplayName,
              const bool& inValue );
```

PARAMETERS:

`inConverter`	A pointer of the converter instance.
`inMatchName`	A unique identifier with ASCII, alphanumeric characters, and no leading numerals.
`inDisplayName`	A label used for the stream.
`inValue`	The value to be serialized as boolean.

Returns

A shared pointer to the `SM_PLUGIN::IMgjsonStaticDataBool` object.

On Error

`SM_PluginException` is thrown in the following cases:

- `inConverter` is NULL.
- `inMatchName` is not valid ASCII.

IMgjsonStaticDataNum

This class is derived from IMgjsonStaticData. It contains static data value of type number. The interface defines the following method:

CreateStaticDataNum()

Creates a `StaticData` node of type number. The node must be later added to `MGJSONGroup` or `MGJSONRoot` to be serialized in the MGJSON file.

```
spIMgjsonStaticDataNum CreateStaticDataNum (ConverterRef inConverter,
              const SM_NAMESPACE::SM_AsciiString& inMatchName ,
              const SM_NAMESPACE::SM_UTF8String& inDisplayName,
              const double& inValue,
              const SM_NAMESPACE::SM_NumberProperties& inNumberProp =
                     SM_NAMESPACE::SM_NumberProperties());
```

PARAMETERS:

`inConverter`	A pointer of the converter instance.
`inMatchName`	A unique identifier with ASCII, alphanumeric characters, and no leading numerals.
`inDisplayName`	A label used for the stream.
`inValue`	The value to be serialized as number.
`inNumberProp`	*(Optional)* Structure containing `NumberProperties` associated with this node. The properties would be calculated if not provided. Otherwise, the provided properties are used and serialized. For more details, refer `public/include/SM_DataTypesCommon.h`.

Returns

A shared pointer to the `SM_PLUGIN::IMgjsonStaticDataNum` object.

On Error

`SM_PluginException` is thrown in the following cases:

- `inConverter` is NULL.
- `inMatchName` is not valid ASCII.
- One of `mDigitsInteger` or `mDigitsDecimal` is not provided.
- One of `mMinimum` and `mMaximum` is provided in `mOccuring` or `mLegal` and other is not.

IMgjsonStaticDataNumArr

This class is derived from IMgjsonStaticData. It contains static data value of type number array. Currently only upto three dimensional data is supported, so the number of elements must not be greater than 3. The interface defines the following method:

CreateStaticDataNumArr()

Creates a `StaticData` node of type number array. The node must be later added to `MGJSONGroup` or `MGJSONRoot` to be serialized in the MGJSON file.

```
spIMgjsonStaticDataNumArr CreateStaticDataNumArr (ConverterRef inConverter,
              const SM_NAMESPACE::SM_AsciiString& inMatchName ,
              const SM_NAMESPACE::SM_UTF8String& inDisplayName,
              const std::vector<double>& inValue,
              const SM_NAMESPACE::SM_NumberArrayProperties& inNumberArrayProp =
                     SM_NAMESPACE::SM_NumberArrayProperties());
```

PARAMETERS:

`inConverter`	A pointer of the converter instance.
`inMatchName`	A unique identifier with ASCII, alphanumeric characters, and no leading numerals.
`inDisplayName`	A label used for the stream.
`inValue`	The value to be serialized as number.
`inNumberArrayProp`	Structure containing `NumberArrayProperties` associated with this node. All other properties are optional. The optional properties are calculated if not provided. Otherwise, the provided properties are used and serialized.

Returns

A shared pointer to the `SM_PLUGIN::IMgjsonStaticDataNumArr` object.

On Error

`SM_PluginException` is thrown in the following cases:

- `inConverter` is NULL.
- `inMatchName` is not valid ASCII.
- If `mCount` is provided and is greater than 3 or does not match with the size of `inValue`.
- If `inValue` size is greater than 3.
- One of `mDigitsInteger` or `mDigitsDecimal` is not provided.
- If `mRanges` is not empty and the size differs from `mCount`.
- In case `mRanges` is not empty and both `mMinimum` and `mMaximum` of `mOccuring` are not provided for all the elements.
- In case `mRanges` is not empty and both `mMinimum` and `mMaximum` of `mLegal` are not provided for all the elements.
- If `mDisplayNames` is not empty and the size differs from `mCount`.

IMgjsonDynamicData

All the three dynamic data classes are derived from this class. Each of the derived class would contain the dynamic info data and the sample values along with the time stamp. Each instance of a derived class would correspond to a dynamic sample set. This interface contains the methods which can be called on objects of all the three dynamic data classes.

This interface defines the following methods:

Commit()

Serializes all added samples to the MGJSON file. This must be the last and mandatory call on any `DynamicData` node. Any changes done to the node after this API call would not take any effect. This should be called only once for each `DynamicData` node.

