

object detector

ObjectDetector interface C++ library

v1.0.0

Table of contents

- [Overview](#)
- [Versions](#)
- [ObjectDetector interface class description](#)
 - [ObjectDetector class declaration](#)
 - [getVersion method](#)
 - [initObjectDetector method](#)
 - [setParam method](#)
 - [getParam method](#)
 - [getParams method](#)
 - [getObjects method](#)
 - [executeCommand method](#)
 - [detect method](#)
 - [encodeSetParamCommand method](#)
 - [encodeCommand method](#)
 - [decodeCommand method](#)
- [Data structures](#)
 - [ObjectDetectorCommand enum](#)
 - [ObjectDetectorParam enum](#)
 - [Object structure](#)
- [ObjectDetectorParams class description](#)
 - [ObjectDetectorParams class declaration](#)
 - [Serialize object detector params](#)
 - [Deserialize object detector params](#)
 - [Read params from JSON file and write to JSON file](#)
- [Build and connect to your project](#)

Overview

ObjectDetector C++ library provides standard interface as well defines data structures and rules for different object detectors (motion detectors, events detectors, neural networks etc.). **ObjectDetector** interface class doesn't do anything, just provides interface and defines data structures. Different object detector classes inherit interface form **ObjectDetector** C++ class. **ObjectDetector.h** file contains **ObjectDetectorParams** class, **ObjectDetectorCommand** enum, **ObjectDetectorParam** enum and includes **ObjectDetector** class declaration. **ObjectDetectorParams** class contains object detector params, list of detected objects and includes methods to encode and decode params. **ObjectDetectorCommand** enum contains IDs of commands. **ObjectDetectorParam** enum contains IDs of params. All object detectors should include params and commands listed in **ObjectDetector.h** file. ObjectDetector class dependency: [Frame](#) class which describes video frame structure and pixel formats, [ConfigReader](#) class which provides methods to work with JSON structures (read/write).

Versions

Table 1 - Library versions.

Version	Release date	What's new
1.0.0	17.07.2023	First version

ObjectDetector interface class description

ObjectDetector class declaration

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

```
class ObjectDetector
{
public:
    /**
     * @brief Get string of current library version.
     * @return String of current library version.
     */
    static std::string getVersion();
    /**
     * @brief Init object detector. All params will be set.
     * @param params Parameters structure.
     * @return TRUE if the object detector init or FALSE if not.
     */
    virtual bool initObjectDetector(ObjectDetectorParams& params) = 0;
    /**
     * @brief Set object detector param.
     * @param id Param ID.
```

```

    * @param value Param value to set.
    * @return TRUE if param was set of FALSE.
    */
virtual bool setParam(ObjectDetectorParam id, float value) = 0;
/**
    * @brief Get object detector param value.
    * @param id Param ID.
    * @return Param value or -1.
    */
virtual float getParam(ObjectDetectorParam id) = 0;
/**
    * @brief Get object detector params structure.
    * @return Object detector params structure.
    */
virtual ObjectDetectorParams getParams() = 0;
/**
    * @brief Get list of objects.
    * @return List of objects. If no detected object the list will be empty.
    */
virtual std::vector<Object> getObjects() = 0;
/**
    * @brief Execute command.
    * @param id Command ID.
    * @return TRUE if the command accepted or FALSE if not.
    */
virtual bool executeCommand(ObjectDetectorCommand id) = 0;
/**
    * @brief Perform detection.
    * @param frame Source video frame.
    * @return TRUE if video frame was processed or FALSE if not.
    */
virtual bool detect(cr::video::Frame& frame) = 0;
/**
    * @brief Encode set param command.
    * @param data Pointer to data buffer. Must have size >= 11.
    * @param size Size of encoded data.
    * @param id Param id.
    * @param value Param value.
    */
static void encodeSetParamCommand(
    uint8_t* data, int& size, ObjectDetectorParam id, float value);
/**
    * @brief Encode command.
    * @param data Pointer to data buffer. Must have size >= 11.
    * @param size Size of encoded data.
    * @param id Command ID.
    */
static void encodeCommand(
    uint8_t* data, int& size, ObjectDetectorCommand id);
/**
    * @brief Decode command.
    * @param data Pointer to command data.
    * @param size Size of data.
    * @param paramId Output command ID.
    * @param commandId Output command ID.

