Video Metadata Framework
Functional Specification
with Usage Examples

Version 3.0
June 2nd, 2016

Revision History

Date	Version	Description	Author
2013-10-31	0.1	Created.	Intel Corporation.
2014-01-05	1.0	Updated for VMF 1.0 release.	Intel Corporation.
2014-12-02	2.0	Updated for VMF 2.0 release.	Intel Corporation.
2016-06-02	3.0	Updated for VMF 3.0 release.	Intel Corporation

Table of Contents

1. Introduction

This document provides a functional specification for Video Metadata Framework (VMF) SDK, which provides C++ programmers with functionality for creating, editing, storing and sharing metadata related to videos.
Metadata is data that describes the characteristics or properties of a resource. VMF SDK allows developers to define and create metadata of any form, and about any resource type. Intel VMF SDK embeds metadata within the host video file or video stream, so that no additional files are needed to store the metadata.

1.1 Purpose

The purpose of this document is to provide developers with an overview of the programming interface of Intel VMF SDK, with a few examples explaining how this SDK can be used in a consumer software.

1.2 Scope

This document describes the programming interface of VMF SDK.
The XMP metadata framework is used to implement the low-level I/O of metadata. The programming interface, or the design, of XMP are not described in this document.

1.3 Definitions, Acronyms and Abbreviations

VMF: Video Metadata Framework, developed by Corporation.
Metadata: All possible information associated with a media file (or parts of the file) which can be incorporated into the media file with using Video Metadata Framework.
Schema: A set of metadata descriptions that define names of metadata, and field names and field types of each metadata. A schema is usually considered as the metadata of metadata.
Descriptor: A structure that defines the name and format of a metadata. The information included in a metadata descriptor includes the name of the metadata, and the value type of the metadata, or the name of the fields and value types of the fields.
Reference: Connections or associations between two or multiple metadata. References are used in VMF to define one to one, one to many and many to many relationships.
XMP: Extensible Metadata Platform is an ISO standard, originally created by Adobe Systems Inc., for creating, processing and interchange of standardized and custom metadata for all kinds of resources. The first version of VMF implementation is built on top of XMP library [XMP-wiki].
XML: Extensible Markup Language

1.4 License

1.5 References

[XMP-wiki]: http://en.wikipedia.org/wiki/Extensible_Metadata_Platform

2. Requirements

This section describes the system and software requirements for using the VMF SDK.

2.1 System Requirements

The VMF SDK is developed as a platform-independent library. It supports the following operating system:

- Microsoft Windows system (version 7 or later)
- GNU Linux
- Google Android system (version 2.3 or later)

2.2 C++ Compiler and Standard

The VMF SDK and the samples are written in C++ language. The code uses some C++ 11 features, which include:

- Auto
- Nullptr, shared_ptr
- Rvalue references and move semantics
- Lambda expressions
- Range for
- Std::thread

The source code and the samples require one of the following C++ compilers:

- C++ compiler 12.1 and newer versions
- Microsoft Visual C++ 2013 and newer versions
- GNU GCC 4.7 and newer versions

3. Overview

This section describes the main features and common applications of VMF SDK.

3.1 Features

The VMF SDK provides a software framework for digital story telling applications to manage metadata within video files. This section describes the main functionalities of the framework.

3.1.1 Metadata Definition and Creation

Metadata data is data about data. The format of metadata can be of any form: a metadata could contain a single value integer, an array of strings, or a complex structure that contains multiple fields, each of which has a different name and value type. The VMF SDK allows developers to define metadata of any form, and allows different types of metadata to be managed by a uniform metadata interface.

The VMF SDK also provides a mechanism that allows metadata to be defined before using. A metadata schema declares metadata formats by defining a set of metadata descriptors. A metadata descriptor defines the name of a metadata type, and the names and value types of all the fields within the metadata. Each instance of metadata is then created based on the associated metadata descriptor, by using the descriptor as a templates to create the internal fields.

The metadata schema mechanism enables sharing of metadata between multiple applications, when the same set of metadata schemas are used by multiple applications.

The metadata schema mechanism also allows metadata defined by one company to be used by applications created by another company.

3.1.2 Metadata Saving and Loading

Once metadata are created for a video sequence or video file, the VMF SDK allows the metadata to be saved within the video, by embedding metadata as part of the video file data. Metadata are carried along with the video when transferring the video from one device to another, or from device to Internet. Since the metadata are embedded within the video file, there is no need to use any auxiliary file for metadata.

Please note that VMF does not generate a new video format for storing the metadata and video, nor does it break the existing video format standards. The video files contained VMF metadata are all standard format of videos, which can be opened by any video players. The VMF supports most commonly used video formats: AVI, mp4 and WAV.

When loading metadata from a video file, VMF allows all metadata to be loaded at once, or a group of metadata to be loaded.

3.1.3 Metadata Querying

The VMF SDK provides multiple querying functions that allow developers to build complex querying function. Some of the functions allow the developers to query about the metadata type or value. Others allow the developers to query the relationships between metadata. The generic query function provided by VMF allows developers to define their own query logic.

The VMF SDK allows multiple querying functions to be used in a sequential way: later querying functions refining the results from early querying functions.

3.1.4 Metadata Shifting and Merging

Digital story telling applications, like other video editing applications, usually need to trim video sequence by cutting off frames from original video files to remove unnecessary video segments. Video summarization applications need to extract video segments from raw video sequences and for a new video.

Those video editing operations (cutting, shifting, merging) typically affect the metadata embedded in the original videos. Some metadata need to be associated with a different frame indices, due to the change of the frame indices. Other metadata may become invalid, due to removal of the video frames associated to the metadata.

The VMF SDK provides functions that allow developers to shift metadata within a video sequence after trimming of a video. The SDK also allows metadata from multiple videos to be merged into a new metadata stream, after merging multiple video files into a new video.

3.1.5 Metadata Serialization

In addition to metadata embedding into media file containers it's possible to serialize the whole MetadataStream or its part (e.g. a single metadata record or a schema description) into s string representation. It can be useful in various scenarios, e.g. for metadata streaming over network.

There is a built-in support for XML and JSON text formats and a possibility for adding user-provides ones.
See the Format class and its derivatives for details.

3.1.6 Metadata Compression

It's possible to compress the whole MetadataStream before saving or serialization. In this case the final string representation of the MetadataStream is processed by either built-in or user-provided algorithm then encoded with base64 and stored as a single metadata record.
See Compressor class and its derivatives and 'compression' sample for details.

3.1.7 Metadata Encryption

It's possible to apply encryption of metadata on various levels before saving or serialization:

- the **specified field** value of the **specified metadata** record
- the entire **specified metadata** record (except frame/time info)
- the **specified field** values of all the specified **metadata channel**
- the specified **metadata channel**
- the specified **metadata schema**
- the **whole metadata**

In this case the final string representation of the required metadata amount is processed by either built-in or user-provided encryption algorithm then encoded with base64 and stored as a special field or even a single metadata record in case of whole metadata encryption.
See Encryptor class and its derivatives and 'encryption sample for details.
NOTE: if both compression and encryption should be used together then compression should happen first for the sake of efficiency.

3.1.8 Metadata Statistics

The dediacetd Stat object coalesces element data from a metadata schema using any of the predefined statistical operations or user-provided code (predefined operations are: min, max, average, count, sum, last value).
Only metadata additions to the stream are processed, if metadata change or remove happened then MetadataStream:: recalcStat() needs to be called in order to clear the existing statistics and re-calculate it again using all the existing metadata items.
The Stat object supports 3 update modes:

- on metadata change
- on specified time interval
- by request (can be sync and async)

The Stat object can stored as a part of MetadataStream (note: user-provided operations may not be registered when loading it back).

3.2 Applications

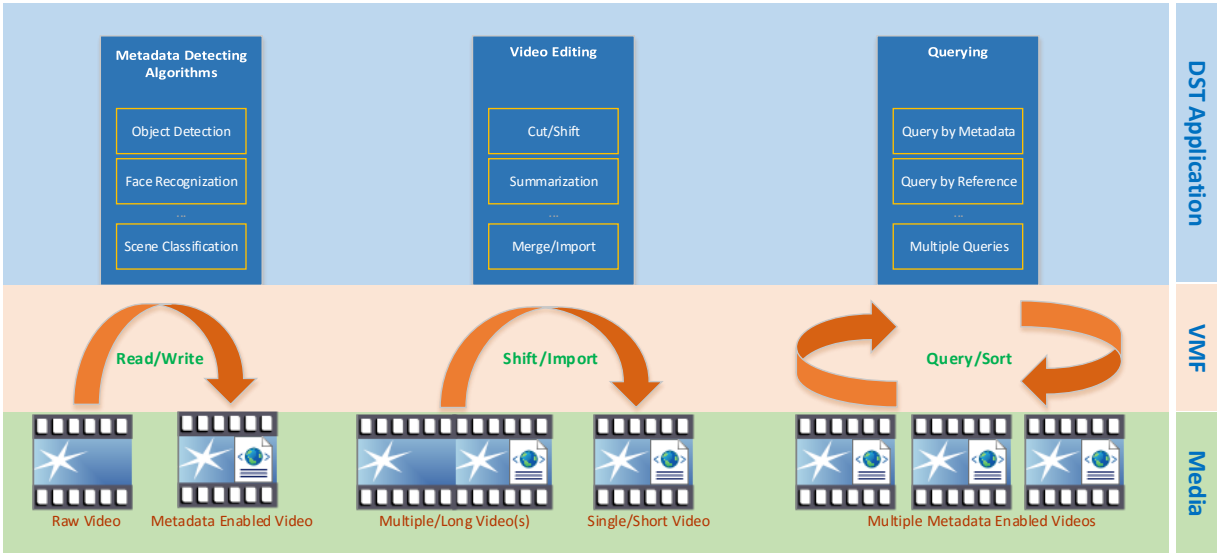
This section describes typical ways of using VMF in digital story telling applications.

3.2.1 Single Application

Most video editing applications or digital story telling applications can benefit from using VMF SDK. Those benefits include:

- Provides storage of metadata, and I/O of metadata to/from video files.
- Allows DST applications to update metadata due to changes made to video frames. The changes can be trimming of the video, or shifting of the video.
- Allows DST applications to import metadata from one video to another video, or merge metadata from multiple videos to a new video.
- Allows DST applications to sort and query metadata.

The following graph illustrates the above roles of VMF SDK in a typical stand-alone digital story telling application. The graph is split into three layers: the top layer consists the DST application logic or algorithms; the middle layer is the VMF library; the bottom layer stands for the media files or in-memory media objects.