```
bool Commit();
```

Returns

A boolean value. `true` in case the commit is successful, `false` in case the node was already committed.

On Error

`SM_PluginException` is thrown in the following cases:

- A node marked as dirty is committed.
- There is a failure in serialization of samples.

SetSampleCount()

Overrides the value of `mSampleCount` passed at the time of creation of node and the provided value is serialized. In case 0 is passed, the calculated count of valid samples is written.

```
void SetSampleCount(SM_NAMESPACE::SM_Uns64 inSampleCount);
```

PARAMETERS:

`inSampleCount`	A number of samples in this sample set.

GetSampleCount()

Returns the calculated count of valid samples successfully added in the dynamic data node.

```
SM_NAMESPACE::SM_Uns64 GetSampleCount() const;
```

Returns

Returns the count of valid samples.

GetType()

Returns the type of dynamic data node.

```
SM_NAMESPACE::eDataValueType GetType() const;
```

Returns

Returns an object of the type `eDataValueType`. It can be one of:

`kSM_ValueType_NUMBER`, `kSM_ValueType_STRING`, or `kSM_ValueType_NUMBER_ARRAY`.

IMgjsonDynamicDataNum

This class is derived from IMgjsonDynamicData. It contains dynamic data info and sample values of type `numberString`.

This interface defines the following methods:

CreateDynamicDataNum()

Creates a `DynamicData` node of type `numberString`. The node must be later added to `MGJSONGroup` or `MGJSONRoot` to be serialized in the MGJSON file.

```
spIMgjsonDynamicDataNum CreateDynamicDataNum(ConverterRef inConverter,
              const SM_NAMESPACE::SM_DynamicDataFields& inDynamicDataFields,
              const SM_NAMESPACE::SM_NumberProperties& inNumberProp =
                     SM_NAMESPACE::SM_NumberProperties());
```

PARAMETERS:

`inConverter`	A pointer of the converter instance.
`inDynamicDataFields`	Structure containing Dynamic Data Fields associated with this node. `mDisplayName`, `inMatchName`, `mSampleSetID`, `mInterpolationType` and `mHasExpectedSampleFrequencyB` are mandatory. `mSampleCount`, `mExpectedMaxInterSampleDuration` are optional. If `mSampleCount` is provided with default value of 0, then the calculated count of valid added samples is serialized. Otherwise, the provided value is serialized. `mTemporalExtent` is also optional. It is not serialized in MGJSON. For more details, refer `public/include/SM_DataTypesCommon.h`.
`inNumberProp`	*(Optional)* Structure containing `NumberProperties` associated with this node. The properties are calculated if not provided. Otherwise, the provided properties are used and serialized. For more details, refer `public/include/SM_DataTypesCommon.h`.

Returns

A shared pointer to the `SM_PLUGIN::IMgjsonDynamicDataNum` object.

On Error

`SM_PluginException` is thrown in the following cases:

- `inConverter` is NULL.
- `inDynamicFileInfo` is not provided in CreateRoot() signifying that Dynamic data would not be present.
- CreateRoot() with `inConverter` is not called before this call.
- `inMatchName` is not valid ASCII.
- `mInterpolationType` is not a valid value.
- One of `mDigitsInteger` or `mDigitsDecimal` is not provided.

- One of mMinimum and mMaximum is provided in mOccuring or mLegal and other is not.

AddSample()

Adds the time-value pair to the DynamicData node. After adding all samples to the DynamicData node, Commit() API must be called in order to serialize the samples to the MGJSON file.

