constant robotics

CvTracker C++ library v8.0.0

programmer's manual

cvtracker

www.constantrobotics.com

CONTENTS

DOCUMENT VERSIONS

Table 1 – Document versions.

Version	Release date	What is new
7.2.0	12.12.2022	Programmer's manual for version 7.2.0 of the CvTracker C++ library.
8.0.0	25.12.2022	Programmer's manual for version 8.0.0 of the CvTracker C++ library.

LIBRARY VERSION

Table 2 – Library versions.

Version	Release date	What is new
7.2.0	12.12.2022	Version 7.2.0.
8.0.0	25.12.2022	**1.** The calculation speed has been increased. **2.** Tracking stability has been improved. **3.** Added support for YUV, YUYV, UYVY, NV12, RGB, BGR video formats for better tracking stability. **4.** The number of configurable parameters has been reduced. **5.** New control commands added.

DESCRIPTION

C++ library **CvTracker** version **8.0.0** is intended for automatic object video tracking. The library is written in C++ (C++17 standard) and uses OpenCV (version 4.5.0 and higher) library to perform forward and backward Fourier transform. The library is compatible **with any processors and operating systems** supporting C++ compiler (C++17 standard) and OpenCV library (version 4.5.0 and higher). The library provides fast calculation, compatibility with low-power processors, high accuracy and contains a lot of additional functions and modes, which allow using it in camera systems of any configuration. The library contains an advanced tracking algorithm **CSRM** developed by ConstantRobotics Ltd. The library provides tracking of low-contrast and small-sized objects against a complex background. The library contains a description of the C++ class **Cvt**. A single instance of the class provides tracking of a single object on video. To track several objects simultaneously, several instances of the **Cvt** class must be created.

LIBRARY FILES

Table 3 – Library source code files.

File	Description
CvtDataStructures.h	Header file containing constants and declaration of data structures.
Cvt.h	Header file containing a description of the Cvt C++ class.
CvtVersion.h	A header file containing a description of the library version.
CvtVersion.h.in	File containing a description of the version of the library needs to configure CMake.
Cvt.cpp	Cvt C++ class implementation file.
CsrmTracker.h	Header file containing a description of the CsrmTracker C++ class that implements the CSRM video tracking algorithm.
CsrmTracker.cpp	CsrmTracker C++ class implementation file.
CMakeLists.txt	CMake file to compile the library.

Table 4 – Library files in case delivery in compiled form without source code.

File	Description
CvtDataStructures.h	Header file containing constants and declaration of data structures.
Cvt.h	Header file containing a description of the Cvt C++ class.
CvtVersion.h	Header file describing the version of the CvTracker library.
Cvt.lib	C++ static library file for Windows operating system.
Cvt.a	C++ static library file for Linux operating systems.

KEY FEATURES AND CAPABILITIES

Table 5 – Key features and capabilities of the library.

Parameter	Value and description
Programming language.	C++ (standard C++17) using the OpenCV library (version 4.5.0 and higher) to perform fast Fourier transforms.
Compatibility with different operating systems.	Compatible with any operating system that supports the C++ compiler (standard C++17) and the OpenCV library (version 4.5.0 and higher).
Maximum size of the tracking rectangle.	**128x128** pixels. It is possible to track part of an object if it does not fit into the tracking rectangle. The shape of the tracking rectangle can be any within the minimum and maximum allowable limits.
Minimum size of the tracking rectangle.	**16x16** pixels. The object can be **2x2** pixels size for normal tracking.
Minimum object size.	**2x2** pixels.
The minimum object contrast.	**5%.** Contrast refers to the ratio of the difference between the average brightness of pixels belonging to the object and the average brightness of pixels belonging to the background. The demo application is designed to evaluate this parameter.
Maximum object offset per 1 frame.	The library provides tracking of objects as they change their position (change the position of the object center) per one video frame of up to **110 pixels** in any direction. The maximum allowable object displacement per video frame is determined by the search window size, which is set by the user.
Discreteness of the calculation of object coordinates.	**1 pixel** when estimating the center of an object (center of a tracking rectangle).
Object size estimation.	The library estimates the size and position of an object within the tracking rectangle to enable automatic adjustment of its position and size at the operator's command.
Auto adjustment of the tracking rectangle position.	The library can automatically adjust the position of the tracking rectangle while tracking an object. This allows to reduce the probability of tracking failure in the case of tracking dynamic maneuvering objects. User can enable/disable this function.
Auto adjustment of the tracking rectangle size.	The library can automatically adjust the size of the tracking rectangle while tracking an object. This allows to reduce the probability of tracking failure in the case of tracking dynamic maneuvering objects. User can enable/disable this function.
Object speed estimation.	The library calculates the horizontal and vertical components of object speed in video frames (pixels per frame).
Changing the parameters.	The library allows user to change parameters of the tracking algorithm even while tracking. Excepts input pixel format and number of color channels for processing.
Supported pixel formats.	Supported pixel formats of input video frames: **Grayscale**, **BGR**, **RGB**, **YUV**, **YUYV**, **UYVY** and **NV12**. For each pixel format the algorithm can use one or more color channels for processing. See section "Supported pixel formats".
Maximum and minimum video frame sizes to be processed.	The maximum size of the video frames for processing is **8192x8192 pixels**, the minimum is **240x240 pixels**.

Parameter	Value and description
Calculation time.	The library performs calculations for each video frame. Calculation speed does not depend on video frame sizes, but depends on library parameters. The main parameters which determine calculation speed are: **1.** search windows size, **2.** input pixel format, **3.** number of color channels for processing. The library does not perform any background tasks. **The library uses only one physical or logical processor core to perform calculations.**
Object loss detection.	The library automatically detects when an object is lost and switches the algorithm into LOST mode – trajectory prediction mode. When the object detection criteria are met, the library automatically recaptures the object (TRACKING mode).
Adaptation to object shape and size changes.	While tracking an object, the library adapts to object shape, size and brightness changes.
Obstacles processing.	If an object is partially (**up to 50%**) blocked by a barrier, there is no tracking loss. The performance of the library in specific situations can be evaluated with the demo application.
Object search window.	The size of the object's search area is set by the user in the library's parameters. The library's tracking algorithm searches object in a search area whose center coincides with the center of the tracking rectangle in the previous video frame. The library allows you to set only the following search area widths: **128** or **256** pixels, and possible search area heights of **128** or **256** pixels in any combination. It is recommended that the search area width and height be set to the same value.
Type of tracking algorithm.	The calculations are performed using a modified correlation tracking algorithm **CSRM** developed by Constant Robotics Ltd.
STOP-FRAME function and compensation for communication delays.	The library allows user to compensate time delays that occur in communication channels when transmitting object capture commands. The library also allows you to implement a STOP-FRAME mode to assist the operator in capturing dynamic objects. The duration of the STOP-FRAME mode can be set in the library parameters (see “Library parameters”).

Note: The values in the table are applied to the concept of video frame(s) and the concept of pixel.

LIBRARY PRINCIPLE

HOW THE TRACKING ALGORITHM WORKS

The principle of operation, implemented in the library algorithms **CSRM** based on correlation search. Correlation search in this case means direct “pixel to pixel” comparison of rectangular parts of processed frame with pattern of tracking object by calculating some measure of comparison (correlation function) with subsequent selection of that part of processed video frame which has the greatest similarity with pattern (has the highest value of correlation function). At the moment of object capturing, the rectangular area of the video frame (capture rectangle) specified in the capture parameters (position and size) is taken as the object reference image, on the basis of which the pattern is formed. The algorithm then searches an object in each frame of the video in particular search window. Search window is area bounded by the algorithm's parameters with the center coinciding with the calculated center of the tracking rectangle on the previous video frame (or the center of the capture rectangle if the first frame after capture is being processed). The calculated most probable position of the tracking object (with highest value of correlation function) in the current video frame (calculated center of the tracking rectangle) is taken as the coordinates of the object. The tracking algorithm considers the probability of object presence (based on the motion parameters of the object) at a particular frame position. Figure 1 shows object search principle.

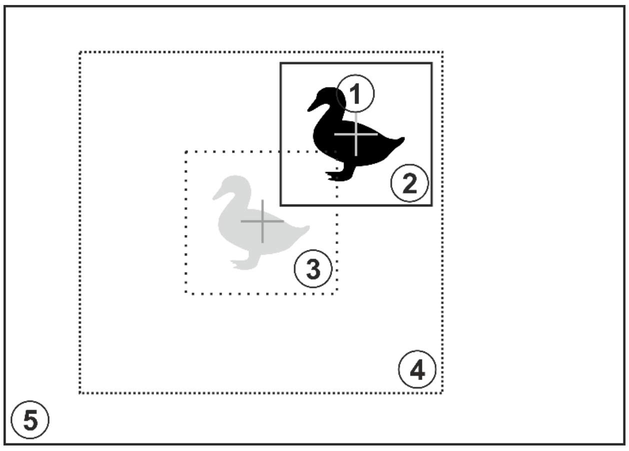


Figure 1 – Object search principle in a video frame.
(1 – object image on the current frame, 2 – tracking rectangle calculated after processing of the current frame, 3 – position of the tracking rectangle on the previous frame, 4 – object search window on the current frame relative to the position of the tracking rectangle on the previous frame, 5 – current video frame)

Figure 1 shows a schematic representation of a video frame (**5**) that contains an image of a object (**1**). Assume that on the previous video frame the object was in the area corresponding to area (**3**), which is the area of the tracking rectangle (the most probable position of the object) in the previous video frame. The library performs object search in the area (**4**) whose center coincides with the position of the center of the tracking rectangle (**3**) in the previous video frame.