3.2.2 Creator-Consumer Model

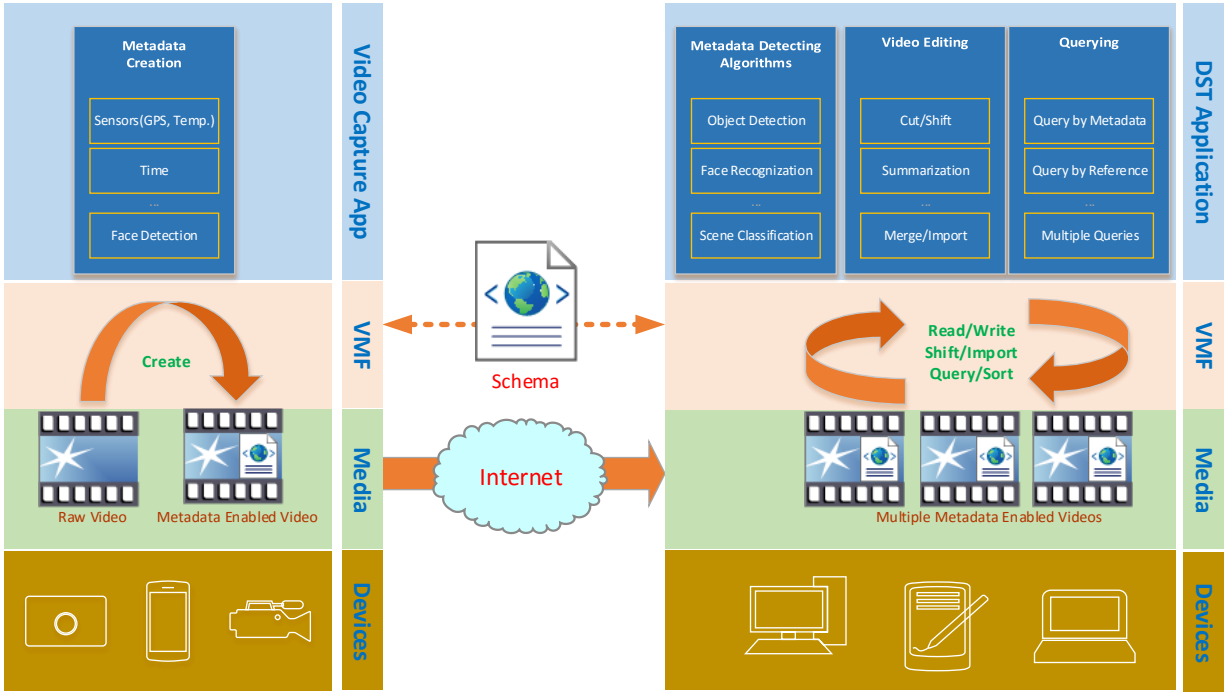
The creator-consumer model is a more common architecture for digital story telling applications to use metadata. In this model, two applications are involved: a metadata creator app, and a metadata consumer application or digital story telling applications.

The metadata creator app is an app running on mobile devices, such as smart phones, digital cameras and digital camcorders. The app captures video from the camera, and embeds metadata collected from sensors on the mobile device. The metadata can be raw information from sensors, such as GPS locations, temperatures and speed. The metadata can also be information detected by the app from the video or other metadata information, such as human face, scene type, etc.

The digital story telling application is the metadata consumer, which imports videos with metadata embedded from internet, and creates digital stories by performing additional operations, such as detecting new metadata, editing and querying. The metadata consumer application usually runs on computers with high computation power and better user input devices, such as tablet PCs, laptops or desktop computers.

It is important that the creator app and the consumer application need to share the same metadata schema, when creating or loading metadata.

The following graph illustrates the architecture of the creator-consumer model:

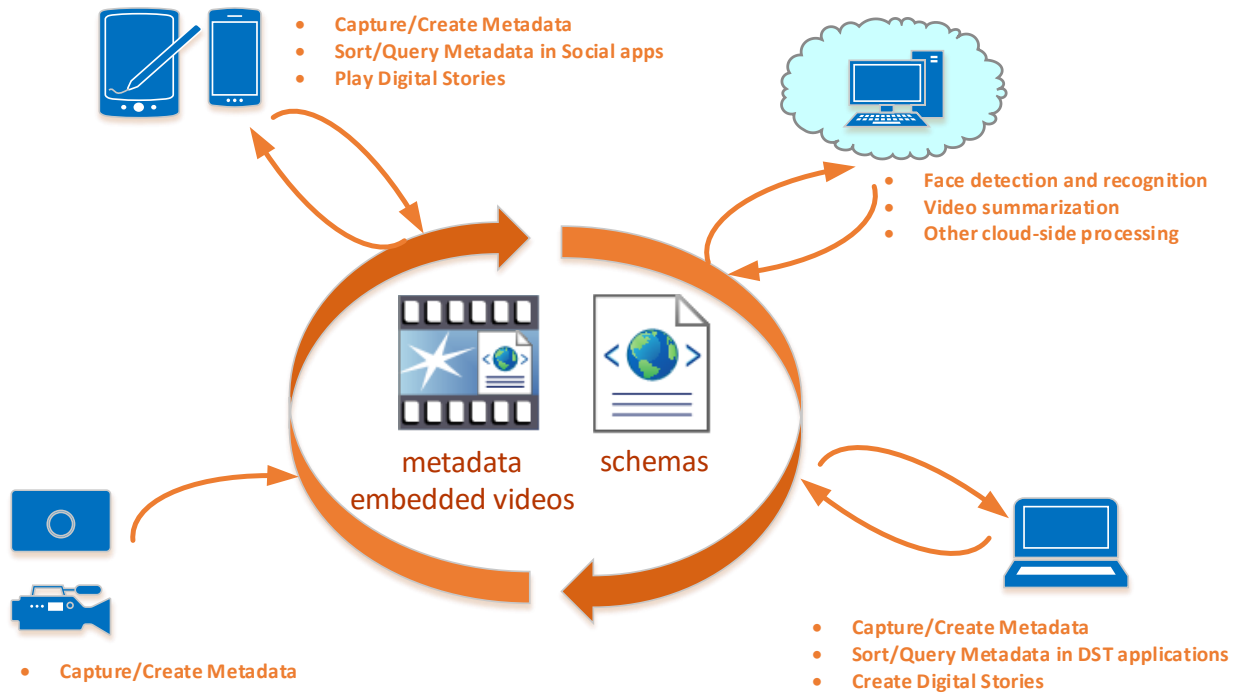


3.2.3 Metadata Ecosystem

In a more general form, multiple metadata creators and multiple metadata consumers are involved in exchanging information through metadata. Applications can consume metadata created by other generators, or add new metadata to videos that already contain metadata from other applications. Schemas are defined and shared to ensure that the metadata generators and metadata consumers understand each other. The application vendors, the schemas, and the metadata together form the metadata ecosystem.

In a healthy metadata ecosystem, schemas defined by popular applications become de facto standards, which attract more new application developers to implement. Thus more and more application can share metadata freely.

Metadata ecosystem also brings new user experience to applications. For example, a social app can allow user to query people from videos based on metadata detected by other digital story telling applications.


Capture/Create Metadata
Sort/Query Metadata in Social apps
Play Digital Stories
Face detection and recognition
Video summarization
Other cloud-side processing
metadata
embedded videos
schemas
Capture/Create Metadata
Capture/Create Metadata
Sort/Query Metadata in DST applications
Create Digital Stories

4. Programming Interface

This section describes the programming interface of VMF SDK.

4.1 Data Model

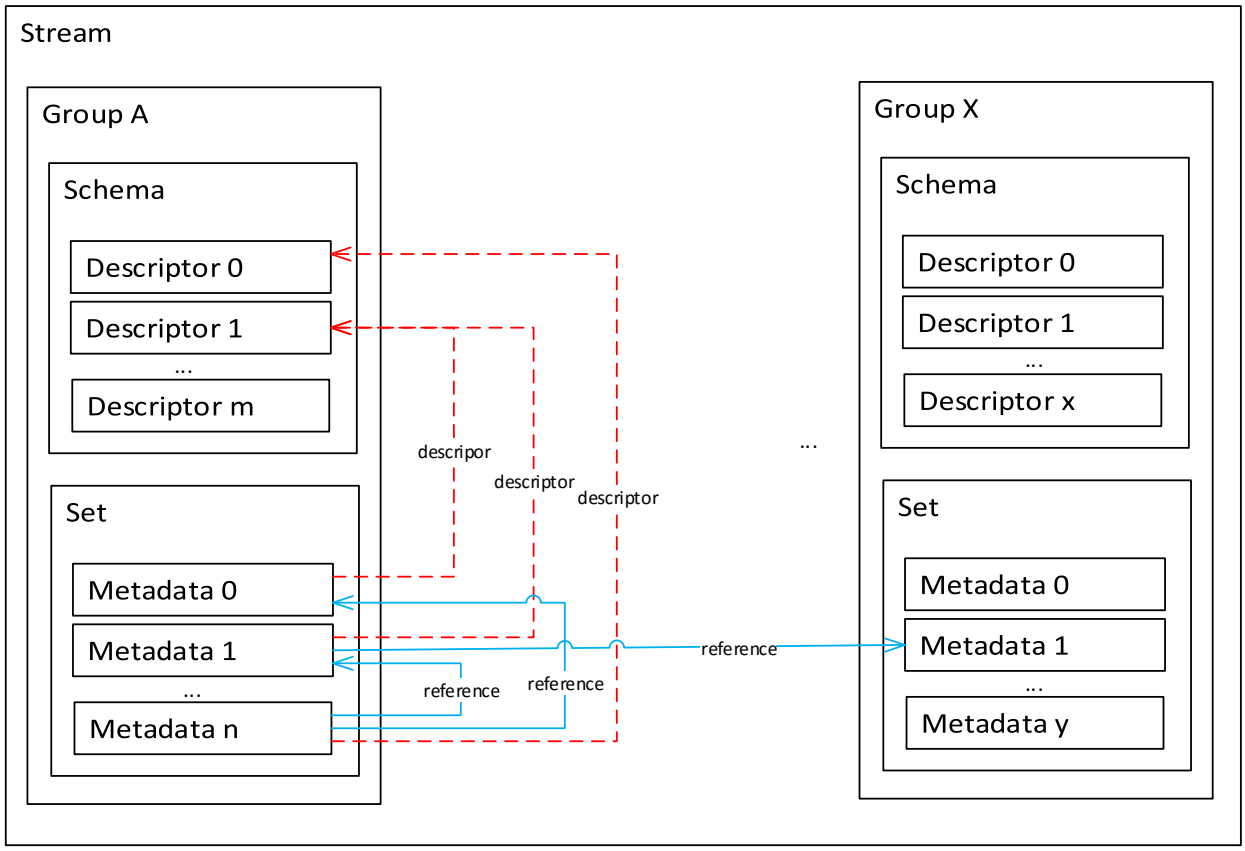
VMF stores the metadata within the host video files. There is no need to carry an additional file along with the media file.