```
bool AddSample(const SM_NAMESPACE::SM_TimeValue& inTime,
               const double& inVal);
```

PARAMETERS:

inTime	One of TimeUTC or TimeSMPTE structures. The time format must be same as that is given in inDynamicFileInfo in CreateRoot(). For more details, refer public/include/SM_PluginTypes.h.
inVal	The value to be serialized as numberString corresponding to the inTime.

Returns

A boolean value. true in case the sample is added successfully. False is returned in case the time format or properties of time such as precisionLength differs from the one set in inDynamicFileInfo in MGJSONRoot.

In case false is returned, sample is not added.

On Error

SM_PluginException is thrown in the following cases:

- There is a failure in calculation of properties.
- There is a failure in storing/serialization of samples.
- The node is already dirty.

IMgjsonDynamicDataNumArr

This class is derived from IMgjsonDynamicData. It contains dynamic data info and sample values of type numberStringArray. Currently only upto three dimensional data is supported, so the number of elements must not be greater than 3.

This interface defines the following methods:

CreateDynamicDataNumArr()

Creates a DynamicData node of type numberStringArray. The maximum number of elements allowed in a number array is 3. The node must be later added to MGJSONGroup or MGJSONRoot to be serialized in the MGJSON file.

```
static spIMgjsonDynamicDataNumArr CreateDynamicDataNumArr(
              ConverterRef inConverter,
              const SM_NAMESPACE::SM_DynamicDataFields& inDynamicDataFields,
              const SM_NAMESPACE::SM_NumberArrayProperties& inNumberArrayProp);
```

PARAMETERS:

inConverter	A pointer of the converter instance.
inDynamicDataFields	Structure containing Dynamic Data Fields associated with this node. mDisplayName, inMatchName, mSampleSetID, mInterpolationType and mHasExpectedSampleFrequencyB are mandatory. mSampleCount, mExpectedMaxInterSampleDuration are optional. If mSampleCount is provided with default value of 0, then the calculated count of valid added samples is serialized. Otherwise, the provided value is serialized. mTemporalExtent is also optional. It is not serialized in MGJSON. For more details, refer public/include/SM_DataTypesCommon.h.
inNumberArrayProp	Structure containing NumberArrayProperties associated with this node. mCount is mandatory. All other properties are optional. The optional properties are calculated if not provided. Otherwise, the provided properties are used and serialized. For more details, refer public/include/SM_DataTypesCommon.h.

Returns

A shared pointer to the SM_PLUGIN::IMgjsonDynamicDataNumArr object.

On Error

SM_PluginException is thrown in the following cases:

- inConverter is NULL.
- inDynamicFileInfo is not provided in CreateRoot() signifying that dynamic data is not present.
- CreateRoot() with inConverter is not called before this call.
- inMatchName is not valid ASCII.
- mInterpolationType is not a valid value.
- If mCount is not provided and is left to default value of 0 or is greater than 3.
- One of mDigitsInteger or mDigitsDecimal is not provided.
- If mRanges is not empty and the size differs from mCount.
- In case mRanges is not empty and both mMinimum and mMaximum of mOccuring are not provided for all the elements.
- In case mRanges is not empty and both mMinimum and mMaximum of mLegal are not provided for all the elements.
- If mDisplayNames is not empty and the size differs from mCount.

AddSample()

Adds the time-value pair to the `DynamicData` node. After adding all the sample to the `DynamicData` node, Commit() API must be called in order to serialize the samples to the MGJSON file.