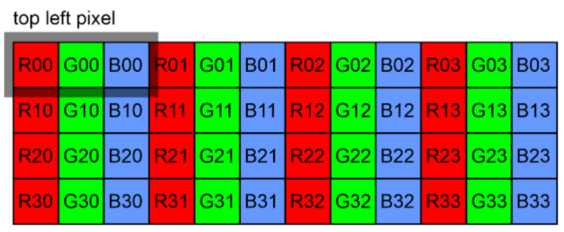
The tracking algorithm does not distinguish between pixels belonging to an object or background within the tracking rectangle immediately after capturing an object. Over time (as several frames are processed), the algorithm estimates whether a pixel within the tracking rectangle belongs to an object or to the background. Based on this information, the algorithm improves the quality of further tracking and estimates the size and position of the object (object image) within the tracking rectangle to enable subsequent automatic parameter adjustments at the operator's command or fully automatic.

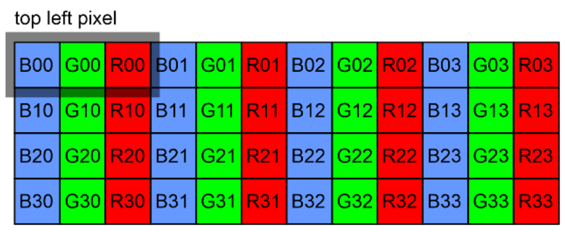
Calculation of object movement (horizontal and vertical velocity components) is performed for each processed video. For each processed video frame, the algorithm calculates the position of the center of the tracking rectangle, the position and size of the object rectangle (the rectangle describes the size of the object image) in the tracking rectangle, and the speed components of the tracking object on the video frames (pixels per frame). The algorithm performs object search in search window (in all possible positions of object within search window). The algorithm generates a surface of the spatial distribution of the probability (correlation surface) of the object presence in the search window. Once the surface is formed, it is analyzed to determine the most probable position of the object on the processed video frame (position of maximum value of the correlation surface).

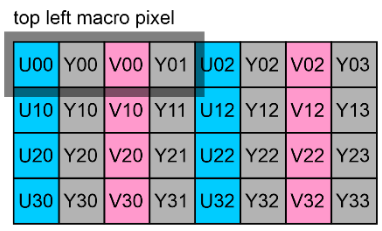
SUPPORTED PIXEL FORMATS

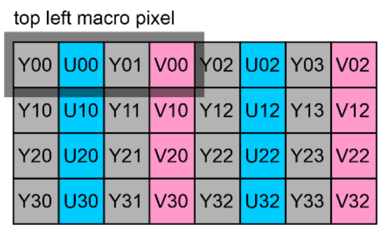
The library supports the following pixel formats: **Grayscale** 8 bit, **RBG** 24 bits (8 bit for each color components), **BGR** 24 bits (8 bit for each color components), **YUV** 24 bits (8 bit for each color components), **YUYV** (**YUY2**, 8 bit for each color components), **UYVY** (8 bit for each color component) and **NV12** (8 bit for each color components). For each input pixel format the user can set number of color channels for processing. Number of channels determines calculation time. Table 6 shows color components layout for each supported pixel format. The library interprets input data according to input pixel format set by user in advance (before video frames processing). **Warning: the input pixel format and number of color channels for processing must be set in advance (before first video frame processing).**

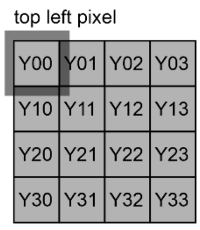
Table 6 – Illustrations of 4x4 pixels image bytes order in various formats.


top left pixel
R00 G00 B00 R01 G01 B01 R02 G02 B02 R03 G03 B03
R10 G10 B10 R11 G11 B11 R12 G12 B12 R13 G13 B13
R20 G20 B20 R21 G21 B21 R22 G22 B22 R23 G23 B23
R30 G30 B30 R31 G31 B31 R32 G32 B32 R33 G33 B33
RGB 24 bits pixel format.

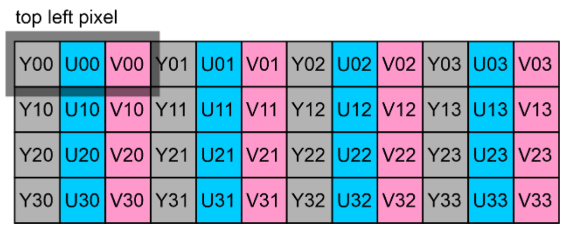

top left pixel
B00 G00 R00 B01 G01 R01 B02 G02 R02 B03 G03 R03
B10 G10 R10 B11 G11 R11 B12 G12 R12 B13 G13 R13
B20 G20 R20 B21 G21 R21 B22 G22 R22 B23 G23 R23
B30 G30 R30 B31 G31 R31 B32 G32 R32 B33 G33 R33
BGR 24 bits pixel format.

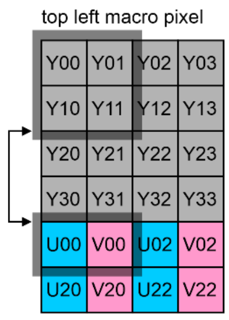

top left macro pixel
U00 Y00 V00 Y01 U02 Y02 V02 Y03
U10 Y10 V10 Y11 U12 Y12 V12 Y13
U20 Y20 V20 Y21 U22 Y22 V22 Y23
U30 Y30 V30 Y31 U32 Y32 V32 Y33
UYVY pixel format.


top left macro pixel
Y00 U00 Y01 V00 Y02 U02 Y03 V02
Y10 U10 Y11 V10 Y12 U12 Y13 V12
Y20 U20 Y21 V20 Y22 U22 Y23 V22
Y30 U30 Y31 V30 Y32 U32 Y33 V32
YUYV (YUY2) pixel format.



Grayscale pixel format.


top left pixel
Y00 U00 V00 Y01 U01 V01 Y02 U02 V02 Y03 U03 V03
Y10 U10 V10 Y11 U11 V11 Y12 U12 V12 Y13 U13 V13
Y20 U20 V20 Y21 U21 V21 Y22 U22 V22 Y23 U23 V23
Y30 U30 V30 Y31 U31 V31 Y32 U32 V32 Y33 U33 V33
YUV 24 bits pixel format.



NV12 pixel format.

For each pixel format the user can set number of color channels. The number of color channels for processing must be set via **setParam(…)** method of **Cvt** C++ class. Table 7 shows possible combination of pixel format and number of channels.

Table 7 – Combinations of pixel format and number of color channels for processing.

Num channels	Description
BGR – maximum 4 channel (3 colors + grayscale)	
1	The library will convert BGR video frame to Grayscale format.
2	The library will use only B and G channels.
3	The library will use B, G and R channels.
4	The library will use B, G and R channels and will make Grayscale channel.
RGB – maximum 4 channel (3 colors + grayscale)	
1	The library will convert BGR video frame to Grayscale format.
2	The library will use only R and G channels.
3	The library will use B, G and R channels.
4	The library will use B, G and R channels and will make Grayscale channel.
YUV – maximum 3 channel (3 color components). If set 4 channels the library will switch to 3 channels automatically.	
1	The library will use Y channel (Grayscale).

2	The library will use Y and U channel.
3	The library will use Y, U and V channels.
YUYV (YUY2) – maximum 3 channel (3 color components). If set 4 channels the library will switch to 3 channels automatically.	
1	The library will use Y channel (Grayscale).
2	The library will use Y and U channel.
3	The library will use Y, U and V channels.
UYVY – maximum 3 channel (3 color components). If set 4 channels the library will switch to 3 channels automatically.	
1	The library will use Y channel (Grayscale).
2	The library will use Y and U channel.
3	The library will use Y, U and V channels.
Grayscale – maximum 1 grayscale channel. If set > 1 channel the library will switch to 1 channel automatically.	
1	The library will use only one Grayscale channel.

STOP-FRAME FUNCTION

If a moving object needs to be captured, this can often be difficult for the user because they need to align the tracking rectangle with an object that keeps changing its position on the video frames. In addition, it is difficult to capture a stationary object in case of camera vibration. To help the user capture an object in challenging dynamic environments, the library has a STOP-FRAME function. This function allows the user to stop the video playback and accurately capture an object on a stopped video frame.