```

```

    * @param value Param or command value.
    * @return 0 - command decoded, 1 - set param command decoded, -1 - error.
    */
    static int decodeCommand(uint8_t* data,
                             int size,
                             ObjectDetectorParam& paramId,
                             ObjectDetectorCommand& commandId,
                             float& value);
};

```

getVersion method

getVersion() method returns string of current version of **ObjectDetector** class. Particular object detector class can have it's own **getVersion()** method. Method declaration:

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

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

```
std::cout << "ObjectDetector class version: " << ObjectDetector::getVersion() <<
std::endl;
```

Console output:

```
ObjectDetector class version: 1.0.0
```

initObjectDetector method

initObjectDetector(...) method initializes object detector. Method declaration:

```
virtual bool initObjectDetector(ObjectDetectorParams& params) = 0;
```

Parameter	Value
params	Object detector parameters class. Object detector should initialize all parameters listed in ObjectDetectorParams.

Returns: TRUE if the object detector initialized or FALSE if not.

setParam method

setParam(...) method designed to set new object detector parameter value. Method declaration:

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

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

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

getParam method

getParam(...) method designed to obtain object detector parameter value. Method declaration:

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

Parameter	Description
id	Parameter ID according to ObjectDetectorParam enum.

Returns: parameter value or -1 if the parameters doesn't exist in particular object detector class.

getParams method

getParams(...) method designed to obtain object detector params structures as well a list of detected objects. Method declaration:

```
virtual ObjectDetectorParams getParams() = 0;
```

Returns: object detector parameters structure (see **ObjectDetectorParams** class description).

getObjects method

getObjects() method designed to obtain list of detected objects. User can obtain list of detected objects via **getParams(...)** method as well. Method declaration:

```
virtual std::vector<Object> getObjects() = 0;
```

Returns: list of detected objects (see **Object** class description). If no detected object the list will be empty.

executeCommand method

executeCommand(...) method designed to execute object detector command. Method declaration:

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

Parameter	Description
id	Command ID according to ObjectDetectorCommand enum.

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

detect method

detect(...) method designed to perform detection algorithm. Method declaration:

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

Parameter	Description
frame	Video frame for processing. Object detector processes only RAW pixel formats (BGR24, RGB24, GRAY, YUYV24, YUYV, UYVY, NV12, NV21, YV12, YU12, see Frame class description).

Returns: TRUE is the video frame was processed FALSE if not. If object detector disabled (see **ObjectDetectorParam** enum description) the method should return TRUE.

encodeSetParamCommand method

encodeSetParamCommand(...) static method designed to encode command to change any parameter for remote object detector. To control object detector 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 **ObjectDetector** class contains static methods for encoding the control command. The **ObjectDetector** 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, ObjectDetectorParam 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 ObjectDetectorParam enum.
value	Parameter value. Value depends on parameter ID.

SET_PARAM command format:

Byte	Value	Description
0	0x01	SET_PARAM command header value.
1	0x01	Major version of ObjectDetector class.
2	0x00	Minor version of ObjectDetector 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.
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 **ObjectDetector** 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 = (float)(rand() % 20);
// Encode command.
ObjectDetector::encodeSetParamCommand(data, size, ObjectDetectorParam::MIN_OBJECT_WIDTH,
outValue);
```

encodeCommand method

encodeCommand(...) static method designed to encode command for remote object detector. To control object detector 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 **ObjectDetector** class contains static methods for encoding the control command. The **ObjectDetector** class provides two types of commands: a parameter change command (SET_PARAM) and an action command (COMMAND).

encodeCommand(...) designed to encode COMMAND (action command). Method declaration:

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

Parameter	Description
data	Pointer to data buffer for encoded command. Must have size >= 11.
size	Size of encoded data. Will be 11 bytes.

Parameter	Description
id	Command ID according to ObjectDetectorCommand enum.

COMMAND format:

Byte	Value	Description
0	0x00	COMMAND header value.
1	0x01	Major version of ObjectDetector class.
2	0x00	Minor version of ObjectDetector 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 **ObjectDetector** 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;
// Encode command.
ObjectDetector::encodeCommand(data, size, ObjectDetectorCommand::RESET);
```

decodeCommand method

decodeCommand(...) static method designed to decode command on object detector side (edge device). Method declaration:

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

Parameter	Description
data	Pointer to input command.
size	Size of command. Should be 11 bytes.
paramId	Parameter ID according to ObjectDetectorParam enum. After decoding SET_PARAM command the method will return parameter ID.
commandId	Command ID according to ObjectDetectorCommand enum. After decoding COMMAND the method will return command ID.

