VFilter C++ interface library

v1.1.0

Table of contents

Overview

The **VFilter** C++ library provides interface as well defines data structures for various video filters implementation. The **VFilter** interface represents video filter module which has video frames as input and output. The **VFilter.h** file contains declaration of data structures: VFilterCommand enum, VFilterParam enum, VFilterParams class and VFilter class declaration. The library depends on Frame library (describes data structures for video frames, Apache 2.0 license) and ConfigReader library (provides methods to read/write JSON config files, Apache 2.0 license). The library uses C++17 standard and doesn't have third-party dependencies to be installed in OS.

Versions

Table 1 - Library versions.

Version	Release date	What's new
1.0.0	20.02.2024	First version of the library.
1.1.0	22.02.2024	- Add setMask method. - Update documentation.

Library files

The library supplied by source code only. The user would be given a set of files in the form of a CMake project (repository). The repository structure is shown below:

```
CMakeLists.txt ------------------- Main CMake file of the library.
3rdparty ------------------------- Folder with third-party libraries.
    CMakeLists.txt --------------- CMake file which includes third-party
libraries.
    ConfigReader ----------------- Source code of the ConfigReader library.
    Frame ------------------------ Source code of the Frame library.
src ------------------------------ Folder with source code of the library.
    CMakeLists.txt --------------- CMake file of the library.
    VFilter.cpp ------------------ Source code file of the library.
    VFilter.h -------------------- Header file which includes VFilter class
declaration.
    VFilterVersion.h ------------- Header file which includes version of the
library.
    VFilterVersion.h.in ---------- CMake service file to generate version file.
test ----------------------------- Folder for internal tests of library.
    CMakeLists.txt --------------- CMake file for tests application.
    main.cpp --------------------- Source code file tests application.
src ------------------------------ Folder with source code of the custom VFilter
implementation.
    CMakeLists.txt --------------- CMake file of the library.
    CustomVFilter.cpp ------------ Source code file of the library.
    CustomVFilter.h -------------- Header file which includes CustomVFilter class
declaration.
```

```
    CustomVFilterVersion.h ------- Header file which includes version of the
library.
    CustomVFilterVersion.h.in ---- CMake service file to generate version file.
```

VFilter interface class description

Class declaration

VFilter interface class declared in **VFilter.h** file. Class declaration:

```
class VFilter
{
public:

    /// Class destructor.
    virtual ~VFilter();

    /// Get the version of the VFilter class.
    static std::string getVersion();

    /// Set the value for a specific library parameter.
    virtual bool setParam(VFilterParam id, float value) = 0;

    /// Get the value of a specific library parameter.
    virtual float getParam(VFilterParam id) = 0;

    /// Get the structure containing all library parameters.
    virtual void getParams(VFilterParams& params) = 0;

    /// Execute a VFilter action command.
    virtual bool executeCommand(VFilterCommand id) = 0;

    /// Process frame.
    virtual bool processFrame(cr::video::Frame& frame) = 0;

    /// Set mask.
    virtual bool setMask(cr::video::Frame mask) = 0;

    /// Encode set param command.
    static void encodeSetParamCommand(uint8_t* data, int& size,
                                      VFilterParam id, float value);

    /// Encode command.
    static void encodeCommand(uint8_t* data, int& size, VFilterCommand id);

    /// Decode command.
    static int decodeCommand(uint8_t* data, int size, VFilterParam& paramId,
                                    VFilterCommand& commandId, float& value);

    /// Decode and execute command.
    virtual bool decodeAndExecuteCommand(uint8_t* data, int size) = 0;
```

```
};
```

getVersion method

The **getVersion()** static method returns string of current class version. Method declaration:

```
static std::string getVersion();
```

Method can be used without **VFilter** class instance:

```
std::cout << "VFilter version: " << cr::video::VFilter::getVersion();
```

Console output:

```
VFilter class version: 1.0.0
```

setParam method

setParam (...) method sets new parameters value. **VFilter** based library should provide thread-safe **setParam(...)** method call. This means that the **setParam(...)** method can be safely called from any thread. Method declaration:

```
virtual bool setParam(VFilterParam id, float value) = 0;
```

Parameter	Description
id	Parameter ID according to VFilterParam enum.
value	Parameter value. Value depends on parameter ID.