VMF manages metadata in a hierarchy structure:

- a media file containing embedded metadata is said to contain a metadata ***stream***,
- a stream contains single or multiple ***groups*** of metadata.
- a metadata ***group*** contains a set of metadata ***descriptors*** and ***sets*** of metadata.

In addition to immediate declarations of metadata, VMF supports ***references*** by which metadata values are relationally associated to other metadata values.

The following diagram illustrates the internal structure of a metadata stream:



4.1.1 Metadata

Metadata describes characteristics of a resource. Metadata can be any information about the host video, or information that is related to the video.

Metadata can be information collected from a sensor, or information calculated by a computer. For example, a video of kids skiing can contain location information captured by the GPS sensor on the video camera as one type of metadata. The same video can also contain people as another type of metadata to describe the people that are in the video. In this case, the people and the video frame region where people are shown up are calculated by using computer algorithms.

Metadata is identified by its **name** and **id**. The id of a metadata is unique within a metadata stream; it is not guaranteed to be unique across metadata streams. There could be multiple metadata objects that are created based on the same descriptor, and therefore have the same metadata name within the host stream.

Metadata Types

Based on the way the metadata associate to the video frame content, metadata can also be classified into ***global***

metadata and ***frame-specific metadata***. A global metadata contains information related to an entire video sequence (i.e., video contiguously contained within a file, though the sequence may have temporal breaks). But not to any specific video frame. A frame-specific metadata contains information about one or multiple specific frames within the video. It is important that we should distinguish global metadata from metadata that is related to every frame of the video. One key difference between a global metadata and a frame-specific metadata is that, a global metadata may also appear in other videos, but a frame-specific metadata is only related to the host video. Based on the number of values and types of values contained within metadata, metadata can also be classified into: ***single-value metadata***, ***array metadata*** and ***structure metadata***. The ability to have an associative reference applies for each type of metadata, but associative references do not apply to elements within array metadata or structure metadata.

Value Type

VMF supports variant type as the basic value type. For any of the metadata categories (single item, array or structure), the type of the item value can be any of the following types:

- vmf::vmf_char: single character variable.
- vmf::vmf_integer: integer value up to 64-bit.
- vmf::vmf_real: double precision floating point value.
- vmf::vmf_string: string value.

Single Value Metadata

A single-value metadata instance contains one element. The data type of the element can be any of the above types.

Array Metadata

An array metadata allows multiple elements, all of the same data type. There is no limit on how many items can be stored within an array metadata.

An array metadata is logically equivalent to a single-value metadata, if the array contains only one element.

The items with the array metadata are identified by indices. The order of the items within an array metadata is maintained by VMF.

Structure Metadata

A structure metadata has multiple groups of elements (also called fields); each field is assigned a name that is unique within the structure metadata. Each field is also assigned with a type for the value it stores. Different fields can have different data types. For example, the following pseudo xml code defines people as structure type metadata (the actual format VMF uses to store metadata is different):

```
<vmf:Metadata id="1001" name=”people”>
 <vmf:Field name=”name” type=”vmf_string”>Jessica</vmf:Field>
 <vmf:Field name=”sex” type=”vmf_char”>F</vmf:Field>
 <vmf:Field name=”age” type=”vmf_integer”>12</vmf:Field>
 <vmf:Field name=”email” type=”vmf_string”>jessica@kids.org</vmf:Field>
</vmf:Metadata>
```

Fields within a structure are identified by field names. A field name must be unique within the same structure.

VMF does not support nested metadata type, which means that the value type for an item within an Array metadata or a Structure metadata cannot be another array or structure.

Metadata Reference

A metadata reference represents one relationship between host metadata and another metadata. The relationship represented by a metadata reference can be “is-a”, “contains”, “related to”, or any other relationship defined by a user or an application. For example, the “people” metadata defined above can contains a list of references to metadata of regions in multiple video frames. Each region defines the location within a specific video frame about the person.

A metadata can contain references to none or multiple metadata. The referenced metadata can be queried by using standard reference-based query functions.

A metadata can refer to other metadata within the same group, or from other metadata groups within the same video file.

4.1.2 Metadata Descriptor

Metadata descriptor is a structure that describes the format of the contents of a Metadata object. It is considered as the metadata of metadata.

A Descriptor structure defines the following properties of a Metadata object:

- Name of the schema the metadata belongs to
- Name of the metadata
- Type of the metadata: single item, array or structure
- Names and types of the fields*****

[*****]*For single-item metadata and array type metadata, there is only one field defined: its name is always empty, and its type defines the value type of the items can be stored.*

A Metadata descriptor is identified by its name, which is the same as the metadata it defines. The name of a metadata descriptor must be unique within the host schema.

Every metadata instance in memory has a reference pointing to a metadata descriptor that defines the format of the metadata. All metadata instances with the same metadata name reference to the same metadata descriptor.

4.1.3 Metadata Set

A metadata set is a collection (array) of metadata instances. Metadata set is usually used to hold the result from a query function. A subset of a metadata set is also a metadata set.

Metadata are stored in its host metadata stream. Removing metadata from a metadata set does not delete the metadata from memory, or remove it from the host metadata stream. The only way to remove a metadata is through the ***remove***() function through metadata stream.

4.1.4 Metadata Schema

A metadata schema is the collection of all metadata descriptors that are belong to the schema. A metadata schema is identified by a name, which should be in form of a Uniform Resource Identifier (URI). For example, the following URI defines the Intel standard schema for digital storytelling applications:

http://ns.intel.com/vmf/std-dst-schema-1.0

Developers can define their own schemas by using either C++ code, XML schema files or JSON schema files.

When the same metadata schema is used to create metadata for multiple videos, the contents of the metadata schema used in each video has to be consistence.

4.1.5 Metadata Group

A metadata group is a virtual container that contains a metadata schema, and all metadata that are created based on descriptors within the schema.

A metadata stream can contain one or multiple metadata groups. Metadata from one group cannot be created based on a descriptor from another group.

Metadata from one group can reference to metadata from other groups within the same metadata stream, or video file.

4.1.6 Metadata Stream

Metadata stream is the collection of all metadata information that are related to a specific video file. It is the root node of the metadata information tree, which stores all metadata and metadata descriptors that are stored within a video file. A metadata stream can contain one or multiple metadata groups.

Metadata stream is the interface for accessing any metadata-related information of a specific video file. All reading and writing operations of metadata from and to a video file are managed by a metadata stream object.

Metadata stream object is the owner and container of all in-memory metadata objects loaded from a specific video file. Metadata stream manages metadata objects and descriptor objects by using std::shared_ptr<T>. Metadata objects contained in a metadata set are just copies of the std::shared_ptr<T>.

4.2 Classes

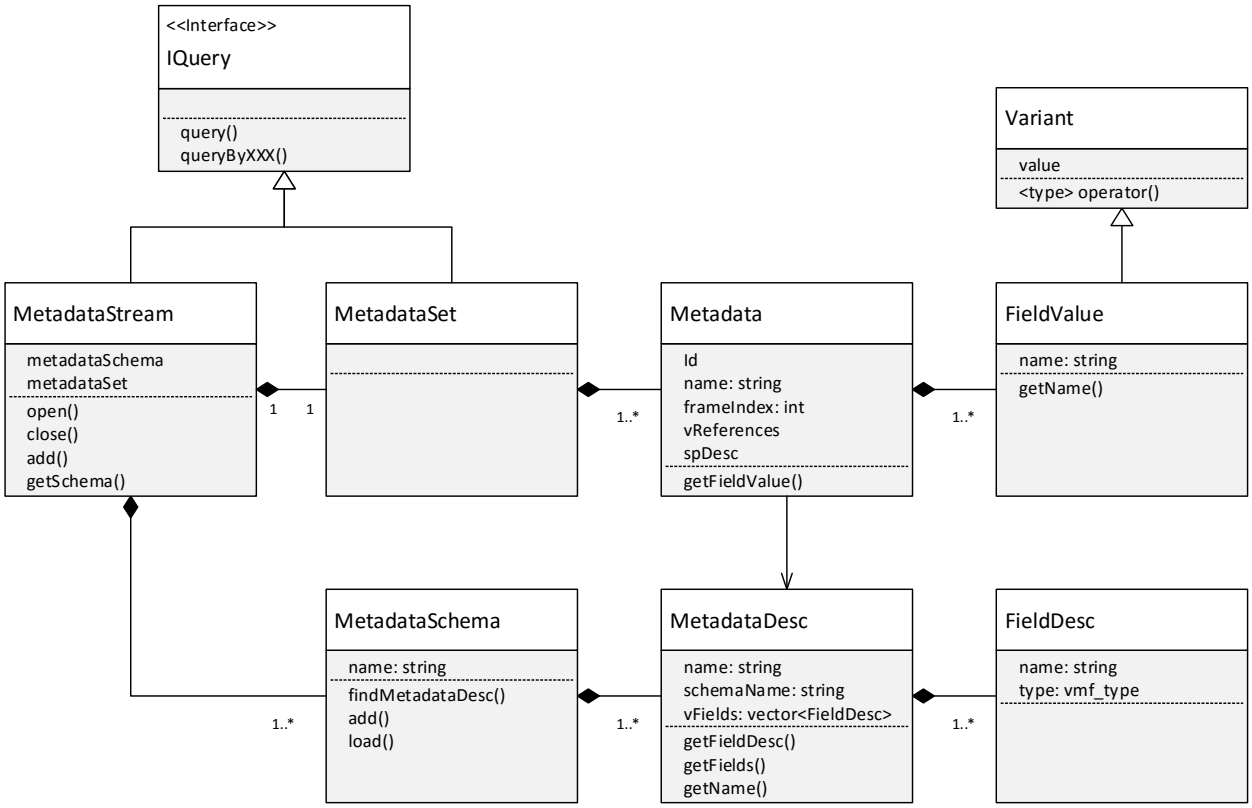
This section describes the design of a few core classes in VMF SDK. The core classes are the interface programmers use to interact with VMF data model and metadata stored within a video file.

Please note that VMF SDK defines everything in ***vmf*** namespace.

The following UML chart describes the relationship of the following core classes in VMF:

- vmf::Variant
- vmf::FieldValue

- vmf::Metadata
- vmf::MetadataSet
- vmf::MetadataDesc
- vmf::MetadataSchema
- vmf::MetadataStream
- vmf::IQuery



4.2.1 vmf::Variant

vmf::Variant class provides a C++ implementation of VARIANT type, which is common in advanced programming languages(such as BASIC).