Parameter	Description
value	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.

Data structures

ObjectDetector.h file defines IDs for parameters (**ObjectDetectorParam** enum) and IDs for commands (**ObjectDetectorCommand** enum).

ObjectDetectorCommand enum

Enum declaration:

```
enum class ObjectDetectorCommand
{
    /// Reset.
    RESET = 1,
    /// Enable.
    ON,
    /// Disable.
    OFF
};
```

Table 2 - Object detector commands description. Some commands maybe unsupported by particular object detector class.

Command	Description
RESET	Reset algorithm.
ON	Enable object detector.
OFF	Disable object detector.

ObjectDetectorParam enum

Enum declaration:

```
enum class ObjectDetectorParam
{
    /// Logging mode. Values: 0 - Disable, 1 - Only file,
    /// 2 - Only terminal (console), 3 - File and terminal (console).
    LOG_MODE = 1,
    /// Frame buffer size. Depends on implementation.
    FRAME_BUFFER_SIZE,
    /// Minimum object width to be detected, pixels. To be detected object's
```

```

/// width must be >= MIN_OBJECT_WIDTH.
MIN_OBJECT_WIDTH,
/// Maximum object width to be detected, pixels. To be detected object's
/// width must be <= MAX_OBJECT_WIDTH.
MAX_OBJECT_WIDTH,
/// Minimum object height to be detected, pixels. To be detected object's
/// height must be >= MIN_OBJECT_HEIGHT.
MIN_OBJECT_HEIGHT,
/// Maximum object height to be detected, pixels. To be detected object's
/// height must be <= MAX_OBJECT_HEIGHT.
MAX_OBJECT_HEIGHT,
/// Minimum object's horizontal speed to be detected, pixels/frame. To be
/// detected object's horizontal speed must be >= MIN_X_SPEED.
MIN_X_SPEED,
/// Maximum object's horizontal speed to be detected, pixels/frame. To be
/// detected object's horizontal speed must be <= MAX_X_SPEED.
MAX_X_SPEED,
/// Minimum object's vertical speed to be detected, pixels/frame. To be
/// detected object's vertical speed must be >= MIN_Y_SPEED.
MIN_Y_SPEED,
/// Maximum object's vertical speed to be detected, pixels/frame. To be
/// detected object's vertical speed must be <= MAX_Y_SPEED.
MAX_Y_SPEED,
/// Probability threshold from 0 to 1. To be detected object detection
/// probability must be >= MIN_DETECTION_PROPABILITY.
MIN_DETECTION_PROPABILITY,
/// Horizontal track detection criteria, frames. By default shows how many
/// frames the objects must move in any(+/-) horizontal direction to be
/// detected.
X_DETECTION_CRITERIA,
/// Vertical track detection criteria, frames. By default shows how many
/// frames the objects must move in any(+/-) vertical direction to be
/// detected.
Y_DETECTION_CRITERIA,
/// Track reset criteria, frames. By default shows how many
/// frames the objects should be not detected to be excluded from results.
RESET_CRITERIA,
/// Detection sensitivity. Depends on implementation. Default from 0 to 1.
SENSITIVITY,
/// Frame scaling factor for processing purposes. Reduce the image size by
/// scaleFactor times horizontally and vertically for faster processing.
SCALE_FACTOR,
/// Num threads. Number of threads for parallel computing.
NUM_THREADS,
/// Processing time for last frame, mks.
PROCESSING_TIME_MKS,
/// Algorithm type. Depends on implementation.
TYPE,
/// Mode. Default: 0 - Off, 1 - On.
MODE,
/// Custom parameter. Depends on implementation.
CUSTOM_1,
/// Custom parameter. Depends on implementation.
CUSTOM_2,
/// Custom parameter. Depends on implementation.