Returns: TRUE if the parameter was set or FALSE if not.

getParam method

getParam(...) method returns parameter value. **VFilter** based library should provide thread-safe **getParam(...)** method call. This means that the **getParam(...)** method can be safely called from any thread. Method declaration:

```
virtual float getParam(VFilterParam id) = 0;
```

Parameter	Description
id	Parameter ID according to VFilterParam enum.

Returns: parameter value or **-1** if the parameters doesn't exist (not supported in particular implementation).

getParams method

getParams(...) method is designed to obtain params structure. **VFilter** based library should provide thread-safe **getParams(...)** method call. This means that the **getParams(...)** method can be safely called from any thread. Method declaration:

```
virtual void getParams(VFilterParams& params) = 0;
```

Parameter	Description
params	Reference to VFilterParams object to store params.

executeCommand method

executeCommand(...) method executes library action command. **VFilter** based library should provide thread-safe **executeCommand(...)** method call. This means that the **executeCommand(...)** method can be safely called from any thread. Method declaration:

```
virtual bool executeCommand(VFilterCommand id) = 0;
```

Parameter	Description
id	Command ID according to VFilterCommand enum.

Returns: TRUE if the command executed or FALSE if not.

processFrame method

processFrame(...) method designed to process frame . **VFilter** based library should provide thread-safe **processFrame(...)** method call. This means that the **processFrame(...)** method can be safely called from any thread. Method declaration:

```
virtual bool processFrame(cr::video::Frame& frame) = 0;
```

Parameter	Description
frame	Reference to **Frame** object.

Returns: TRUE if frame processed or FALSE if not.

setMask method

setMask(...) method designed to set mask. Method declaration:

```
virtual bool setMask(cr::video::Frame mask) = 0;
```

Parameter	Description
mask	Filter mask is **Frame** object with GRAY pixel format. Filter omits image segments, where filter mask pixel values equal 0.

Returns: TRUE if the detection mask was set or FALSE if not.

encodeSetParamCommand method

encodeSetParamCommand(...) static method encodes command to change any **VFilter** parameter value. To control any video filter remotely, the developer has to design his own protocol and according to it encode the command and deliver it over the communication channel. To simplify this, the **VFilter** class contains static methods for encoding the control command. The **VFilter** class provides two types of commands: a parameter change command (SET_PARAM) and an action command (COMMAND). **encodeSetParamCommand(...)** designed to encode SET_PARAM command. Method declaration:

```
static void encodeSetParamCommand(uint8_t* data, int& size, VFilterParam id,
float value);
```

Parameter	Description
data	Pointer to data buffer for encoded command. Must have size >= 11.
size	Size of encoded data. Will be 11 bytes.
id	Parameter ID according to VFilterParam enum.
value	Parameter value.

SET_PARAM command format:

Byte	Value	Description
0	0x01	SET_PARAM command header value.
1	Major	Major version of VFilter class.
2	Minor	Minor version of VFilter class.
3	id	Parameter ID **int32_t** in Little-endian format.
4	id	Parameter ID **int32_t** in Little-endian format.
5	id	Parameter ID **int32_t** in Little-endian format.

Byte	Value	Description
6	id	Parameter ID **int32_t** in Little-endian format.
7	value	Parameter value **float** in Little-endian format.
8	value	Parameter value **float** in Little-endian format.
9	value	Parameter value **float** in Little-endian format.
10	value	Parameter value **float** in Little-endian format.