A vmf::Variant variable can be used to store values with different types. The variable stores both the value and the type of the value, so that the raw type value can be retrieved later.

vmf::Variant class supports the following basic value types:

- vmf::vmf_integer: integer value up to 64-bit.
- vmf::vmf_real: double precision floating point value.
- vmf::vmf_string: string value.
- vmf::vmf_vec2d: 2-D vector with double precision floating point value.
- vmf::vmf_vec3d: 3-D vector with double precision floating point value.
- vmf::vmf_vec4d: 4-D vector with double precision floating point value.
- vmf::vmf_rawbuffer: raw binary data with variable size.

vmf::Variant class also supports array of most abovementioned types, except for the vmf::vmf_rawbuffer type:

- vmf::vmf_integer_vector: array of integer values up to 64-bit.
- vmf::vmf_real_vector: array of double precision floating point values.
- vmf::vmf_string_vector: array of string values.
- vmf::vmf_vec2d_vector: array of 2-D vectors with double precision floating point value.
- vmf::vmf_vec3d_vector: array of 3-D vectors with double precision floating point value.

- vmf::vmf_vec4d_vector: array of 4-D vectors with double precision floating point value.

The names of those types explain the type of data and the size or precision of the data a vmf::Variant variable stores.

vmf::Variant is the base class of vmf::FieldValue, which is the basic value type of a metadata object.

Constructors

vmf::Variant class provides multiple constructors that can construct vmf::Variant objects from raw value types. For example, the following code construct a vmf::Variant object with type of vmf_integer:

```
vmf::Variant var(32);
```

Another example: the following code construct a variable with type of vmf_string:

```
vmf::Variant var("This is a string");
```

Type Casting

vmf::Variant class provides a set of type casting operators that can convert vmf::Variant objects to raw type values automatically. For example:

```
vmf::Variant varPi( 3.14 );
double fPi = varPi;

vmf::Variant varString("Hello World!");
std::string strHelloWorld = varString;
```

Note: If a vmf::Variant object is casted to a type that is not compatible to the value type it holds, an exception would be thrown. For example, the following code would throw an exception:

```
try
{
  vmf::Variant varString("Hello World!");
  double fTemp = varString; // Cannot convert a string to a double floating number!
}
catch( vmf::exception& e)
{
  std::cout << "Failed to convert variant value to a different type!";
}
```

Type Converting

A vmf::Variant variable can be converted to a different value type that is ***convertible*** from current value type. For example, the following code can convert a vmf::Variant variable from vmf_integer to vmf_real:

```
vmf::Variant var( 1024 );
var.converTo( vmf::vmf_real ); // Ok to convert
```

The member function vmf::Variant::isCompatible(Type srcType, Type tarType) tells if one can convert from a source type to a target type.

Note: converTo() routine throws an exception if the source type is not convertible to the target type. The routine may also throw an exception for a target type that is convertible from the source type, but if the range of the value exceeds the range of the new value type. For example, the following code convert a vmf_integer to a convertible type, but will throw an exception:

```
try
{
  vmf::Variant varTemp( 1e100 ); // The type is vmf_real
  if( vmf::Variant::isConvertible( varTemp.getType(), vmf::vmf_integer ) )
  {
    varTemp.converTo( vmf::vmf_ integer); // Exception: 1e100 exceeds the range of vmf_integer
  }
}
catch( vmf::exception& e)
{
```

```
    std::cout << "Value to be converted exceeds the range of the target type!";
  }
```

4.2.2 *vmf::FieldValue*

vmf::FieldValue class derives from vmf::Variant class, by adding an additional "***name***" member variable. A vmf::FieldValue can be used as a vmf::Variant object, and it has all the properties that vmf::Variant provides. The name member of a vmf::FieldValue object can be accessed by vmf::FieldValue::getName() routine.
The following example demonstrates how to create a vmf::FieldValue object, and how to access its name member:

```
vmf::FieldValue fieldAge( "age", vmf::Variant( 12 ));
std::string strFieldName = fieldAge.getName(); // The value of strFieldName is set to "age"
int nFieldValue = fieldAge;    // The value of nFieldValue is set to 12
```

Objects of vmf::FieldValue class form the values of vmf::Metadata.

4.2.3 *vmf::MetadataDesc*

vmf::MetadataDesc class implements the metadata descriptor concept. A vmf::MetadataDesc object defines one type of metadata in a specific schema. The vmf::MetadataDesc class contains a name that defines the metadata in the scope of its host schema, and a list of field descriptors, each of which contains a field name and a field type members.
vmf::MetadataDesc object can be constructed in one of the two ways:

Single Item or Array Type

To construct a single item metadata or array type metadata, the name and the value type are needed:

```
std::shared_ptr<vmf::MetadataDesc> spMetadataDesc( new vmf::MetadataDesc( "my-metadata", vmf::vmf_string ));
```

Please note that most of pointers used in VMF are shared pointers based on std::shared_ptr.

Structure Type

To construct a structure type metadata, the name, field names, and field value types have to be provided. A helper structure vmf::FieldDesc is defined to hold information (name and type) about a field:

```
struct FieldDesc
{
  std::string      name;
  Variant::Type    type;
};
```

The following code sample constructs a "people" metadata descriptor:

```
std::shared_ptr< vmf::MetadataSchema> spSchema; // Need to initialize the object in real code.
std::vector< vmf::FieldDesc > vFields;
vFields.emplace_back( vmf::FieldDesc( "name", vmf::Variant::type_string ));
vFields.emplace_back( vmf::FieldDesc( "age", vmf::Variant::type_integer ));
vFields.emplace_back( vmf::FieldDesc( "sex", vmf::Variant::type_string));
vFields.emplace_back( vmf::FieldDesc( "email", vmf::Variant::type_string ));
std::shared_ptr< vmf::MetadataDesc > spDesc( new vmf::MetadataDesc( "people", vFields ));
spSchema->add( spDesc );
```

The last statement adds the newly constructed descriptor object into a schema object, which is basically a list of metadata descriptors.

Access Field Information

A vmf::MetadataDesc object contains two name properties: one for the metadata it represents, and another name for the host schema name. The following two routines return the names:

```
std::string  getMetadataName() const;
std::string getSchemaName() const;
```

vmf::MetadataDesc class provides two ways to access the information about the fields. One method allows the caller to access the field information about a specific field; another method allows the caller to get the whole set of field information:

```
std::vector<FieldDesc> getFields() const;
bool getFieldDesc( FieldDesc& field, const std::string& sFieldName = "" ) const;
```

The getFieldDesc() routine allows the caller to pass "" as the field name parameter. This is designed for single item

metadata and array type metadata, which do not have field name.

4.2.4 vmf::MetadataSchema

vmf::MetadataSchema class implements the schema model. A metadata schema is essentially an array of vmf::MetadataDesc objects.

Schema Name

Each metadata schema should have a unique name. We recommend using URI (uniform resource identifier) format to declare a metadata name.

Search for Descriptor

A metadata schema object usually contains many metadata descriptor objects. To search for a specific descriptor, the following findMetadataDesc() function can be used:

```
std::shared_ptr< vmf::MetadataSchema> spSchema; // Need to initialize the object in real code.
if( !spSchema->findMetadataDesc( “my-metadata” ) )
{
  std::cout << “Metadata <my-metadata> was not defined!”;
}
```

4.2.5 vmf::Metadata

vmf::Metadata class provides the storage for metadata objects. The class is derived from std::vector<T> class:

```
class vmf::Metadata : public std::vector< vmf::FieldValue >
```

Metadata Value(s)

vmf::Metadata class can represent three types of metadata: Single-Value, Array and Structure.

A single value metadata contains only one value. The value can be set by using either vmf::Metadata:addValue() member function, or std::vector::push_back() function:

```
std::shared_ptr<vmf::Metadata> spMetadata( new vmf::Metadata( spDescriptor ));
spMetadata->addValue( vmf::Variant( 100 ));
```

or,

```
spMetadata->push_back( vmf::Variant( 100 ));
```

An Array type metadata contains multiple values of the same type. The following code creates a metadata that contains a list of names:

```
std::shared_ptr<vmf::Metadata> spMetadata( new vmf::Metadata( spNameDescriptor ));
spMetadata->addValue( vmf::Variant( “Jessica” ));
spMetadata->addValue( vmf::Variant( “Alan” ));
spMetadata->addValue( vmf::Variant( “Mike” ));
```

Since Metadata class stores its values in std::vector base class, the std::vector member function can be used to access the value(s) of a Metadata object.

To access the value of a metadata that stores a single integer:

```
if( spMetadata->size() > 0 )
{
  int nValue = spMetadata->at(0);
}
```

The iterator interface of std::vector<T> is also available to vmf::Metadata class. The following code demonstrates how to access values in string array type metadata by using the iterator interface (begin() and end() routines).

```
std::for_each( spMetadata->begin(), spMetadata->end(), [&]( vmf::FieldValue& value )
{
  std::cout << value.getName() << “: “ << (std::string)value << std::endl;
});
```

A structure type metadata contains multiple fields, each of which may have different value type. The following member functions allow callers to access field names, or a specific field:

```
std::vector< std::string > getFieldNames() const;
vmf::Variant getFieldValue( const std::string& sName ) const;
iterator findField( const std::string& sFieldName );
void setFieldValue( const std::string& sFieldName, const vmf::Variant& value );
```

To analyze what fields are in an unknown type of metadata, one can use getFieldNames() together with getFieldValue() routine. For example, the following code dump all field names and field values of an unknown type of metadata:

```
void dump( const std::shared_ptr<Metadata>& spMetadata )
{
  std::vector<std::string> vFieldNames = spMetadata->getFieldNames();
  std::for_each( vFieldNames->begin(), vFieldNames ->end(), [&]( const std::string& fieldName )
  {
    vmf::Variant fieldValue = spMetadata->getFieldValue( fieldName );
    std::cout << "Field name = [" << fieldName << "], field value = [", << fieldValue.toString() << "]" << std::endl;
  });
}
```

To set field value, the name of the field to be set, and the value to be set are both needed:

```
spMetadata->setFieldValue( "name", "Jessica" );
spMetadata->setFieldValue( "age", 12 );
spMetadata->setFieldValue( "sex", 'F' );
```

Please note that raw type values can be used in places where a vmf::Variant objects are required. A proper vmf::Variant constructor will be invoked to construct the vmf::Variant object on the fly, if raw type values are used.
The std::vector<T> members are not recommended to access field values of a structure type of metadata. The setFieldValue() function provides additional check that makes sure proper filed and type are used. If member functions of std::vector<T> is used to add a field value to a metadata object, the check will be delayed till the moment when the metadata is added to a metadata stream.
In additional to the value items stored in the vector, vmf::Metadata adds a few more members:

Other Members

Id: each vmf::Metadata object has an identification number, which is unique within the host metadata stream. The Id of a metadata is assigned by the stream when the metadata object is added to the stream. Before adding to a stream, the Id of any newly created metadata is -1;
The Id can be accessed by vmf::Metadata::getId() routine.
Id can also be used to retrieve the metadata object from the host stream:

```
std::shared_ptr<vmf::Metadata> spMetadata = stream.getById( nMetadataId );
```

Please note that getId() may fail to get a metadata if that metadata has not yet been loaded from video file.
Name: The name of a metadata indicates the metadata descriptor based on which the metadata was created. So the name of a metadata should be the same of the name of its metadata descriptor. This name is unique for the descriptor within the host schema.
Multiple metadata be created based on the same descriptor. In that case, multiple instance of the same type of metadata share the same name.
The name member can be accessed by vmf::Metadata::getName() routine.

```
std::string strName = spMetadata->getName();
```

Schema Name: A metadata object also contains the name of its host schema. The schema name, together with the metadata name, can be used to retrieve the descriptor object of a metadata object:

```
std::shared_ptr<vmf::MetadataSchema> spSchema = stream.getSchema( spMetadata->getSchemaName() );
std::shared_ptr<vmf::MetadataDesc> spDescriptor = spSchema->findMetadataDesc( spMetadata->getName() );
```

FrameIndex and NumOfFrames: The FrameIndex and NumOfFrames members define the video frame(s) to which the host metadata is associated. The FrameIndex defines the index of the starting video frame. The NumOfFrames defines the number of frames in sequence that contain the same metadata.
For global metadata that is not associated to any specific video frame, the FrameIndex is set to -1.
References: Metadata references are pointers to other metadata objects within the same metadata stream. References are created by using vmf::Metadata::addReference() routine:

```
std::shared_ptr<vmf::Metadata> spMetadataA, spMetadataB;
spMetadataA->addReference( spMetadataB );
```

The reference metadata (spMetadataB) has to be added to the same host stream, before it can be used as a

reference of other metadata.
A metadata can have zero or multiple metadata as its references. The reference metadata can be accessed by using one of the following routines:

```
std::shared_ptr<Metadata> getFirstReference( const std::string& sMetadataName ) const;
MetadataSet getReferencesByMetadata( const std::string& sMetadataName = "" ) const;
bool isReference( const Metadata::IdType& id ) const;
void addReference( const std::shared_ptr< Metadata >& spMetadata, bool bEnsureUnique = false );
void removeReference( const IdType& id );
```

The routine ***getFirstReference***() is usually used when a metadata only have one reference to a type of metadata. For example, if we assume that a region usually associate with one people metadata at most, and one location metadata at most, the following code can print all people that shows up in a specific location:

```
vmf::MetadataStream stream;
// Get all metadata that have name of “region” from the stream
vmf::MetadataSet set = stream.queryByName( “region” );
std::for_each( set.begin(), set.end(), [&]( std::shared_ptr<Metadata>& spMetadata )
{
  std::shared_ptr<Metadata> spPeople = spMetadata->getFirstReference( “people” );
  std::shared_ptr<Metadata> spLocation = spMetadata->getFirstReference( “location” );
  if( spPeople != nullptr && spLocation != nullptr && spLocation->getFieldName(“name”) == “San Jose” )
  {
    std::cout << spPeople->getFieldValue(“name”).toString() << “ was in San Jose” << std::endl;
  }
});
```

4.2.6 vmf::MetadataSet and vmf::IQuery

vmf::MetadataSet class is the standard container class for metadata objects. It provides the basic functions for querying metadata. The interface of all query functions are actually defined in one of the base classes of vmf::MetadataSet, the vmf::IQuery. Another base class of vmf::MetadataSet is std:vector<std::shared_ptr<vmf::Metadata>> class. The vmf::MetadataSet class does not contain any additional member variables.

```
class MetadataSet : public std::vector< std::shared_ptr< Metadata >>, public IQuery
```

The interface defined by IQuery includes the following functions:

```
MetadataSet query( std::function< bool( const std::shared_ptr<Metadata>& spMetadata )> filter ) const;
MetadataSet queryByReference( std::function< bool( const std::shared_ptr<Metadata>& spMetadata,
  const std::shared_ptr<Metadata>& spReference )> filter ) const;
MetadataSet queryByFrameIndex( size_t index ) const;
MetadataSet queryBySchema( const std::string& sSchemaName ) const;
MetadataSet queryByName( const std::string& sName ) const;
MetadataSet queryByNameAndValue( const std::string& sMetadataName, const vmf::FieldValue& value ) const;
MetadataSet queryByNameAndFields( const std::string& sMetadataName,
  const std::vector< vmf::FieldValue>& vFields ) const;
MetadataSet queryByReference( const std::string& sReferenceName ) const;
MetadataSet queryByReference( const std::string& sReferenceName, const vmf::FieldValue& value ) const;
MetadataSet queryByReference( const std::string& sReferenceName,
  const std::vector< vmf::FieldValue>& vFields ) const;
```

The query functions have similar pattern: i) all functions are const, which means that the function do not impact the host MetadataSet object. ii) all functions return another MetadataSet object as query results. Since vmf::MetadataSet contains only pointers (std::shared_ptr<vmf::Metadata>) to the metadata objects, copying metadata pointers from one set to another set does not create new metadata objects(It just increase the internal counter inside each shared_ptr pointers.).
The function ***query***() is a generic query function, which requires a lambda expression as a parameter, allowing the caller to provide the actual filtering/querying algorithm. The lambda expression function needs to have the following prototype:

```
std::function< bool( const std::shared_ptr<Metadata>& spMetadata )> filter
```

Basically, the lambda expression defines a function that takes a ***const shared_ptr*** to ***vmf::Metadata*** object, and returns a boolean value, to indicate that the metadata object passed by the spMetadata parameter meets the condition, and should be included in the return set.

The ***query***() function can be used to construct unlimited custom query functions. Actually, most of the other query functions in class vmf::MetadataSet are implemented by using the ***query***() function. For example:

```
MetadataSet MetadataSet::queryByName( const std::string& sName ) const
{
  MetadataSet set = query( [&]( const std::shared_ptr< Metadata >& spItem )->bool
  {
    return ( spItem->getName() == sName );
  });

  return set;
}
```

Multiple query functions can be stacked over each other to create a complicated query. Query functions can also be mixed with customer querying algorithms or functions, to achieve a more accurate query result. The samples in later section demonstrates this ability.

Other than the generic ***query***() function, here are a few other commonly used query functions:

Function ***queryBySchema***() returns all metadata that belongs to a specific schema. Function ***queryByName***() returns all metadata that have a specific name. Since a metadata type name is unique only within its host schema, there could be another type of metadata in another schema that has the same name. So the result from ***queryByName***() does not guarantee that the result metadata have the same type. Typically ***queryByName***() is used in conjunction with ***queryBySchema***(). For example:

```
vmf::MetadataStream stream; // MetadataStream class also implement IQuery interface
std::string strSchemaName("vmf://ns.intel.com/core/1.0" );
// Get all metadata belong to a specific schema
vmf::MetadataSet setSchema = stream.queryBySchema( strSchemaName );
// Now find all people metadata from that schema
vmf::MetadataSet setPeople = setSchema.queryByName("people");
```

The above two queries can also be implemented by using the generic query function:

```
vmf::MetadataStream stream; // MetadataStream class also implement IQuery interface
std::string strSchemaName("vmf://ns.intel.com/core/1.0" );
// Get all people metadata belong to a specific schema
vmf::MetadataSet setAll = stream.getAll();
vmf::MetadataSet setPeople = stream.query([&]( const std::shared_ptr<vmf::Metadata>& spMetadata )
{
  return spMetadata->getName() == "people" && spMetadata->getSchemaName = strSchemaName;
});
```

Shift

Another important feature provided by vmf::MetadataSet class is to be able to adjust the FrameIndex and NumOfFrames properties of metadata due to changes of video frames as results of a video editing operations(such as shrinking, cutting, shifting, etc).

This feature is provided by the shift() function:

```
size_t shift( long long nTarFrameIndex, long long nSrcFrameIndex,
  long long nNumOfFrames = FRAME_COUNT_ALL, MetadataSet* pSetFailure = NULL);
```

The functions takes four parameters:

- **nTarFrameIndex**: the new index of the original frame indicated by nSrcFrameIndex.
- **nSrcFrameIndex/nNumOfFrames**: the range of frame indices in the original video will be kept.
- **pSetFailure**: a set of metadata that have fallen out the new valid frame ranges, defined by nTargetFrameIndex and nNumOfFrames.

For a video that has 3000 frames, for example, the following code will relocate frames 500-2499, to a new location: frames 0-1999:

```
set.shift( 0, 500, 2000, &setFailure );
```

Please note the nTarFrameIndex is set to 0, which indicates that the original frame 500 will become the first frame in the new video.

For the metadata collected in ***setFailure*** (which have failed to be shifted), the typical operation is to remove them from the stream, or apply another shift operation.

4.2.7 vmf::MetadataStream

class vmf::MetadataStream implements the metadata stream object mentioned in the data model section. A MetadataStream object corresponds to one metadata-enabled video sequence. The video sequence could be a video file on the hard disk, or a newly created video object in the memory.

vmf::MetadataStream is the only interface that can add, delete, load, save metadata to/from a video file. The vmf::MetadataStream contains the full set of metadata objects that are loaded from host video file or added by in-memory logic. It can be considered as a MetadataSet object that contains the complete set of metadata of a stream. Since it also implements the vmf::IQuery interface, so it is usually used as the starting set of most query operations.

Other than the IQuery interface, MetadataStream class also provide the following functionalities:

Load/Save

vmf::MetadataStream class provides the basic I/O functions for loading and saving metadata from a video file. To be able to access the I/O functions, a vmf::MetadataStream object has to be connected to a video file, by using the open() function:

```
vmf::MetadataStream stream;
bool bSucceed = stream.open( “my-video.avi”, vmf::MetadataStream:: ReadWrite );
```

vmf::MetadataStream class supports three modes to open video files: ***InMemory***, ***ReadWrite***, and ***ReadOnly***. The ***InMemory*** mode is used when a metadata stream is not attached to a video file (newly created stream), or the host video file has been closed by calling ***close***() function. A stream can be reopened by calling ***reopen***() function, if it has been previously opened and closed.

After attaching a vmf::MetadataStream object to a video file successfully, we can load the metadata within the video file:

```
bool bSucceed = stream.load();
```

The ***load***() function loads all metadata, and metadata-related from the video file. If there is no metadata in the video, the routine returns false.