```

```
CUSTOM_3
};
```

Table 3 - Object detector params description. Some params maybe unsupported by particular object detector class.

Parameter	Access	Description
LOG_MODE	read / write	Logging mode. Values: 0 - Disable, 1 - Only file, 2 - Only terminal, 3 - File and terminal.
FRAME_BUFFER_SIZE	read / write	Frame buffer size. Depends on implementation. It can be buffer size for image filtering or can be buffer size to collect frames for processing.
MIN_OBJECT_WIDTH	read / write	Minimum object width to be detected, pixels. To be detected object's width must be \geq MIN_OBJECT_WIDTH.
MAX_OBJECT_WIDTH	read / write	Maximum object width to be detected, pixels. To be detected object's width must be \leq MAX_OBJECT_WIDTH.
MIN_OBJECT_HEIGHT	read / write	Minimum object height to be detected, pixels. To be detected object's height must be \geq MIN_OBJECT_HEIGHT.
MAX_OBJECT_HEIGHT	read / write	Maximum object height to be detected, pixels. To be detected object's height must be \leq MAX_OBJECT_HEIGHT.
MIN_X_SPEED	read / write	Minimum object's horizontal speed to be detected, pixels/frame. To be detected object's horizontal speed must be \geq MIN_X_SPEED.
MAX_X_SPEED	read / write	Maximum object's horizontal speed to be detected, pixels/frame. To be detected object's horizontal speed must be \leq MAX_X_SPEED.
MIN_Y_SPEED	read / write	Minimum object's vertical speed to be detected, pixels/frame. To be detected object's vertical speed must be \geq MIN_Y_SPEED.
MAX_Y_SPEED	read / write	Maximum object's vertical speed to be detected, pixels/frame. To be detected object's vertical speed must be \leq MAX_Y_SPEED.
MIN_DETECTION_PROBABILITY	read / write	Probability threshold from 0 to 1. To be detected object detection probability must be \geq MIN_DETECTION_PROBABILITY. For example: neural networks for each detection result calculates detection probability from 0 to 1. MIN_DETECTION_PROBABILITY parameters used to filter detection results.

Parameter	Access	Description
X_DETECTION_CRITERIA	read / write	Horizontal track detection criteria, frames. By default shows how many frames the objects must move in any(+/-) horizontal direction to be detected.
Y_DETECTION_CRITERIA	read / write	Vertical track detection criteria, frames. By default shows how many frames the objects must move in any(+/-) vertical direction to be detected.
RESET_CRITERIA	read / write	Track reset criteria, frames. By default shows how many frames the objects should be not detected to be excluded from results.
SENSITIVITY	read / write	Detection sensitivity. Depends on implementation. Default from 0 to 1.
SCALE_FACTOR	read / write	Frame scaling factor for processing purposes. Reduce the image size by scaleFactor times horizontally and vertically for faster processing.
NUM_THREADS	read / write	Num threads. Number of threads for parallel computing.
PROCESSING_TIME_MKS	read only	Processing time for last frame, mks.
TYPE	read / write	Algorithm type. Depends on implementation.
MODE	read / write	Mode. Default: 0 - Off, 1 - On.
CUSTOM_1	read / write	Custom parameter. Depends on implementation.
CUSTOM_2	read / write	Custom parameter. Depends on implementation.
CUSTOM_3	read / write	Custom parameter. Depends on implementation.

Object structure

Object structure used to describe detected object. Object structure declared in **ObjectDetector.h** file. Structure declaration:

```
typedef struct Object
{
    /// Object ID. Must be uniques for particular object.
    int id{0};
    /// Object type. Depends on implementation.
    int type{0};
}
```

```

    /// Object rectangle width, pixels.
    int width{0};
    /// Object rectangle height, pixels.
    int height{0};
    /// Object rectangle top-left horizontal coordinate, pixels.
    int x{0};
    /// Object rectangle top-left vertical coordinate, pixels.
    int y{0};
    /// Horizontal component of object velocity, +-pixels/frame.
    float vx{0.0f};
    /// Vertical component of object velocity, +-pixels/frame.
    float vy{0.0f};
    /// Detection probability from 0 (minimum) to 1 (maximum).
    float p{0.0f};
} object;

```

Table 4 - Object structure fields description.