encodeSetParamCommand(...) is static and used without **VFilter** class instance. This method used on client side (control system). Command encoding example:

```
// Buffer for encoded data.
uint8_t data[11];
// Size of encoded data.
int size = 0;
// Random parameter value.
float outValue = static_cast<float>(rand() % 20);
// Encode command.
VFilter::encodeSetParamCommand(data, size, VFilterParam::LEVEL, outValue);
```

encodeCommand method

encodeCommand(...) static method encodes command for **VFilter** remote control. To control any video filter remotely, the developer has to design his own protocol and according to it encode the command and deliver it over the communication channel. To simplify this, the **VFilter** class contains static methods for encoding the control command. The **VFilter** class provides two types of commands: a parameter change command (SET_PARAM) and an action command (COMMAND). **encodeCommand(...)** designed to encode COMMAND command (action command). Method declaration:

```
static void encodeCommand(uint8_t* data, int& size, VFilterCommand id);
```

Parameter	Description
data	Pointer to data buffer for encoded command. Must have size >= 7.
size	Size of encoded data. Will be 7 bytes.
id	Command ID according to VFilterCommand enum.

COMMAND format:

Byte	Value	Description
0	0x00	COMMAND header value.
1	Major	Major version of VFilter class.

Byte	Value	Description
2	Minor	Minor version of VFilter class.
3	id	Command ID **int32_t** in Little-endian format.
4	id	Command ID **int32_t** in Little-endian format.
5	id	Command ID **int32_t** in Little-endian format.
6	id	Command ID **int32_t** in Little-endian format.

encodeCommand(...) is static and used without **VFilter** class instance. This method used on client side (control system). Command encoding example:

```
// Buffer for encoded data.
uint8_t data[7];
// Size of encoded data.
int size = 0;
// Encode command.
VFilter::encodeCommand(data, size, VFilterCommand::RESTART);
```

decodeCommand method

decodeCommand(...) static method decodes command on image filter side. Method declaration:

```
static int decodeCommand(uint8_t* data, int size, VFilterParam& paramId,
VFilterCommand& commandId, float& value);
```

Parameter	Description
data	Pointer to input command.
size	Size of command. Must be 11 bytes for SET_PARAM and 7 bytes for COMMAND.
paramId	VFilter parameter ID according to VFilterParam enum. After decoding SET_PARAM command the method will return parameter ID.
commandId	VFilter command ID according to VFilterCommand enum. After decoding COMMAND the method will return command ID.
value	VFilter parameter value (after decoding SET_PARAM command).

Returns: 0 - in case decoding COMMAND, **1** - in case decoding SET_PARAM command or **-1** in case errors.

decodeAndExecuteCommand method

decodeAndExecuteCommand(...) method decodes and executes command encoded by encodeSetParamCommand(...) and encodeCommand(...) methods on video filter side. The particular implementation of the VFilter must provide thread-safe **decodeAndExecuteCommand(...)** method call. This means that the **decodeAndExecuteCommand(...)** method can be safely called from any thread. Method declaration:

```
virtual bool decodeAndExecuteCommand(uint8_t* data, int size) = 0;
```

Parameter	Description
data	Pointer to input command.
size	Size of command. Must be 11 bytes for SET_PARAM or 7 bytes for COMMAND.

Returns: TRUE if command decoded (SET_PARAM or COMMAND) and executed (action command or set param command).

Data structures

VFilterCommand enum

Enum declaration:

```
enum class VFilterCommand
{
    /// Restart image filter algorithm.
    RESTART = 1,
    /// Enable filter.
    ON,
    /// Disable filter.
    OFF
};
```

Table 2 - Action commands description.

Command	Description
RESTART	Restart image filter algorithm.
ON	Enable video filter.
OFF	Disable video filter.

VFilterParam enum

Enum declaration:

```
enum class VFilterParam
{
    /// Current filter mode, usually 0 - off, 1 - on.
    MODE = 1,
    /// Enhancement level for particular filter, as a percentage in range from
    /// 0% to 100%.
    LEVEL,
    /// Processing time in microseconds. Read only parameter.
    PROCESSING_TIME_MCSEC,
    /// Type of the filter. Depends on the implementation.
    TYPE,
    /// VFilter custom parameter. Custom parameters used when particular image
    /// filter has specific unusual parameter.
    CUSTOM_1,
    /// VFilter custom parameter. Custom parameters used when particular image
    /// filter has specific unusual parameter.
    CUSTOM_2,
    /// VFilter custom parameter. Custom parameters used when particular image
    /// filter has specific unusual parameter.
    CUSTOM_3
};
```

Table 3 - Params description.