We can also choose to load metadata that belong to a specific schema:

```
bool bSucceed = stream.load( “vmf://ns.mycompany.com/my-schame” );
```

Or, we can load metadata of a specific type, by providing both the schema name, and the metadata name:

```
bool bSucceed = stream.load( “vmf://ns.mycompany.com/my-schame”, “my-metadata-type” );
```

Saving metadata from a stream to a video file is simple:

```
bool bSucceed = stream.save();
stream.close();
```

Reopen

The ***close***() function discussed above disconnects a stream object from its host video file, and sets the internal mode to ***InMemory***. When a steam is closed, the metadata stored inside the stream are still in the system memory. The metadata stream can be attached to the same video file later, by calling ***reopen***() function.

```
stream.close();
// Perform some operations here.

// Re open the file
If( stream.reopen( vmf::MetadataStream::ReadWrite ))
{
  stream.save();
  stream.close();
```

```
}
```

Access Schema

A metadata stream object may contain single or multiple metadata schemas. Call the following function to get access to a specific schema:

```
const std::shared_ptr< MetadataSchema > getSchema( const std::string& sSchemaName ) const;
```

vmf::MetadataStream::***addSchema***() routine adds a new schema to a stream. The name of the new schema has to be unique and not have been added to the stream.

One can also delete a schema by using ***deleteSchema***() routine. When deleting a schema, there should be metadata in the same stream that belong to the schema.

Access Metadata

To add a new metadata to a stream, call add routine:

```
Metadata::IdType add( std::shared_ptr< Metadata >& spMetadata );
```

The routine returns the Id of the newly added metadata. The routine also validates the metadata before adding it to the stream. If the metadata is invalid, an exception would be thrown. A metadata is invalid if one of the following conditions is true:

- Contains no value.
- FrameIndex is greater than -1, but NumOfFrames is 0.
- No descriptor.
- Missing field values.
- Contains undefined field.
- Incorrect field type.
- Defined as array type but contains values with different types.

Once added to a stream, a metadata can be retrieved by its Id:

```
auto spMetadata = stream.getById( id );
```

The ***getAll***() function returns a MetadataSet object that contains all metadata within the stream:

```
MetadataSet getAll() const;
```

Since vmf::MetadataStream implements vmf::IQuery interface, you can query a stream in the same way as you may query a MetadataSet.

To remove a metadata from the host stream, the Id of the metadata needs to be provided:

```
bool remove( const Metadata::IdType& id );
```

You can also remove multiple metadata by providing a MetadataSet object that contains the metadata to be deleted:

```
void remove( const MetadataSet& set );
```

Import

When composing a digital story by pulling video segments from multiple video sources, metadata that belong to the original source movies are often needed to be imported into the new video. The import function allows developers to import metadata from one stream to another stream.

```
bool import( MetadataStream& srcStream, MetadataSet& srcSet, long long nTarFrameIndex, long long nSrcFrameIndex, long long
nNumOfFrames, MetadataSet* pSetFailure );
```

Typically, the ***import***() function are used together with the shift operation, to allow importing metadata of a specific range of frames from a source video to a specific location in the target video. So the last three parameters of ***import***() function are the same as vmf::MetadataSet::***shift***() function.

The following code import metadata from two source videos into a new stream:

```
vmf::MetadataStream streamA, streamB, streamNew;
if(streamA.open( “video-A.avi”, vmf::MetadataStream::ReadOnly ) &&
  streamB.open( “video-B.avi”, vmf::MetadataStream::ReadOnly ) &&
  streamNew.open( “video-New.avi”, vmf::MetadataStream::ReadWrite ))
{
  // Load metadata from both source videos
  streamA.load();
  streamB.load();
```

```
    vmf::MetadataSet setA = streamA.getAll();
    vmf::MetadataSet setB = streamB.getAll();

    // Import 3000 frames from index 500, to new index of 0;
    streamNew.import( streamA, setA, 0, 500, 3000, NULL );

    // Shift all frames from index 1000, to new index of 3000;
    streamNew.import( streamB, setB, 3000, 1000 );

    // Save the changes
    streamNew.save();
}

// Always close stream objects.
streamNew.close();
streamB.close();
streamA.close();
```

4.2.8 vmf::Format

vmf::Format C++ interface provides possibility to serialize the whole MetadataStream or its part (e.g. a single metadata record or a schema description) to a string representation that can be useful in various scenarios in particular for metadata live streaming over network. The VMF includes built-in support for XML and JSON text formats (vmf::FormatXML and vmf::FormatJSON classes respectively) but it's possible to make and use user-provided formats via vmf::Format implementation.

4.2.9 vmf::Compressor

vmf::Compressor C++ interface provides possibility to compress metadata before saving it into a media file or serialization. When used it processes the final text representation with either built-in or user-provided algo then encodes with base64 and stores as a single metadata record (for compatibility with older VMF versions). A ZLib-based compression algo is available as a default / built-in one.

The 'compression' sample demonstrates working with both built-in and user-defined compression algo-s.

4.2.10 vmf::Encryptor

vmf::Encryptor C++ interface provides possibility to encrypt metadata before saving it into a media file or serialization on the following levels:

- a field in a metadata record,
- a complete metadata record,
- a field in all metadata records of the type,
- all metadata records of the type,
- all metadata records of the schema,
- all metadata records.

When used it processes the final text representation with either built-in or user-provided algo then encodes with base64 and stores as a single metadata record (for compatibility with older VMF versions). A crypto++-based encryption algo is available as a default / built-in one.

The 'encryption' sample demonstrates working with both built-in and user-defined encryption algo-s.

NOTE: if both compression and encryption should be used together then compression should happen first for the sake of efficiency.

4.2.11 vmf::Stat, vmf::StatField

vmf::Stat C++ class provides a configurable dashboard-like object that coalesces data from added metadata using a number of built-in statistical operations (last value, average, sum, count, min/max) and user-defined ones.

It updates asynchronously either on every metadata addition or on the specified time interval or by request.

The ‘metadata-statistics’ sample demonstrates working with both built-in and user-defined statistics operations.

5. VMF Sample

This section describes using of VMF library in one sample application. Each of the sub-sections describes one specific topic.

The sample application opens one video file and attaches it to a metadata stream. The application then defines a schema with a list of metadata descriptors. A few metadata are then created based on the descriptors, and different types of queries are applied.

5.1 Load Metadata

This section demonstrates the high level framework of a typical application that uses VMF. The application opens a video file as the host of the metadata stream. The application then performs a few operations and close the metadata. We will discuss the details of each operations in later sections.

Please note that the stream is opened in ***Update*** mode, which will allow changes made to the stream be saved to the video file later.

Here is the code:

```
#include "vmf.h"
#include <iostream>
#include <vector>
#include <algorithm>
#include <limits>

void main( void )
{
  try
  {
    vmf::MetadataStream stream;
    if( stream.open( "my-video.avi", vmf::MetadataStream::Update) )
    {
      // Declare schema name
      std::string sSchemaName( "vmf:http://ns.intel.com/vmf/1.0" );
      initSchema( stream, sSchemaName );
      // Create metadata for two events: ski and birthday party
      addMetadata( stream, sSchemaName );
      // Query demo
      queryDemo( stream, sSchemaName );
      // Save and close the stream
      stream.save();
      stream.close();
    }
  }
  catch( std::exception& e)
  {
    std::cout << e.what() << std::endl;
  }
}
```

5.2 Define Schema

This section analyzes the routine ***initSchema***(), which defines a few metadata descriptors as the content of a schema.

The routine defines five descriptors: people, region, gps, time and event. All except event metadata are structure type of metadata. The event metadata is a single value metadata, which does not have any field.

Please note that a helper structure, vmf::FieldDesc, is used to carry field description information to the

vmf::MetadataDesc constructor.

```
void initSchema( vmf::MetadataStream& stream, const std::string& sSchemaName )
{
  std::shared_ptr< vmf::MetadataSchema > spSchema( new vmf::MetadataSchema( sSchemaName ));

  // Add people metadata description.
  std::vector< vmf::FieldDesc > vFields;
  vFields.emplace_back( vmf::FieldDesc( "name", vmf::Variant::type_string ));
  vFields.emplace_back( vmf::FieldDesc( "age", vmf::Variant::type_integer ));
  vFields.emplace_back( vmf::FieldDesc( "sex", vmf::Variant::type_char ));
  vFields.emplace_back( vmf::FieldDesc( "email", vmf::Variant::type_string ));
  std::shared_ptr< vmf::MetadataDesc > spDesc( new vmf::MetadataDesc( "people", vFields ));
  spSchema->add( spDesc );

  // Add region metadata desc
  vFields.clear();
  vFields.emplace_back( vmf::FieldDesc( "left", vmf::Variant::type_integer ));
  vFields.emplace_back( vmf::FieldDesc( "top", vmf::Variant::type_integer ));
  vFields.emplace_back( vmf::FieldDesc( "width", vmf::Variant::type_integer ));
  vFields.emplace_back( vmf::FieldDesc( "height", vmf::Variant::type_integer ));
  spDesc = std::shared_ptr< vmf::MetadataDesc >( new vmf::MetadataDesc( "region", vFields ));
  spSchema->add( spDesc );

  // Add gps desc
  vFields.clear();
  vFields.emplace_back( vmf::FieldDesc( "lat", vmf::Variant::type_string ));
  vFields.emplace_back( vmf::FieldDesc( "long", vmf::Variant::type_string ));
  vFields.emplace_back( vmf::FieldDesc( "alt", vmf::Variant::type_real ));
  spDesc = std::shared_ptr< vmf::MetadataDesc >( new vmf::MetadataDesc( "gps", vFields ));
  spSchema->add( spDesc );

  // Add time metadata desc
  vFields.clear();
  vFields.emplace_back( vmf::FieldDesc( "year", vmf::Variant::type_integer ));
  vFields.emplace_back( vmf::FieldDesc( "month", vmf::Variant::type_integer ));
  vFields.emplace_back( vmf::FieldDesc( "day", vmf::Variant::type_integer ));
  vFields.emplace_back( vmf::FieldDesc( "hour", vmf::Variant::type_integer ));
  vFields.emplace_back( vmf::FieldDesc( "minute", vmf::Variant::type_integer ));
  vFields.emplace_back( vmf::FieldDesc( "second", vmf::Variant::type_integer ));
  vFields.emplace_back( vmf::FieldDesc( "millisecond", vmf::Variant::type_integer ));
  spDesc = std::shared_ptr< vmf::MetadataDesc >( new vmf::MetadataDesc( "time", vFields ));
  spSchema->add( spDesc );

  // Add event metadata desc
  spDesc = std::shared_ptr< vmf::MetadataDesc >( new vmf::MetadataDesc( "event", vmf::Variant::type_string ));
  spSchema->add( spDesc );
```

A schema has to be added to the host stream before any descriptor within the schema can be used.

```
// Add schema to stream
stream.addSchema( spSchema );
```

```
}
```

5.3 Create Metadata

The ***addMetadata***() routine creates a set of metadata instances based on the descriptors defined in the ***initSchema***() routine.