Field	Type	Description
id	int	Object ID. Must be unique for particular object. Object detector must assign unique ID for each detected object. This is necessary for control algorithms to distinguish different objects from frame to frame.
type	int	Object type. Depends on implementation. For example detection neural networks can detect multiple type of objects.
width	int	Object rectangle width, pixels. Must be MIN_OBJECT_WIDTH <= width <= MAX_OBJECT_WIDTH (see ObjectDetectorParam enum description).
height	int	Object rectangle height, pixels. Must be MIN_OBJECT_HEIGHT <= height <= MAX_OBJECT_HEIGHT (see ObjectDetectorParam enum description).
x	int	Object rectangle top-left horizontal coordinate, pixels.
y	int	Object rectangle top-left vertical coordinate, pixels.
vX	float	Horizontal component of object velocity, +-pixels/frame. if it possible object detector should estimate object velocity on video frames.
vY	float	Vertical component of object velocity, +-pixels/frame. if it possible object detector should estimate object velocity on video frames.
p	float	Detection probability from 0 (minimum) to 1 (maximum). Must be p >= MIN_DETECTION_PROBABILITY (see ObjectDetectorParam enum description).

ObjectDetectorParams class description

ObjectDetectorParams class declaration

ObjectDetectorParams class used for object detector initialization (**initObjectDetector(...)** method) or to get all actual params (**getParams()** method). Also **ObjectDetectorParams** provide structure to write/read params from JSON files (**JSON_READABLE** macro, see [ConfigReader](#) class description) and provide methods to encode and decode params. Class declaration:

```
class ObjectDetectorParams
{
public:
    /// Init string. Depends on implementation.
    std::string initString{""};
    /// Logging mode. Values: 0 - Disable, 1 - Only file,
    /// 2 - Only terminal (console), 3 - File and terminal (console).
    int logMode{0};
    /// Frame buffer size. Depends on implementation.
    int frameBufferSize{1};
    /// Minimum object width to be detected, pixels. To be detected object's
    /// width must be >= minObjectWidth.
    int minObjectWidth{4};
    /// Maximum object width to be detected, pixels. To be detected object's
    /// width must be <= maxObjectWidth.
    int maxObjectWidth{128};
    /// Minimum object height to be detected, pixels. To be detected object's
    /// height must be >= minObjectHeight.
    int minObjectHeight{4};
    /// Maximum object height to be detected, pixels. To be detected object's
    /// height must be <= maxObjectHeight.
    int maxObjectHeight{128};
    /// Minimum object's horizontal speed to be detected, pixels/frame. To be
    /// detected object's horizontal speed must be >= minXSpeed.
    float minXSpeed{0.0f};
    /// Maximum object's horizontal speed to be detected, pixels/frame. To be
    /// detected object's horizontal speed must be <= maxXSpeed.
    float maxXSpeed{30.0f};
    /// Minimum object's vertical speed to be detected, pixels/frame. To be
    /// detected object's vertical speed must be >= minYSpeed.
    float minYSpeed{0.0f};
    /// Maximum object's vertical speed to be detected, pixels/frame. To be
    /// detected object's vertical speed must be <= maxYSpeed.
    float maxYSpeed{30.0f};
    /// Probability threshold from 0 to 1. To be detected object detection
    /// probability must be >= minDetectionProbability.
    float minDetectionProbability{0.5f};
    /// Horizontal track detection criteria, frames. By default shows how many
    /// frames the objects must move in any(+/-) horizontal direction to be
    /// detected.
    int xDetectionCriteria{1};
    /// Vertical track detection criteria, frames. By default shows how many
    /// frames the objects must move in any(+/-) vertical direction to be
    /// detected.
    int yDetectionCriteria{1};
    /// Track reset criteria, frames. By default shows how many
    /// frames the objects should be not detected to be excluded from results.
    int resetCriteria{1};
```

```

    /// Detection sensitivity. Depends on implementation. Default from 0 to 1.
    float sensitivity{0.04f};
    /// Frame scaling factor for processing purposes. Reduce the image size by
    /// scaleFactor times horizontally and vertically for faster processing.
    int scaleFactor{1};
    /// Num threads. Number of threads for parallel computing.
    int numThreads{1};
    /// Processing time for last frame, mks.
    int processingTimeMks{0};
    /// Algorithm type. Depends on implementation.
    int type{0};
    /// Mode. Default: false - Off, on - On.
    bool enable{true};
    /// Custom parameter. Depends on implementation.
    float custom1{0.0f};
    /// Custom parameter. Depends on implementation.
    float custom2{0.0f};
    /// Custom parameter. Depends on implementation.
    float custom3{0.0f};
    /// List of detected objects.
    std::vector<Object> objects;

    JSON_READABLE(ObjectDetectorParams, initString, logMode, frameBufferSize,
                  minObjectWidth, maxObjectWidth, minObjectHeight, maxObjectHeight,
                  minXSpeed, maxXSpeed, minYSpeed, maxYSpeed, minDetectionProbability,
                  xDetectionCriteria, yDetectionCriteria, resetCriteria, sensitivity,
                  scaleFactor, numThreads, type, enable, custom1, custom2, custom3);

    /**
     * @brief operator =
     * @param src Source object.
     * @return ObjectDetectorParams object.
     */
    ObjectDetectorParams& operator= (const ObjectDetectorParams& src);

    /**
     * @brief Encode params. Method doesn't encode initString.
     * @param data Pointer to data buffer.
     * @param size Size of data.
     * @param mask Pointer to parameters mask.
     */
    void encode(uint8_t* data, int& size,
               ObjectDetectorParamsMask* mask = nullptr);

    /**
     * @brief Decode params. Method doesn't decode initString;
     * @param data Pointer to data.
     * @return TRUE is params decoded or FALSE if not.
     */
    bool decode(uint8_t* data);
};