Parameter	Access	Description
MODE	read / write	Current filter mode, usually 0 - off, 1 - on (typical). Can have another meaning depends on implementation.
LEVEL	read / write	Enhancement level for particular filter, as a percentage in range from 0% to 100% (typical). Can have another meaning depends on implementation. The video filter should keep this value in memory. After enable video filter this value should be implemented.
PROCESSING_TIME_MCSEC	read only	Processing time in microseconds. Read only parameter. Used to control performance of video filter.
TYPE	read / write	Type of the filter. Depends on the implementation.
CUSTOM_1	read / write	VFilter custom parameter. Custom parameters used when particular image filter has specific unusual parameter.

Parameter	Access	Description
CUSTOM_2	read / write	VFilter custom parameter. Custom parameters used when particular image filter has specific unusual parameter.
CUSTOM_3	read / write	VFilter custom parameter. Custom parameters used when particular image filter has specific unusual parameter.

VFilterParams class description

Class declaration

VFilterParams class is used to provide video filter parameters structure. Also **VFilterParams** provides possibility to write/read params from JSON files (**JSON_READABLE** macro) and provides methods to encode and decode params. **VFilterParams** interface class declared in **VFilter.h** file. Class declaration:

```
class VFilterParams
{
public:
    /// Current filter mode, usually 0 - off, 1 - on.
    int mode{ 0 };
    /// Enhancement level for particular filter, as a percentage in range from
    /// 0% to 100%.
    int level{ 0 };
    /// Processing time in microseconds. Read only parameter.
    int processingTimeMcSec{ 0 };
    /// Type of the filter. Depends on the implementation.
    int type{ 0 };
    /// VFilter custom parameter. Custom parameters used when particular image
    /// filter has specific unusual parameter.
    float custom1{ 0.0f };
    /// VFilter custom parameter. Custom parameters used when particular image
    /// filter has specific unusual parameter.
    float custom2{ 0.0f };
    /// VFilter custom parameter. Custom parameters used when particular image
    /// filter has specific unusual parameter.
    float custom3{ 0.0f };

    /// Macro from ConfigReader to make params readable/writable from JSON.
    JSON_READABLE(VFilterParams, mode, level, processingTimeMcSec, type,
                  custom1, custom2, custom3)

    /// operator =
    VFilterParams& operator= (const VFilterParams& src);

    // Encode (serialize) params.
    bool encode(uint8_t* data, int bufferSize, int& size,
```

```
                                                VFilterParamsMask* mask = nullptr);
    // Decode (deserialize) params.
    bool decode(uint8_t* data, int dataSize);
};
```

Table 4 - **VFilterParams** class fields description is related to VFilterParam enum description.

Field	type	Description
mode	int	Current filter mode, usually 0 - off, 1 - on (typical). Can have another meaning depends on implementation.
level	int	Enhancement level for particular filter, as a percentage in range from 0% to 100% (typical). Can have another meaning depends on implementation. The video filter should keep this value in memory. After enable video filter this value should be implemented.
processingTimeMcSec	int	Processing time in microseconds. Read only parameter. Used to control performance of video filter.
type	int	Type of the filter. Depends on the implementation.
custom1	float	VFilter custom parameter. Custom parameters used when particular image filter has specific unusual parameter.
custom2	float	VFilter custom parameter. Custom parameters used when particular image filter has specific unusual parameter.
custom3	float	VFilter custom parameter. Custom parameters used when particular image filter has specific unusual parameter.

None: *VFilterParamsclass fields listed in Table 4* ***have to*** *reflect params set/get by methods setParam(...) and getParam(...).*

Serialize VFilter params

VFilterParams class provides method **encode(...)** to serialize VFilter params. Serialization of **VFilterParams** is necessary in case when image filter parameters have to be sent via communication channels. Method provides options to exclude particular parameters from serialization. To do this method inserts binary mask (1 byte) where each bit represents particular parameter and **decode(...)** method recognizes it. Method declaration:

```
bool encode(uint8_t* data, int bufferSize, int& size, VFilterParamsMask* mask =
nullptr);
```

Parameter	Value
data	Pointer to data buffer. Buffer size must be >= 48 bytes.
bufferSize	Data buffer size. Buffer size must be >= 48 bytes.