When creating a metadata instance, the metadata descriptor is required by the vmf::Metadata constructor. This design ensures that each metadata object is associated with the information of its descriptor. When stream.***add***() routine is called later to add a metadata object, each field name and field value type are checked against the corresponding descriptor, to make sure the format of the metadata matches the format defined by the descriptor. The code also demonstrates that trying to set a field with incorrect type, or trying to set undefined field, will trigger runtime exceptions.

```
void addMetadata( vmf::MetadataStream& stream, const std::string& sSchemaName )
{
  const std::shared_ptr< vmf::MetadataSchema > spSchema = stream.getSchema( sSchemaName );
  if( spSchema == nullptr || spSchema->size() == 0 )
    throw std::exception( "Schema was not found, or schema has no metadata defined!" );

  const std::shared_ptr< vmf::MetadataDesc > spEventDesc = spSchema->findMetadataDesc( "event" );
  const std::shared_ptr< vmf::MetadataDesc > spPeopleDesc = spSchema->findMetadataDesc( "people" );
  const std::shared_ptr< vmf::MetadataDesc > spRegionDesc = spSchema->findMetadataDesc( "region" );
  const std::shared_ptr< vmf::MetadataDesc > spTimeDesc = spSchema->findMetadataDesc( "time" );
  const std::shared_ptr< vmf::MetadataDesc > spGpsDesc = spSchema->findMetadataDesc( "gps" );

  if( spPeopleDesc == nullptr || spRegionDesc == nullptr || spTimeDesc == nullptr || spGpsDesc == nullptr || spEventDesc ==
nullptr )
    throw std::exception( "No all metadata description defined" );

  //---------------------------------------------------------------------------
  // Create people metadata
  std::shared_ptr< vmf::Metadata > spJessica( new vmf::Metadata( spPeopleDesc ));
  spJessica->setFieldValue( "name", "Jessica" );
  spJessica->setFieldValue( "age", 12 );
  spJessica->setFieldValue( "sex", 'F' );
  spJessica->setFieldValue( "email", "jessica@kidsmail.com" );
```

VMF lib checks field name and field type against descriptor. If invalid type or undefined field name are used, exception will be thrown.

```
  // Cannot set field value with different type than it was defined in the description
  try
  {
    spJessica->setFieldValue( "age", 12.0 );
  }
  catch( std::exception& e )
  {
    std::cout << e.what() << std::endl;
  }

  // Cannot set field not defined.
  try
  {
    spJessica->setFieldValue( "address", "123 4th av, San Jose, CA" );
  }
```

```
catch( std::exception &e )
{
   std::cout << e.what() << std::endl;
}

// Create another metadata
std::shared_ptr< vmf::Metadata > spAlan( new vmf::Metadata( spPeopleDesc ));
spAlan->setFieldValue( "name", "Alan" );
spAlan->setFieldValue( "age", 8 );
spAlan->setFieldValue( "sex", 'M' );
spAlan->setFieldValue( "email", "alan@kidsmail.com" );

// Create another metadata
std::shared_ptr< vmf::Metadata > spMike( new vmf::Metadata( spPeopleDesc ));
spMike->setFieldValue( "name", "Mike" );
spMike->setFieldValue( "age", 2 );
spMike->setFieldValue( "sex", 'M' );
spMike->setFieldValue( "email", "mike@kidsmail.com" );

// Add all metadata to stream
stream.add( spJessica );
stream.add( spAlan );
stream.add( spMike );
```

Since the people metadata are global metadata, the frame index are not set for the above metadata objects. In the following code, a metadata of type of “event” is associated to video frame range 1000~3999.

```
//---------------------------------------------------------------------
// Create event metadata
std::shared_ptr< vmf::Metadata > spSki( new vmf::Metadata( spEventDesc ));
spSki->addValue( "Ski" );
// 3000 frames starting from frame 1000
spSki->setFrameIndex( 1000, 3000 );
spSki->addReference( spJessica );
spSki->addReference( spAlan );
stream.add( spSki );
```

Please note that the above code adds reference from the “Ski” event metadata to two people metadata: “Jessica” and “Alan”. These two references define the fact that Jessica and Alan went to the ski event.
The following code creates another event, “Birthday”, which reference to all three people metadata defined above.

```
std::shared_ptr< vmf::Metadata > spBirthday( new vmf::Metadata( spEventDesc ));
spBirthday->addValue( "Birthday" );
// 2500 frame starting from frame 4001
spBirthday->setFrameIndex( 4001, 2500 );
spBirthday->addReference( spJessica );
spBirthday->addReference( spAlan );
spBirthday->addReference( spMike );
stream.add( spBirthday );
```

The following code creates a few region metadata, and associate each with different people metadata. Please note that some region metadata are associated with multiple frames, indicating that the subject was in the same region across multiple frames.

```
//---------------------------------------------------------------------
// Create region metadata
```

```
std::shared_ptr< vmf::Metadata > spRegion( new vmf::Metadata( spRegionDesc ));
spRegion->setFieldValue( "left", 100 );
spRegion->setFieldValue( "top", 100 );
spRegion->setFieldValue( "width", 50 );
spRegion->setFieldValue( "height", 50 );
// 25 frames since frame index 1000
spRegion->setFrameIndex( 1000, 25 );
stream.add( spRegion );
// Link this to Jessica
spRegion->addReference( spJessica );

// Add a region
spRegion = std::shared_ptr< vmf::Metadata >( new vmf::Metadata( spRegionDesc ));
spRegion->setFieldValue( "left", 150 );
spRegion->setFieldValue( "top", 110 );
spRegion->setFieldValue( "width", 50 );
spRegion->setFieldValue( "height", 50 );
// 10 frames since frame index 1010
spRegion->setFrameIndex( 1020, 10 );
stream.add( spRegion );
// Link this to Alan
spRegion->addReference( spAlan );

// Add a region
spRegion = std::shared_ptr< vmf::Metadata >( new vmf::Metadata( spRegionDesc ));
spRegion->setFieldValue( "left", 300 );
spRegion->setFieldValue( "top", 200 );
spRegion->setFieldValue( "width", 50 );
spRegion->setFieldValue( "height", 50 );
// Only in frame 1030 and 1031
spRegion->setFrameIndex( 1030, 2 );
stream.add( spRegion );
spRegion->addReference( spJessica );

// Add a region
spRegion = std::shared_ptr< vmf::Metadata >( new vmf::Metadata( spRegionDesc ));
spRegion->setFieldValue( "left", 200 );
spRegion->setFieldValue( "top", 120 );
spRegion->setFieldValue( "width", 300 );
spRegion->setFieldValue( "height", 200 );
// Only in frame 1010
spRegion->setFrameIndex( 5000 );
stream.add( spRegion );
// Link this to all kids
spRegion->addReference( spJessica );
spRegion->addReference( spAlan );
spRegion->addReference( spMike );
```

The next two metadata are gps metadata, which are associated to only one frame each.

```
//---------------------------------------------------------------------------
// Create gps metadata
// Lake Tahoe: Lat: 38.959409 N | Lng: -119.906845 W | Alt: 2272 m
```

```
std::shared_ptr< vmf::Metadata > spGps = std::shared_ptr< vmf::Metadata >( new vmf::Metadata( spGpsDesc ));
spGps->setFieldValue( "lat", "38.959409 N" );
spGps->setFieldValue( "long", "-119.906845 W" );
spGps->setFieldValue( "alt", 2272.f );
spGps->setFrameIndex( 1010 );
stream.add( spGps );

// Lake Tahoe: Lat: 38.946326 N | Lng: -119.894829 W | Alt: 2741 m
spGps = std::shared_ptr< vmf::Metadata >( new vmf::Metadata( spGpsDesc ));
spGps->setFieldValue( "lat", "38.946326 N" );
spGps->setFieldValue( "long", "-119.894829 W" );
spGps->setFieldValue( "alt", 2761.f );
spGps->setFrameIndex( 1020 );
stream.add( spGps );
```

The last gps metadata is associated with multiple frames, indicating that the location of all those frames are the same.

```
// Palo Alto:  Lat: 37.433840 N | Lng: -122.128143 W | Alt: 26 m
spGps = std::shared_ptr< vmf::Metadata >( new vmf::Metadata( spGpsDesc ));
spGps->setFieldValue( "lat", "37.433840 N" );
spGps->setFieldValue( "long", "-122.128143 W" );
spGps->setFieldValue( "alt", 26.f );
spGps->setFrameIndex( 4001, 2500 );
stream.add( spGps );

//---------------------------------------------------------------------
// Create a timestamp metadata
// January 3, 2013
std::shared_ptr< vmf::Metadata > spTime = std::shared_ptr< vmf::Metadata >( new vmf::Metadata( spTimeDesc ));
spTime->setFieldValue( "year", 2013 );
spTime->setFieldValue( "month", 1 );
spTime->setFieldValue( "day", 3 );
spTime->setFieldValue( "hour", 9 );
spTime->setFieldValue( "minute", 15 );
spTime->setFieldValue( "second", 30 );
spTime->setFrameIndex( 1030 );
stream.add( spTime );

// Feb 19, 2013
spTime = std::shared_ptr< vmf::Metadata >( new vmf::Metadata( spTimeDesc ));
spTime->setFieldValue( "year", 2013 );
spTime->setFieldValue( "month", 2 );
spTime->setFieldValue( "day", 19 );
spTime->setFieldValue( "hour", 11 );
spTime->setFieldValue( "minute", 45 );
spTime->setFieldValue( "second", 30 );
spTime->setFrameIndex( 4030 );
stream.add( spTime );
}
```

5.4 Query Metadata

This section demonstrates the query functions.