```

Table 5 - ObjectDetectorParams class fields description.

Field	Type	Description
initString	string	Init string. Depends on implementation.

Field	Type	Description
logMode	int	Logging mode. Values: 0 - Disable, 1 - Only file, 2 - Only terminal, 3 - File and terminal.
frameBufferSize	int	Frame buffer size. Depends on implementation. It can be buffer size for image filtering or can be buffer size to collect frames for processing.
minObjectWidth	int	Minimum object width to be detected, pixels. To be detected object's width must be \geq minObjectWidth.
maxObjectWidth	int	Maximum object width to be detected, pixels. To be detected object's width must be \leq maxObjectWidth.
minObjectHeight	int	Minimum object height to be detected, pixels. To be detected object's height must be \geq minObjectHeight.
maxObjectHeight	int	Maximum object height to be detected, pixels. To be detected object's height must be \leq maxObjectHeight.
minXSpeed	float	Minimum object's horizontal speed to be detected, pixels/frame. To be detected object's horizontal speed must be \geq minXSpeed.
maxXSpeed	float	Maximum object's horizontal speed to be detected, pixels/frame. To be detected object's horizontal speed must be \leq maxXSpeed.
minYSpeed	float	Minimum object's vertical speed to be detected, pixels/frame. To be detected object's vertical speed must be \geq minYSpeed.
maxYSpeed	float	Maximum object's vertical speed to be detected, pixels/frame. To be detected object's vertical speed must be \leq maxYSpeed.
minDetectionProbability	float	Probability threshold from 0 to 1. To be detected object detection probability must be \geq minDetectionProbability. For example: neural networks for each detection result calculates detection probability from 0 to 1. minDetectionProbability parameters used to filter detection results.
xDetectionCriteria	int	Horizontal track detection criteria, frames. By default shows how many frames the objects must move in any(+/-) horizontal direction to be detected.
yDetectionCriteria	int	Vertical track detection criteria, frames. By default shows how many frames the objects must move in any(+/-) vertical direction to be detected.