The function works as follows: video frames is put to the library for processing frame-by-frame. The library places frames in a ring buffer of the size specified by user in library parameters (see "Library parameters"). The tracking data contains an index corresponding to the position of the frame added to the ring buffer and is transmitted to the control system via communication channel. The user of the control system sees the video from the cameras on the monitor. Each video frame has its own identifier assigned by the tracking algorithm. The user can stop video playback, move the capture rectangle to an object on a stopped video frame and capture. When a capture command is formed, it includes a video frame identifier corresponding to the displayed video frame. When a capture command is received by the library, the object is captured on the frame in the frame buffer according to the identifier specified in the capture command. The video frame on which the object is captured will be some time behind the current video frame from the camera (the number of frames corresponding to the time the control system operator "stops" the video, with the addition of the delay time in the communication channels). After a capture, the algorithm switches to tracking mode. When processing subsequent frames, the library sequentially processes the frames in the frame buffer, starting with the frame where the object was captured. When a processing method of the library method is called, multiple frames are processed to "catch up" the current video frame. In this way the library "catches up" the current video frame from the camera in a short time and enters normal tracking mode. This function significantly reduces the skills requirements for users of tracking systems. **WARNING:** until the library has "caught up" the current video frame, it is not recommended to turn the pan-tilt systems. This may result in incorrect operation of the tracking systems.

COMMUNICATION CHANNEL DELAY COMPENSATION

When controlling the tracking system remotely (via communication channels), communication delays negatively affect the quality of the object being captured by the user. Video captured by the tracker device is compressed and transmitted to the control system with some delay. When the operator captures object the generated capture command also arrives at the tracker device with some delay. The capture command contains the coordinates of the capture rectangle center. When capturing a dynamic object, due to time delays in the communication channels, the captured area will not match the object. Figure 2 shows the displacement between capture rectangle and real object position.

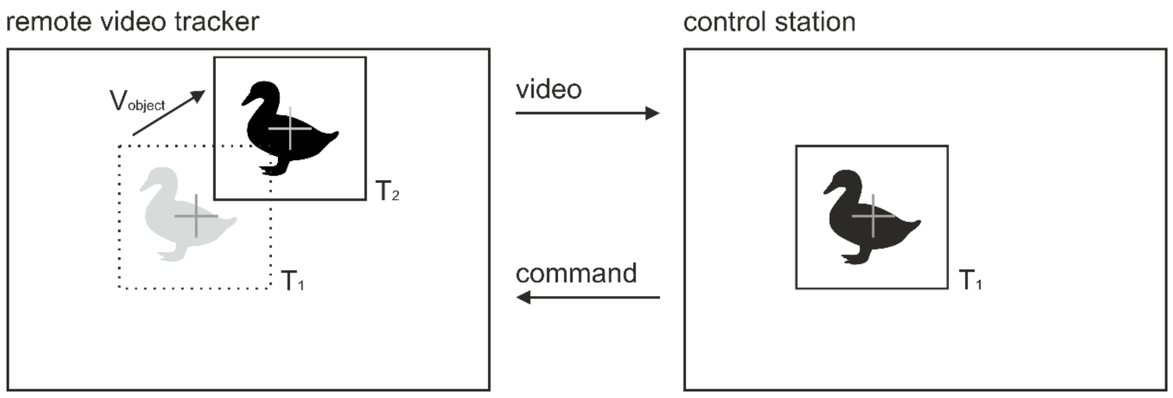


Figure 2 – Position error of the capture rectangle.

Figure 2 shows a video frame coming into the tracker device from the camera (left) and a video frame displayed to a user of the control system (right) at the same point of time. Assume that the video frame captured from the camera is simultaneously put to the tracking algorithm and sent to the control system. At time **T1**, the object was at one frame position. This frame after some time (encoding time + video frame transmission time + decoding time + display time) is displayed to the operator (user), who performs object capture. For operator the object position is corresponding to the time **T1**. Once an object has been captured, the capture command is sent to the tracking algorithm with some delay (command generation time + command transmission time + command decoding time) (time moment **T2**). The algorithm captures an object in the current video frame. If the object is moving, the error (horizontally and vertically) in the position of the capture rectangle will be as follows:

$$\Delta X = (T_2 - T_1) * V_x, \qquad (2)$$

$$\Delta Y = (T_2 - T_1) * V_y, \qquad (3)$$

where: ΔX – horizontal position error in pixels; ΔY – vertical position error in pixels; T_1 – the point in time corresponding to the frame displayed to the user; T_2 – the point in time corresponding to the current video frame; V_x – the horizontal component of the speed of an object in video frames (pixels per frame); V_y – the vertical component of the speed of the object in the video frames (pixels per frame).

To compensate for errors that occur, a video frame identifier must be included in the capture command. The principle for compensating of time delays in communication channels is identical to that of the STOP-FRAME function (see “STOP-FRAME function”).

OPERATION MODES

Table 8 – Tracking algorithm operating modes.

Mode	**Description**
FREE – free mode.	In this mode, the library does not perform any calculations. The library only adds video frames to the frame buffer. Conditions for entering FREE mode: 1. Once the Cvt C++ class has been initialized. This mode is the default mode. 2. Automatically when the automatic tracking reset criteria are met (see “Criteria for automatic change of operation modes”). 3. After command “RESET” (see “Library control commands”).
TRACKING – tracking mode.	In this mode the library calculates the automatic tracking and updates all calculated (estimated) object parameters. Criteria for entering TRACKING mode: 1. After the “CAPTURE” command.

Mode	Description
	2. Automatically from LOST mode when object detection criteria are met (see "Criteria for automatic change of operating modes").
LOST – object loss mode.	In this mode, the library searches object for automatic recapturing (switching to TRACKING mode) and updates it's coordinates in one of the ways specified in the parameters. LOST mode contains the following additional modes: **0.** Tracking rectangle coordinates are not updated (remain the same as before entering LOST mode). **1.** The tracking rectangle coordinates are updated based on the components of the object's speed calculated before going into LOST mode. When the tracking rectangle reaches any edge of the frame, the coordinate update in the corresponding direction stops. **2.** The tracking rectangle coordinates are updated based on the components of the speed of objects in the video frames calculated before going into LOST mode. When the tracking reset criteria is met, the device switches to FREE mode. Criteria for entering LOST mode: 1. Automatically when object loss is detected (see section "Criteria for automatic change of operating modes"). 2. On command from TRACKING mode. 3. On command from INERTIAL mode. 4. On command from STATIC mode.
INERTIAL – inertial tracking mode.	In this mode the library does not search for an object to recapture automatically, but only updates the coordinates of the tracking rectangle based on the previously calculated velocity components of the objects. Criteria for entering INERTIAL mode: 1. On command from TRACKING mode. 2. On command from LOST mode. 3. On command from STATIC mode.
STATIC – static mode.	This mode does not perform any calculations and the tracking rectangle coordinates remain the same as before going into this mode. This mode is necessary to "freeze" the tracking algorithm for a certain number of frames. For example, if the tracking system is exposed to strong vibrations, it is possible to "freeze" the tracking algorithm until the vibration ends.

Figure 3 shows the operating mode graph and the possible transitions between them. Also shown in the graph are commands (according to the **CvtCommand** structure, see "Library control commands") designed to change the modes of operation. The words **auto** in figure 3 indicate the ability to change the mode automatically if the relevant criteria are met (see "Criteria for automatic change of operating modes ").

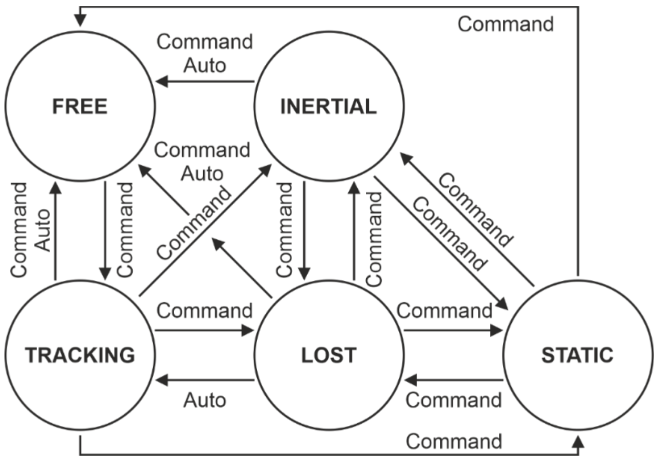


Figure 3 – Operation modes of the tracking algorithm.
(Auto – automatic mode change capability)

CRITERIA FOR AUTOMATIC CHANGE OF OPERATION MODES

Figure 3 shows the graph of operation modes. There are the following conditions for automatic mode changes (word “AUTO” in figure 3):

1. Automatic switching from **TRACKING** to **FREE** mode is possible only if the tracking rectangle center has touched (coincided in coordinates) any of the video frame edges.
2. The automatic switching from **TRACKING** to **LOST** mode is possible when an object loss is detected – when the calculated object detection probability falls below the threshold.
3. Automatic switching from **LOST** to **TRACKING** mode is possible when an object is detected again after a loss – when the calculated object detection probability exceeds the threshold.
4. Automatic reset of tracking in the **LOST** mode (switch to **FREE** mode) is possible when the center of the tracking rectangle touches the edge of the video frame (if the LOST mode option set to 2), as well as when the number of frames specified in the parameters has expired, at which the algorithm is continuously in **LOST** mode.
5. Automatic reset of tracking in **INERTIAL** mode (switch to **FREE** mode) is possible when the center of the tracking rectangle reaches the edge of the frame.