Parameter	Value
size	Size of encoded data.
mask	Parameters mask - pointer to **VFilterParamsMask** structure. **VFilterParamsMask** (declared in VFilter.h file) determines flags for each field (parameter) declared in VFilterParams class. If the user wants to exclude any parameters from serialization, he can put a pointer to the mask. If the user wants to exclude a particular parameter from serialization, he should set the corresponding flag in the **VFilterParamsMask** structure.

Returns: TRUE if params encoded (serialized) or FALSE if not.

VFilterParamsMask structure declaration:

```
struct VFilterParamsMask
{
    bool mode{ true };
    bool level{ true };
    bool processingTimeMcSec{ true };
    bool type{ true };
    bool custom1{ true };
    bool custom2{ true };
    bool custom3{ true };
};
```

Example without parameters mask:

```
// Prepare parameters.
cr::video::VFilterParams params;
params.level = 80.0f;

// Encode (serialize) params.
int bufferSize = 128;
uint8_t buffer[128];
int size = 0;
params.encode(buffer, bufferSize, size);
```

Example with parameters mask:

```
// Prepare parameters.
cr::video::VFilterParams params;
params.level = 80.0;

// Prepare mask.
cr::video::VFilterParams mask;
// Exclude level.
mask.level = false;

// Encode (serialize) params.
int bufferSize = 128;
uint8_t buffer[128];
int size = 0;
```

```
params1.encode(buffer, bufferSize, size, &mask);
```

Deserialize VFilter params

VFilterParams class provides method **decode(...)** to deserialize params. Deserialization of VFilterParams is necessary in case when it is needed to receive params via communication channels. Method automatically recognizes which parameters were serialized by **encode(...)** method. Method declaration:

```
bool decode(uint8_t* data, int dataSize);
```

Parameter	Value
data	Pointer to data buffer with serialized params.
dataSize	Size of command data.

Returns: TRUE if params decoded (deserialized) or FALSE if not.

Example:

```
// Prepare parameters.
cr::video::VFilterParams params1;
params1.level = 80;

// Encode (serialize) params.
int bufferSize = 128;
uint8_t buffer[128];
int size = 0;
params1.encode(buffer, bufferSize, size);

// Decode (deserialize) params.
cr::video::VFilterParams params2;
params2.decode(buffer, size);
```

Read params from JSON file and write to JSON file

VFilter depends on open source ConfigReader library which provides method to read params from JSON file and to write params to JSON file. Example of writing and reading params to JSON file:

```
// Write params to file.
cr::utils::ConfigReader inConfig;
cr::video::VFilterParams in;
inConfig.set(in, "vFilterParams");
inConfig.writeToFile("VFilterParams.json");

// Read params from file.
cr::utils::ConfigReader outConfig;
if(!outConfig.readFromFile("VFilterParams.json"))
{
    cout << "Can't open config file" << endl;
    return false;
}
```

VFilterParams.json will look like:

```
{
    "vFilterParams":
    {
        "level": 50,
        "mode": 2,
        "processingTimeMcSec": 23,
        "type": 1,
        "custom1": 0.7f,
        "custom2": 12.0f,
        "custom3": 0.61f
    }
}
```

Build and connect to your project

Typical commands to build **VFilter**:

```
git clone https://github.com/ConstantRobotics-Ltd/VFilter.git
cd VFilter
git submodule update --init --recursive
mkdir build
cd build
cmake ..
make
```

If you want connect **VFilter** to your CMake project as source code you can make follow. For example, if your repository has structure:

```
CMakeLists.txt
src
    CMakeList.txt
    yourLib.h
    yourLib.cpp
```