Field	Type	Description
resetCriteria	int	Track reset criteria, frames. By default shows how many frames the objects should be not detected to be excluded from results.
sensitivity	float	Detection sensitivity. Depends on implementation. Default from 0 to 1.
scaleFactor	int	Frame scaling factor for processing purposes. Reduce the image size by scaleFactor times horizontally and vertically for faster processing.
numThreads	int	Num threads. Number of threads for parallel computing.
processingTimeMks	int	Processing time for last frame, mks.
type	int	Algorithm type. Depends on implementation.
enable	bool	Mode: false - Off, true - On.
custom1	float	Custom parameter. Depends on implementation.
custom2	float	Custom parameter. Depends on implementation.
custom3	float	Custom parameter. Depends on implementation.
objects	std::vector	List of detected objects.

None: *ObjectDetectorParams* class fields listed in Table 5 **must** reflect params set/get by methods *setParam(...)* and *getParam(...)*.

Serialize object detector params

ObjectDetectorParams class provides method **encode(...)** to serialize object detector params (fields of *ObjectDetectorParams* class, see Table 5). Serialization of object detector params necessary in case when you need to send params via communication channels. Method provides options to exclude particular parameters from serialization. To do this method inserts binary mask (3 bytes) where each bit represents particular parameter and **decode(...)** method recognizes it. Method doesn't encode *initString*. Method declaration:

```
void encode(uint8_t* data, int& size, ObjectDetectorParamsMask* mask = nullptr);
```

Parameter	Value
data	Pointer to data buffer. Buffer size should be at least 43 bytes.
size	Size of encoded data. 43 bytes by default.

Parameter	Value
mask	Parameters mask - pointer to ObjectDetectorParamsMask structure. ObjectDetectorParamsMask (declared in ObjectDetector.h file) determines flags for each field (parameter) declared in ObjectDetectorParams 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 ObjectDetectorParamsMask structure.

ObjectDetectorParamsMask structure declaration:

```
typedef struct ObjectDetectorParamsMask
{
    bool logMode{true};
    bool frameBufferSize{true};
    bool minObjectWidth{true};
    bool maxObjectWidth{true};
    bool minObjectHeight{true};
    bool maxObjectHeight{true};
    bool minXSpeed{true};
    bool maxXSpeed{true};
    bool minYSpeed{true};
    bool maxYSpeed{true};
    bool minDetectionProbability{true};
    bool xDetectionCriteria{true};
    bool yDetectionCriteria{true};
    bool resetCriteria{true};
    bool sensitivity{true};
    bool scaleFactor{true};
    bool numThreads{true};
    bool processingTimeMks{true};
    bool type{true};
    bool enable{true};
    bool custom1{true};
    bool custom2{true};
    bool custom3{true};
    bool objects{true};
} ObjectDetectorParamsMask;
```

Example without parameters mask:

```
// Prepare random params.
ObjectDetectorParams in;
in.logMode = rand() % 255;
in.objects.clear();
for (int i = 0; i < 5; ++i)
{
    Object obj;
    obj.id = rand() % 255;
    obj.type = rand() % 255;
    obj.width = rand() % 255;
    obj.height = rand() % 255;
    obj.x = rand() % 255;
```

```

    obj.y = rand() % 255;
    obj.vx = rand() % 255;
    obj.vy = rand() % 255;
    obj.p = rand() % 255;
    in.objects.push_back(obj);
}

// Encode data.
uint8_t data[1024];
int size = 0;
in.encode(data, size);
cout << "Encoded data size: " << size << " bytes" << endl;

```

Example with parameters mask:

```

// Prepare random params.
ObjectDetectorParams in;
in.logMode = rand() % 255;
in.objects.clear();
for (int i = 0; i < 5; ++i)
{
    Object obj;
    obj.id = rand() % 255;
    obj.type = rand() % 255;
    obj.width = rand() % 255;
    obj.height = rand() % 255;
    obj.x = rand() % 255;
    obj.y = rand() % 255;
    obj.vx = rand() % 255;
    obj.vy = rand() % 255;
    obj.p = rand() % 255;
    in.objects.push_back(obj);
}

// Prepare mask.
ObjectDetectorParamsMask mask;
mask.logMode = false;

// Encode data.
uint8_t data[1024];
int size = 0;
in.encode(data, size, &mask)
cout << "Encoded data size: " << size << " bytes" << endl;

```

Deserialize object detector params

ObjectDetectorParams class provides method **decode(...)** to deserialize params (fields of **ObjectDetectorParams** class, see Table 5). Deserialization of params necessary in case when you need to receive params via communication channels. Method automatically recognizes which parameters were serialized by **encode(...)** method. Method doesn't decode **initString**. Method declaration:

```

bool decode(uint8_t* data);

```

Parameter	Value
data	Pointer to encode data buffer.