HOW TO USE THE LIBRARY

HOW TO USE THE LIBRARY

The library consists of a few source code files: **Cvt.h**, **Cvt.cpp**, **CsrmTracker.h**, **CsrmTracker.cpp**, **CvtDataStructures.h** and **CvtVersion.h**. When delivered compiled version, the library includes four files: **Cvt.h**, **Cvt.a** (for Linux) or **Cvt.lib** (for Windows), **CvtDataStructures.h** and **CvtVersion.h**. To use the library, the developer must include the listed files in C++ project. The **Cvt.h** file contains the declaration of the Cvt C++ class, which implements the tracking algorithm. The way of use of the library (source code of simple test application given in section “Simple demo application”):

1. Connect the tracking library files and OpenCV library to your project.
2. Create an instance of the **Cvt** C++ class.
3. Set parameters of the library through calling the **setParam(...)** method, if necessary.
4. Call the **processFrame(...)** method to process the next video frame.
5. To perform commands use **executeCommand(...)** method.
6. The **getResults(...)** method is used to get the tracking results.

LIBRARY PARAMETERS

The library and tracking algorithm parameters are set by calling the setParam(...) method of the Cvt C++ class. The list of available parameters is contained in the CvtParam enum declared in the CvtDataStructures.h file. The declaration of CvtParam enum:

```
enum class CvtParam
{
    FRAME_BUFFER_SIZE = 1,
    TRACKING_RECTANGLE_WIDTH,
    TRACKING_RECTANGLE_HEIGHT,
    SEARCH_WINDOW_X,
    SEARCH_WINDOW_Y,
    SEARCH_WINDOW_WIDTH,
    SEARCH_WINDOW_HEIGHT,
    LOST_MODE_OPTION,
    MAXIMUM_NUM_FRAMES_IN_LOST_MODE,
    USE_TRACKING_RECTANGLE_AUTO_SIZE,
    USE_TRACKING_RECTANGLE_AUTO_POSITION,
    INPUT_FORMAT,
    NUM_CHANNELS
};
```

Table 9 – Description of the library parameters.

Parameter	Description
FRAME_BUFFER_SIZE	Frame buffer size. **Default value 2**. When processing the next video frame, the library copies the frame data to the buffer for further processing according to input pixel format and number of color channels for processing. The frame buffer is required for the STOP-FRAME function and to compensate time delay in communication channels (see “STOP-FRAME function”). The size of the frame buffer determines the time delay that the library can compensate for object capture. The possible delay can be calculated as follows: $$T = FRAME\_BUFFER\_SIZE * \frac{1000}{fps},$$ where: T – the maximum time delay (msec) that the library can compensate for; fps – the number of frames per second coming from the video source. **Valid value from 2 to 1024.**
TRACKING_RECTANGLE_WIDTH	Width of tracking rectangle. **The default value is 64**. **Valid values are from 16 to 128.**
TRACKING_RECTANGLE_HEIGHT	Height of tracking rectangle. **The default value is 64**. **Valid values are from 16 to 128.**
SEARCH_WINDOW_X	Horizontal position of the center of the object search window. The center of the search area after the next frame is processed always coincides with the center of the tracking rectangle. The user can change the position of the search windows center for the next video frame by setting the coordinate value using the setParam(…) method. After processing the next video frame, the position of the center of the search area will again coincide with the calculated center of the tracking rectangle.
SEARCH_WINDOW_Y	Vertical position of the center of the object search window.
SEARCH_WINDOW_WIDTH	The width of the search area. The default value is **256** pixels. The parameter can only take values of **128** or **256**. **Note:** The width of the search area must be less than or equal to the width of the tracking rectangle (TRACKING_RECTANGLE_WIDTH parameter). The size of the search area determines the speed of the calculation: the smaller the search area, the faster the calculation.
SEARCH_WINDOW_HEIGHT	The height of the search area. The default value is **256** pixels. The parameter can only take values of **128** or **256**. **Note:** The height of the search area must be less than or equal to the height of the tracking rectangle (TRACKING_RECTANGLE_HEIGHT parameter). The size of the search area determines the speed of the calculation: the smaller the search area, the faster the calculation.

Parameter	Description
LOST_MODE_OPTION	Parameter that defines the behavior of the tracking algorithm in **LOST** mode. **Default is 0**. Possible values: 0. In **LOST** mode, the coordinates of the center of the tracking rectangle are not updated and remain the same as before entering **LOST** mode. 1. The coordinates of the center of the tracking rectangle are updated based on the components of the object's speed calculated before going into **LOST** mode. If the tracking rectangle "touches" the edge of the video frame, the coordinate updating for this component (horizontal or vertical) will stop. 2. The coordinates of the center of the tracking rectangle are updated based on the components of the object's speed calculated before going into **LOST** mode. The tracking is reset if the center of the tracking rectangle touches any of the edges of the video frame.
MAXIMUM_NUM_FRAMES_IN _LOST_MODE	Maximum number of continuous video frames in **LOST** mode, after which the library performs an automatic tracking reset (see "Criteria for automatic mode changes"). **The default value is 256**.
USE_TRACKING_RECTANGLE _AUTO_SIZE	The flag of necessity of automatic tracking rectangle size adjustment during object tracking. Automatic tracking rectangle size adjustment allows to track maneuvering objects effectively when their size and shape change. **The default value is 0 (the function is disabled)**. Any value other than 0 indicates the necessity to use this function.
USE_TRACKING_RECTANGLE _AUTO_POSITION	The flag of necessity of automatic tracking rectangle position adjustment during object tracking. Automatic adjustment of tracking rectangle position allows to track maneuvering objects effectively when their angle is changed. The algorithm smoothly shifts the tracking rectangle to the center of the object. **The default value is 1 (enabled).** Any value other than 0 indicates the necessity to use this function.
INPUT_FORMAT	Index of input pixel format: **0** – Grayscale (default). **1** – NV12. **2** – BGR. **3** – RGB. **4** – YUV. **5** – YUYV (YUY2). **6** – UYVY. **Warning: the input pixel format must be set in advance before first video frame processing.** See "Supported pixel formats" section.
NUM_CHANNELS	Number of color channels for processing. See "Supported pixel formats" section. **Warning: the number of color channels for processing must be set in advance before first video frame processing.**

LIBRARY CONTROL COMMANDS

The library and tracking algorithm are controlled using the executeCommand(...) method. The list of commands available listed in union CvtCommand declared in the file CvtDataStructure.h. The declaration of the CvtCommand enum:

```
enum class CvtCommand
{
    CAPTURE = 1,
    RESET,
    SET_INERTIAL_MODE,
    SET_LOST_MODE,
    SET_STATIC_MODE,
    SET_TRACKING_RECTANGLE_AUTO_SIZE,
    SET_TRACKING_RECTANGLE_AUTO_POSITION,
    MOVE_TRACKING_RECTANGLE,
    SET_TRACKING_RECTANGLE_POSITION,
    SET_TRACKING_RECTANGLE_POSITION_PERCENTS,
    CHANGE_TRACKING_RECTANGLE_SIZE,
    CAPTURE_PERCENTS
};
```

Table 10 – Library control commands.

Command	Description
CAPTURE	Object capture command. If the tracking algorithm is not in **FREE** mode, the reset command will be executed first and then the object will be captured.
RESET	Tracking reset command. Switches the tracking algorithm to **FREE** mode from any other mode.
SET_INERTIAL_MODE	Command to switch the tracking algorithm into I**NERTIAL** mode from **TRACKING**, **LOST** and **STATIC** modes.
SET_LOST_MODE	Command to switch the tracking algorithm into **LOST** mode from **TRACKING**, **INERTIAL** and **LOST** modes.
SET_STATIC_MODE	Command to switch the algorithm into **STATIC** mode from **TRACKING**, **LOST** and **INERTIAL** modes.
SET_TRACKING_RECTANGLE _AUTO_SIZE	Command to automatically adjust the position and size of the tracking rectangle. In **TRACKING** mode, for each processed video frame, the library calculates the position and size of the object rectangle within the tracking rectangle. If an object has not been captured optimally (e.g. over the edge of the object) or if the tracking rectangle has moved to the edge of the object during tracking, the operator can correct the position and size of tracking rectangle without reset and re-capture object, and without having to manually control the size and position of the tracking rectangle. This command shifts the position of the tracking rectangle so that the object is in the center of the tracking rectangle and sets the size of the tracking rectangle to 50% larger horizontally and vertically than the object size.
SET_TRACKING_RECTANGLE _AUTO_POSITION	The command to automatically correct the position of the tracking rectangle in **TRACKING** mode. In **TRACKING** mode, the library calculates the position and size of the object rectangle in the tracking rectangle for each processed video frame. When this command is executed, the library shifts the tracking rectangle so that the object is in the center of the tracking rectangle.
MOVE_TRACKING_RECTANGLE	Tracking rectangle position correction command in **TRACKING** mode. If an object is captured over the edge of the object or if the tracking rectangle is displaced during tracking, the operator has the option of correcting the position of the tracking rectangle without having to reset and re-capture object.
SET_TRACKING_RECTANGLE _POSITION	Set tracking rectangle position in **FREE** mode. In command the user must set horizontal and vertical position of the tracking rectangle center. In **FREE** mode the algorithm doesn't change

Command	Description
	position of tracking rectangle but user can manual change tracking rectangle position in library's result data.
SET_TRACKING_RECTANGLE _POSITION_PERCENTS	Set tracking rectangle position in **FREE** mode. In command the user must set horizontal and vertical position of the tracking rectangle center in precents of frame width/height multiple by 1000. In FREE mode the algorithm doesn't change position of tracking rectangle but user can manually change tracking rectangle position in library's result data.
CHANGE_TRACKING _RECTANGLE_SIZE	Change current tracking rectangle size in all modes. In command the user must set add (+/-) to the size of the rectangle.
CAPTURE_PERCENTS	Capture object with coordinates in percent of frame width/height multiple by 1000.