```
bool AddSample(const SM_NAMESPACE::SM_TimeValue& inTime,
               const std::vector<double>& inVal);
```

PARAMETERS:

`inTime`	One of `TimeUTC` or `TimeSMPTE` structures. The time format must be same as that given in `inDynamicFileInfo` in CreateRoot(). For more details, refer `public/include/SM_PluginTypes.h`.
`inVal`	The value to be serialized as `numberArrayString` corresponding to the `inTime`.

Returns

A boolean value. `true` in case the sample is added successfully. `false` is returned in the following cases:

- The time format or properties of time such as `precisionLength` differs from the one set in `inDynamicFileInfo` in `MGJSONRoot`.
- The size of `inVal` does not match `mCount`.

In case `false` is returned, sample is not added.

On Error

`SM_PluginException` is thrown in the following cases:

- There is a failure in calculation of properties.
- There is a failure in storing/serialization of samples.
- The node is already dirty.

IMgjsonDynamicDataString

This class is derived from IMgjsonDynamicData. It contains dynamic data info and sample values of type string.

This interface defines the following methods:

CreateDynamicDataString()

Creates a `DynamicData` node of type `paddedString`. The node must be later added to `MGJSONGroup` or `MGJSONRoot` to be serialized in the MGJSON file.

```
static spIMgjsonDynamicDataString CreateDynamicDataString(
              ConverterRef inConverter,
              const SM_NAMESPACE::SM_DynamicDataFields& inDynamicDataFields,
              const SM_NAMESPACE::SM_StringProperties& inStringProp =
                     SM_NAMESPACE::SM_StringProperties());
```

PARAMETERS:

`inConverter`	A pointer of the converter instance.
`inDynamicDataFields`	Structure containing Dynamic Data Fields associated with this node. `mDisplayName`, `inMatchName`, `mSampleSetID`, `mInterpolationType` and `mHasExpectedSampleFrequencyB` are mandatory. `mSampleCount`, `mExpectedMaxInterSampleDuration` are optional. If `mSampleCount` is provided with default value of 0, then the calculated count of valid added samples is serialized. Otherwise, the provided value is serialized. `mTemporalExtent` is optional. It is not serialized in MGJSON. For more details refer, `public/include/SM_DataTypesCommon.h`.
`inStringProp`	*(Optional)* Structure containing `StringProperties` associated with this node. The properties are calculated if not provided. Otherwise, the provided properties are used and serialized. For more details, refer `public/include/SM_DataTypesCommon.h`.

Returns

A shared pointer to the `SM_PLUGIN::IMgjsonDynamicDataString` object.

On Error

`SM_PluginException` is thrown in the following cases:

- `inConverter` is NULL.
- `inDynamicFileInfo` is not provided in CreateRoot() signifying that dynamic data is not present.
- CreateRoot() with this `inConverter` is not called before this call.
- `inMatchName` is not valid ASCII.
- `mInterpolationType` is not a valid value.
- One of `mMaxLen` and `mMaxLenValDigit` is provided and other is not.

AddSample()

Adds the time-value pair to the `DynamicData` node. After adding all the sample to the `DynamicData` node, Commit() API must be called in order to serialize the samples to the MGJSON file.

```
bool AddSample(const SM_NAMESPACE::SM_TimeValue& inTime,
               const SM_NAMESPACE::SM_UTF8String& inVal);
```

PARAMETERS:

`inTime`	One of `TimeUTC` or `TimeSMPTE` structures. The time format must be same as that given in `inDynamicFileInfo` in CreateRoot(). For more details, refer `public/include/SM_PluginTypes.h`.
`inVal`	The value to be serialized as string corresponding to the `inTime`.

Returns

A boolean value. `true` in case the sample is added successfully. `false` is returned in case the time format or properties of time such as `precisionLength` differs from the one set in `inDynamicFileInfo` in `MGJSONRoot`. In case false is returned, sample is not added.

On Error

`SM_PluginException` is thrown in the following cases:

- There is a failure in calculation of properties.
- There is a failure in storing/serialization of samples.
- The node is already dirty.

Note: None of the writer APIs is thread safe.

Note: In case `SM_PluginException` is thrown from `AddSample()` or Commit(), `DynamicData` node is marked as dirty and is invalidated.

Error handling

All of the MGJSON Writer APIs which are not marked as `__NOTHROW__` can throw exception of type SM_PluginException under various scenarios. You can take an appropriate action by catching the exceptions or allow them to propagate to the product.

The various methods which are to be implemented in the plug-in and the parsing of native file can also throw exceptions of the type `SM_PluginException`.

For throwing an exception you should use the following macros defined in `\public\include\SM_PluginTypedefs.h`:

- `THROW_PLUGIN_EXCEPTION(errId)`: Recommended for throwing any exception.
- `THROW_PLUGIN_EXCEPTION_IN_PARSING(errId, offset, lineNo)`: Recommended for throwing exceptions occurring during parsing of native files so that the offset and line number of the native file can also be provided.

The `errId` is of type `SM_NAMESPACE::eErrorCode` present in SM_PluginException. You can use the existing `PLUGIN ERROR CODES` and `WRITER ERROR CODES` present in `public\include\SM_Const.h`.

You can also add your own error codes after `kSMClientPlugins_ErrorCode`.

SM_PluginException