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

Example:

```
// Prepare random params.
ObjectDetectorParams in;
in.logMode = rand() % 255;
for (int i = 0; i < 5; ++i)
{
    Object obj;
    obj.id = rand() % 255;
    obj.type = rand() % 255;
    obj.width = rand() % 255;
    obj.height = rand() % 255;
    obj.x = rand() % 255;
    obj.y = rand() % 255;
    obj.vX = rand() % 255;
    obj.vY = rand() % 255;
    obj.p = rand() % 255;
    in.objects.push_back(obj);
}

// Encode data.
uint8_t data[1024];
int size = 0;
in.encode(data, size);
cout << "Encoded data size: " << size << " bytes" << endl;

// Decode data.
ObjectDetectorParams out;
if (!out.decode(data))
{
    cout << "Can't decode data" << endl;
    return false;
}
```

Read params from JSON file and write to JSON file

ObjectDetector library depends on **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:

```
// Prepare random params.
ObjectDetectorParams in;
in.logMode = rand() % 255;
in.objects.clear();
for (int i = 0; i < 5; ++i)
{
    Object obj;
```

```

    obj.id = rand() % 255;
    obj.type = rand() % 255;
    obj.width = rand() % 255;
    obj.height = rand() % 255;
    obj.x = rand() % 255;
    obj.y = rand() % 255;
    obj.vX = rand() % 255;
    obj.vY = rand() % 255;
    obj.p = rand() % 255;
    in.objects.push_back(obj);
}

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

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

ObjectDetectorParams out;
if(!outConfig.get(out, "ObjectDetectorParams"))
{
    cout << "Can't read params from file" << endl;
    return false;
}

```

ObjectDetectorParams.json will look like:

```

{
  "ObjectDetectorParams": {
    "custom1": 57.0,
    "custom2": 244.0,
    "custom3": 68.0,
    "enable": false,
    "frameBufferSize": 200,
    "initString": "sfsfsfsfsf",
    "logMode": 111,
    "maxObjectHeight": 103,
    "maxObjectWidth": 199,
    "maxXSpeed": 104.0,
    "maxYSpeed": 234.0,
    "minDetectionProbability": 53.0,
    "minObjectHeight": 191,
    "minObjectWidth": 149,
    "minXSpeed": 213.0,
    "minYSpeed": 43.0,
    "numThreads": 33,
    "resetCriteria": 62,
    "scaleFactor": 85,
  }
}

```

```

        "sensitivity": 135.0,
        "type": 178,
        "xDetectionCriteria": 224,
        "yDetectionCriteria": 199
    }
}

```

Build and connect to your project

Typical commands to build **ObjectDetector** library:

```

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

```

If you want connect **ObjectDetector** library 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 **ObjectDetector** as submodule by commands:

```

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

```

In you repository folder will be created folder **3rdparty/ObjectDetector** which contains files of **ObjectDetector** repository with subrepositories **Frame** and **ConfigReader**. New structure of your repository:

```

CMakeLists.txt
src
    CMakeList.txt
    yourLib.h
    yourLib.cpp
3rdparty
    ObjectDetector

```

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(${PARENT}_SUBMODULE_OBJECT_DETECTOR ON CACHE BOOL "" FORCE)
if (${PARENT}_SUBMODULE_OBJECT_DETECTOR)
    SET(${PARENT}_OBJECT_DETECTOR ON CACHE BOOL "" FORCE)
    SET(${PARENT}_OBJECT_DETECTOR_TEST OFF CACHE BOOL "" FORCE)
endif()

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

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

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

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 ObjectDetector library in your **src/CMakeLists.txt** file:

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

Done!