OBTAINING IMAGES FROM LIBRARY

The library allows the user to obtain images of the internal matrices of the tracking algorithm. The available matrices are listed in the enum CvtImage, declared in the file CvtDataStructures.h. The declaration of the CvtImage enum:

```
enum class CvtImage
{
    PATTERN_IMAGE = 1,
    MASK_IMAGE,
    CORRELATION_SURFACE_IMAGE
};
```

Table 11 – Internal matrices of the tracking algorithm.

Matrix	Description
PATTERN_IMAGE	The reference image of the object. The reference image is generated at the moment the object is captured and updated during tracking. The size of the reference image is always **256x256** pixels.
MASK_IMAGE	Object mask image. Object mask is matrix from which the algorithm calculates the position and size of the object (object rectangle) into the tracking rectangle. The size of the object mask is always **256x256** pixels.
CORRELATION _SURFACE_IMAGE	The image of the surface of the probability distribution (correlation surface) relative to the position of the center of the tracking rectangle on the previous video frame. The size of the probability distribution surface is always equal to the maximum size of the search window of **256x256** pixels.

RESULTS STRUCTURE

For each processed video frame, the library updates the CvtResults structure declared in the CvtDataStructures.h file. The CvtResults structure:

```
typedef struct
{
    int mode;
    int trackingRectangleCenterX;
    int trackingRectangleCenterY;
    int trackingRectangleWidth;
    int trackingRectangleHeight;
    int objectCenterX;
    int objectCenterY;
    int objectWidth;
    int objectHeight;
    int frameCounterInLostMode;
    int frameCounter;
    int frameWidth;
    int frameHeight;
    int searchWindowWidth;
    int searchWindowHeight;
    int searchWindowCenterX;
    int searchWindowCenterY;
    int lostModeOption;
    int frameBufferSize;
    int maximumNumberOfFramesInLostMode;
    int trackerFrameID;
    int bufferFrameID;
    float horizontalObjectVelocity;
    float verticalObjectVelocity;
    float objectDetectionProbability;
    bool useTrackingRectangleAutoSize;
    bool useTrackingRectangleAutoPosition;
} CvtResults;
```

Table 12 – Description of the CvtResults structure fields.

Field	Description
mode	Tracking algorithm mode: **0 – FREE**, **1 – TRACKING**, **2 – LOST**, **3 – INERTIAL**, **4 – STATIC**. See “Operational modes”.
trackingRectangleCenterX	Horizontal coordinate of the center of the tracking rectangle.
trackingRectangleCenterY	Vertical coordinate of the center of the tracking rectangle.
trackingRectangleWidth	Width of the tracking rectangle.
trackingRectangleHeight	Height of the tracking rectangle.
objectCenterX	Horizontal coordinate of the object rectangle center. During tracking, the algorithm estimates the position and size of the object in the tracking rectangle. This information can be used by the user to adjust the size and position of the tracking rectangle.
objectCenterY	Vertical coordinate of the object rectangle center.
objectWidth	Width of the object rectangle.
objectHeight	Height of the object rectangle.
frameCounterInLostMode	Frame counter in **LOST** mode. The counter is reset to 0 when you go to **LOST** mode and counts the number of frames while the algorithm is in this mode.
frameCounter	Frame counter from the moment an object is captured.
frameWidth	Width of video frames.
frameHeight	Height of video frames.
searchWindowWidth	Width of search window.
searchWindowHeight	Height of search window.
searchWindowCenterX	The horizontal coordinate of the center of the search window for the next video frame.
searchWindowCenterY	The vertical coordinate of the center of the search window for the next video frame.

Field	Description
lostModeOption	Mode of updating coordinates of object rectangle center in **LOST** mode (see “Tracking algorithm options”).
frameBufferSize	Frame buffer size.
maximumNumberOf FramesInLostMode	Maximum number of video frames in **LOST** mode before automatic tracking reset.
trackerFrameID	Identifier of the last video frame processed by the tracking algorithm. The video frame ID is the index of the frame in the frame buffer.
bufferFrameID	Identifier of the last frame added to the buffer. The video frame identifier is an index of the frame in the frame buffer and is used to implement the STOP-FRAME function and to compensate for communication delays (see “STOP-FRAME function” and “Communication channel delay compensation”).
horizontalObjectValocity	The horizontal component of the velocity of an object in video frames (pixels per frame).
verticalObjectVelocity	The vertical component of the velocity of an object in video frames (pixels per frame).
objectDetectionProbability	The current value of the object detection probability calculated by the library for the last video frame processed.
useTrackingRectangle AutoSize	The flag of the active mode of automatic adjustment of the tracking rectangle size during object tracking.
useTrackingRectangle AutoPosition	The flag of the active mode of automatic adjustment of the tracking rectangle position during object tracking.

TRACKING ALGORITHM CLASS DESCRIPTION

Declaration of the Cvt class

The **Cvt** C++ class is declared in the Cvt.h file. The declaration of the Cvt C++ class:

```
namespace cr
{
namespace vtracker
{

class Cvt
{
public:

    static std::string getVersion();

    Cvt();

    ~Cvt();

    bool setParam(CvtParam id, float value);

    float getParam(CvtParam id);

    bool processFrame(uint8_t *frame, int width, int height, int timeoutMsec = 0);
    bool executeCommand(CvtCommand id, int arg1 = -1, int arg2 = -1,
                        int arg3 = -1, uint8_t* arg4 = nullptr);

    CvtResults getResults();

    bool getImage(CvtImage type, uint8_t* image);
};
}
}
```

Table 13 – Cvt C++ class methods.

Method	Description
Cvt()	Class constructor.
~Cvt()	Class destructor.
setParam(…)	Set library parameter.
getParam(…)	Get library parameter.
processFrame(…)	Process video frame.
executeCommand(…)	Execute command.
getResults()	Get results data.
getImage(…)	Get image from library.
getVersion()	Get library version.

setParam(…) method

The setParam(…) method is intended to modify library parameters. The declaration of the method:

```
bool setParam(CvtParam id, float value);
```

Parameters:

id	Parameter ID (see “Library parameters”).
value	Parameter value.

Return value:
The method returns **TRUE** if the parameter is set. Otherwise, the method returns **FALSE**.

getParam(…) method

The getParam(...) method is intended to get the value of a library parameter. The declaration of the method:

```
float getParam(CvtParam id);
```

Parameters:

id	Parameter ID to retrieve (see “Library parameters”).

Return value:
The method returns the value of the parameter or returns **-1** if the parameter with this identifier does not exist.

processFrame(…) method

The processFrame(...) method processes the video frame. The declaration of the method:

```
bool processFrame(uint8_t *frame, int width, int height, int timeoutMsec = 0);
```

Parameters:

frame	Pointer to frame data according to pixel format set in advance (see “Supported pixel formats”).
width	Width of video frames to be processed.
height	Height of video frames to be processed.
timeoutMsec	The time limit in which the method must return control. If the value is 0, it tells the method to process all new video frames added to the buffer. The time interval must be set based on the frequency of the frames to be processed. The time interval must correspond to at least two periods of video frames for the **STOP-FRAME** function to be implemented.

The first time the method is called, the library allocates memory for the frame buffer. The method copies the video frame data to the frame buffer and processes it if the tracking algorithm is not in **FREE** mode. If a frame has a different size than the last frame (width and heigh), the library resets tracking, releases frame buffer memory and reinitialize it according to the new video frame size. If the processed frame is not the last frame added to the buffer, the method processes the frames in sequence until the last frame added to the buffer is processed or until the specified time (timeoutMsec parameter) expires.

Return value:
Method returns **TRUE** if frame is successfully added to buffer and processed. Method returns **FALSE**, if width and/or height of video frame are **0**.

executeCommand(…) method

The executeCommand(...) method is used to execute a command. The declaration of the method:

```
bool executeCommand(CvtCommand id, int arg1 = -1, int arg2 = -1, int arg3 = -1,
                    uint8_t* arg4 = nullptr);
```

Parameters:

id	Command ID according to CvtCommand enum (see “Library control commands”).
arg1	The first argument. Has different meaning depends on the command:

CAPTURE	Horizontal coordinate of center of tracking rectangle.
RESET	not used.
SET_INERTIAL_MODE	not used.
SET_LOST_MODE	not used.
SET_STATIC_MODE	not used.
SET_TRACKING_ RECTANGLE_AUTO_SIZE	not used.
MOVE_TRACKING_RECTANGLE	Horizontal offset of the tracking rectangle.
SET_TRACKING _RECTANGLE_POSITION	Horizontal position of tracking rectangle center.
SET_TRACKING_RECTANGLE _POSITION_PERCENTS	Horizontal position of tracking rectangle center.