```
class SM_PluginException  {

    public:


        SM_PluginException(SM_NAMESPACE::eErrorCode errorID,
                           const SM_NAMESPACE::SM_Uns64& fileOffset = 0,
                           const SM_NAMESPACE::SM_Uns64& lineNo = 0) :
                           mErrorID(errorID),mFileOffset(fileOffset),
                           mLineNo(lineNo) {};


        /// Retrieves the numeric code from an SM_PluginException.

        inline SM_NAMESPACE::eErrorCode GetErrorCode() const { return mErrorID; };


        /// Retrieves the file offset, if any

        inline const SM_NAMESPACE::SM_Uns64& GetFileOffset() const { return mFileOffset; }



        /// Retrieves the file line no., if any

        inline const SM_NAMESPACE::SM_Uns64& GetFileLineNo() const { return mLineNo; }



    private:

        /// Error codes. See SM_Const.h.

        SM_NAMESPACE::eErrorCode mErrorID;


         /// fileOffset offset in file where error occurs, can be used during
         /// parsing of native file

        SM_NAMESPACE::SM_Uns64 mFileOffset;


        /// line number where error occurs, can be used during parsing of native file.

        SM_NAMESPACE::SM_Uns64 mLineNo;

    };
```

Walk-through writer APIs

```
//structure of dynamic fields

SM_NAMESPACE::SM_DynamicFileInfo  dynamicInfo;

dynamicInfo.mTimeInfo = SM_NAMESPACE::SM_UTCInfo(3, true);
```

```
// create root
spIMgjsonRoot root = IMgjsonRoot_v1::CreateRoot(this, getMGJSONPath().c_str(),
                                                &dynamicInfo);
root->SetCreator("Sample code");



// Add static data node
spIMgjsonStaticData staticdata1 = IMgjsonStaticDataString_v1::CreateStaticDataString(
                                  this, "Season", "Season", "2018");
root->AddStaticData(staticdata1);

spIMgjsonStaticData staticdata2 = IMgjsonStaticDataNum_v1::CreateStaticDataNum(
                                  this, "RaceNumber", "Racer", 6);
root->AddStaticData(staticdata2);

spIMgjsonStaticData staticdata3 = IMgjsonStaticDataBool_v1::CreateStaticDataBool(
                                  this, "clearWeather", "clearWeather", true);
root->AddStaticData(staticdata3);

spIMgjsonStaticData staticdata4 = IMgjsonStaticDataNumArr_v1::CreateStaticDataNumArr(
                                  this, "Teammates", "Teammates",
                                  std::vector<double>{8, 17, 21 });
root->AddStaticData(staticdata4);

// create group

spIMgjsonGroup group = IMgjsonGroup_v1::CreateGroup(this, "telemetry", "telemetry");

/*****************************************************************/

//Add dynamic samples of type number
SM_NAMESPACE::SM_DynamicDataFields inDynamicDataFields1("Speed", "Speed", "001",
SM_NAMESPACE::kSM_InterpolationType_LINEAR, 0);

//specifying legal range
SM_NAMESPACE::SM_NumberProperties
numProp(SM_NAMESPACE::SM_NumberRange(SM_NAMESPACE::SM_NumberMinMax(0, 200)));
```

```
spIMgjsonDynamicDataNum dynamicNumData = IMgjsonDynamicDataNum_v1::CreateDynamicDataNum(
                                          this, inDynamicDataFields1, numProp);