A query is a function applied to the stream object, or a MetadataSet object. The very first query usually applies to

the stream object. Since this demo is about metadata within a specific schema, so ***queryBySchema***() routine is used to create the set of metadata for other queries.

```
void queryDemo( vmf::MetadataStream& stream, const std::string& sSchemaName )
{
  //-------------------------------------------------------------------------
  // Find all metadata of schema
  vmf::MetadataSet set = stream.queryBySchema( sSchemaName );
  std::cout << "There are " << set.size() << " metadata in the stream for schema " << sSchemaName << std::endl << std::endl;
```

The following query demonstrates using of generic query function, which takes a lambda expression as the input logic. In this case, the same functionality can be achieved by calling function ***queryByMetadata***().

```
  //-------------------------------------------------------------------------
  // Find all people
  vmf::MetadataSet setPeople = set.query( []( const std::shared_ptr< vmf::Metadata >& spMetadata )->bool
  {
    return spMetadata->getName() == "people";
  });
  std::cout << "There are " << setPeople.size() << " people found in the stream: ";
  std::for_each( setPeople.begin(), setPeople.end(), []( const std::shared_ptr< vmf::Metadata>& spPeople )
  {
    std::cout << (std::string)spPeople->getFieldValue( "name" ) << ", ";
  });
  std::cout << std::endl << std::endl;
```

The following code demonstrates that user-defined logic can be combined with query functions to refine the query result.

```
  //-------------------------------------------------------------------------
  // Find all boys
  std::cout << "Boys are: ";
  std::for_each( setPeople.begin(), setPeople.end(), []( const std::shared_ptr< vmf::Metadata>& spPeople )
  {
    if( (char)spPeople->getFieldValue( "sex" ) == 'M' )
      std::cout << (std::string)spPeople->getFieldValue( "name" ) << ", ";
  });
  std::cout << std::endl << std::endl;
```

Here is an example of query based on selective field values. The routine ***queryByNameAndFields***() allows caller to compare values based on a selection of fields. The same function can also be achieved by using the generic query function, with user defined lambda expression logic.

```
  //-------------------------------------------------------------------------
  // Find boy at age 8
  std::vector< vmf::FieldValue > vFieldValues;
  vFieldValues.emplace_back( vmf::FieldValue( "sex", 'M' ));
  vFieldValues.emplace_back( vmf::FieldValue( "age", 8 ));
  vmf::MetadataSet setBoyAge8 = set.queryByNameAndFields( "people", vFieldValues );
  std::cout << "There are " << setBoyAge8.size() << " at age 8: ";
  std::for_each( setBoyAge8.begin(), setBoyAge8.end(), []( const std::shared_ptr< vmf::Metadata>& spPeople )
  {
    std::cout << (std::string)spPeople->getFieldValue( "name" ) << ", ";
  });
  std::cout << std::endl << std::endl;
```

Here is another example of mixing custom logic with query function. The first query collects all metadata of type "event" into ***setEvents***. Then a ***for_each*** loop checks referenced metadata of all event, and prints the name value of metadata object that has type of "people".

```
//---------------------------------------------------------------------------
// Find all events and people who went to that event
vmf::MetadataSet setEvents = set.queryByName( "event" );
std::cout << "There are " << setEvents.size() << " events in the stream: " << std::endl;
std::for_each( setEvents.begin(), setEvents.end(), []( const std::shared_ptr< vmf::Metadata>& spEvent )
{
   std::cout << (std::string)spEvent->at(0) << ": ";
   vmf::MetadataSet setRefPeople = spEvent->getReferencesByMetadata( "people" );
   std::for_each( setRefPeople.begin(), setRefPeople.end(), []( const std::shared_ptr< vmf::Metadata>& spPeople )
   {
      std::cout << (std::string)spPeople->getFieldValue( "name" ) << ", ";
   });
   std::cout << std::endl;

});
std::cout << std::endl;
```

The query below searches for all boys in video frame of 5000. The query is based on the generic query function. The lambda expression logic defines that the metadata has to meet two conditions: i) it has to be a "people" metadata; ii) its frame index covers frame 5000.

```
//---------------------------------------------------------------------------
// Find all boys in a frame
int nFrameIndex = 5000;
// Find all region in that frame
vmf::MetadataSet setRegionsInFrame = set.query( [&]( const std::shared_ptr< vmf::Metadata >& spMetadata )->bool
{
   return spMetadata->getFrameIndex() >= nFrameIndex &&
      nFrameIndex < spMetadata->getFrameIndex() + spMetadata->getNumOfFrames() &&
      spMetadata->getName() == "region";
});
```

The following is a query that does not call any of the ***IQuery*** functions. It is pure custom logic and c++ std algorithms.
The goal of the query is to find all boys that are referenced by any of the regions returned from the above query. The query starts by searching all people metadata that are referenced by metadata in the ***setRegionsInFrame*** set, which is the result of the previous query.
Two layers of ***for_each()*** loops are used in this query: the first layer loops through all region metadata found so far; the second layer loops through the people metadata referenced by each of the region metadata. The core logic checks to see if the "***sex***" field of the people metadata is '***M***', and adds the metadata data to the result set if so.

```
// Find all boys that are referenced by any of the regions
vmf::MetadataSet setBoysInFrame;
std::for_each( setRegionsInFrame.begin(), setRegionsInFrame.end(), [&]( std::shared_ptr<vmf::Metadata>& spRegion )
{
   vmf::MetadataSet setPeople = spRegion->getReferencesByMetadata( "people" );
   std::for_each( setPeople.begin(), setPeople.end(), [&]( std::shared_ptr<vmf::Metadata>& spPeople )
   {
      if( (char)spPeople->getFieldValue( "sex" ) == 'M' )
      {
```

```
            setBoysInFrame.push_back( spPeople );
        }
    });
});
```

The above logic collects all boys shown up in the target regions. However, there might be boys shown up in multiple regions. The following code removes any duplicated names from the result.

```
// Remove redundant entry
std::sort( setBoysInFrame.begin(), setBoysInFrame.end() );
setBoysInFrame.erase( std::unique( setBoysInFrame.begin(), setBoysInFrame.end() ), setBoysInFrame.end());

// Print result
std::cout << "There are " << setBoysInFrame.size() << " boys in frame " << nFrameIndex << ": ";
std::for_each( setBoysInFrame.begin(), setBoysInFrame.end(), []( const std::shared_ptr< vmf::Metadata>& spPeople )
{
   std::cout << (std::string)spPeople->getFieldValue( "name" ) << ", ";
});
std::cout << std::endl << std::endl;
}
```

5.5 Serialization

This section demonstrates the metadata Serialization possibilities.

The whole MetadataStream can be converted to a string representation with the following code:

```
vmf::MetadataStream stream;
// …
vmf::FormatXML xml;
std::string text = stream.serialize(xml);
```

And once serialized it can be deserealized back in the same or another process with the code:

```
vmf::MetadataStream stream;
vmf::FormatXML xml;
stream.deserialize(text, xml);
```

When e.g. just a metadata record needs to be serialized and then deserialized back (e.g. in a streaming scenario) the code looks a bit differently:

```
// serialize
vmf::FormatJSON json;
vmf::Metadata m = …
std::string text = json->store({m});
// …
// …
// deserialize
std::vector<vmf::MetadataInternal> mi;
std::vector<std::shared_ptr<vmf::MetadataSchema>> schemas;
std::vector< std::shared_ptr<vmf::MetadataStream::VideoSegment>> segments;
std::vector<std::shared_ptr<vmf::Stat>> stats;
vmf::Format::AttribMap attribs;
vmf::Format::ParseCounters actual = json->parse(text, mi, schemas, segments, stats, attribs);
```

5.6 Compression

This section demonstrates the metadata Compression possibilities.

```
vmf::MetadataStream mdStream;
// …
// Saving metadata using builtin ZLib Compressor
mdStream.save(vmf::Compressor::builtinId());
// …
// loading in this case won’t require any code efforts and happens automatically
```

```
mdStream.load(fileName);
// …
```

With user defined Compressor:

```
class MyCustomCompressor : public vmf::Compressor
{
// …
};
shared_ptr<Compressor> compressor = make_shared< MyCustomCompressor >();
Compressor::registerNew(compressor);
MetadataStream mdStream;
// …
mdStream.save(compressor->getId());
// …
// before loading the custom compressed data the custom compressor needs to be registered (if not yet registered)
Compressor::registerNew(compressor);
mdStream.load(fileName);
// …
```

5.7 Encryption

This section demonstrates the metadata Encryption possibilities.

```
MetadataStream mdStream;
// …
std::shared_ptr<Encryptor> encryptor = std::make_shared<EncryptorDefault>("ThereIsNoSpoon");
mdStream.setEncryptor(encryptor);

if(scope == EncryptionScope::OneField)
    encryptField = true;
if(scope == EncryptionScope::OneMetadata)
    encryptRecord = true;
{
  gpsMetadata->push_back(FieldValue(GPS_COORD_LAT_FIELD, lat, encryptField));
  gpsMetadata->push_back(FieldValue(GPS_COORD_LNG_FIELD, lng));
  gpsMetadata->setTimestamp(time);
  gpsMetadata->setUseEncryption(encryptRecord);
}

if(scope == EncryptionScope::Whole)
    mdStream.setUseEncryption(true);
else if(scope == EncryptionScope::Schema)
    gpsSchema->setUseEncryption(true);
else if(scope == EncryptionScope::MetaDesc)
    metadesc->setUseEncryption(true);
else if(scope == EncryptionScope::FieldDesc)
    field.useEncryption = true;

mdStream.save();
```

5.8 Statistics

This section demonstrates the metadata Statistics possibilities.

```
vmf::MetadataStream mdStream;
// …
// Set up statistics object(s)
```

```
std::vector< vmf::StatField > fields;
fields.emplace_back( GPS_COUNT_COORD_NAME, GPS_SCHEMA_NAME, GPS_DESC, GPS_COORD_FIELD,
vmf::StatOpFactory::builtinName( vmf::StatOpFactory::BuiltinOp::Count ));
fields.emplace_back( GPS_COUNT_TIME_NAME, GPS_SCHEMA_NAME, GPS_DESC, GPS_TIME_FIELD,
vmf::StatOpFactory::builtinName( vmf::StatOpFactory::BuiltinOp::Count ));
mdStream.addStat( make_shared<vmf::Stat>( GPS_STAT_NAME, fields, vmf::Stat::UpdateMode::Disabled ));

mdStream.getStat(GPS_STAT_NAME)->setUpdateTimeout( 50 );
mdStream.getStat(GPS_STAT_NAME)->setUpdateMode( vmf::Stat::UpdateMode::OnTimer );

// Add to metadata a new item
mdStream.add(gpsMetadata);

mdStream.getStat(GPS_STAT_NAME)->update();

std::vector< std::string > fieldNames = mdStream.getStat(GPS_STAT_NAME)->getAllFieldNames();
for( auto& fieldName : fieldNames )
{
   const vmf::StatField& field = stat->getField( fieldName );
   std::cout << "  name='" << field.getName() << "'"
                "  operation='" << field.getOpName() << "'"
                "  metadata='" << field.getMetadataName() << "'"
                "  field='" << field.getFieldName() << "'"
                "  value=" << field.getValue()
             << std::endl;
}
```