	CHANGE_TRACKING _RECTANGLE_SIZE	Add to tracking rectangle width.
	CAPTURE_PERCENTS	Horizontal position of tracking rectangle center in percents multiple by 1000.
arg2	The second argument. Has different values depending on the command:	
	CAPTURE	Vertical coordinate of center of tracking rectangle.
	RESET	not used.
	SET_INERTIAL_MODE	not used.
	SET_LOST_MODE	not used.
	SET_STATIC_MODE	not used.
	SET_TRACKING_RECTANGLE _AUTO_SIZE	not used.
	MOVE_TRACKING_RECTANGLE	Vertical offset of the tracking rectangle in pixels.
	SET_TRACKING _RECTANGLE_POSITION	Vertical position of tracking rectangle center.
	SET_TRACKING_RECTANGLE _POSITION_PERCENTS	Vertical position of tracking rectangle center.
	CHANGE_TRACKING _RECTANGLE_SIZE	Add to tracking rectangle width.
	CAPTURE_PERCENTS	Vertical position of tracking rectangle center in percents multiple by 1000.
arg3	The third argument. Only used in the CAPTURE command and is the identifier of the frame on which you want to perform a object capture. A value of -1 instructs the library to perform a tracking object capture on the last frame added to the buffer. If the frame ID is greater than or equal to 0, the **arg4** parameter will be ignored.	
arg4	Not used.	

Return value:
The method returns **TRUE** if the command was executed successfully. Otherwise, it returns **FALSE**.

getResults(…) method

The getResults(...) method is designed to get the current tracker result data (see “Results structure”). The declaration of the method:

```
CvtResults getResults();
```

Return value:
Method returns CvtResults structure (see “Results structure”).

getImage(…) method

The getImage(...) method is designed to retrieve images of the internal matrices of the tracking algorithm. The declaration of the method:

```
bool getImage(CvtImage type, uint8_t* image);
```

Parameters:

type	The type of image to be obtained (see “Obtaining images”). The following are available to the user: **PATTERN_IMAGE** – reference image of an object, **MASK_IMAGE** – object mask image, **CORRELATION_SURFACE_IMAGE** – image of the distribution surface of the probability of object detection.
image	A pointer to the image buffer to be filled. All generated images are in Grayscale format (1 byte per pixel in grayscale). Image buffer size should correspond to 256x256 pixels in mono8 format = 65536 bytes.

Return value:
Method returns **TRUE** if image buffer is full. The method returns **FALSE** if the specified identifier of the received image type does not exist.

getVersion() method

The static method getVersion() is designed to get the string of the current library version. The declaration of the method is given below.

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

Return value:
The method returns a string of the current software library version in the format **“8.0.0”**.

The method can be called without creating an object of the Cvt class, as shown below.

```
cout << "Cvt lib v" << Cvt::getVersion() << endl;
```

EXAMPLE OF USING THE LIBRARY

SIMPLE DEMO APPLICATION

Below is the source code for a simple demo application to test the library. It uses the OpenCV open source library to capture video frames and create the user interface.

```
#include <opencv2/opencv.hpp>
#include "Cvt.h"

// Link namespaces.
using namespace cv;
using namespace std;
using namespace cr::vtracker;

// Global variables.
Cvt g_tracker;
int g_frameWidth = 0;
int g_frameHeight = 0;
```

```cpp
// Prototype of mouse callback function.
void MouseCallBackFunc(int event, int x, int y, int flags, void* userdata);

// Entry point.
int main(void)
{
    cout << "======================================================" << endl;
    cout << "Simple Demo Application for CVT v" << Cvt::getVersion() << endl;
    cout << "======================================================" << endl;

    // Dialog to enter video source init string.
    string initString = "0";
    cout << "Enter video source init string "
    << "(camera num, rtsp string, video file): ";
    cin >> initString;
    cout << initString << endl;

    // Open video source.
    VideoCapture videoSource;
    bool result = false;
    if (initString.size() < 5)
        result = videoSource.open(stoi(initString));
    else
        result = videoSource.open(initString);
    if (result == false)
    {
        cout << "ERROR: Video source not open" << endl;
        return -1;
    }

    // Init variables.
    Mat frame;
    Mat yuvFrame;
    CvtResults trackerData;

    // Set tracker parameters.
    g_tracker.setParam(CvtParam::TRACKING_RECTANGLE_WIDTH, 72);
    g_tracker.setParam(CvtParam::TRACKING_RECTANGLE_HEIGHT, 72);
    g_tracker.setParam(CvtParam::LOST_MODE_OPTION, 0);
    // Set YUV input data format.
    g_tracker.setParam(CvtParam::INPUT_FORMAT, 4);
    // Set 3 channels (Y, U and V).
    g_tracker.setParam(CvtParam::NUM_CHANNELS, 3);

    // Init OpenCV window.
    namedWindow("Simple demo application", WINDOW_AUTOSIZE);
    // Set mouse callback.
    setMouseCallback("Simple demo application", MouseCallBackFunc, nullptr);

    // Main loop.
    while (true)
    {
        // Capture next video frame.
        videoSource >> frame;
        if (frame.empty())
        {
            // If we have video file we can set initial position to replay.
            videoSource.set(CAP_PROP_POS_FRAMES, 0);
            continue;
        }

        // Update frame size.
        g_frameWidth = frame.size().width;
        g_frameHeight = frame.size().height;

        // Convert to YUV format.
        cvtColor(frame, yuvFrame, COLOR_BGR2YUV);

        // Process video frame.
        g_tracker.processFrame(
        yuvFrame.data, yuvFrame.size().width, yuvFrame.size().height);

        // Get current tracker data.
        trackerData = g_tracker.getResults();

        // Choose tracking rectangle color according to tracker mode.
        Scalar rectColor;
        switch (trackerData.mode) {
        case CVT_FREE_MODE_INDEX: rectColor = Scalar(255, 255, 255); break;
        case CVT_TRACKING_MODE_INDEX: rectColor = Scalar(0, 0, 255); break;
        case CVT_LOST_MODE_INDEX: rectColor = Scalar(255, 0, 0); break;
        default: rectColor = Scalar(255, 255, 255); break; }

        // Draw tracking rectangle.
        Rect rect(trackerData.trackingRectangleCenterX -
                  trackerData.trackingRectangleWidth / 2,
                  trackerData.trackingRectangleCenterY -
                  trackerData.trackingRectangleHeight / 2,
                  trackerData.trackingRectangleWidth,
                  trackerData.trackingRectangleHeight);
        rectangle(frame, rect, rectColor, 2);

        // Show video with tracker result information.
        imshow("Simple demo application", frame);

        // Wait keyboard events.
        switch (waitKey(25))
        {
        // ESC - Exit.
        case 27:
            destroyAllWindows();
            return 1;
        // W - Increase tracking rectangle height.
        case 119:
            g_tracker.executeCommand(
            CvtCommand::CHANGE_TRACKING_RECTANGLE_SIZE, 0, 8);
            break;
        // S - Decrease tracking rectangle height.
        case 115:
            g_tracker.executeCommand(
            CvtCommand::CHANGE_TRACKING_RECTANGLE_SIZE, 0, -8);
            break;
        // D - Increase tracking rectangle width.
        case 100:
            g_tracker.executeCommand(
            CvtCommand::CHANGE_TRACKING_RECTANGLE_SIZE, 8, 0);
            break;
        // A - Decrease tracking rectangle width.
        case 97:
            g_tracker.executeCommand(
            CvtCommand::CHANGE_TRACKING_RECTANGLE_SIZE, -8, 0);
            break;
        // T - Move strobe UP (change position in TRACKING mode).
        case 116:
            g_tracker.executeCommand(
            CvtCommand::MOVE_TRACKING_RECTANGLE, 0, 4);
            break;
        // G - Move strobe DOWN (change position in TRACKING mode).
        case 103:
            g_tracker.executeCommand(
            CvtCommand::MOVE_TRACKING_RECTANGLE, 0, -4);
            break;
        // H - Move strobe RIGHT (change position in TRACKING mode).
        case 104:
            g_tracker.executeCommand(
            CvtCommand::MOVE_TRACKING_RECTANGLE, -4, 0);
            break;
        // F - Move strobe LEFT (change position in TRACKING mode).
        case 102:
            g_tracker.executeCommand(
            CvtCommand::MOVE_TRACKING_RECTANGLE, 4, 0);
            break;
        }
    }
}

// Mouse callback function.
void MouseCallBackFunc(int event, int x, int y, int flags, void* userdata)
{
    // Set mouse position in any case.
    g_tracker.executeCommand(CvtCommand::SET_TRACKING_RECTANGLE_POSITION, x, y);

    switch (event)
    {
    /// Capture object.
    case cv::EVENT_LBUTTONDOWN:
        g_tracker.executeCommand(CvtCommand::CAPTURE, x, y);
        break;
    /// Reset tracker.
    case cv::EVENT_RBUTTONDOWN:
        g_tracker.executeCommand(CvtCommand::RESET);
        break;
    case cv::EVENT_MBUTTONDOWN:
        break;
    case cv::EVENT_MOUSEMOVE:
        break;
    }
}
```