/*     Add two samples of type number

*      time:2017-06-02T12:40:25.000Z, value: 166

*      time:2017-06-02T12:40:26.000Z, value: 168

*/


SM_NAMESPACE::SM_TimeValue time1 = SM_NAMESPACE::SM_DateTimeUTC(2017, 06, 02, 12,
40, 25, 'Z', 0, 0, 0);

SM_NAMESPACE::SM_TimeValue time2 = SM_NAMESPACE::SM_DateTimeUTC(2017, 06, 02, 12,
40, 26, 'Z', 0, 0, 0);


dynamicNumData->AddSample(time1, 166);

dynamicNumData->AddSample(time2, 168);


SM_NAMESPACE::SM_Uns64 cnt = dynamicNumData->GetSampleCount();


if(cnt > 0) {

    dynamicNumData->SetSampleCount(cnt);

    dynamicNumData->Commit();

    //Add dynamic node in a group

    group->AddDynamicData(dynamicNumData);

}


/*****************************************************************/


//Add dynamic samples of type string

SM_NAMESPACE::SM_DynamicDataFields inDynamicDataFields2("Gear", "Gear", "002",
SM_NAMESPACE::kSM_InterpolationType_HOLD, 0, 2);

spIMgjsonDynamicDataString dynamicStringData =
IMgjsonDynamicDataString_v1::CreateDynamicDataString(this, inDynamicDataFields2);


/*     Add two samples of type number

*      time:2017-06-02T12:40:25.000Z, value: Fourth

*      time:2017-06-02T12:40:26.000Z, value: Fifth

*/
```

```
dynamicStringData->AddSample(time1, "Fourth");

dynamicStringData->AddSample(time2, "Fifth");

dynamicStringData->Commit();

group->AddDynamicData(dynamicStringData);


/*****************************************************************/


//Add dynamic samples of type number array

SM_NAMESPACE::SM_DynamicDataFields inDynamicDataFields3("GPS (Lat\/Lon)", "GPS",
"003", SM_NAMESPACE::kSM_InterpolationType_LINEAR, 1);

SM_NAMESPACE::SM_NumberArrayProperties prop(2, 0, SM_NAMESPACE::SM_NumberArrayRange(),
SM_NAMESPACE::SM_StringArray { "Latitude", "Longitude" });

spIMgjsonDynamicDataNumArr dynamicNumArrData =
IMgjsonDynamicDataNumArr_v1::CreateDynamicDataNumArr(this, inDynamicDataFields3,
prop);


/*     Add two samples of type number

*      time:2017-06-02T12:40:25.000Z, value: 043.99416346, 011.36938006

*      time:2017-06-02T12:40:26.000Z, value: 043.99418535, 011.36939430

*/

dynamicNumArrData->AddSample(time1, std::vector<double>{ 043.99416346, 011.36938006
});

dynamicNumArrData->AddSample(time2, std::vector<double>{ 043.99418535, 011.36939430
});

dynamicNumArrData->Commit();

group->AddDynamicData(dynamicNumArrData);


/*****************************************************************/


root->AddGroup(group);

root->Commit();


// client can process the exception thrown by writer apis.

// client can throw any exception using the macro THROW_PLUGIN_EXCEPTION.

/* client can throw the exception occuring during the native file parsing using the
macro THROW_PLUGIN_EXCEPTION_IN_PARSING so that the offset and line number of the
native file could also be provided.

*/
```

Sample converter

The SDK provides a `JSONConverter` as a sample. The sample `JSONConverter` derives from SM_PluginBase and implements the required static methods: `initialize()`, `terminate()`, and `checkFileFormat` and other required virtual method: `convertToMGJSON()`.

For building steps refer *Sensor Manager Plug-in SDK Overview Guide*.

Refer `JsonConverter\include\JsonConverter.h` and `JsonConverter\source\JsonConverter.cpp` for the implementation details.

***Important**: The sample JSON converter uses rapidjson to parse JSON content, you may use another JSON parser if you choose. Adobe is not responsible for any vulnerabilities in rapidjson.*