You can add repository **VFilter** as submodule by commands:

```
cd <your respository folder>
git submodule add https://github.com/ConstantRobotics-Ltd/VFilter.git
3rdparty/VFilter
git submodule update --init --recursive
```

In your repository folder will be created folder **3rdparty/VFilter** which contains files of **VFilter** repository with subrepository **ConfigReader**. New structure of your repository:

```
CMakeLists.txt
src
    CMakeList.txt
    yourLib.h
    yourLib.cpp
3rdparty
    VFilter
```

Create CMakeLists.txt file in **3rdparty** folder. CMakeLists.txt should contain:

```
cmake_minimum_required(VERSION 3.13)

################################################################################
## 3RD-PARTY
## dependencies for the project
################################################################################
project(3rdparty LANGUAGES CXX)

################################################################################
## SETTINGS
## basic 3rd-party settings before use
################################################################################
# To inherit the top-level architecture when the project is used as a submodule.
SET(PARENT ${PARENT}_YOUR_PROJECT_3RDPARTY)
# Disable self-overwriting of parameters inside included subdirectories.
SET(${PARENT}_SUBMODULE_CACHE_OVERWRITE OFF CACHE BOOL "" FORCE)

################################################################################
## CONFIGURATION
## 3rd-party submodules configuration
################################################################################
SET(${PARENT}_SUBMODULE_VFILTER                         ON  CACHE BOOL "" FORCE)
if (${PARENT}_SUBMODULE_VFILTER)
    SET(${PARENT}_VFILTER                               ON  CACHE BOOL "" FORCE)
    SET(${PARENT}_VFILTER_TEST                          OFF CACHE BOOL "" FORCE)
    SET(${PARENT}_VFILTER_EXAMPLE                       OFF CACHE BOOL "" FORCE)
endif()

################################################################################
## INCLUDING SUBDIRECTORIES
## Adding subdirectories according to the 3rd-party configuration
################################################################################
if (${PARENT}_SUBMODULE_VFILTER)
    add_subdirectory(VFilter)
endif()
```

File **3rdparty/CMakeLists.txt** adds folder **VFilter** to your project and excludes test application and example (VFilter class test application and example of custom **VFilter** class implementation) from compiling. Your repository new structure will be:

```
CMakeLists.txt
src
    CMakeList.txt
    yourLib.h
    yourLib.cpp
3rdparty
    CMakeLists.txt
    VFilter
```

Next you need include folder 3rdparty in main **CMakeLists.txt** file of your repository. Add string at the end of your main **CMakeLists.txt**:

```
add_subdirectory(3rdparty)
```

Next you have to include **VFilter** library in your **src/CMakeLists.txt** file:

```
target_link_libraries(${PROJECT_NAME} VFilter)
```

Done!

How to make custom implementation

The **VFilter** class provides only an interface, data structures, and methods for encoding and decoding commands and params. To create your own implementation of the video filter, VFilter repository has to be included in your project (see **Build and connect to your project** section). The catalogue **example** (see **Library files** section) includes an example of the design of the custom VFilter algorithm. All the methods of the VFilter interface class have to be included. Custom VFilter class declaration:

```
class CustomVFilter : public VFilter
{
public:

    // Class constructor.
    CustomVFilter();

    // Class destructor.
    ~CustomVFilter();

    // Get the version of the VFilter class.
    static std::string getVersion();

    // Set the value for a specific library parameter.
    bool setParam(VFilterParam id, float value) override;

    // Get the value of a specific library parameter.
    float getParam(VFilterParam id) override;
```

```
    // Get the structure containing all library parameters.
    void getParams(VFilterParams& params) override;

    // Execute a VFilter command.
    bool executeCommand(VFilterCommand id) override;

    // Process frame.
    bool processFrame(cr::video::Frame& frame) override;

    // Set mask.
    bool setMask(cr::video::Frame mask) override;

    // Decode and execute command.
    bool decodeAndExecuteCommand(uint8_t* data, int size);

private:

    /// Parameters structure (default params).
    VFilterParams m_params;
    /// Mutex for parameters access.
    std::mutex m_paramsMutex;
};
```