EXPLANATION OF SIMPLE DEMO APPLICATION

The demo application opens a video file, captures video frames and passes them to the tracking library for processing. The application only requires the OpenCV library header files and the tracking library to be connected:

```
#include <opencv2/opencv.hpp>
#include "Cvt.h"

// Link namespaces.
using namespace cv;
using namespace std;
using namespace cr::vtracker;
```

After the header files are included, the following global variables are declared: mouse pointer coordinates to control tracking capture and reset and instance of the Cvt class:

```
// Global variables.
Cvt g_tracker;
int g_frameWidth = 0;
int g_frameHeight = 0;
```

The prototype mouse event processing function is then declared:

```
// Prototype of mouse callback function.
void MouseCallBackFunc(int event, int x, int y, int flags, void* userdata);
```

In the main function the user must set name of video source. It can be video file or camera num. After the video source name is set the application opens video source.

```
// Dialog to enter video source init string.
string initString = "0";
cout << "Enter video source init string "
```

```
<< "(camera num, rtsp string, video file): ";
cin >> initString;
cout << initString << endl;

// Open video source.
VideoCapture videoSource;
bool result = false;
if (initString.size() < 5)
    result = videoSource.open(stoi(initString));
else
    result = videoSource.open(initString);
if (result == false)
{
    cout << "ERROR: Video source not open" << endl;
    return -1;
}
```

Variables for video frames are declared after opening a video file:

```
// Init variables.
Mat frame;
Mat yuvFrame;
CvtResults trackerData;
```

After that, the initial size of the tracking rectangle is set (default dimensions are 128x128 pixels) through a call to the setParam(...) method, as well the YUV input pixel form and 3 color channels for processing (see “Library parameters”):

```
// Set tracker parameters.
g_tracker.setParam(CvtParam::TRACKING_RECTANGLE_WIDTH, 72);
g_tracker.setParam(CvtParam::TRACKING_RECTANGLE_HEIGHT, 72);
g_tracker.setParam(CvtParam::LOST_MODE_OPTION, 0);
// Set YUV input data format.
g_tracker.setParam(CvtParam::INPUT_FORMAT, 4);
// Set 3 channels (Y, U and V).
g_tracker.setParam(CvtParam::NUM_CHANNELS, 3);
```

Next, the video output windows are declared, and the mouse event handler is registered:

```
// Init OpenCV window.
namedWindow("Simple demo application", WINDOW_AUTOSIZE);
// Set mouse callback.
setMouseCallback("Simple demo application", MouseCallBackFunc, nullptr);
```

Next is an endless loop of video frame capture, processing, and display. In the loop, the next video frame is captured first. When the end of the video file is reached, playback starts again.

```
// Capture next video frame.
videoSource >> frame;
if (frame.empty())
```

```
{
    // If we have video file we can set initial position to replay.
    videoSource.set(CAP_PROP_POS_FRAMES, 0);
    continue;
}
```

Captured video frame with OpenCV library is in BGR 24 bits format. To be processed by the library, it must be converted to YUV format:

```
// Convert to YUV format.
cvtColor(frame, yuvFrame, COLOR_BGR2YUV);
```

The resulting video frame in YUV format is sent to the library for processing. After processing we get the results of work (tracking data).

```
// Process video frame.
g_tracker.processFrame(
yuvFrame.data, yuvFrame.size().width, yuvFrame.size().height);

// Get current tracker data.
trackerData = g_tracker.getResults();
```

Next, draw the position of the tracking rectangle on the original video. In **FREE** mode, draw a white capture rectangle with the center coinciding with the position of the mouse pointer:

```
// Choose tracking rectangle color according to tracker mode.
Scalar rectColor;
switch (trackerData.mode) {
case CVT_FREE_MODE_INDEX: rectColor = Scalar(255, 255, 255); break;
case CVT_TRACKING_MODE_INDEX: rectColor = Scalar(0, 0, 255); break;
case CVT_LOST_MODE_INDEX: rectColor = Scalar(255, 0, 0); break;
default: rectColor = Scalar(255, 255, 255); break; }

// Draw tracking rectangle.
Rect rect(trackerData.trackingRectangleCenterX -
          trackerData.trackingRectangleWidth / 2,
          trackerData.trackingRectangleCenterY -
          trackerData.trackingRectangleHeight / 2,
          trackerData.trackingRectangleWidth,
          trackerData.trackingRectangleHeight);
rectangle(frame, rect, rectColor, 2);
```

Next, display the video and handle keyboard events (exit the application by pressing ESC).

```
// Show video with tracker result information.
imshow("Simple demo application", frame);

// Wait keyboard events.
switch (waitKey(25))
{
```

```
// ESC - Exit.
case 27:
    destroyAllWindows();
    return 1;
// W - Increase tracking rectangle height.
case 119:
    g_tracker.executeCommand(
    CvtCommand::CHANGE_TRACKING_RECTANGLE_SIZE, 0, 8);
    break;
// S - Decrease tracking rectangle height.
case 115:
    g_tracker.executeCommand(
    CvtCommand::CHANGE_TRACKING_RECTANGLE_SIZE, 0, -8);
    break;
// D - Increase tracking rectangle width.
case 100:
    g_tracker.executeCommand(
    CvtCommand::CHANGE_TRACKING_RECTANGLE_SIZE, 8, 0);
    break;
// A - Decrease tracking rectangle width.
case 97:
    g_tracker.executeCommand(
    CvtCommand::CHANGE_TRACKING_RECTANGLE_SIZE, -8, 0);
    break;
// T - Move strobe UP (change position in TRACKING mode).
case 116:
    g_tracker.executeCommand(
    CvtCommand::MOVE_TRACKING_RECTANGLE, 0, 4);
    break;
// G - Move strobe DOWN (change position in TRACKING mode).
case 103:
    g_tracker.executeCommand(
    CvtCommand::MOVE_TRACKING_RECTANGLE, 0, -4);
    break;
// H - Move strobe RIGHT (change position in TRACKING mode).
case 104:
    g_tracker.executeCommand(
    CvtCommand::MOVE_TRACKING_RECTANGLE, -4, 0);
    break;
// F - Move strobe LEFT (change position in TRACKING mode).
case 102:
    g_tracker.executeCommand(
    CvtCommand::MOVE_TRACKING_RECTANGLE, 4, 0);
    break;
}
```

In the mouse event processing function, objects are captured and reset using the executeCommand(...) method. Pressing the left mouse button performs a tracking capture if the algorithm is in **FREE** mode or a tracking reset in other cases.

```
void MouseCallBackFunc(int event, int x, int y, int flags, void* userdata)
```

```
{
    // Set mouse position in any case.
    g_tracker.executeCommand(CvtCommand::SET_TRACKING_RECTANGLE_POSITION, x, y);

    switch (event)
    {
    /// Capture object.
    case cv::EVENT_LBUTTONDOWN:
        g_tracker.executeCommand(CvtCommand::CAPTURE, x, y);
        break;
    /// Reset tracker.
    case cv::EVENT_RBUTTONDOWN:
        g_tracker.executeCommand(CvtCommand::RESET);
        break;
    case cv::EVENT_MBUTTONDOWN:
        break;
    case cv::EVENT_MOUSEMOVE:
        break;
    }
}
```

PROTOCOL PARSER LIBRARY

DESCRIPTION

In many applications there we need remote control of the tracking algorithm. For example, when designing an automatic tracking device, the designer needs to be able to control the device through communication channels. To do so, it needs to define a control protocol and implement message encoding and decoding functions. The message parser software library for the CvTracker Parser library is designed to facilitate this task. The library is delivered as source code files including: **CvtParser.h**, **CvtParser.cpp**, **CvtDataStructures.h** (from the tracking library) and **CvtParserVersion.h**. The CvtParser.h file contains the declaration of the **CvtParser** C++ class, which implements the message encoding and decoding functions. To use the library, the developer needs to include source code files in C++ project.

MESSAGE FORMATS

Message types

The protocol parser library defines three types of messages:
1. SET_PARAM – command to set parameter (see “Library parameters”).
2. COMMAND – command to the tracking library (see “Library control commands”).
3. DATA – results data (see “Results structure”).

SET_PARAM message format

byte:	0	1	2	3	4	5	6	7	8	9	10
hex:	0x01	0x08	0x00	Int32_t id				float value			

Message size 11 bytes.

Message fields:

int32_t id	Parameter identifier (see “Library parameters”). The value is in **little endian** format.
float value	Parameter value.

COMMAND message format

byte:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
hex:	**0x02**	**0x08**	**0x00**	int32_t id				int32_t arg1				int32_t arg2			
byte:	**15**	**16**	**17**	**18**											
hex:	int32_t arg3														

Message size 19 bytes.

Message fields:

int32_t id	Command ID (see “Library control commands”).
int32_t arg1	The value of the first argument of the command (see “executeCommand(...) method”). The value is in **little endian** format.
int32_t arg2	The value of the second argument of the command (see “executeCommand(...) method”). The value is in **little endian** format.
int32_t arg3	The value of the third argument of the command (see “executeCommand(...) method”). The value is in **little endian** format.

DATA message format

The DATA message includes values of fields of CvtResults structure declared in CvtDataStructures.h file. Number of fields from CvtResults structure included in message can vary. Below is an example of data packet with all fields included.

byte:		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
hex:		0x00	0x80	0x00	fields mask = **0xFFFFFFFF**				trackingRectangleCenterX				trackingRectangleCenterY			
byte:	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**
hex:	trackingRectangleWidth				trackingRectangleHeight				objectCenterX				objectCenterY			
byte:	**31**	**32**	**33**	**34**	**35**	**36**	**37**	**38**	**39**	**40**	**41**	**42**	**43**	**44**	**45**	**46**
hex:	objectWidth				objectHeight				frameCounterInLostMode				frameCounter			
byte:	**47**	**48**	**49**	**50**	**51**	**52**	**53**	**54**	**55**	**56**	**57**	**58**	**59**	**60**	**61**	**62**
hex:	frameWidth				frameHeight				searchWindowWidth				searchWindowHeight			
byte:	**63**	**64**	**65**	**66**	**67**	**68**	**69**	**70**	**71**	**72**	**73**	**74**	**75**	**76**	**77**	**78**
hex:	searchWindowCenterX				searchWindowCenterY				lostModeOption				frameBufferSize			
byte:	**79**	**80**	**81**	**82**	**83**	**84**	**85**	**86**	**87**	**88**	**89**	**90**	**91**	**92**	**93**	**94**
hex:	maximumNumberOfFramesInLostMode				trackerFrameID				bufferFrameID				horizontalObjectVelocity			
byte:	**95**	**96**	**97**	**98**	**99**	**100**	**101**	**102**	**103**	**104**	**105**	**106**	**107**	**108**	**109**	**110**
hex:	verticalObjectVelocity				objectDetectionProbability				mode				**A***	**B***		

A* = useTrackingRectangleAutoSize (0 == false, 1 == true).
B* = useTrackingRectangleAutoPosition (0 == false, 1 == true).

Message fields:

field mask	Bitmask of the CvtResults structure fields included in the message. The bit mask is formed as follows: a single value of each bit starting from the left bit of byte #3 indicates the inclusion of the corresponding field of the CvtResults structure in the packet in order of declaration. The last four bits of the byte 5 group are not used and can have any value. In the above example, all bits of the mask are single, indicating that all fields of the CvtResults structure are included in the packet. When decoding a DATA message, the receiver reads the bitmask and decodes the message accordingly.
data fields	CvtResults structure data fields according to declaration.

Assuming, that we want to encode only trackingRectangleCenterX and trackingRectangleCenterY fields, then to do this we need to form an appropriate bitmask and include only the specified fields of the CvtResults structure in the message. To form the bitmask, we must define the sequence number of the declaration of the corresponding field of the CvtResults structure. The trackingRectangleCenterX field is declared first and corresponds to the leftmost bit of the bitmask. The trackingRectangleCenterY field corresponds to the second bit. So, the bitmask looks as follow:

byte:	**3**								**4**								**5**								**6**							
bit:	**1**	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

So, the message will be 15 bytes long, as shown below:

byte:	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
hex:	**0x00**	**0x08**	**0x01**	**0xC0**	**0x00**	**0x00**	**0x00**	trackingRectangleCenterX				trackingRectangleCenterY			

PROTOCOL PARSER CLASS DESCRIPTION

CvtParser class

The CvtParser C++ class is declared in the CvtParser.h file and contains methods of encoding and decoding messages for the tracker library. The class declaration is given below.

```
namespace cr
{
namespace vtracker
{
class CvtParser
{
public:

    CvtParser();

    ~CvtParser();

    void encodeParam(uint8_t* data, int& size, CvtParam id, float value);

    void encodeCommand(uint8_t* data, int& size, CvtCommand id,
                       int arg1 = -1, int arg2 = -1, int arg3 = -1);
    void encodeResults(uint8_t* data, int& size, CvtResults& results,
                       CvtResultsMask* mask = nullptr);

    int decodePacket(uint8_t* data, int size);

    CvtResults getResults();

    void getParam(CvtParam& id, float& value);

    void getCommand(CvtCommand& id, int& arg1, int& arg2, int& arg3);

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

Table 14 – Methods of CvtParser class.

Method	Description
encodeParam(…)	SET_PARAM command encoding method.
encodeCommand(…)	COMMAND encoding method.
encodeResults(…)	DATA encoding method.
decodePacket(…)	Method for decoding input messages.
getResults()	Method for retrieving the results structure after decoding using decodePacket(...) method.
getParam(…)	Method for retrieving the decoded library parameter after decoding using decodePacket(...) method.
getCommand(…)	Method to retrieve decoded command for library after decoding by decodePacket(...) method.
getVersion()	Static method to retrieve the string of the current library version.

encodeParam(…) method

The encodeParam(...) method is intended to encode SET_PARAM command. The declaration of the method:

```
void encodeParam(uint8_t* data, int& size, CvtParam id, float value);
```

Parameters:

data	Pointer to buffer to be filled with message data. Size of data buffer must be >= 11
size	Message buffer size. Must be at least 15 bytes.
id	Parameter ID to be included in the message (see “Library parameters”).
value	Parameter value to be included in the message.

encodeCommand(…) method

The encodeCommand(...) method is used to encode COMMAND messages. The declaration of the method:

```
void encodeCommand(uint8_t* data, int& size, CvtCommand id,
                    int arg1 = -1, int arg2 = -1, int arg3 = -1);
```

Parameters:

data	Pointer to message buffer. Size of data buffer must be >= 19.
size	Size of the generated message. 19 bytes.
id	Command ID to be included in the message.
arg1	The first argument of the command (see “executeCommand(...) method”).
arg2	The second argument of the command (see “executeCommand(...) method”).
arg3	The third argument of the command (see “executeCommand(...) method”).

encodeResults(…) method

The encodeResults(...) method is used to encode DATA messages. The declaration of the method:

```
void encodeResults(uint8_t* data, int& size, CvtResults& results,
                   CvtResultsMask* mask = nullptr);
```

Parameters:

data	Pointer to message buffer. Data buffer size must be >= 109.
size	The size of the result message.
results	Results structure (see “Results structure”).

mask	Pointer to the CvtResultsMask structure. This structure is declared in the CvtDataStructures.h file and contains fields with names similar to those of the CvtResults structure, but of a logical type. The declaration of the structure is given below.
	```
	typedef struct
	{
	    bool mode;
	    bool trackingRectangleCenterX;
	    bool trackingRectangleCenterY;
	    bool trackingRectangleWidth;
	    bool trackingRectangleHeight;
	    bool objectCenterX;
	    bool objectCenterY;
	    bool objectWidth;
	    bool objectHeight;
	    bool frameCounterInLostMode;
	    bool frameCounter;
	    bool frameWidth;
	    bool frameHeight;
	    bool searchWindowWidth;
	    bool searchWindowHeight;
	    bool searchWindowCenterX;
	    bool searchWindowCenterY;
	    bool lostModeOption;
	    bool frameBufferSize;
	    bool maximumNumberOfFramesInLostMode;
	    bool trackerFrameID;
	```

	```
	    bool bufferFrameID;
	    bool horizontalObjectVelocity;
	    bool verticalObjectVelocity;
	    bool objectDetectionProbability;
	    bool useTrackingRectangleAutoSize;
	    bool useTrackingRectangleAutoPosition;
	} CvtResultsMask;
	```
	 By default all fields of CvtResults structure are included into DATA message, but when the user uses the field mask, only those fields of CvtResults structure which are marked as TRUE in corresponding fields of CvtResultsMask structure will be included into message (see “DATA message format”).

decodePacket(…) method

The decodePacket(...) method is designed to decode input messages. The declaration of the method:

```
int decodePacket(uint8_t* data, int size);
```

Parameters:

data	Pointer to the packet data.
size	The size of the packet data.

Return value:
The method returns **0** if the DATA message is successfully decoded, **1** if the SET_PARAM message is successfully decoded, **2** if the COMMAND message is successfully decoded and **-1** if there are any errors in the message. The value of decoded data can be retrieved using getResults(), getParam(...) and getCommand(...) methods.

getResults() method

The getResults() method is designed to get the decoded results structure (after decoding the message by decodePacket(...) method). The declaration of the method:

```
CvtResults getResults();
```

Return value:
The method returns the results structure (see “Results structure”) according to the decoded message using the decodePacket(...) method.

getParam(…) method

The getParam(...) method is designed to get the decoded parameter from the SET_PARAM message. The declaration of the method:

```
void getParam(CvtParam& id, float& value);
```

Parameters:

id	Parameter ID (see “Library parameters”).
value	Parameter value.

getCommand(…) method

The getCommand(...) method is designed to get the decoded COMMAND message data. The declaration of the method:

```
void getCommand(CvtCommand& id, int& arg1, int& arg2, int& arg3);
```

Parameters:

id	Command ID (see “Library control commands”).
arg1	First command argument (see “executeCommand(...) method”).
arg2	Second command argument (see “executeCommand(...) method”).
arg3	Third command argument (see “executeCommand(...) method”).

getVersion() method

The static getVersion() method is designed to get the string of the current software library version. The declaration of the method is given below.

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

Return value:
The method returns a string of the current software library version in the format **“8.0.0”**.

The method can be called without creating an object of the CvtParser class, as shown below.

```
cout << "CvtParser lib v" << CvtParser::getVersion() << endl;
```